To our customers,

Old Company Name in Catalogs and Other Documents

On April 1st, 2010, NEC Electronics Corporation merged with Renesas Technology Corporation, and Renesas Electronics Corporation took over all the business of both companies. Therefore, although the old company name remains in this document, it is a valid Renesas Electronics document. We appreciate your understanding.

Renesas Electronics website: http://www.renesas.com

April 1st, 2010
Renesas Electronics Corporation

Issued by: Renesas Electronics Corporation (http://www.renesas.com)

Send any inquiries to http://www.renesas.com/inquiry.

Notice

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

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

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

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

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

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

Application Notes

C Compiler Package for 740 Family

Application Notes

M3T-ICC740

Renesas Electronics
www.renesas-electoronics.com

Rev.1.00 2006.09

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 C compiler package.

Note that the contents described in this application note are detailed in each related manual listed below. Refer to the respective manuals for more information.

IMA Assembler Programming Guide (Mitsubishi 740 Family) Rev. 2
IAR C Library Function Reference Guide
ICC Compiler Programming Guide (Mitsubishi 740 Family) Rev. 2
740 Family 740 Family Sample Programs
740 Family Software Manual

This application note contains the information reproduced from the “ICC Compiler Programming Guide” and “IMA Assembler Programming Guide” with the permission of IAR Systems.
Furthermore, this application note was created based on the “740 Family Programming Guidelines <C Language Part> REJ05B0468-0200/Rev. 2.00.”

This application note is composed of the following chapters.
Chapter 1: Introduction to C language.
Chapter 2: Explains about project settings.
Chapter 3: Describes the C compiler ICC740.
Chapter 4: Describes the assembler A740.
Chapter 5: Describes the linker XLINK.
Chapter 6: Describes the debugger.
Chapter 7: Provides tips for coding.
Chapter 8: Explains how to estimate the stack.
Chapter 9: Explains about interrupt handling.

<Symbols and Conventions used in this Application Note>

(RET): Indicates the Return (Enter) key is to be pressed.

△ : Indicates one or more spaces or tabs.

[] : Indicates that the enclosed item can be omitted.
abc : Italics denote the value or label that must actually be input as part of a command.
{a | b} : Indicates that either of the two is to be selected.
… : Indicates that the immediately preceding item is specified one or more times.
H : Integer constants followed by H are in hexadecimal.
0x : Integer constants preceded by 0x are in hexadecimal.
[Menu->Menu Option] : The boldface and -> denote a menu option.

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 used.

(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.

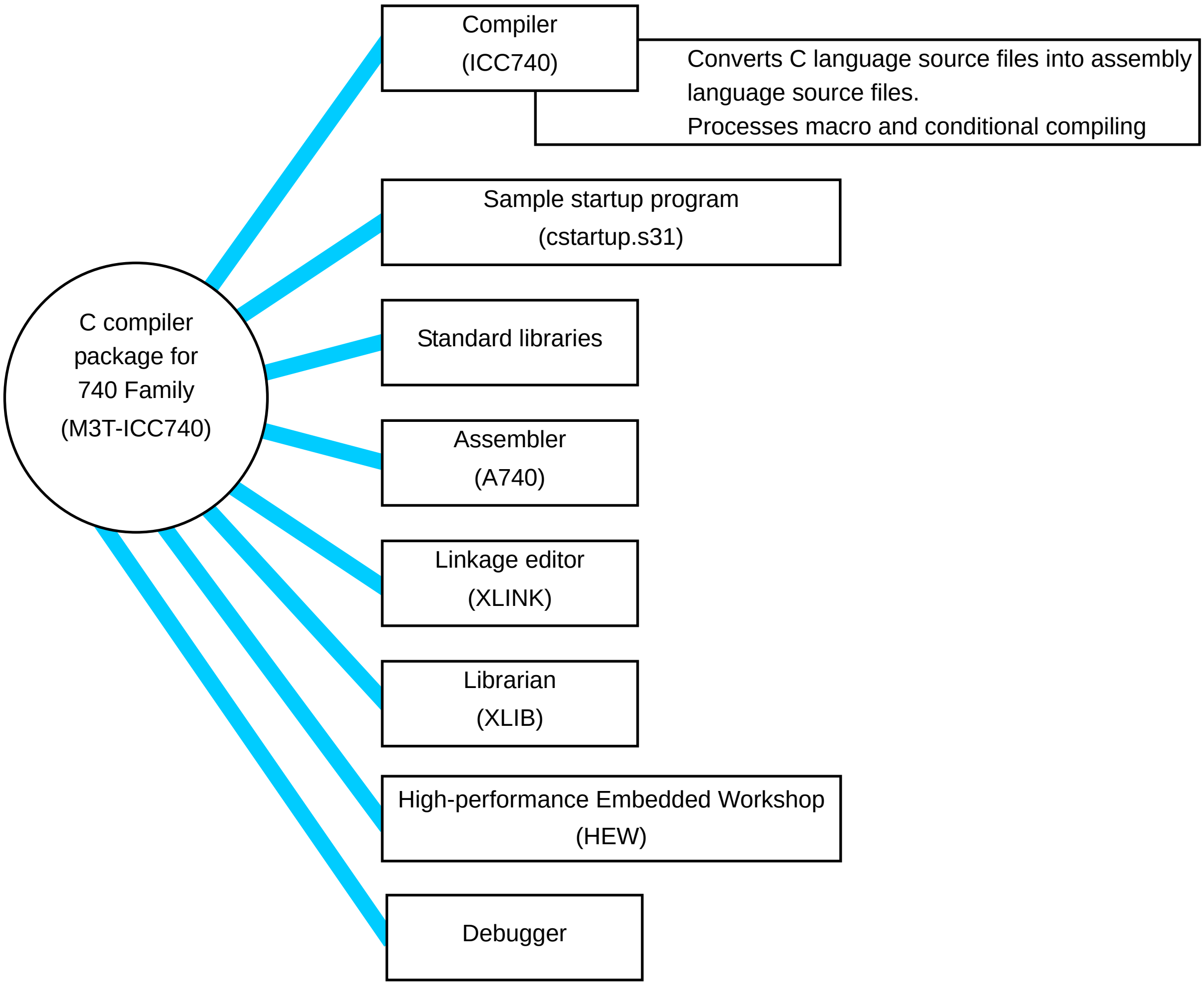
	C language	Assembly language
Basic unit of program (Method of description)	Function (Function name ( ) { })	Subroutine (Subroutine name:)
Format	Based on ANSI C language in free format	1 instruction/line
Discrimination between uppercase and lowercase	Uppercase and lowercase are discriminated	Not discriminated
Allocation of data area	Specified by type	specified by size (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 "compile". The software provided for performing this operation is called a "compiler".
This section explains the procedure for developing a program by using the C compiler package (M3T-ICC740) for the 740 family of Renesas 8-bit single-chip microcomputers.

C Compiler Package (M3T-ICC740) for 740 Family Product List

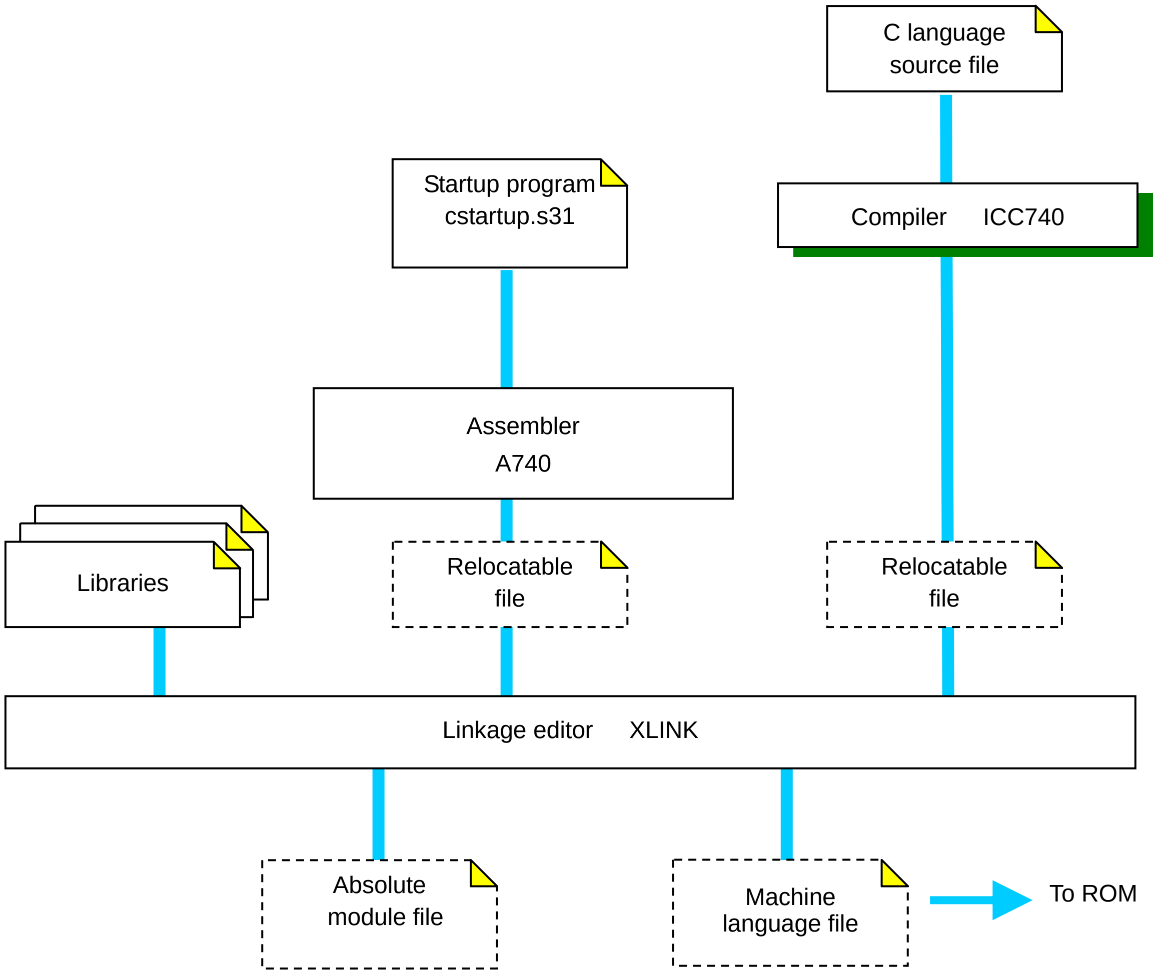
The following lists the products included in the C compiler package (M3T-ICC740) for the Renesas 8-bit single-chip microcomputers 740 family.



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 five items that need to be observed when writing a C language program:

(1) Separate executable statements in a program with a semicolon ";".
(2) Enclose execution units of functions or control statements with brackets "{" and "}"
(3) Functions and variables require type declaration.
(4) Reserved words cannot be used in identifiers (e.g., function names and variable names).
(5) 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

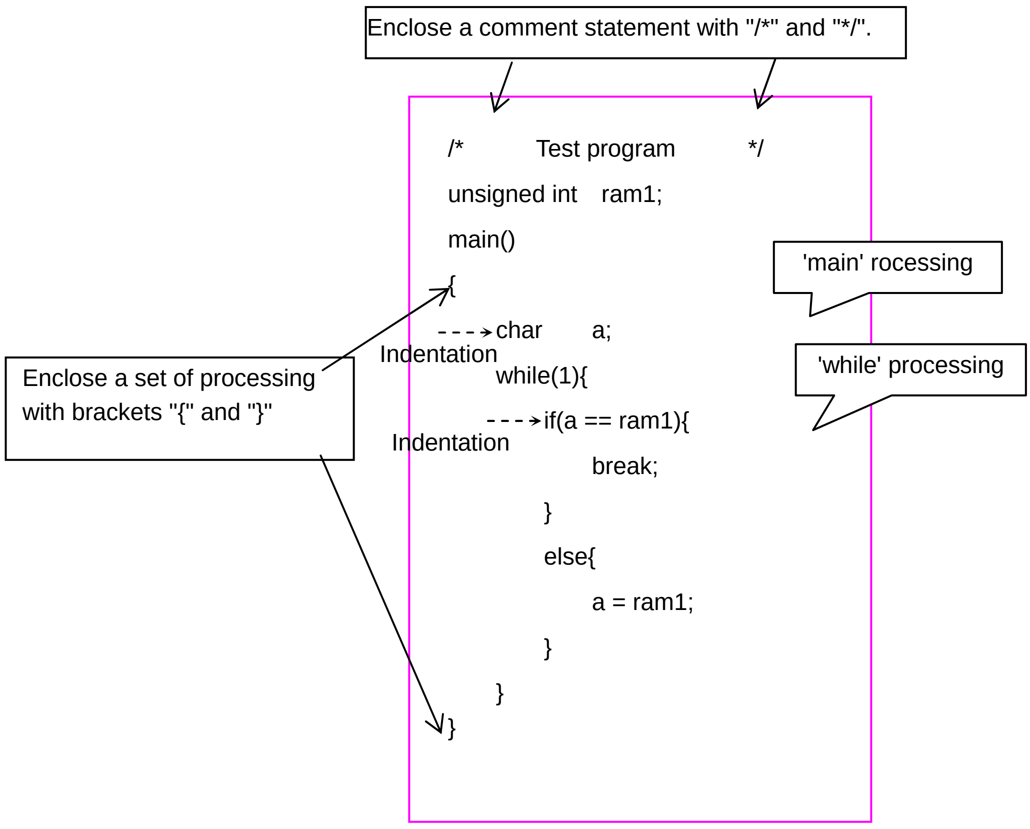
The following schematically shows a configuration of a general C language source file. For each item in this file, refer to the section indicated with an arrow.

Reading header file	→ Refer to 1.9, "Preprocess Commands".
Type declaration of functions used;	→ Refer to 1.5, "Functions".
Macro definition	→ Refer to 1.9, "Preprocess Commands".
Declaration of external variables	→ Refer to 1.2, "Date Types" and 1.6, "Storage Classes".
Type function name (dummy argument, ...)	→ Refer to 1.5, "Functions".
{	
Internal variables definition;	→ Refer to 1.2, "Date Types" and 1.6, "Storage Classes".
Statement;	→ Refer to 1.3, "Operators" and 1.4, "Control Statements".
⋮	
}	

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	

* To use "-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.

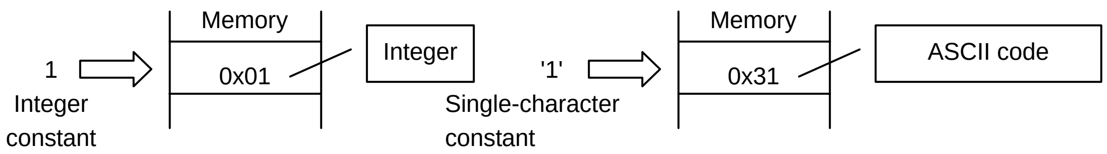
Numeration	Method of writing	Examples
Decimal	Normal mathematical notation (nothing added)	127, +127, -56
Hexadecimal	Numerals are preceded by 0x or 0X	0x3b, 0x3B
Octal	Numerals are preceded by 0 (zero)	07, 041

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

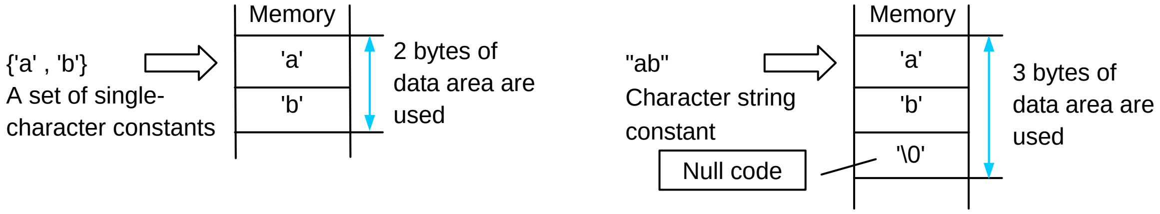
Single-character constants must be enclosed with single quotations ('). In addition to alphanumeric characters, control codes can be handled as single-character constants.
Inside the microcomputer, all of these constants are handled as ASCII code, as shown below.



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 and Definition of Variables

Variables are declared and defined 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 defined.

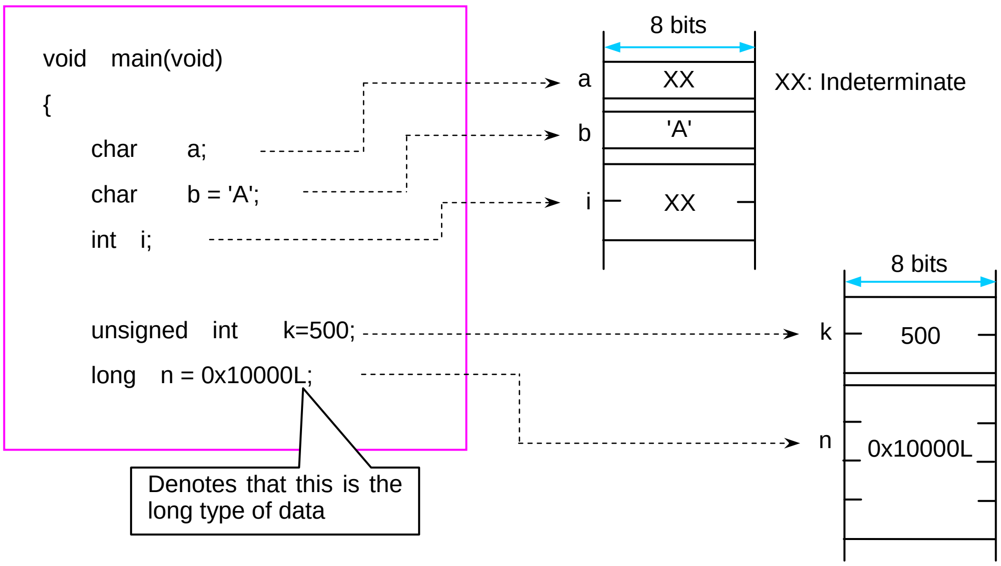
Example: To set 'A' to variable a of char type as its initial value

char a = 'A';

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

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 types 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)
{
       char   a;                      Synonymous with "unsigned char   a;"
       signed   chars_a;

       int   b;                       Synonymous with "signed int   b;"
       unsigned   int      u_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;
char volatile 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
Storage class specifier (described later)	Type qualifier	Type specifier	
static	const	char	
register	volatile	short	
auto		int	
extern		long	
typedef		float	
		struct	data name
		union	
		enum	
		void	
		signed	
		unsigned	

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.

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

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 expression) variable ++ (postfix expression)	Increments the value of an expression.
--	-- variable (prefix expression) variable -- (postfix expression)	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 expression, prefix or postfix, is used when writing the operator.
<Examples>
Prefix expression: The value is increment or decrement before assignment.
b = ++a; → a = a + 1; b = a;
Postfix expression: 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.

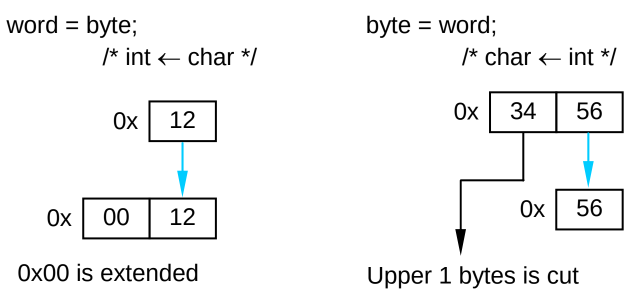
Column Implicit Type Conversion

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;
```



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 1	Returns the value of the expression 1 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	False if the expression is true, or true if the expression is false.

1.3.5 Other Operators

This section explains six 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
(type)	(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.

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

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, the following implicit conversions are performed.
(A) The data types are adjusted to that of data which is long in bit length.
(B) When all the data shorter than int type is used for the arithmetic operators.
(C) The constant is handled as int.
To ensure that the program will operate as expected, write explicit type conversion using the cast operator.
An example to operate as expected (if statement becomes true) is shown.

(1)

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

Since the cast operator for the entire expression (a-1) is used, on the left side,
unsigned 0x00 - unsigned 0x01 = unsigned 0xFF = 255.
Therefore, comparison is performed on 255 >= 5, so the result is found to be true.

Secondly, an example not to operate as expected (if statement becomes false) is shown.

(2)

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

The expression (a - 1) becomes an expression unsigned char - signed int. By implicit conversion, unsigned char has its type changed to signed int, on the left side, signed 0x0000 - signed 0x0001 = signed 0x0000 + signed 0xFFFF = signed 0xFFFF = -1.
Therefore, comparison is performed on -1 >= 5, so the result is found to be false.

(3)

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

Since the constant 1 has its type changed to unsigned char, on the left side, the expression becomes an expression unsigned char - unsigned char. By int extension of (B), unsigned char has its type changed to signed int and the same calculation as (2) is performed.
Therefore, comparison is performed on -1 >= 5, so the result is found to be false.

Check in the list file and assembly file eventually. The program size may be changed or runtime library may be called.

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 "()."

```
int a = 5;
if(a & 0x10 == 0){

}
else{

}
```

Because between bitwise operator "&" and relational operator "==", precedence is higher for the relational operator "==" and, hence, the comparison result of 0x10==0 (false: 0) and the variable a are AND's, so the operation of the if statement conditional expression always results in false.

Therefore

```
int a = 5;
if((a & 0x10) == 0){

}
else{

}
```

To ensure that the operation of a & 0x10 has precedence, add expressional "()."

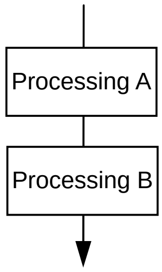
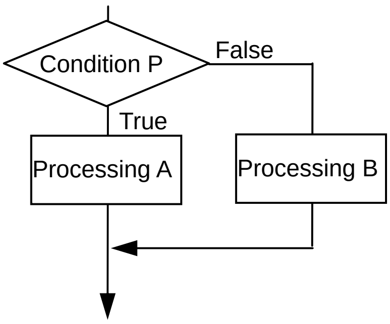
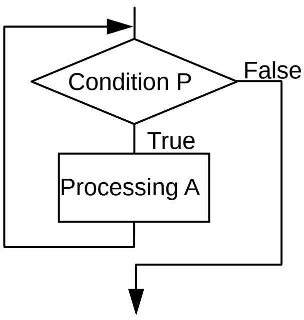
1.4 Control Statements

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.

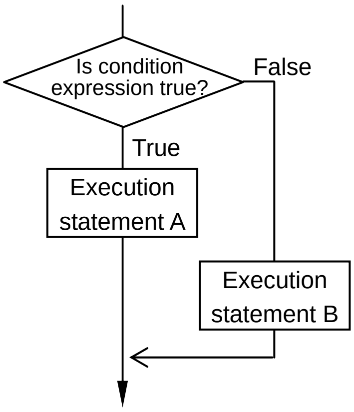
Sequential processing		Executed top down, from top to bottom.
Branch processing		Branched to processing A or processing B depending on whether condition P is true or false.
Repeat processing		Processing A is repeated as long as condition P is met.

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.

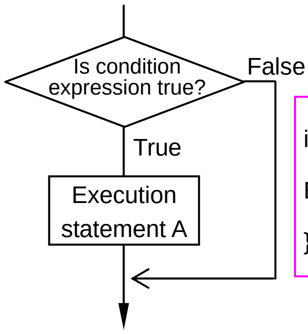
if-else Statement

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)

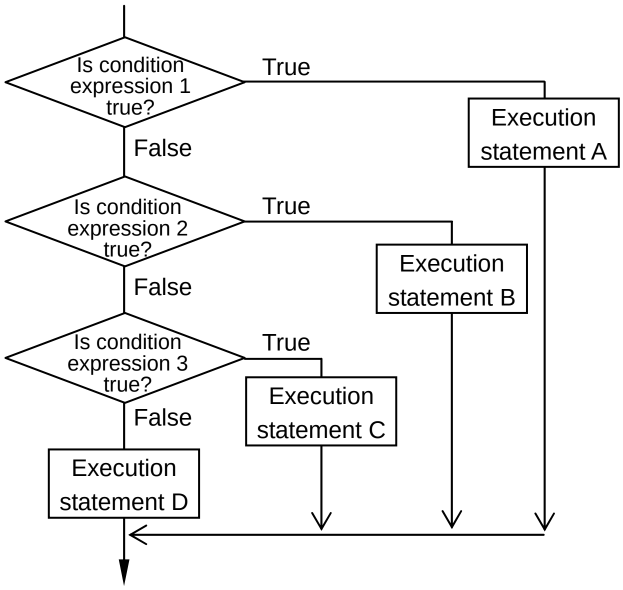
In this example, the program counts up a seconds counter "second" and a minutes counter "minute". When this program module is called up every 1 second, it functions as a clock.

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

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

- void count_up(void); — Declares "count_up" function. (Refer to Section 1.5, "Functions".)
- unsigned int second = 0; — Declares variables for "second" (seconds counter) and "minute" (minutes counter).
- void count_up(void) — Defines "count_up" function.
- if(second >= 59){ — If 59 seconds or more, the module resets "second" and counts up "minute".
- else{ — 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 B
}
else if(condition expression 3) {
        Execution statement C
}
else{
        Execution statement D
}
```

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.

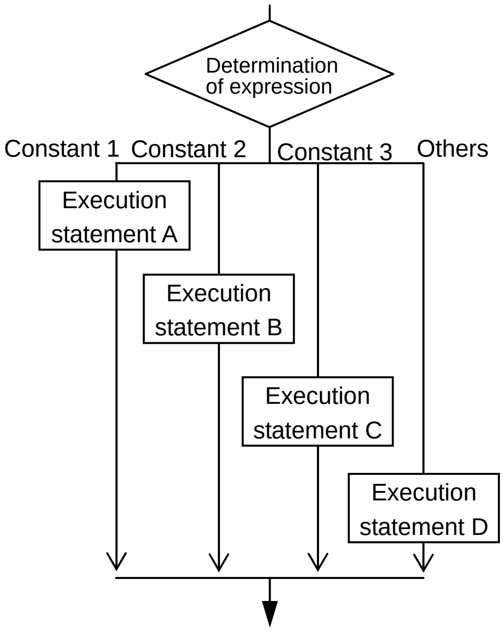
If the content of "sw" is 2,
the program multiplies data.

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);

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

void select(void)
{
    switch(sw){

        case 0 : ans = a + b;
                 break;
        case 1 : ans = a - b;
                 break;
        case 2 : ans = a * b;
                 break;
        case 3 : ans = a / b;
                 break;
        default: error();
                 break;
    }
}
```

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

Declares the variables used.

Defines "select" function.

Deterrmines the content of "sw".

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.

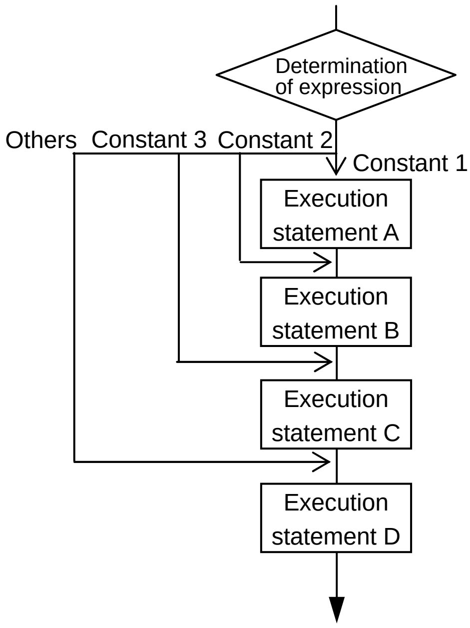
If the content of "sw" is 3, the program divides data.

If the content of "sw" is 4 or greater,
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.



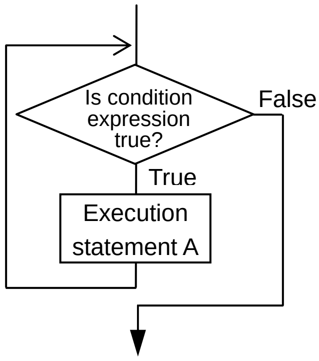

```
switch(expression) {

    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)

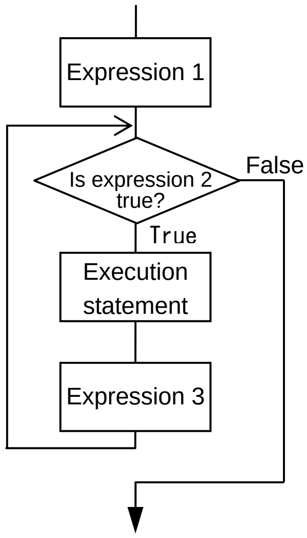
In this example, the program finds the sum of integers from 1 to 100.

```
void sum(void);

unsigned int total = 0;

void sum(void)
{
    unsigned int i = 1;

    while(i <= 100){
        total += i;
        i++;
    }
}
```

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 reaches 100.
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)

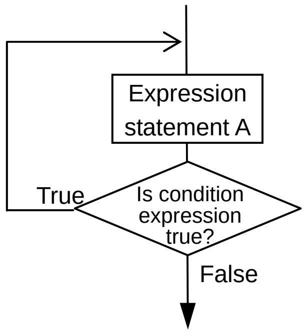
In this example, the program finds the sum of integers from 1 to 100.

```
void sum(void);

unsigned int total = 0;

void sum(void)
{
    unsigned int i = 1;

    for(i = 1; i <= 100; i++){
        total += i;
    }
}
```

- `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 counter variables.
- `for(i = 1; i <= 100; i++){ ... }` — 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);
}
```

- 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 increments from 1 to 100.

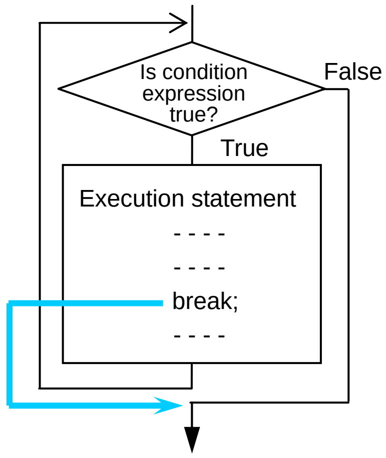
1.4.4 Suspending Processing

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

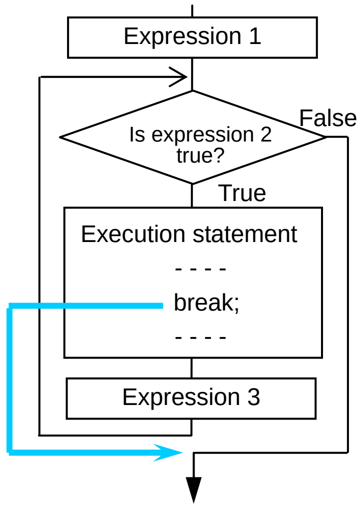
Use this statement in repeat processing or in a switch-case statement. When "break;" is executed, the program suspends processing and exits only one block.

• When used in a while statement




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

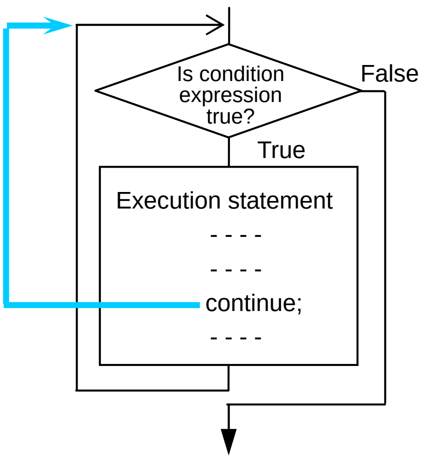
• When used in a for statement



continue Statement

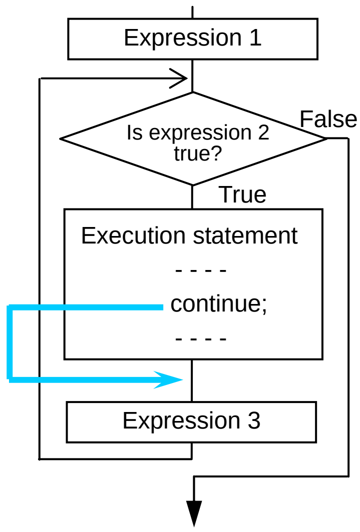
Use this statement in repeat processing. When "continue;" is executed, the program suspends processing. After being suspended, the program returns to condition determination when continue is used in a while statement or executes expression 3 before returning to condition determination when used in a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

```
void    main(void)
{
        while(1){
          - - - -
                   while (…){
                           if(…){
                                   goto    err;
                           }
                   }
        }
        err:           ;
}
```

Entering a label
label: execution statement;

If no operation need to be performed,
label: ; (dummy statement)

1.5 Functions

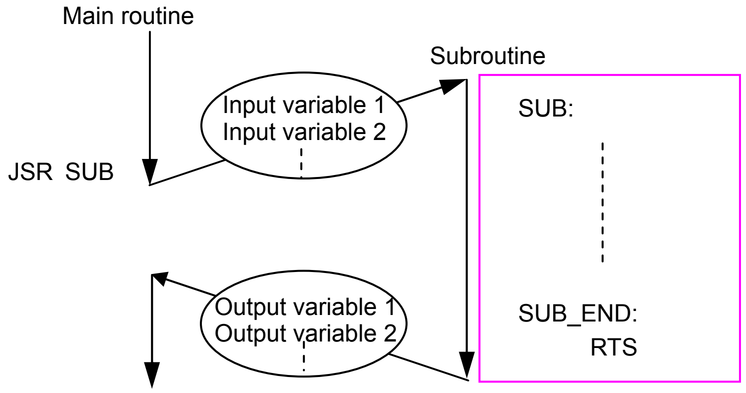
1.5.1 Functions and Subroutines

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.

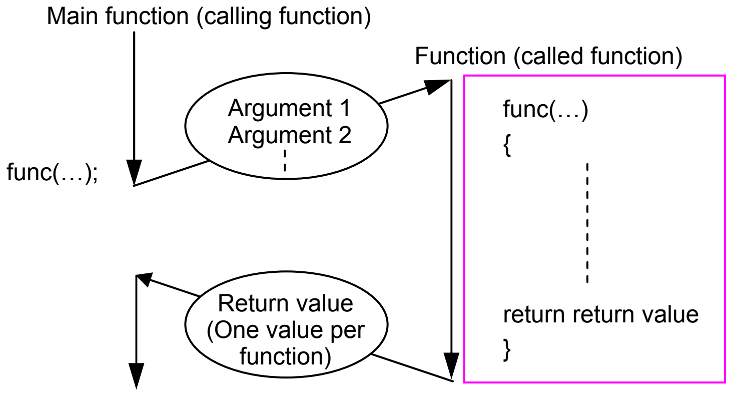
Arguments and Return Values

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 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 an assignment operator to receive a return value from the called function.

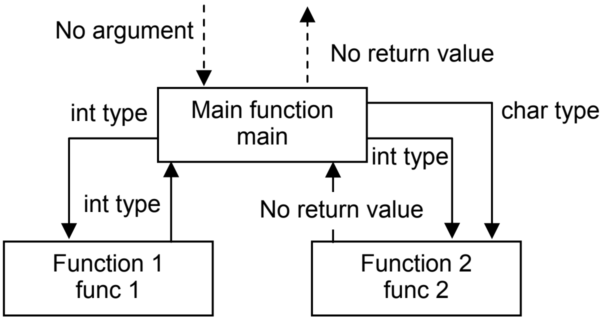
```
function name (argument 1, ...);
```

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 value (dummy argument), there is no possibility of damaging the variable 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.

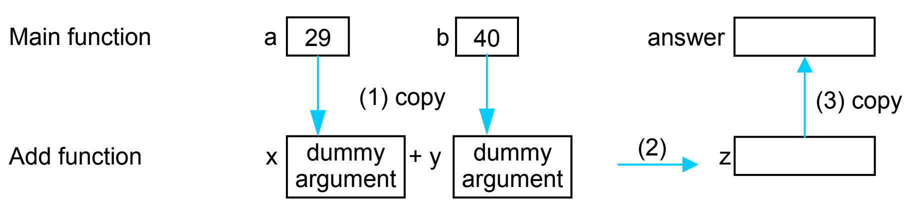
```
/*   Prototype declaration   */
void   main(void);
long   add(int, int);

/*   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 can change effective ranges depending on their nature, e.g., whether they are used in the entire program or in only one function. Specifying these effective ranges of variables and functions is called "storage classes (or scope)".
This section explains the types of storage classes of variables and functions and how to specify them.

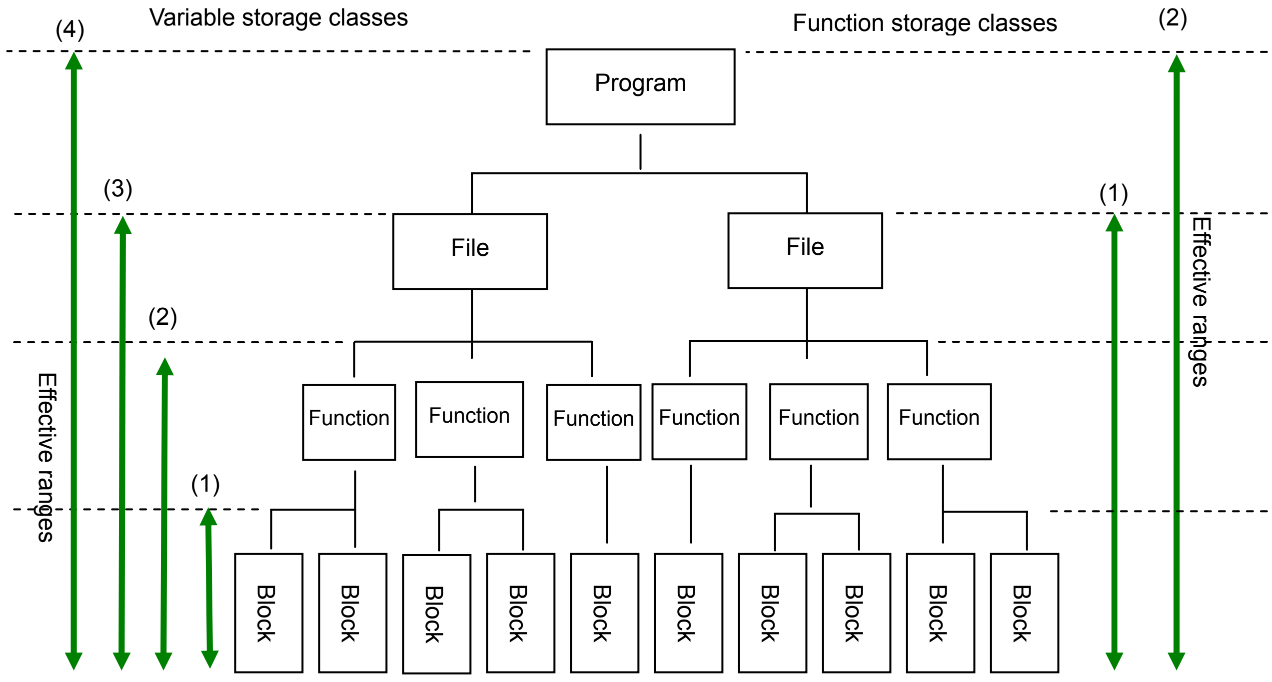
Effective Range of Variables and Functions

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 four storage classes for a variable:
(1) Effective in the block
(2) Effective in only a function
(3) Effective in only a file
(4) 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 close the variables or functions that have been created or conversely open them among the members of a team.

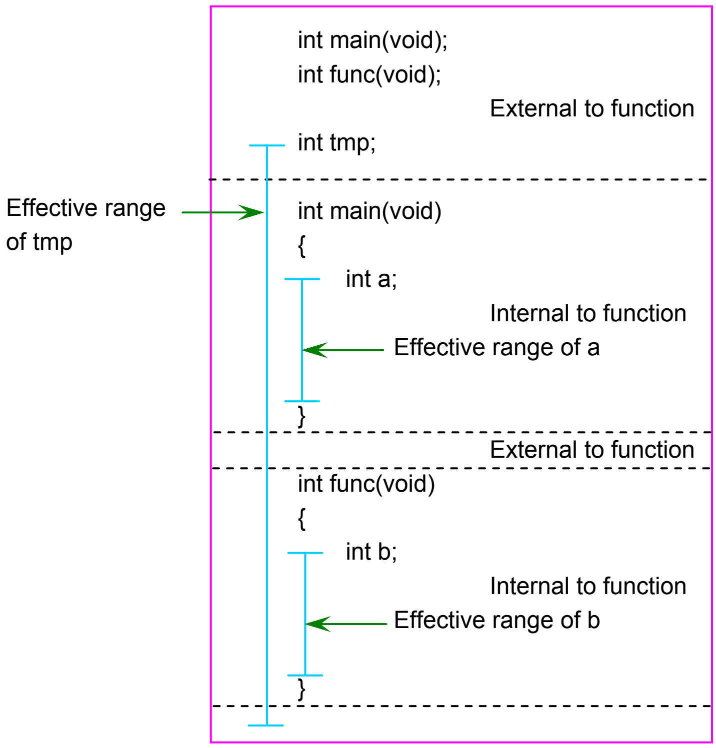


1.6.2 Storage Classes of Variables

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.



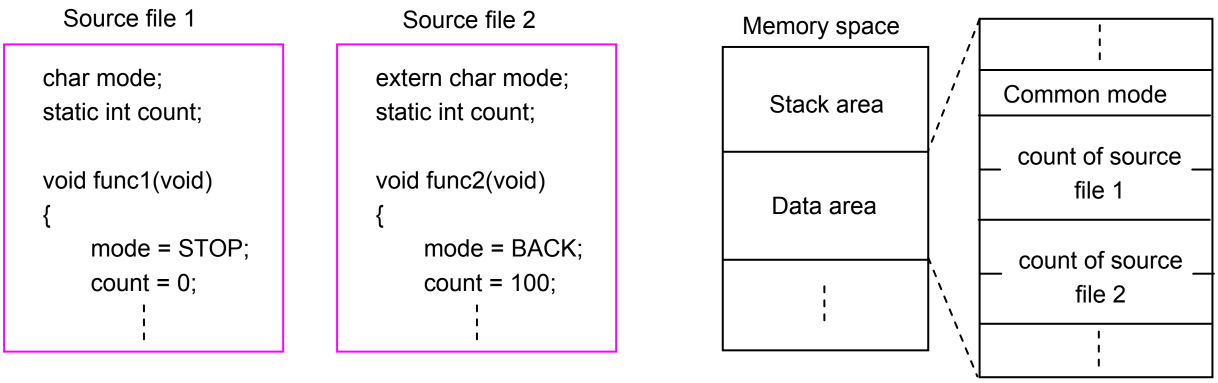
Storage Class Specifiers

The storage class specifiers that can be used for variables are auto, static, register, and extern. The following shows the format of a storage class specifier.

```
storage class specifier data type variable name;
```

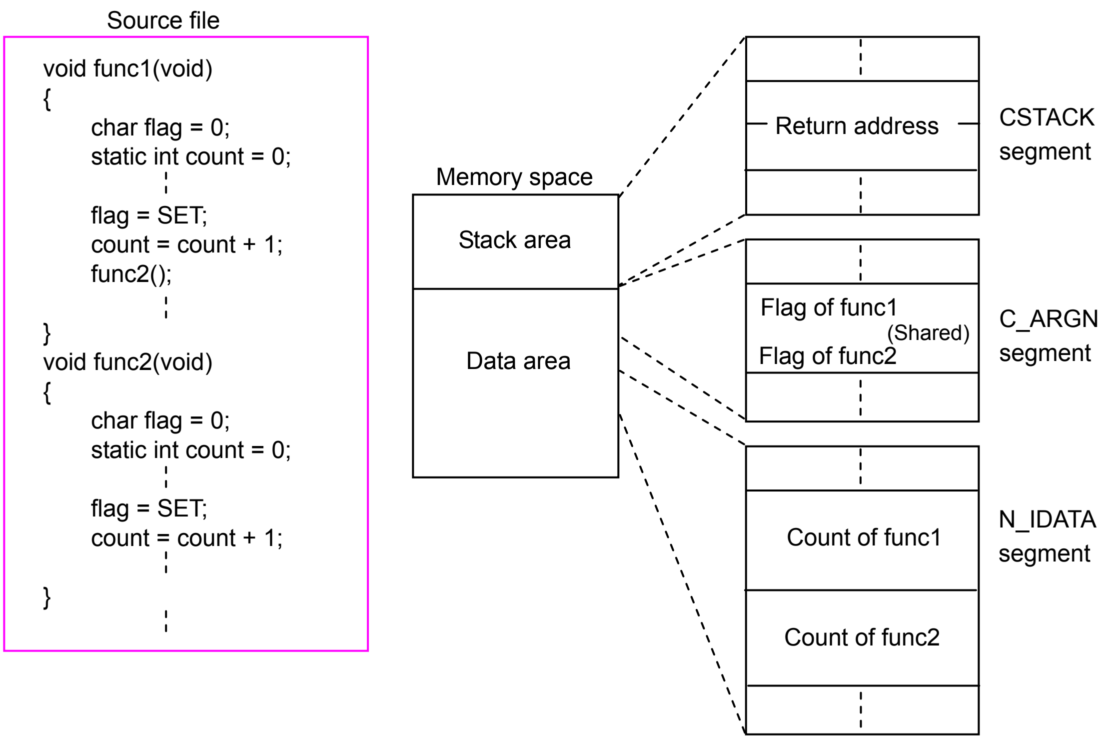
Storage Classes of External Variable

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.



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 as 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 initialized to 0 and assigned to the N_UDATA segment on data area, and which set the initial value are initialized to the set value and assigned to the N_IDATA segment on data area. The variable is initialized only once when starting up the program.



1.6.3 Storage Classes of Functions

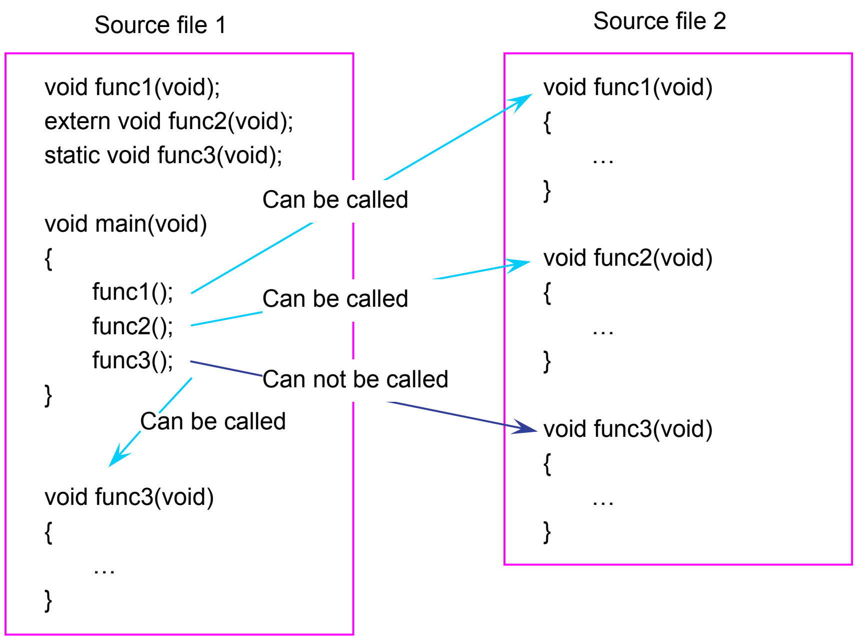
The storage class of a function is specified on both function defining and function declaring sides.
The storage class specifiers, static and extern can be used here.
This section explains how to specify the storage class of a function.

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 "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".

(3) 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.



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]	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 declaring side]

1.7 Arrays and Pointers

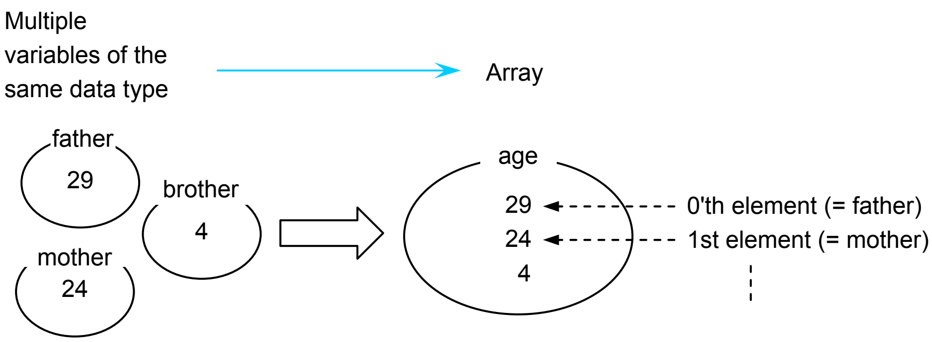
1.7.1 Arrays

This section describes how to think arrays.

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).

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

        total = father + mother + 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 sister 1 = 1;
        int sister 2 = 1;
                :
        int total;

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

1.7.2 Creating an Array

Arrays handled in the C language have one-dimensional array and two-dimensional array.
This section describes how to create and reference each type of array.

One-dimensional Array

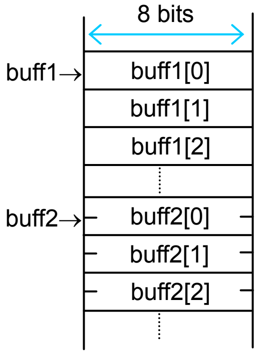
A one-dimensional array has a one-dimensional (linear) expanse. The following shows the declaration format of a one-dimensional array.

```
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.

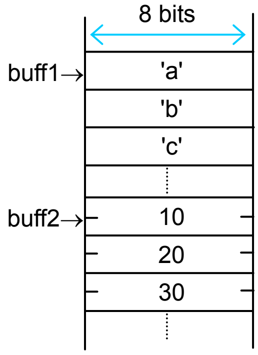
• Declaration of one-dimensional array

```
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 "rows" and " columns". 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

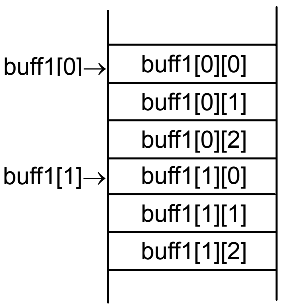
Columns→

Rows ↓

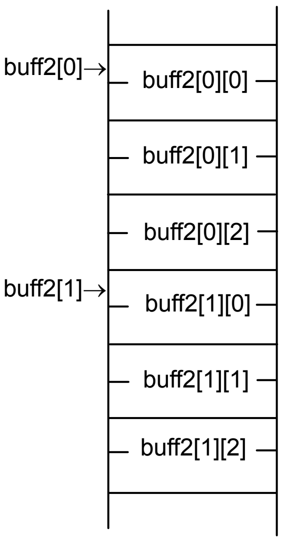
Row 0 Column 0	Row 0 Column 1	Row 0 Column 2
Row 1 Column 0	Row 1 Column 1	Row 1 Column 2

• Declaration of two-dimensional array

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

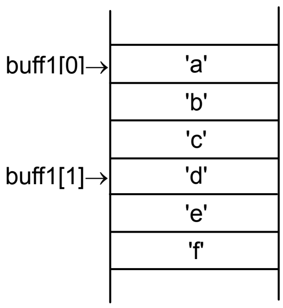



```
int buff2[2][3];
```



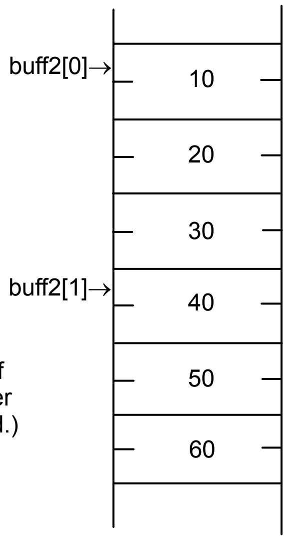
• Declaration and initialization of two-dimensional array

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




```
int buff2[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. The pointer should be useful to handle the array.
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.

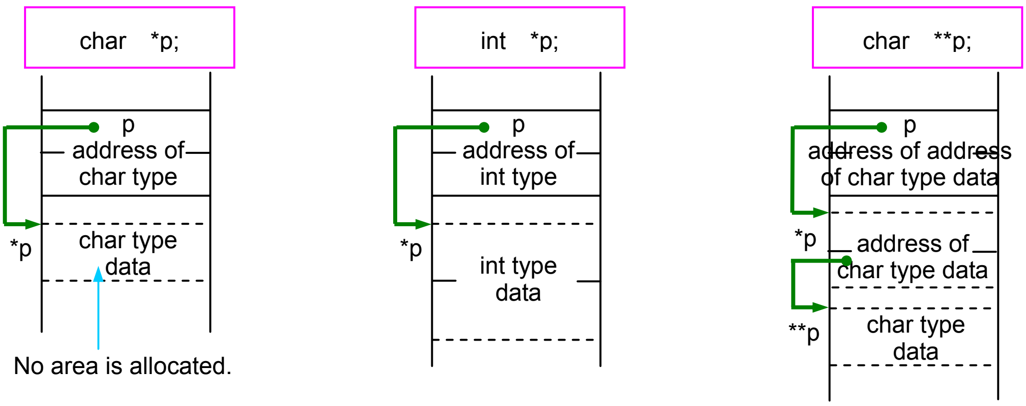
Declaring a Pointer Variable

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 to int type for variable of int type a as an example.

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

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

Address modifier
↓
This "&a" indicates the address of variable a.
This "*p" indicates the content of variable a.

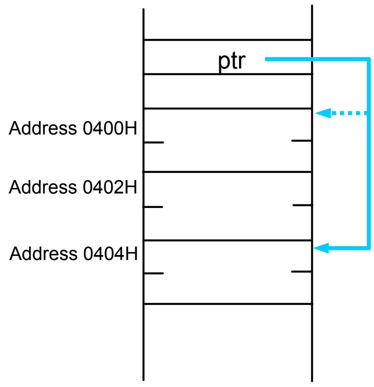
a is "5"

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 calculations 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, ptr + 2 points x sizeof(int) 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 generally accessed by the microprocessor used.

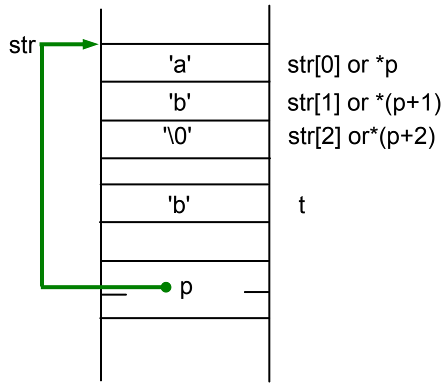
1.7.4 Using Pointers

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.

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

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

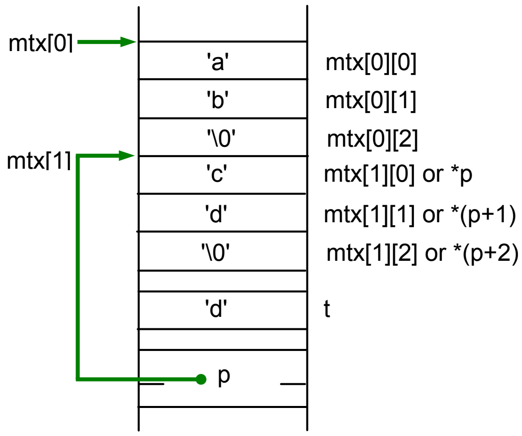


The start address of a one-dimensional array can be obtained by str.
(Address modifier '&' is unnecessary.)

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.

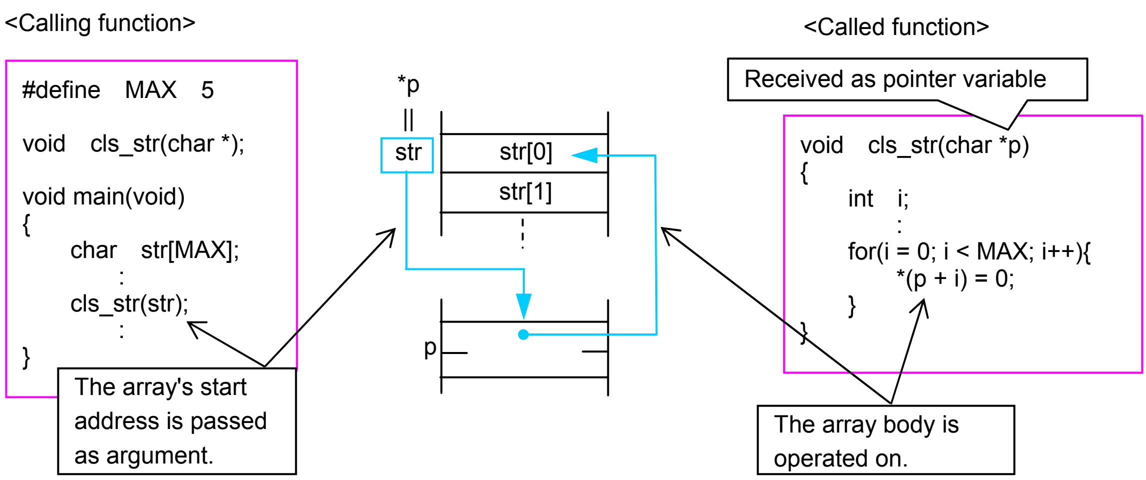
```
void main(void)
{
      char mtx[2][3] = {
            "ab", "cd"
      };
      char *p;
      char t;

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



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 area in the calling function is rewritten directly by rewriting the pointer variable in the called function.
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 Speed

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.

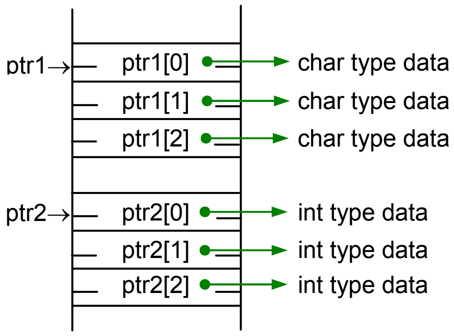
Pointer Array Declaration

The following shows how to declare a pointer array.

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

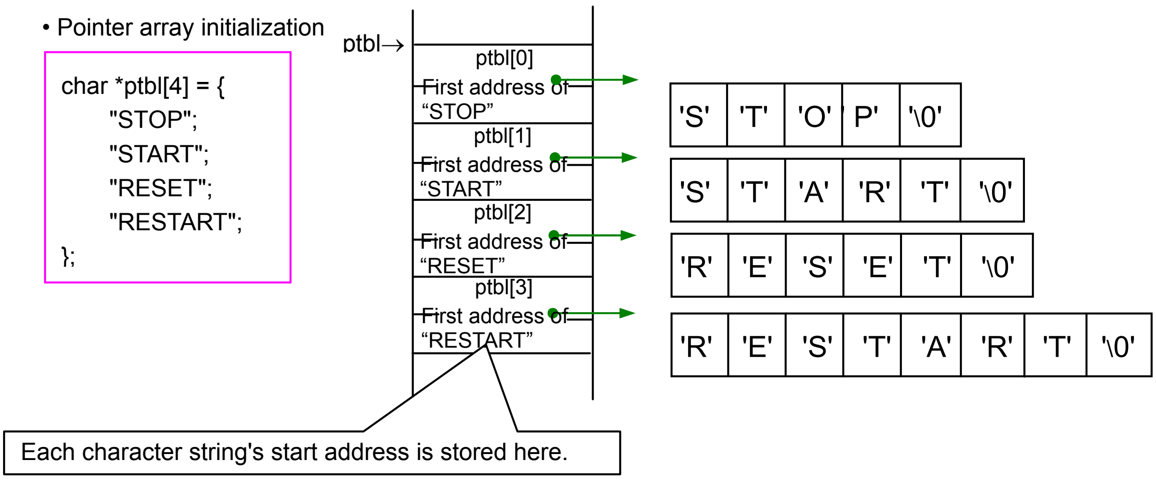
• Pointer array declaration

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



• Pointer array initialization

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

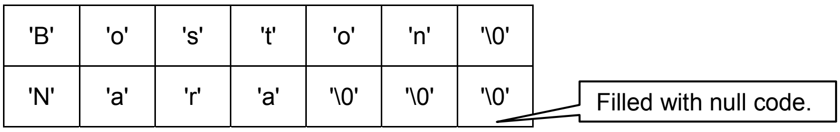


Pointer Array and Two-dimensional Array

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.

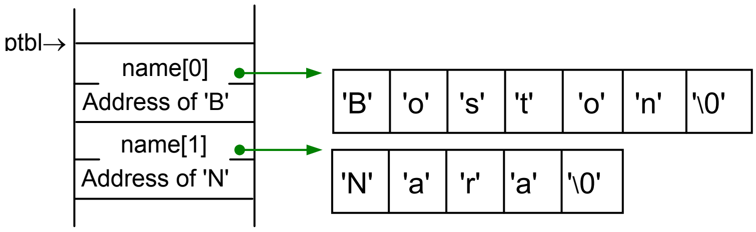
- Two-dimensional array

```
char name[2][7] = {
        "Boston",
        "Nara",
};
```



- Pointer array

```
char   *name[2] = {
        "Boston",
        "Nara",
};
```



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.

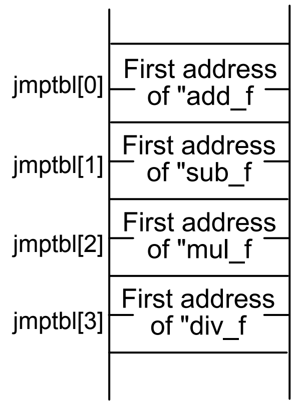
<Declaration format> Type of return value (*function pointer name) (data type of argument);
<Reference format> Variable in which to store return value = (*function pointer name) (argument);

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



Setting of jump address

Function call using a function pointer

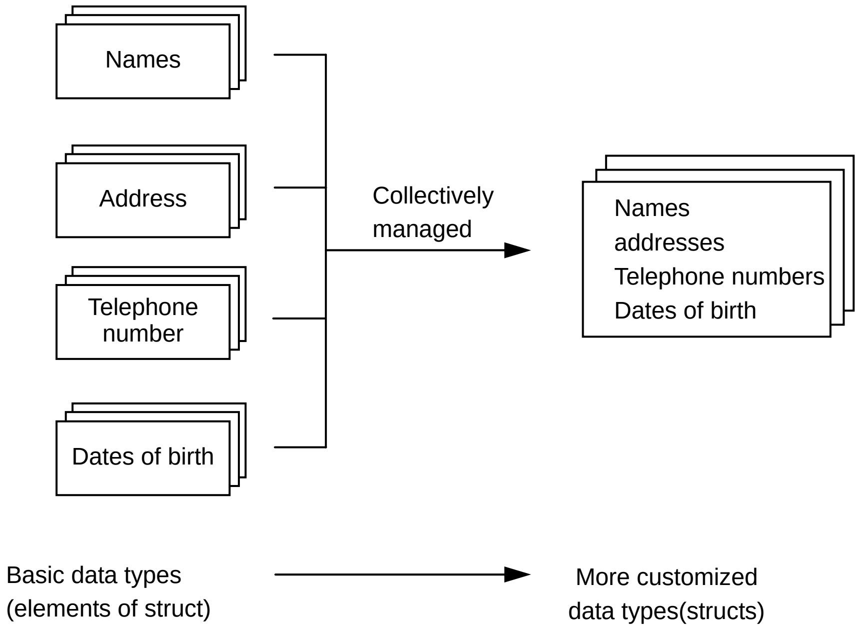
1.8 Structures and Unions

1.8.1 Structures and Unions

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 which combines 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 customized 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 of Struct

To define a struct type, write "struct".

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

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

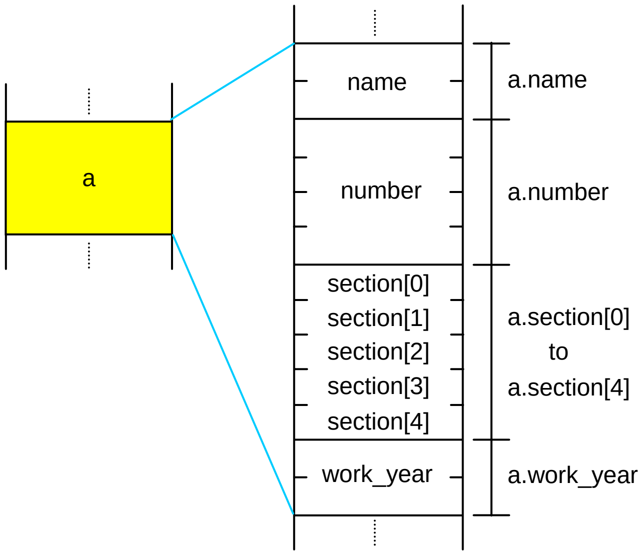
```
struct   struct tag   struct variable name;
```

Referencing Struct

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

```
struct variable name.member name
```

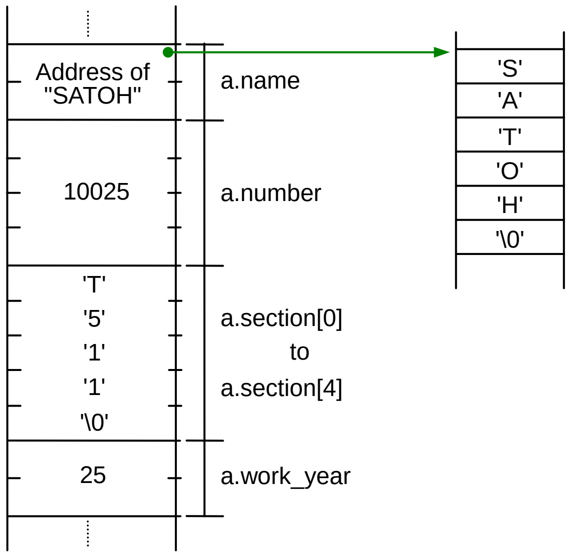
```
struct   person{
      char   *name;
      long   number;
      char   section[5];
      int   work_year;
};

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
};
```

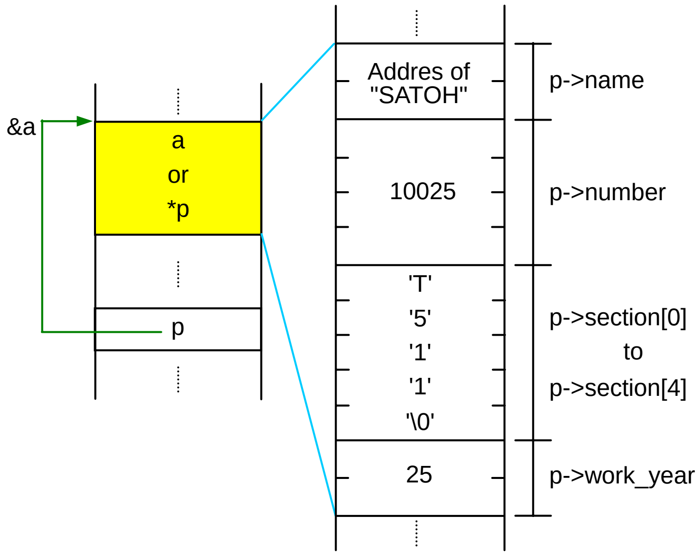


Example for Referencing Members Using a Pointer

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

```
Pointer->member name
```

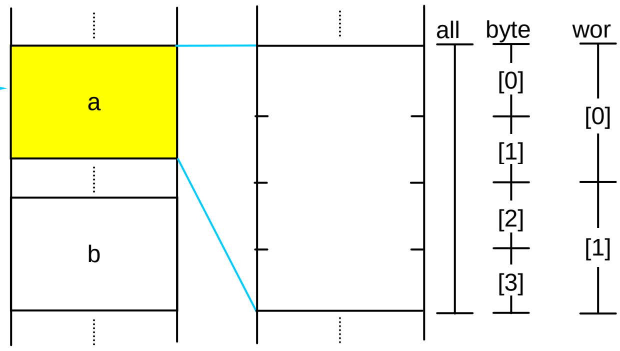
```
#define    LYEAR    20
struct    person{
        char    *name;
        long    number;
        char    section[5];
        int    work_year;
};

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

void    main(void)
{
        struct person *p;
        p = &a;
        if( p->work_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 bits 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 "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.

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

struct data   sdata, *sptr;
              ⋮
```

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

```
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 definition, 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. Methods of description vary 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.

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

#include     <stdio.h>

#include     "usr_global.h"

/*main function**********/
void main ( void )
{
        ⋮
}
```

The standard include file is read from the standard directory.

The header of a global variable is read from the current directory.

1.9.3 Macro Definition

Use the "#define identifier" for character string replacement and macro definition. Uppercase letters are generally used 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

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

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

A string can be assigned a name.

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 eight 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 Constant expression A #else B #endif	If the constant expression is true (not 0), ICC740 compiles block A; if false, it compiles block B.
#ifdef Macro name A #else B #endif	If a macro name is defined, ICC740 compiles block A; if not defined, it compiles block B.
#ifndef Macro name A #else B #endif	If a macro name 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
Explains about project settings

This section describes memory models, segment configuration, stack area, object format and C startup module, and explains how to set values in a special area.

2.1 Set Content

Program development with ICC740 starts by setting up a processor group, memory model and stack area first. The following lists the set content of each item.

Item	Choices	Default
Processor group	With MUL/DIV instruction	○
	Without MUL/DIV instruction	
	With MUL/DIV instruction and extended data access	
Memory model	Large model	○
	Tiny model	
	Zero-page model	
Stack area	1 page (100H–1FFH)	○
	0 page (00H-FFH)	
Object format	UBROF(IAR format)	
	IEEE695(for HEW)	○
	intel-standard(for ROM)	
	motorola(for ROM)	

2.2 Description of Memory Models

ICC740 uses the following memory models to create a project.

1) Large model
   The Large model is located in areas whose default variable placement position is in other than zero-page (addresses beginning with 0x100).

2) Tiny model
   The Tiny model is located in areas whose default variable placement position is in zero-page (addresses 0x0–0xFF).

3) Zero-page model
   The Zero-page model can only use zero-page.

2.2.1 Details of Memory Models

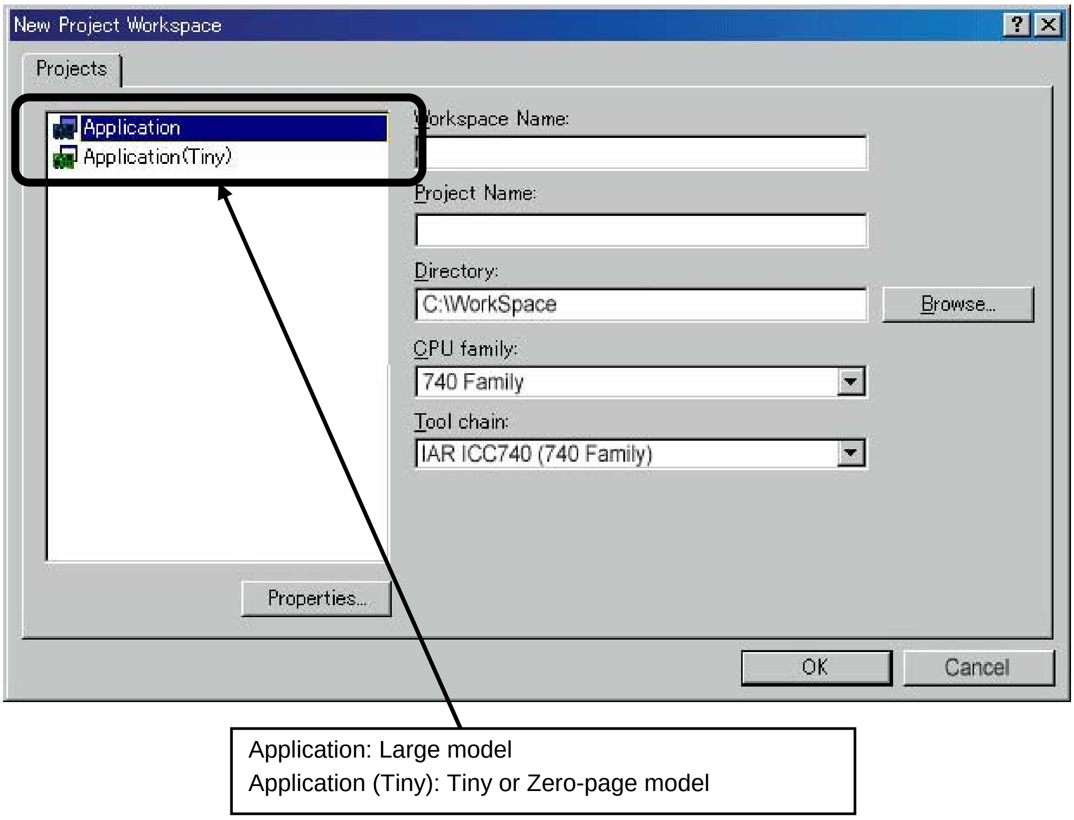
The differences between each memory model are summarized in the table below.

Item	Large model	Tiny model	Zero-page model
Places where variables located	Addresses beginning with 0x100	Addresses up to and including 0xFF	Addresses up to and including 0xFF
Placement at addresses beginning with 0x100 in C language	–	Definition using npage npage int v1; extern npag int v2;	Not locatable
Placement at addresses up to and including 0xFF in C language	Definition using zpage zpage int v3; extern zpag int v4;	–	–
Method for accessing addresses beginning with 0x100 in assembly source program	–	Operand np: used Ida np:v1	Inaccessible
Method for accessing addresses up to and including 0xFF in assembly source program	Code size reducing by using operand zp: Ida zp:v3	–	–

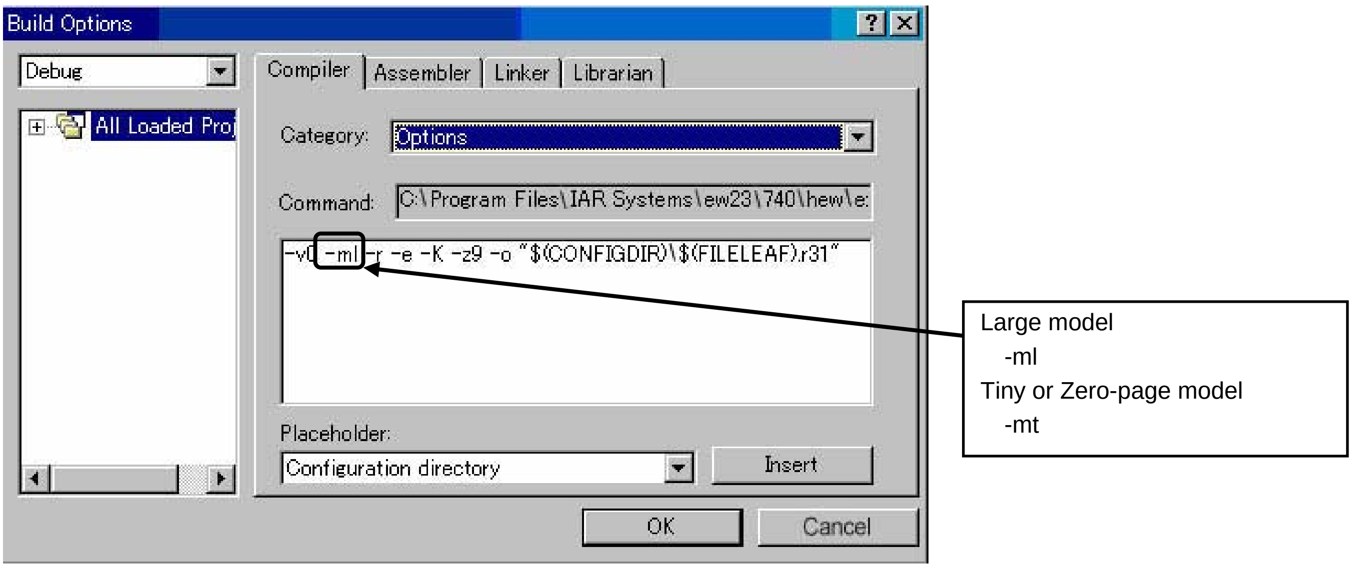
The extended keywords zpage and npage are specifiable in external variables, auto-variables and arguments to functions.

2.2.2 Changing Memory Models

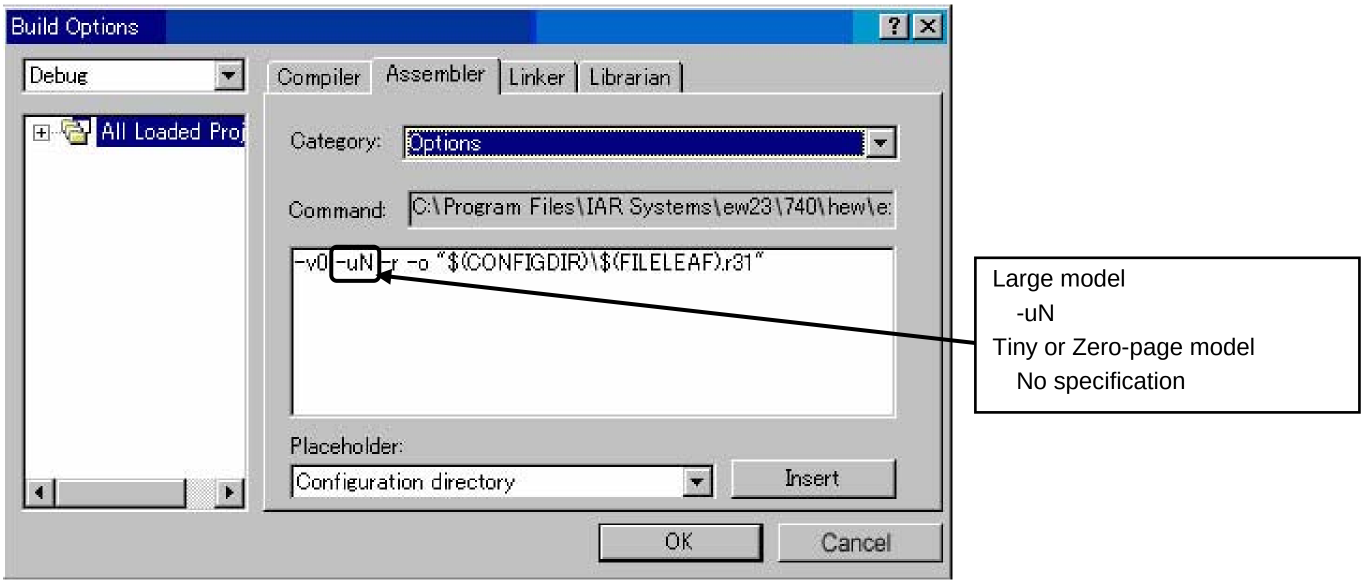
To change memory models, specify a project type when creating a new project.
The memory model of default <Application> is set to “Large model.”
To change it to Tiny model or Zero-page model, specify <Application (Tiny)>.



To check the set content, choose IAR ICC740 Toolchain from the Build menu to open Build Options.



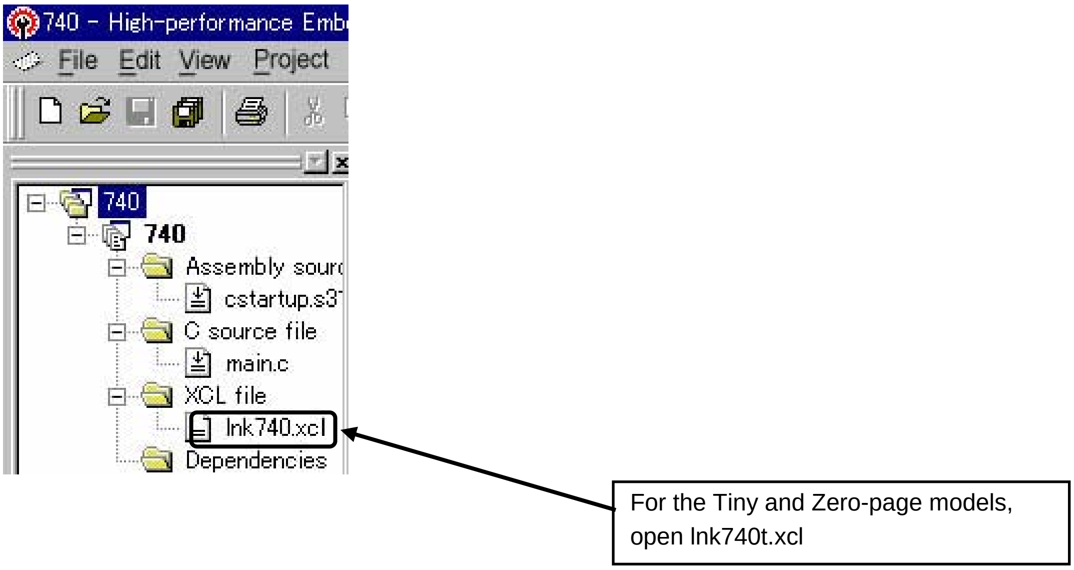
The “-m” option on the Compiler tab is set to “-ml” for the Large model. This option is set to “-mt” for the Tiny or Zero-page model.



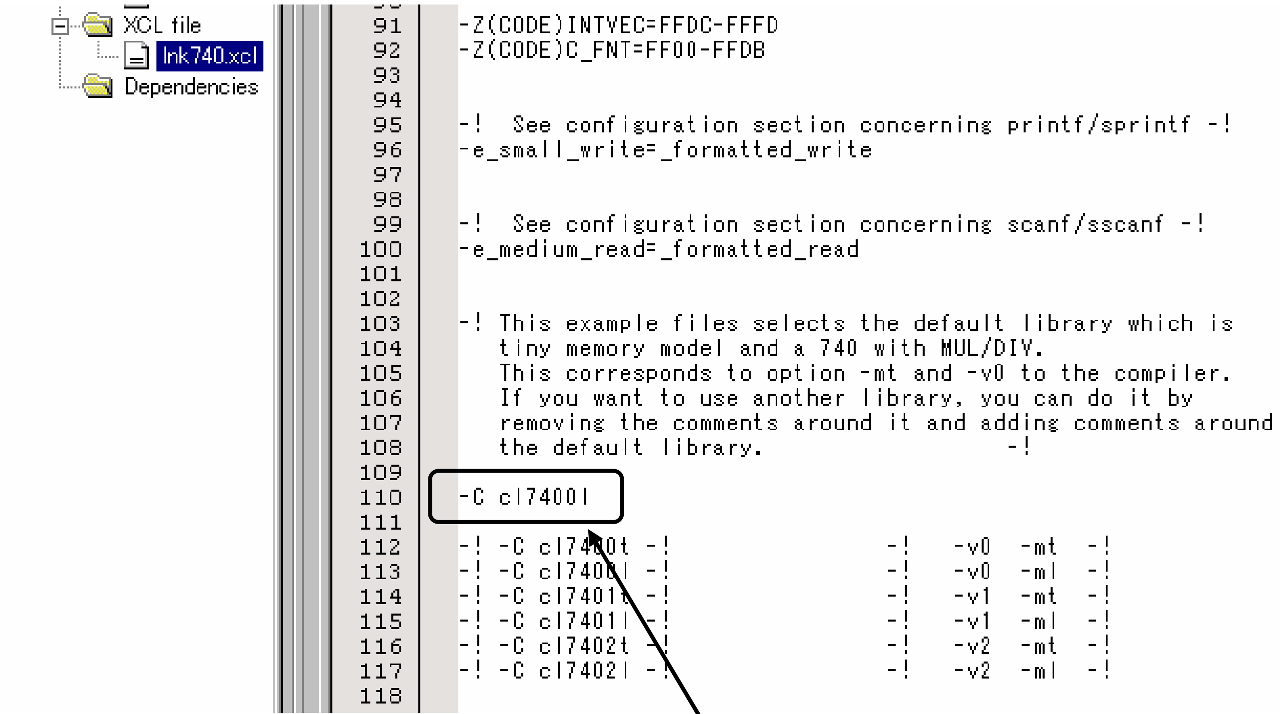
The -uN option specified on the Assembler tab is for the Large model. The -uN option is not specifiable for the Tiny and Zero-page models.
The -uN option uses 16-bit addressing. The following are not affected.

8-bit addressing specification:
LDA ZP:label
16-bit addressing specification:
LDA NP:label

Next, check how the linker is set.
Check the link command file for linker setup, and not the Linker tab. To inspect the link command file, open the lnk740.xcl file in the workspace. For the Tiny and Zero-page models, open the lnk740t.xcl file.



The lnk740.xcl file will have a library specified with the -C option in the last part of it.
For the Large model, this library is set c174001.r31, whereas for the Tiny and Zero-page models, it is set to c17400t.r31. For the Zero-page model, the location of the CSTACK segment must be changed from that of the Tiny model. Refer to page 85 for details.




```
-Z(CODE)INTVEC=FFDC-FFFD
-Z(CODE)C_FNT=FF00-FFDB


-!  See configuration section concerning printf/sprintf -!
-e_small_write=_formatted_write


-!  See configuration section concerning scanf/sscanf -!
-e_medium_read=_formatted_read


-! 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  -!
```

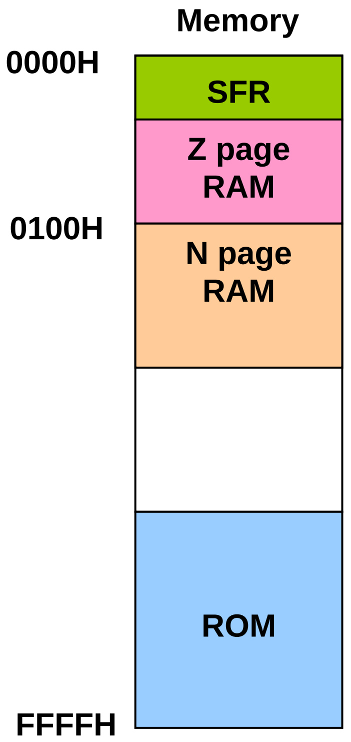
Large model
-C c174001
Tiny and Zero-page models
-C c17400t

2.3 Segment Configuration

2.3.1 Segment Configuration of ICC740

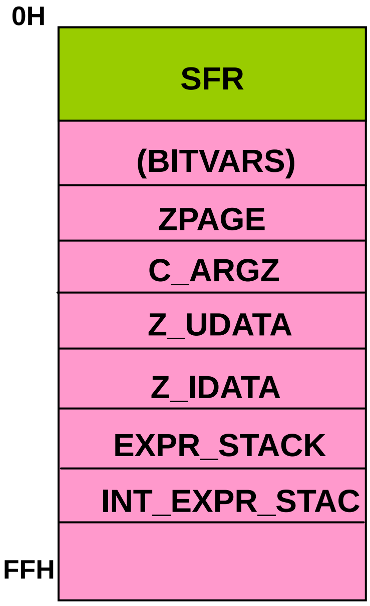
The segment configuration of ICC740 is classified below.

- To ensure that the resources of the 740 family are effectively used, ICC740 classifies memory into the following.
  - Z page RAM (0H–FFH): Zero page
  - N page RAM (100H and on): Normal page or non-zero page
  - ROM (up to and including FFFFH)
- Segments are set in each for data management.



2.3.2 Segment Map: Z Page RAM (0H–FFH)

This section shows the segment map: Z page RAM (0H–FFH).



Bit variables whose addresses are undetermined yet

Library data (located in zero page)

Auto variables and arguments using zpage: func (zpage int a)

External variables using zpage, not subject to initialization: zpage short c;

External variables using zpage, subject to initialization: zpage char b=1;

Expression stack: Manipulated with register X. Specify its size in lnk740.xcl.

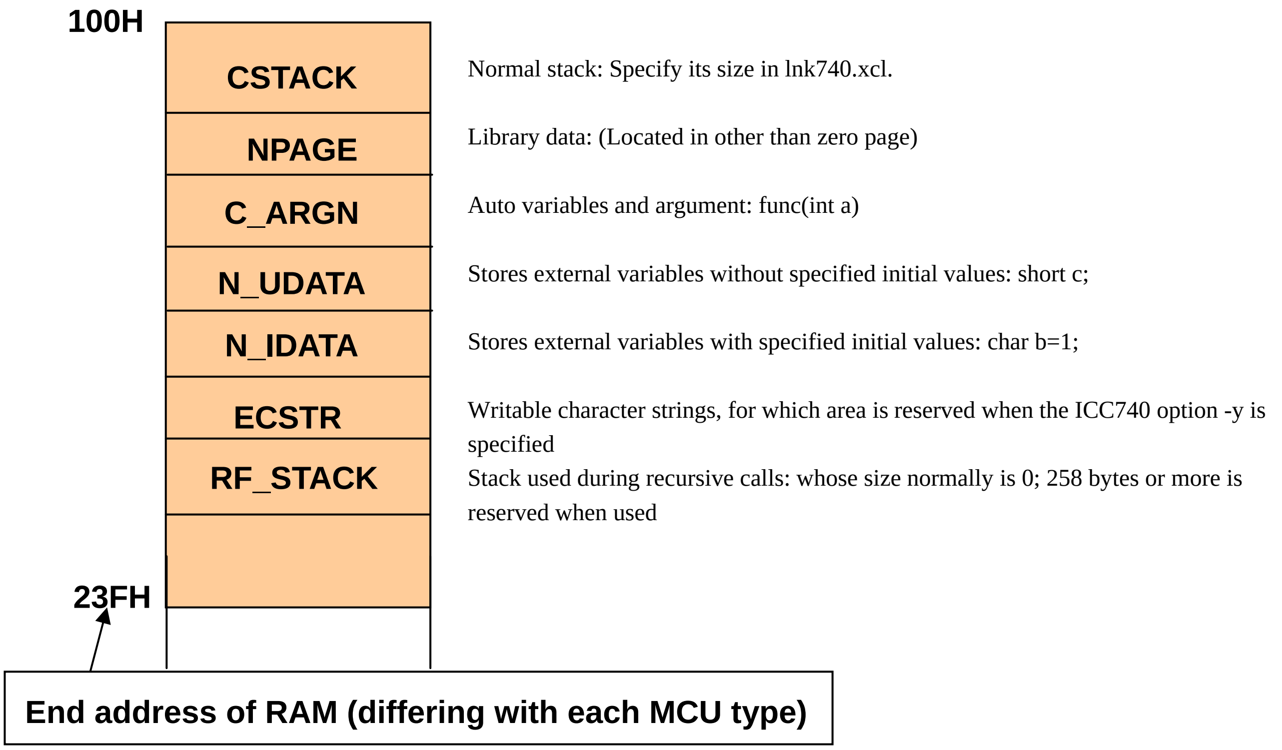
Interrupt-time expression stack: Manipulated with register X. Specify its size in lnk740.xcl.

Name (outline)	Type	Description
BITVAR (Static bit variable storage)	Read/write	Assembly-accessible. Holds a static bit variable. This segment must be located in Zero Page (0–0xFF).
ZPAGE (Zero Page assembler library data)	Read/write	Compiler-only. Holds the Zero Page internal library variables. This segment must be located in Zero Page (0–0xFF).
C_ARGZ (Local variables)	Read/write	Assembly-accessible. Holds static auto-variables. This segment must be located in Zero Page (0–0xFF).
Z_UDATA (Non-initialized static variables)	Read/write	Assembly-accessible. Holds variables in memory that are not explicitly initialized; these are implicitly initialized to all zero, which is performed by **CSTARTUP**. This segment must be located in Zero Page (0–0xFF).
Z_IDATA (Initialized static variables)	Read/write	Assembly-accessible. Holds static variables in internal data memory that are automatically initialized from **Z_CDATA** in **cstartup.s31**. See also **Z_CDATA** [Note 1]. This segment must be located in Zero Page (0–0xFF).
EXPR_STACK (Expression stack)	Read/write	Assembly-accessible. Holds temporary results while evaluating expressions during normal processing. This segment must be located in Zero Page (0–0xFF).
INT_EXPR_STACK (Interrupt expression stack)	Read/write	Assembly-accessible. Holds temporary results while evaluating expressions during interrupt processing. This segment must be located in Zero Page (0–0xFF).

Note 1: **Z_CDATA** (initialization constant)
Type: Read-only
Description: Accessible in assembly
CSTARTUP copies initial values from this segment into the Z_IDATA segment.

2.3.3 Segment Map: N Page RAM (beginning with 100H)

The segment map of the N page RAM (beginning with 100H) is shown below.



Name (outline)	Type	Description
CSTACK (Data stack)	Read/write	Assembly-accessible. Holds the hardware stack. This segment must be located in Zero Page (**0–0xFF**), or page 1 (**0x100–0x1FF**), depending on the position set by **CSTARTUP**. This segment and length is normally defined in the XLINK file by the command: **-Z(DATA)CSTACK + *nn* = *start*** where ***nn*** is the length and ***start*** is the location.
NPAGE (Normal page assembler library data)	Read/write	Compiler-only. Holds the non-Zero Page internal library variables.
C_ARGN (Local variables)	Read/write	Assembly-accessible. Holds auto variables. This segment must be located in N page (Addressed beginning with 0x100).
N_UDATA (Non-initialized static variables)	Read/write	Assembly-accessible. Holds variables in memory that are not explicitly initialized; these are implicitly initialized to all zero, which is performed by **CSTARTUP**.
N_IDATA (Initialized static data)	Read/write	Assembly-accessible. Holds static variables in internal data memory that are automatically initialized from **N_CDATA** in **cstartup.s31**. See also **N_CDATA** of “2.3.4. Segment Map ROM (up to FFFFH)” on page 81.

ECSTR (Writable copies of string literals)	Read/write	Assembly-accessible. Holds writable copies of C string literals. For more information refer to the C compiler **Writable strings** (**-y**) option (Note 1) below, and “-y” on page 103. See also **WCSTR** (Note 2), and **CSTR** *of* “2.3.4. Segment Map ROM (up to FFFFH)” on page 81 and **CCSTR** on page 82.
RF_STACK (Recursive stack)	Read/write	Assembly-accessible. Holds the local variables and parameters for enclosed calls of recursive functions. Because **SF_STACK** is located in page 1 and subsequent pages and because the compiler reserves 256 bytes of storage for it, recursive calls cannot be used in the Zero-page model.

(Note 1) Writable strings (-y)

Syntax: -y

Causes the compiler to compile string literals and other constants as initialized variables.
Normally, string literals and constants are compiled as read-only. If you want to be able to write to them, use the Writable strings (-y) option, causing them to be compiled as writable variables.
Note that arrays initialized with strings (ie char c[] = "string") are always compiled as initialized variables, and are not affected by the Writable strings (-y) option.

(Note 2) WCSTR (Writable string literals)

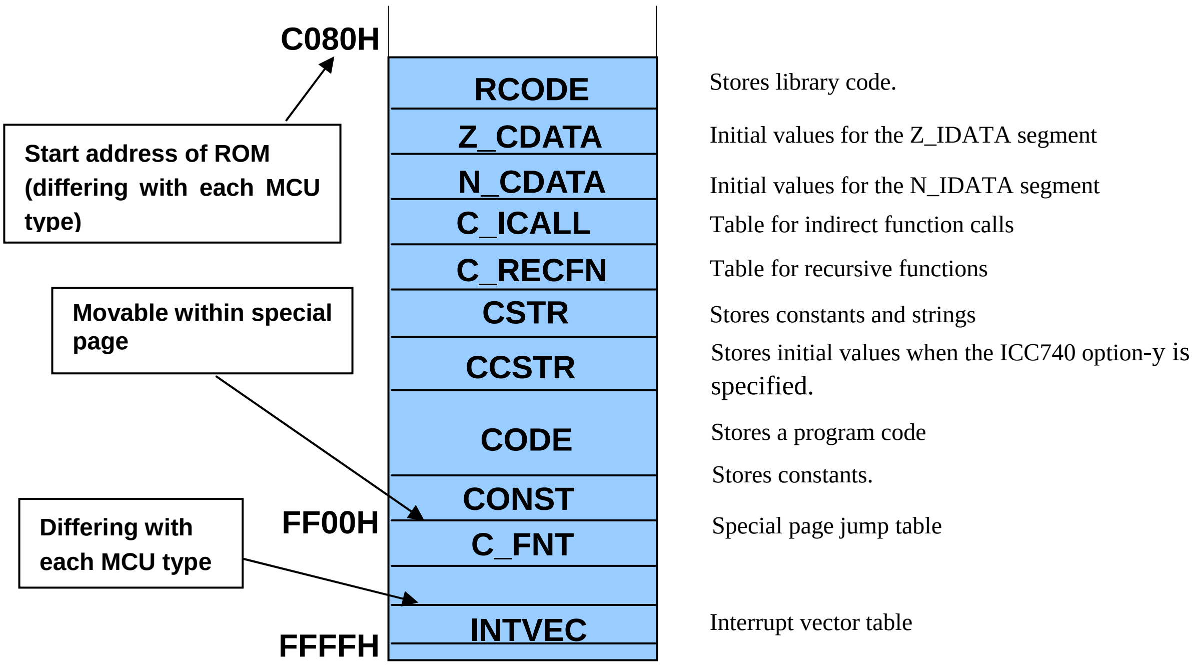
TYPE: Read-write

DESCRIPTION: Assembly-accessible.

Normally strings are placed in the **CSTR** (ROM) or **WCSTR** (RAM) area. If you have specified writable and PROMable strings, a special segment **CCSTR** (ROM) holds the string while the **ECSTR** (RAM) has the same amount of space. At run time, **CCSTR** is assumed to be copied to **ECSTR**.

2.3.4 Segment Map: ROM (up to FFFFH)

The segment map of the ROM (up to FFFFH) is shown below.



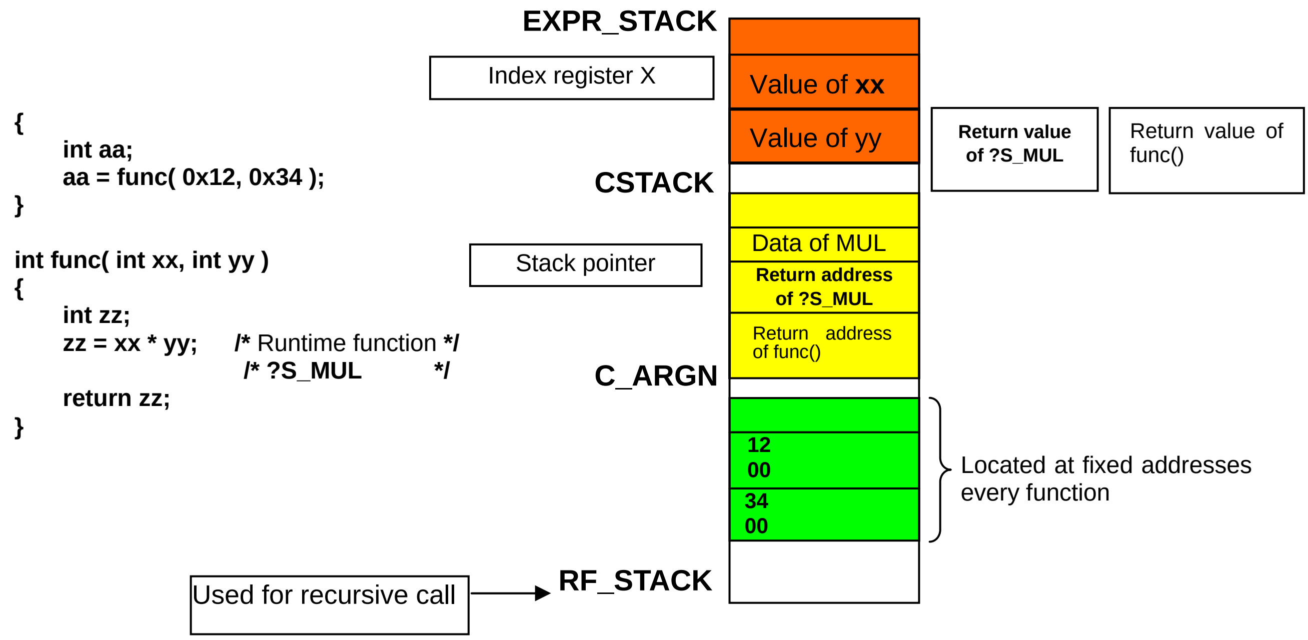
Name (outline)	Type	Description
RCODE (Startup code)	Read-only	Assembly-accessible code used by code generator intrinsic functions. This segment can also be used for user-written assembler code that is not called from C (interrupt handlers and similar resident code).
Z_CDATA (Initialization constants)	Read-only	Assembly-accessible. **CSTARTUP** copies initialization values from this segment to the **Z_IDATA** segment.
N_CDATA (Initialization constants)	Read-only	Assembly-accessible. **CSTARTUP** copies initialization values from this segment to the **N_IDATA** segment.
C_ICALL (Table of indirect function calls)	Read-only	Compiler-only. Holds a table of indirect function calls.
C_RECFN (Table of recursive functions)	Read-only	Compiler-only. Holds a table of recursive functions.
CSTR (String literals)	Read-only	Assembly-accessible. Holds C string literals when the C ompiler Writable strings (-y) option is active. For more information, see "-y" on page 103 of this application notes. See also **ECSTR** and **WCSTR** (Note 2) of "2.3.3. Segment Map: N Page RAM (beginning with 100H)" on page 80 and **CCSTR** of "2.3.4. Segment Map: ROM (up to FFFFH)"on page 82.

CCSTR (String literals)	Read-only	Assembly-accessible. Holds C string literals. For more information, see "-y" on page 103 of this application notes. See also **ECSTR** and **WCSTR** (Note 2) of "2.3.3. Segment Map: N Page RAM (beginning with 100H)" on page 80 and **CSTR** of "2.3.4. Segment Map: ROM (up to FFFFH)"on page 81.
CODE (Code)	Read-only	Assembly-accessible. Holds user program code and various library routines.
CONST (Constants)	Read-only	Assembly-accessible. Used for storing **const** objects. Can be used in assembly language routines for declaring constant data.
C_FNT (Special page branch table)	Read-only	Assembly-accessible. Holds the address of the function invoked according to tiny_func calling rules. This segment must be located in a special page (FF00H–FFFFH).
INTVEC (Interrupt vectors)	Read-only	Assembly-accessible. Holds the interrupt vector table generated by the use of the **interrupt** extended keyword (which can also be used for user-written interrupt vector table entries). The start of this segment should be the start address of the vectors for your particular processor option.

2.4 Description of the Stack Area

2.4.1 Stack Management of ICC740

ICC740 uses multiple segments to exercise stack management.
The stack management of ICC740 is schematically shown below.



EXPR_STACK and INT_EXPR_STACK segments

Used as areas in which the return values and temporary variables of functions are stored.
Also used as areas in which the arguments, temporary variables and return values of C runtime functions are stored.
These segments are manipulated using the index register X, and are located in the address range 0x00 to 0xFF.
These segments are automatically switched to EXPR_STACK during normal operation or switched to INT_EXPR_STACK during interrupt.
* Refer to EXPR_STACK and INT_EXPR_STACK in Section 2.3.2, “Segment Map: Z Page RAM (0H–FFH),” on page 78.

CSTACK segment

Used by JSR, PHA, MUL and DIV instructions.
Also used to save the return address and registers when an interrupt occurs.
This segment is located in the address range 0x00 to 0xFF or 0x100 to 0x1FF as set by the CPU mode register.
* Refer to CSTACK in Section 2.3.3, “Segment Map: N Page RAM (beginning with 100H),” on page 79.

C_ARGN and C_ARGZ segments

An area for the local variables of functions.
Each local variable is located statically (at addresses specific to each).
Since a large amount of RAM is needed if all local variables are located at separate addresses, the linker XLINK locates local variables at shared addresses based on the directive command DEFFN output by the compiler, taking care not to destroy the local variables of upper-level functions. That way, it reduces the necessary RAM size.
* Refer to C_ARGZ segment in Section 2.3.2, “Segment Map: Z Page RAM (0H–FFH),” on page 78, and the C_ARGN segment in Section 2.3.3, “Segment Map: N Page RAM (beginning with 100H),” on page 79.

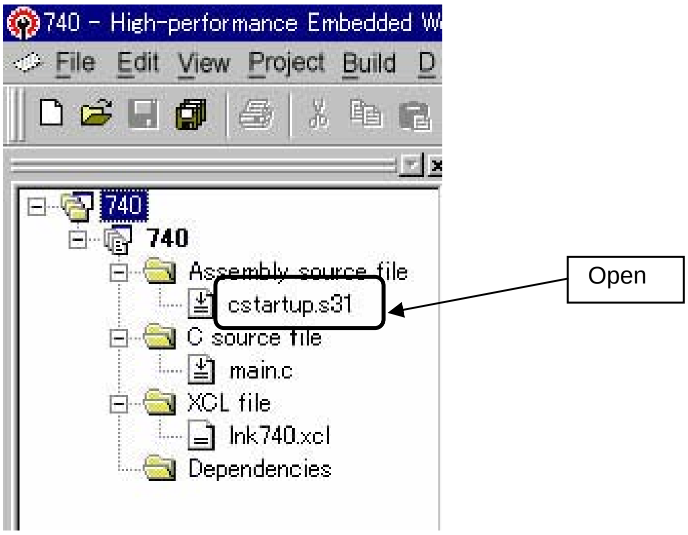
RF_STACK segment

An area for the local variables of functions during recursive calls. This segment is located at address 0x100 and those that follow.
The linker determines whether the call is recursive, according to which the linker set 0 bytes of area when there are no recursive calls or 256 bytes of area (256 bytes for local variables and 2 bytes for management use) when there are recursive calls.
* Refer to RF_STACK in Section 2.3.3, “Segment Map: N Page RAM (beginning with 100H),” on page 79.

2.4.2 Altering the CSTACK Segment

The following shows how to change the CSTACK segment to zero page.

First, open cstartup.s31 in the workspace.



The stack page in cstartup.s31 is set to page 1 (3803 group). To change it to zero page, rewrite “#0CH” on the 137th line to “#08H.”

cstartup.s31:
137 line

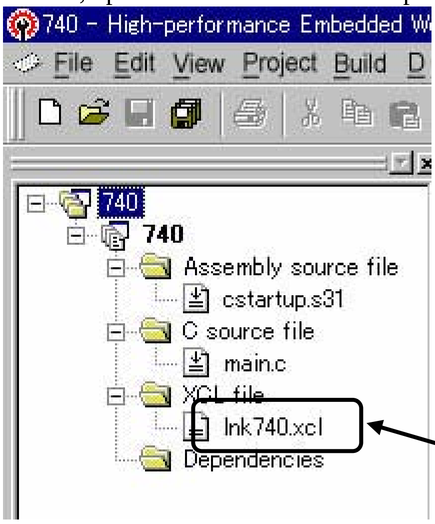
```
        RSEG    RCODE:ROOT
init_C
        CLD
        CLT
        LDM     #0CH, 3BH
        LDX     #LOW (SFE(CSTACK)-1)
        TXS
```

CPU mode register (3803 group)

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

Manual change

Next, open lnk740.xcl in the workspace.



For Tiny or Zero-page model,
lnk740t.xcl

CSTACK is set to page 1.
To change it to zero page, change “NPAGE” on the 57th line to “ZPAGE” and delete the address specification “= 100.” This change causes CSTACK to be located as the last segment of zero page.

lnk740.xcl:
57 line

-Z(NPAGE)CSTACK+40=100

Zero-page model

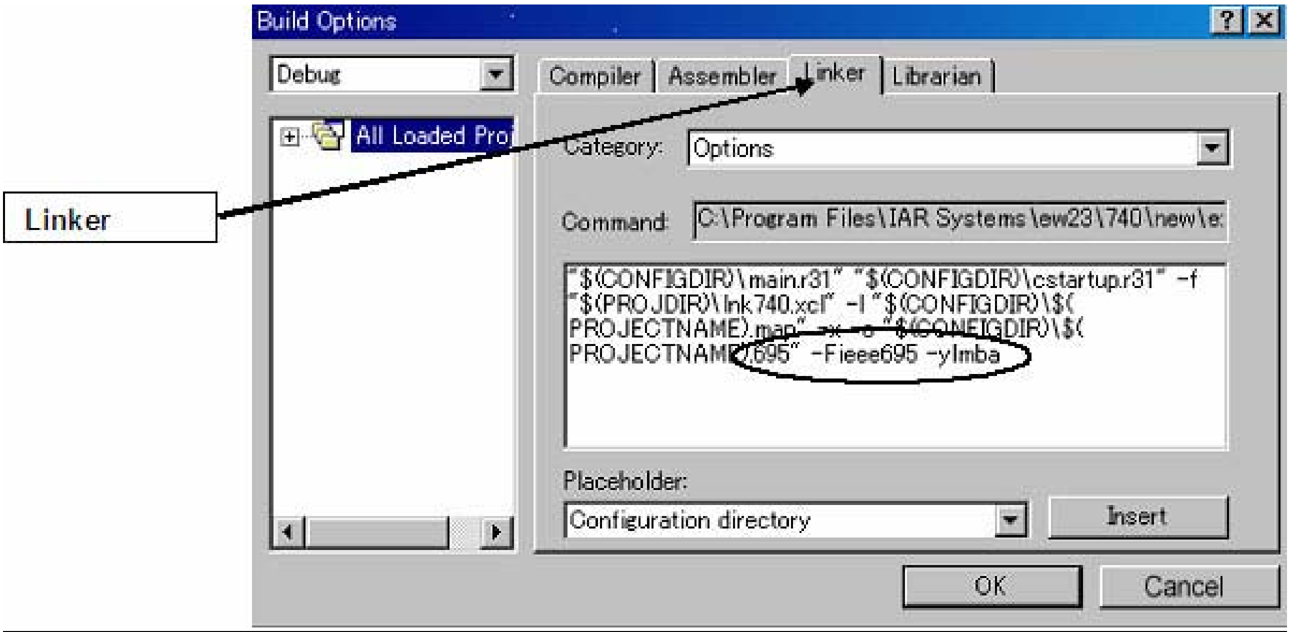
-Z(ZPAGE)CSTACK+20

2.5 Description of the Object Format

2.5.1 Altering the Object Format

When an object is created, its format is set to IEEE695 format Note 1.
To check the set content, choose IAR ICC740 Toolchain from the Build menu to open Build Options. Then select the Linker tab.

Note 1: The IEEE695 format is suitable for debugging with the debugger operating in HEW.



To alter the object format, change the string enclosed with a circle above.

To select the Intel Hex format, change this string as shown below.
-o "$(CONFIGDIR)\$(PROJECTNAME).hex" -Fintel-standard -Y0

To select the Motorola format, change this string as shown below.
-o "$(CONFIGDIR)\$(PROJECTNAME).mot" -Fmotorola

2.6 Description of the C Startup Module

2.6.1 Description of the C Startup Module

M3T-ICC740 uses the C startup module named “cstartup.s31” for project development.
cstartup.s31 is set up as shown below.

Microcomputer	3803 group
Stack pointer operating area	Page 1 (100H–1FFH)

This section describes cstartup.s31 while at the same time explaining which part of it to change and how, as necessary.

The C startup module needs to be changed in the following cases:

1. The processor mode register of the init_C routine is changed (as when the stack pointer area is changed to zero-page memory).
2. Ports and other SFRs are set immediately after reset.
3. A target microcomputer whose interrupt vector information differs from the default vector information is used.

Note that cstartup.s31 does not need to be rewritten for a change of memory models.

Note also that although the library contains the cstartup module, M3T-ICC740 uses this cstartup.s31. M3T-ICC740 has the following line written in lnk740.xcl to ensure that the cstartup module included in the library will not be used.

```
-C c174001
```

Do not delete this -C.

For the contents of segments used in the description of cstartup.s31, refer to pages 76–80.

Description of cstartup.s31

cstartup.s31: Lines 1–40

```
;---------------------------------------------------------------
;
;              cstartup.s31
;
; This file contains the MELPS 740 C startup routine
; and must usually be tailored to suit customer's hardware.
;
;
; You probably want to set up the mode register and you may
; have to change the reset vector.
;
; $Id: cstartup.s31 1.16 2001/07/16 14:16:13Z IJON Exp $
;
;---------------------------------------------------------------


#if 0
#if ((__TID__ >> 4) & 15) == 0
#define MELPS_MULDIV
#endif

#if ((__TID__ >> 4) & 15) == 2
#define MELPS_37600
#endif
#endif

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

Initial setting

M3T-ICC740 doe not use this part.

To use multiple interrupts (interrupt within another interrupt), do not delete the semicolon (;) at the beginning of the line.
When not using multiple interrupts, the semicolon (;) at the beginning of the line may be deleted. This will help to spare 1 byte of RAM and 4 bytes of ROM.

cstartup.s31: Lines 41–80

```
        NAME    CSTARTUP

        EXTERN    main                          ; where to begin execution
        EXTERN    __low_level_init
        DEFFN     __low_level_init(32768,0,0,0)
        EXTERN    exit                          ; where to go when program is don
        DEFFN     exit(32770,0,0,0)

        PUBLIC    ?INTERRUPT_EXPR_STACK     ; Start address for interrupt

        PUBLIC    ?CSTARTUP_INTVEC ; start (base address) of interrupt vector
        PUBLIC    ?CSTARTUP_RESETVEC        ; Location of reset vec

;-------------------------------------------------------------;
;    CSTACK - The C stack segment                              ;
;                                                              ;
;    Please, see in the link file lnk*.xcl how to increment    ;
;    the stack size without having to reassemble cstartup.s31 !  ;
;-------------------------------------------------------------;

        RSEG    CSTACK:ROOT
        BLKB    0

;-------------------------------------------------------------;
;    EXPR_STACK - The expression stack segment                 ;
;                                                              ;
;    Please, see in the link file lnk*.xcl how to increment    ;
;    the stack size without having to reassemble cstartup.s31 !  ;
;-------------------------------------------------------------;

        RSEG      EXPR_STACK:ROOT
        BLKB      0


;-------------------------------------------------------------;
;    INT_EXPR_STACK - The interrupt expression stack segment   ;
;                                                              ;
```

Starting the program module CSTARTUP

380th line: PUBLIC exit
405th line: PUBLIC __low_level_init
Although declared as “PUBLIC” in the same source file, these each are a separate module.
The IAR assembler A740 permits multiple modules to be written in one and the same file. For symbols to be referenced between those modules, however, “EXTERN” and “PUBLIC” are required.

The mark “?” attached at the beginning of a symbol name denotes that this is a reserved symbol name of the compiler. Therefore, do not use the symbol names that begin with “?.”

Declaring a segment

Declaring the hardware stack segment CSTACK

The specific size is set in the lnk740.xcl file, so that 0 is specified as a temporary value here.

Declaring an expression stack segment

The specific size is set in the lnk740.xcl file, so that 0 is specified as a temporary value here.

cstartup.s31: Lines 81–120

```
;    Please, see in the link file lnk*.xcl how to increment          ;
;    the stack size without having to reassemble cstartup.s31 !      ;
;---------------------------------------------------------------;

          RSEG      INT_EXPR_STACK:ROOT
          BLKB      0

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

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


;---------------------------------------------------------------;
; This will insert the information needed by interrupts who use ;
; the interrupt expression stack. Do not alter it!              ;
;---------------------------------------------------------------;

          RSEG    CONST
?INTERRUPT_EXPR_STACK:
          BYTE    SFE(INT_EXPR_STACK)

;---------------------------------------------------------------;
;    Forward declarations of segment used during initialization    ;
;---------------------------------------------------------------;
          RSEG    Z_UDATA
          RSEG    Z_IDATA
          RSEG    Z_CDATA
          RSEG    N_UDATA
          RSEG    N_IDATA
```

Declaring an interrupt expression stack segment

The specific size is set in the lnk740.xcl file, so that 0 is specified as a temporary value here.

Setting multiple interrupts

For multiple interrupts to be enabled in ICC740, the -h option is required.

Setting the interrupt stack

The tail address of the INT_EXPR_STACK segment is set in ?INTERRUPT_EXPR_STACK.
SEF() indicates the end address of the segment.

Declaring various segments used by ICC740.

cstartup.s31: Lines 121–160

```
        RSEG    N_CDATA
        RSEG    ECSTR
        RSEG    RF_STACK

        RSEG    CCSTR
        RSEG    CONST
        RSEG    CSTR


;---------------------------------------------------------------;
;   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

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

;---------------------------------------------------------------;
; If hardware must be initiated from assembly or if interrupts  ;
; should be on when reaching main, this is the place to insert  ;
; such code.                                                    ;
;                                                               ;
; NOTE: You probably want to initialize the mode register here. ;
;---------------------------------------------------------------;
;---------------------------------------------------------------;
; Call __low_level_init to perform initialization before        ;
```

At power-on, the program starts from here. (This address is written in the reset vector.)

The stack operation page is set to page 1.
To change it to zero page,
LDM #08h, 3Bh"; set stack page: 3803 Group
Make sure each bit in the CPU mode register is set as suitable for the working environment.

The trailing –1 of the CSTACK segment is set in the stack pointer S.
LOW() indicates the lower byte of the address.

Setting the data for multiple interrupts

Initializes the multiple interrupt management variable ?IES_USAGE.

Although only a comment is written as the default here, the necessary processing if any to be performed before variables are initialized may be written below this comment.
For example, this will include port settings that need to be set immediately after power-on or determination of hot start.

cstartup.s31: Lines 161–200

```
; initializing segments and calling main.                          ;
; If the function returns 0 no segment initialization should       ;
; take place.                                                      ;
;                                                                  ;
; Link with your own version of __low_level_init to override       ;
; the default action: to do nothing but return 1.                  ;
;-----------------------------------------------------------;

        LDX   #SFE(EXPR_STACK)          ; set up expression stack
        JSR   __low_level_init
        TAY                           ; test return value
        BEQ   skip_seg_init


;-----------------------------------------------------------;
; If it is not a requirement that static/global data is set        ;
; to zero or to some explicit value at startup, the following      ;
; line refering to seg_init can be deleted, or commented.          ;
;-----------------------------------------------------------;

        JSR   seg_init                 ; initialize data segments
        LDX       #SFE(EXPR_STACK)      ; set up expression stack (again)
                                        ; as seg_init destroys it
skip_seg_init

;-----------------------------------------------------------;
; Set up expression stack                                          ;
;-----------------------------------------------------------;

expr_stack_start EQU   SFE(EXPR_STACK)

        LIMIT   expr_stack_start,0,100h,"Expression stack out of range"

        LDX       #expr_stack_start & 255   ; load initial expr stack pointer

        JSR   main          ; execute main()

;-----------------------------------------------------------;
; Now when we are ready with our C program we must perform a       ;
; system-dependent action.   In this case we just stop.             ;
```

Before global variables are initialized (segment initialization) and the Main function is called, the __low_level_init routine is called.
By default, __low_level_init is written to only return the value 1 without executing any action (403th line and on). If the returned value of __low_level_init is 0 (i.e., there are no segments to be initialized), the global variable initialization routine is skipped.

Calls the global variable initialization routine seg_init.

The end address of the EXPR_STACK segment is reassigned to the index register X.
The index register X is used in data processing of the EXPR_STACK segment.

LIMIT is the directive to check whether the symbol is within a specified range. It is written in the form *LIMIT label*, *minimum*, *maximum*, *message*.
If the symbol is assigned any value that is outside a specified range, an error message is output.
More specifically, it checks to see if expr_stack_start is located between 0 to 100h, i.e., located in zero page correctly.
Note that this part is not output to the object file.

Calls the main() function.

cstartup.s31: Lines 201–240

```
;-------------------------------------------------------------;
; DO NOT CHANGE THE NEXT LINE OF CSTARTUP IF YOU WANT TO RUN      ;
; YOUR SOFTWARE WITH THE HELP OF THE C-SPY HLL DEBUGGER.          ;
;-------------------------------------------------------------;

        JMP    REFFN exit

;-------------------------------------------------------------;
; Copy initialized PROMmed code to shadow RAM and clear        ;
; uninitialized variables.                                   ;
;-------------------------------------------------------------;

        EXTERN    ?DP0_L00, ?DP1_L00
seg_init

;-------------------------------------;
; Initialize recursive stack RF_STACK     ;
;-------------------------------------;
        LDA #LOW(SFB(RF_STACK)+2)
        STA np:SFB(RF_STACK)+0
        LDA #HIGH(SFB(RF_STACK)+2)
        STA np:SFB(RF_STACK)+1


;-------------------------------------;
; Zero out Z_UDATA                          ;
;-------------------------------------;
        LDY       #SIZEOF(Z_UDATA)
        BEQ       skip1
        LDX       #SFB(Z_UDATA)
        LDA       #0
loop1   STA       0,X
        INX
        DEY
        BNE       loop1
skip1


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

Jumps to the processing routine to be executed when the main() function processing is completed.

A data pointer. It is used in data transfers between segments.

Initializing the recursive stack (RF_STACK segment)

Registers the address of the RF_STACK segment.
SFB() indicates the start address of the segment.
HIGH() indicates the lower byte of the address.

Initialization processing of global variables that have no initial values assigned (zero-page memory)

The Z_UDATA segment is cleared to 0 using the index registers X and Y.

cstartup.s31: Lines 241–280

```
; Copy Z_CDATA into Z_IDATA                  ;
;--------------------------------------;
         LDY      #SIZEOF(Z_CDATA)
         BEQ   skip2
loop2:   LDA      NP:SFB(Z_CDATA)-1,Y
         STA      NP:SFB(Z_IDATA)-1,Y
         DEY
         BNE   loop2
skip2

;--------------------------------------;
; Zero out N_UDATA                          ;
;--------------------------------------;
         LDM       #LOW(SFB(N_UDATA)),?DP0_L00
         LDM       #HIGH(SFB(N_UDATA)),?DP0_L00+1
         LDA      #0
         TAY
         LDX      #HIGH(SIZEOF(N_UDATA))
         BEQ       skip3
loop3    STA      (?DP0_L00),Y
         INY
         BNE       loop3
         INC      ZP:?DP0_L00+1
         DEX
         BNE       loop3
skip3
         LDX      #LOW(SIZEOF(N_UDATA))
         BEQ       skip4
loop4    STA      (?DP0_L00),Y
         INY
         DEX
         BNE       loop4
skip4


;--------------------------------------;
; Copy CCSTR into ECSTR                        ;
;--------------------------------------;
         LDM       #LOW(SFB(CCSTR)),?DP0_L00
         LDM       #HIGH(SFB(CCSTR)),?DP0_L00+1
```

Initialize global variables with initial values specified (zero page memory)

Initial values are copied from the Z_CDATA segment to the Z_IDATA segment using the index register Y.

Initialize global variables with no initial values specified (N page memory)

The N_UDATA segment is cleared to 0 using the data pointer ?DP0_L00.

Initializers for C string literals when –y compiler option is used in ICC740.

cstartup.s31: Lines 281–320

```
            LDM         #LOW(SFB(ECSTR)),?DP1_L00
            LDM         #HIGH(SFB(ECSTR)),?DP1_L00+1
            LDX         #HIGH(SIZEOF(CCSTR))
data pointers?DP0_L00               BEQ         skip5
            JSR         large_copy_mem
skip5
            LDX         #LOW(SIZEOF(CCSTR))
            BEQ         skip6
            JSR         small_copy_mem
skip6

;--------------------------------------;
; Copy N_CDATA into N_IDATA                          ;
;--------------------------------------;
            LDM         #LOW(SFB(N_CDATA)),?DP0_L00
            LDM         #HIGH(SFB(N_CDATA)),?DP0_L00+1
            LDM         #LOW(SFB(N_IDATA)),?DP1_L00
            LDM         #HIGH(SFB(N_IDATA)),?DP1_L00+1
            LDX         #HIGH(SIZEOF(N_CDATA))
            BEQ         skip7
            JSR         large_copy_mem
skip7
            LDX         #LOW(SIZEOF(N_CDATA))
            BEQ         return

; fall into small_copy_mem to do final copy

;--------------------------------------;
; Copy memory                                        ;
;--------------------------------------;
small_copy_mem
            LDY         #0
small_loop
            LDA         (?DP0_L00),Y
            STA         (?DP1_L00),Y
            INY
            DEX
            BNE         small_loop
return      RTS
```

Initial values are copied from the CCSTR segment to the ECSTR segment using the data pointers ?DP0_L00 and ?DP1_L00.

Initialize global variables with initial values specified (N page memory)

Initial values are copied from the N_CDATA segment to the N_IDATA segment using the data pointers ?DP0_L00 and ?DP1_L00.

Copy processing routine

cstartup.s31: Lines 321–360

```
large_copy_mem
          LDY      #0
large_loop
          LDA      (?DP0_L00),Y
          STA      (?DP1_L00),Y
          INY
          BNE      large_loop
          INC      ?DP0_L00+1                ; update high pointers
          INC      ?DP1_L00+1
          DEX
          BNE      large_loop                ; no, move next block
          RTS


;---------------------------------------------------------------;
; Interrupt vectors must be inserted here by the user.          ;
;                                                                ;
; It is assumed that the interrupt vector segment starts        ;
; at address xxE0 on all chips except 37600 where it is         ;
; starts at xxC0.   The reset vector is assumed to be located   ;
; at xxF?.   We simply skip to xxF? and insert the reset vector. ;
;                                                                ;
; Chip group     Default reset vector                           ;
; -----------------------------------------                     ;
;      -v0        FFFC                                           ;
;      -v1        FFFE                                           ;
;      -v2        FFFA                                           ;
;                                                                ;
; If this does not match your specific chip derivative, you     ;
; have to make changes below.                                    ;
;---------------------------------------------------------------;

          COMMON   INTVEC

?CSTARTUP_INTVEC:
          BLKB     0FFFEH - 0FFDCH -2        ; 3803 Group
#if 0
#if defined(MELPS_37600)
          BLKB     40H - 6        ; FFFA   ( FFC0 + 40 - 6)   (-v2)
```

Setting the INTVEC segment

Declares the INTVEC segment as a common segment.
Common segments are overwritten.
The start address of the INTVEC segment is written in lnk740.xcl.

Sets a memory size equal to the interrupt vector minus the reset interrupt vector.
Here, the memory size is calculated as shown below.

End address of interrupt vector – Initial address of interrupt vector – Size of reset interrupt vector

The value of each interrupt vector is set in interrupt [ ] that is written in ICC740.

interrupt [16] void intr_timer2(void)

In the above example, the address of intr_timer2 is written over at the INTVEC segment start address + 16 when linked.

cstartup.s31: Lines 361–400

```
#elif defined(MELPS_MULDIV)
        BLKB      20H - 4        ; FFFC
#else
        BLKB      20H - 2        ; FFFE
#endif
#endif
?CSTARTUP_RESETVEC:
        WORD      init_C
        ENDMOD    init_C


;---------------------------------------------------------------;
; Function/module: exit (int code)                              ;
;                                                               ;
; When C-SPY is used this code will automatically be replaced   ;
; by a 'debug' version of exit().                               ;
;---------------------------------------------------------------;
        MODULE    exit

        PUBLIC    exit
        DEFFN     exit(0,0,0,0,32770,0,0,0)
        PUBLIC    ?C_EXIT

        RSEG    RCODE

?C_EXIT
exit

;---------------------------------------------------------------;
; The next line can be replaced by user defined code.           ;
;---------------------------------------------------------------;
        BRA    *

        ENDMOD


;---------------------------------------------------------------;
; Function/module: default __low_level_init                     ;
; You can replace this routine by linking with your own version.;
; The default action is to do nothing and return 1.             ;
```

Sets the reset vector.
The program starts from init_C after reset.

Defines termination of the Init_C module.

Module exit

Defines start of the module exit.

This asterisk (*) denotes a location counter. The jump address is the counter itself, comprising an endless loop.

Defines termination of the module exit.

Module lowinit

cstartup.s31: Lines 401–413

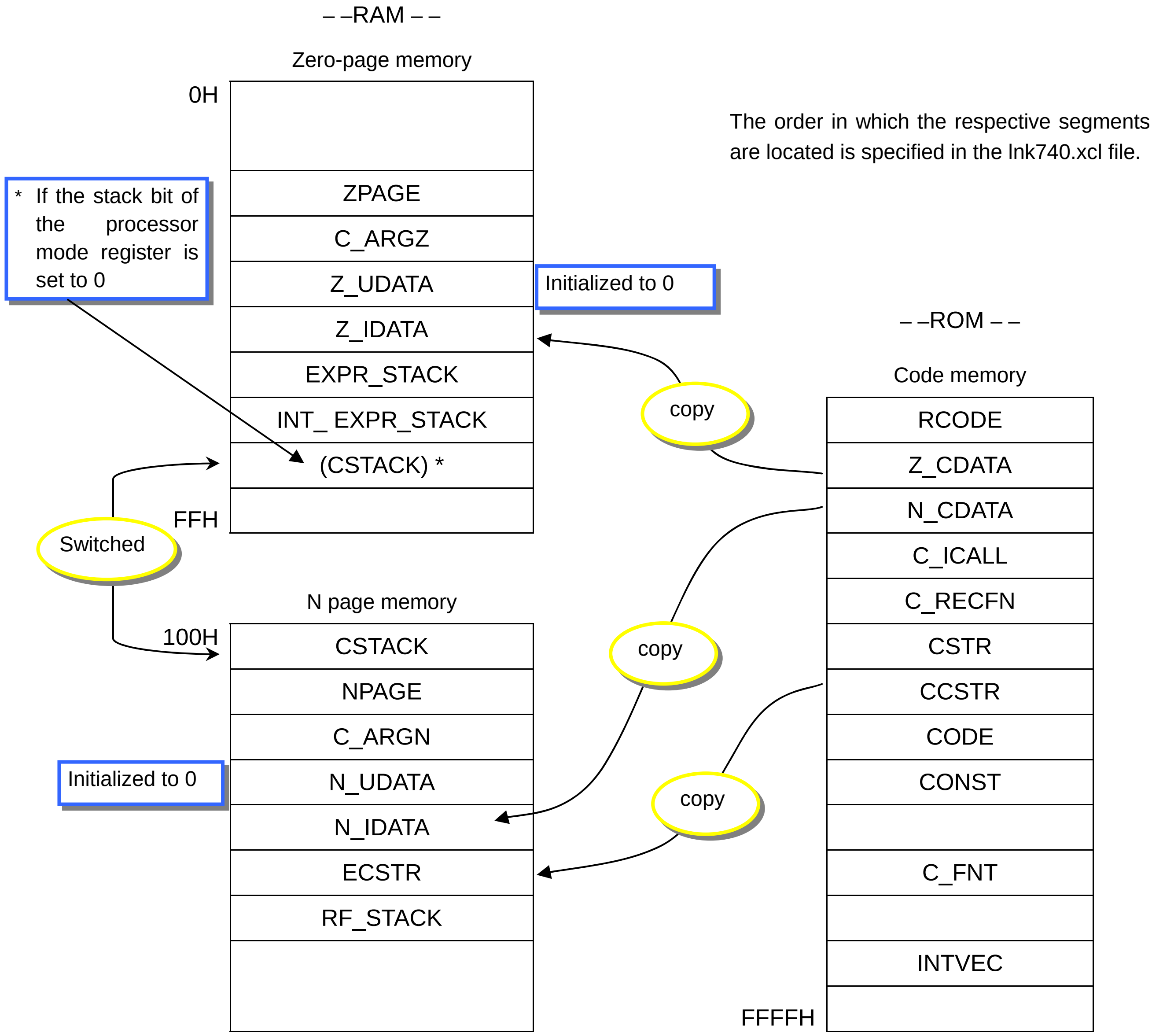
```
;---------------------------------------------------------------;

        MODULE   lowinit

        PUBLIC   __low_level_init
        DEFFN    __low_level_init(0,0,0,0,32768,0,0,0)
        RSEG     RCODE
__low_level_init
        LDA      #1
        RTS

        END
```

Indicates start of the module lowinit.

Processing of the lowinit module is written here.
By default, it only returns the value 1.
If all segments need to be initialized here, alter the processing written here to return the value 0.

Defines end of the source file.

Processing of the C startup mode is schematically shown below.



2.7 Setting Values in a Special Area

2.7.1 Setting Values in a Special Area

To set values in an area used for special purposes such as ID code, create a segment for that area in the lnk740.xcl file and set values in an assembly language file.
This area differs with each product. Consult the data sheet of each MCU used.

An example of an assembly language file and lnk740.xcl setup is shown below.

･Example for the 7542 group (flash memory version)

FFD4h	ID1
FFD5h	ID2
FFD6h	ID3
FFD7h	ID4
FFD8h	ID5
FFD9h	ID6
FFDAh	ID7
FFDBh	ROM code protect

lnk740.xcl file

```
  -Z(CODE)RCODE, … (line next to this line)
  -Z(CODE)ID_CODE=FFD4-FFDB
```

Assembly language file:

```
  RSEG      ID_CODE
  BYTE      0FFH
  BYTE      0FFH
  BYTE      0FFH
  BYTE      0FFH
  BYTE      0FFH
  BYTE      0FFH
  BYTE      0FFH
ROMCP:
  BYTE      0FFH                ; ROM Code Protect
```

Set the ID code and ROM code protect values.

･Example for the 7545 group (QzROM version)

FFD4h	Area used for shipping inspection by Renesas
FFD5h	Area used for shipping inspection by Renesas
FFD6h	Area used for shipping inspection by Renesas
FFD7h	Area used for shipping inspection by Renesas
FFD8h	Area used for shipping inspection by Renesas
FFD9h	Area used for shipping inspection by Renesas
FFDAh	Function setting ROM data
FFDBh	ROM code protect

lnk740.xcl file

```
  -Z(CODE)RCODE, …(line next to this line)
  -Z(CODE)RESERVE1,FUNCTION_SET_ROM,RESERVE2=FFD4-FFDB
```

Assembly language file:

```
  RSEG        RESERVE1
  BLKB        01H
  BLKB        01H
  BLKB        01H
  BLKB        01H
  BLKB        01H
  BLKB        01H
  RSEG      FUNCTION_SET_ROM
  BYTE      12H                 ; Function Set Rom Data
  RSEG      RESERVE2
  BLKB      01H                 ; ROM Code Protect
```

Do not write anything to the area used for shipping inspection by Renesas.
Set the function setting ROM data value.
Use the ROM writer to set the ROM code protect value.

Chapter 3

C Compiler: ICC740

This chapter describes the options and functionality of the C compiler, ICC740.

3.1 Description of Basic Options

3.1.1 Summary of the Compiler Options

The following is a summary of all the compiler options.

Option	Description
-A*prefix*	Assembly output to prefixed filename.
-a *filename*	Assembly output to named file.
-b	Make a LIBRARY module.
-C	Allow nested comments.
-c	‘char’ is ‘signed char’.
-D*symb[xx]*	Defined symbols.
-e	Enable language extensions.
-F	Form feed after function.
-f *filename*	Extend the command line.
-G	Open standard input as source.
-g[0][A]	Global strict type checking.
-h	Manages the INT_EXPR_STACK segment during multiple interrupts.
-H*name*	Set object module name.
-I*prefix*	Include paths.
-i	Add **#include** file text.
-K	Allow ‘***//***’ comments.
-L[*prefix*]	List to prefixed source name.
-l *filename*	List to named file.
-m[tl]	Memory model.
-N*prefix*	Preprocessor to prefixed filename.
-n *filename*	Preprocessor to named file.
-O*prefix*	Set object filename prefix.
-o *filename*	Set object filename.
-P	Generate PROMable code.
-p*lines*	Formats a list into a page.
-q	Insert mnemonics.
-R*name*	Code segment.
-r[012inre]	Generate debug information.
-S	Set silent operation.
-s[0-9]	Optimize for speed.
-T	Active lines only.
-t*n*	Tab spacing.
-U*symb*	Undefine symbol.
-v*n*	Processor configuration.
-w	Disable warnings.
-X	Explain C declarations.
-x[D][F][T][2]	Cross reference.
-y	Writable strings.
-z[0-9]	Optimize for size.

3.2 Language extensions

This chapter summarizes the extensions provided in the 740 C Compiler to support specific features of the 740 microprocessor.

Introduction

The extensions are provided in three ways:

- As extended keywords. By default, the compiler conforms to the ANSI specifications and 740 extensions are not available. The command line option **-e** makes the extended keywords available, and hence reserves them so that they cannot be used as variable names.
- As **#pragma** keywords. These provide **#pragma** directives which control how the compiler allocates memory, whether the compiler allows extended keywords, and whether the compiler outputs warning messages.
- As intrinsic functions. These provide direct access to very low-level processor details.

3.2.1 Extended keywords summary

The extended keywords provide the following facilities:

Addressing control

Variables may be forced into the Zero Page area with **zpage**, or out of Zero Page with **npage**.
Also, variables can be declared as a single-bit zero page variable depending on a bit.
zpage, npage

Non-volatile ram

Variables may be placed in non-volatile RAM by using the following data type modifier:
no_init

I/O access

The **sfr** data type can be used to declare byte I/O identifiers.
sfr

Interrupt routines

Interrupt handlers and non-interruptable routines may be written in C using the following keywords:
interrupt monitor

Calling procedure

Functions can have the calling sequences altered using the keyword give below.
tiny-func

3.2.2 Pragma directive summary

#pragma directives provide control of extension features while remaining within the standard language syntax.
Note that **#pragma** directives are available regardless of the **-e** option.
The following categories of **#pragma** functions are available:

Bitfield orientation

#pragma bitfields=default
#pragma bitfields=reversed

Extention control

#pragma language=default
#pragma language=extended

Function attribute

#pragma function=default
#pragma function=interrupt
#pragma function=intrinsic
#pragma function=monitor
#pragma function=tiny_func

Memory usage

#pragma codeseg
#pragma memory=constseg
#pragma memory=dataseg
#pragma memory=default
#pragma memory=no_init
#pragma memory=zpage
#pragma memory=npage

Warning message control

#pragma warnings=default
#pragma warnings=off
#pragma warnings=on

3.2.3 Predefined symbolssummary

Predefined symbols allow inspection of the compile-time environment.

Function	Description
DATE	Current date in **Mmm dd yyyy** format.
FILE	Current source filename.
_IAR_SYSTEMS_ICC_	IAR C compiler identifier.
LINE	Current source line number.
STDC	ANSI C compiler identifier.
TID	Target identifier.
TIME	Current time in **hh:mm:ss** format.
VER	Returns the version number as an **int**.

3.2.4 Other extensions

$ character

The character **$** has been added to the set of valid characters in identifiers for compatibility with DEC/VMS C.

Use of sizeof at compile time

The ANSI-specified restriction that the **sizeof** operator cannot be used in **#if** and **#elif** expressions has been eliminated.

Chapter 4

Assembler: A740

This chapter describes the options and other features of the assembler, A740.

4.1 Description of Basic Options

4.1.1 Outline of the Assembler Options

The following is a summary of all the assembler options.

Option	Description
-B	Outputs macro execution information.
-b	Make a LIBRARY module.
-c{DMEAO}	Sets list options.
-D*symb*[=*xx*]	Defined symbols.
-d	Does not check whether #ifdef and #endif are in pairs.
-E*number*	Sets the maximum number of errors.
-f *filename*	Extend the command line.
-G	Open standard input as source.
-I*prefix*	Adds include search prefix.
-i	Lists include files.
-L[*prefix*]	Generates a list in the prefixed source name.
-l *filename*	Generates a list in the named file.
-Mab	Sets macro argument quoted character
-N	Does not attach a header to a list.
-O*prefix*	Set object filename prefix.
-o *filename*	Set object filename.
-p*nn*	Sets the number of lines in each page.
-r[en]	Enables debugger output in an object.
-S	Set silent operation.
-s{+|-}	Discriminates user symbols between uppercase and lowercase.
-T	Active lines only.
-t*n*	Tab spacing.
-v*n*	Processor configuration.
-U*symb*	Undefine symbol.
-uN	Sets 16-bit addressing.
-w[*string*]	Disables warning.
-x{DI2}	Generates a cross reference list.

4.2 Assembly Language Interface

This section describes the interface between assembly language subroutines and C language functions that is used when calling an assembly language subroutine from a C language function and vice versa.

4.2.1 Function Declaration

No matter which side whether a function is called from the C language or the assembly language side, a declaration of the called function must be written on the assembly language side.
For this function declaration, write the assembler directive DEFFN in an assembly language source file. DEFFN is required for the calculation of the C_ARGZ and C_ARGN segment sizes.

1) To call an assembly language subroutine from C language
   Set the auto-variable and argument sizes in the assembler directive DEFFN.
   Furthermore, if a function call is involved, write the called function in “call1,...”

Auto-variable size to be set in the C_ARGZ segment

Argument size to be set in the C_ARGZ segment

DEFFN   sub1(a, 0, b, 0, 0x8000+x, 0, y, 0) [, call1, call2, ...]

Argument size to be set in the C_ARGN segment

Auto-variable size to be set in the C_ARGN segment

If the function is to be used in interrupt handling, be sure to set the first parameter as shown below.

DEFFN   sub1(0x200+a, 0, b, 0, 0x8000+x, 0, y, 0) [, call1, call2, ...]

2) To call a C language function from assembly language
   Set the argument sizes in the assembler directive DEFFN.

Argument size to be set in the C_ARGZ segment

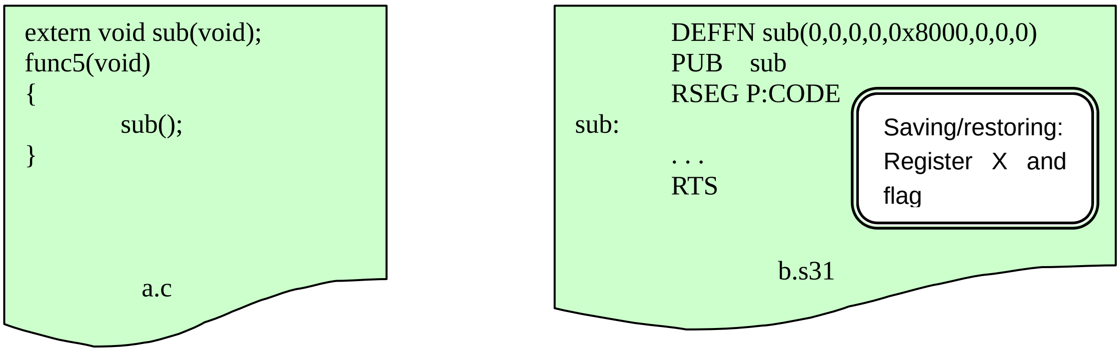
DEFFN   sub2(0x8000+x, 0, y, 0)

Argument size to be set in the C_ARGN segment

4.2.2 Calling an Assembly Language Subroutine from C Language

1) To call an assembly language subroutine without arguments and return values from C language

The following shows an example of how to write a program fragment for calling an assembly language subroutine without arguments and return values from C language.

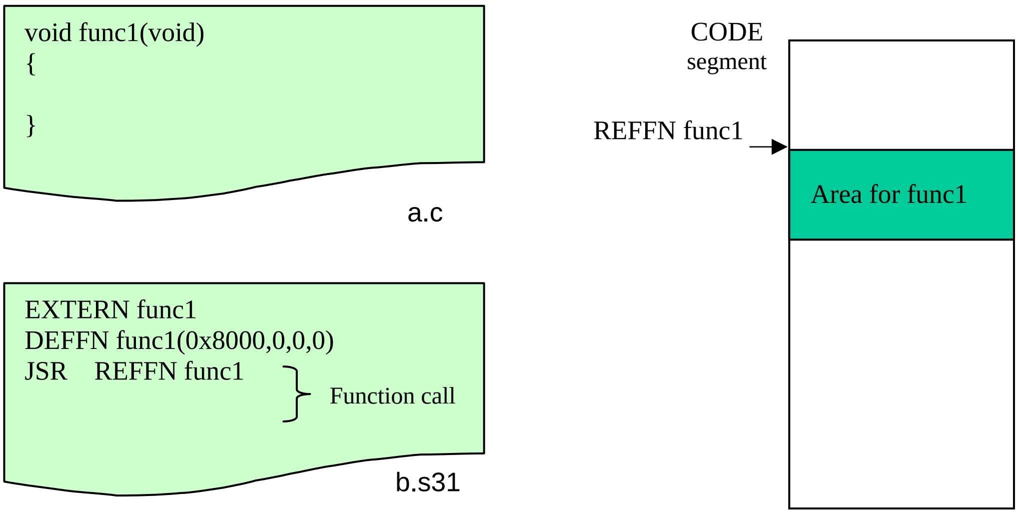


In the C language source, declare the assembly language subroutine as *extern* before it is called. In the called assembly language subroutine, write processes for saving and restoring the index register X and processor status register.

4.2.3 Calling a C Language Function from Assembly Language

1) To call a C language function without arguments and return values from assembly language

The following shows an example of how to write a program fragment for calling a C language function without arguments and return values from assembly language.

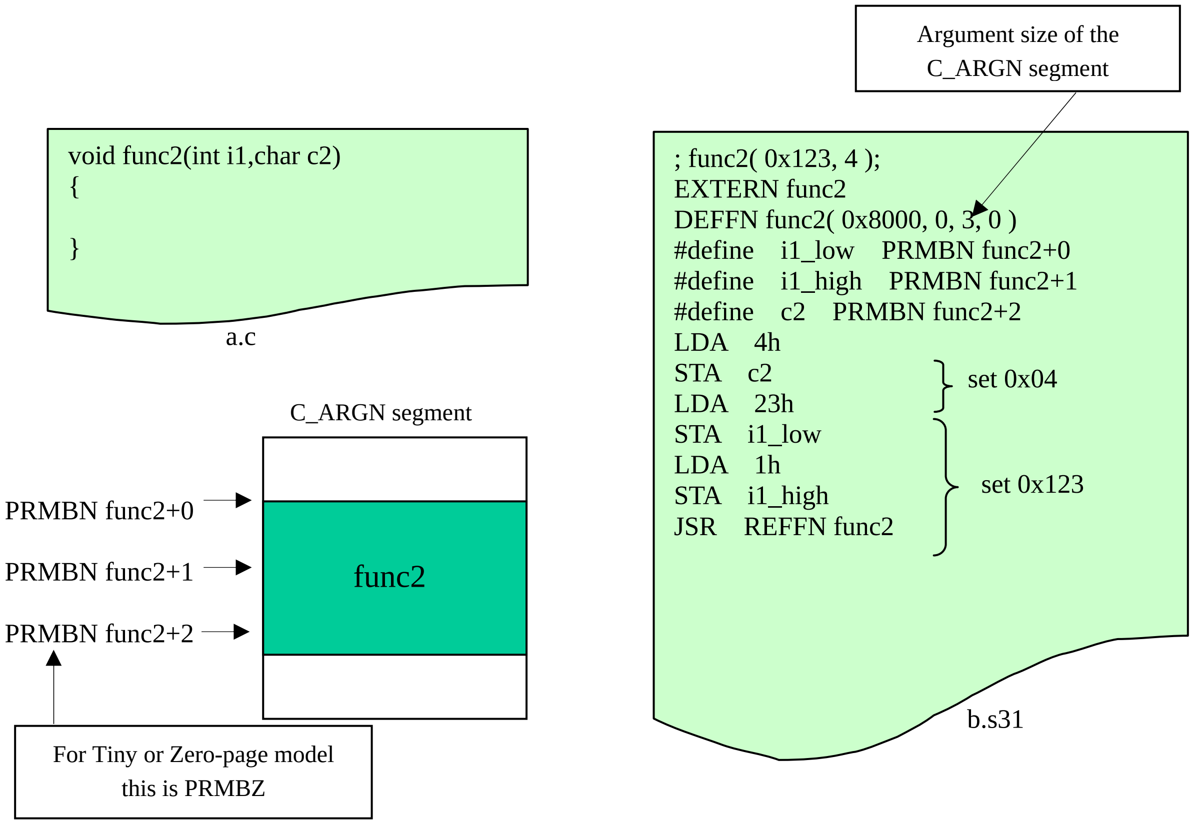


To call a C language function in an assembly language subroutine, add REFFN in front of the function to be called.
In the C language function, the EXPR_STACK segment (or INT_EXPR_STACK segment if an interrupt occurs) is used for C runtime function calls, etc. This segment is manipulated with the index register X.
If the index register X is not indicating the EXPR_STACK segment, set up the index register X before a function call.

Example: LDX #LOW(SFE(EXPR_STACK))

2) To call a C language function with arguments from assembly language (Large model)

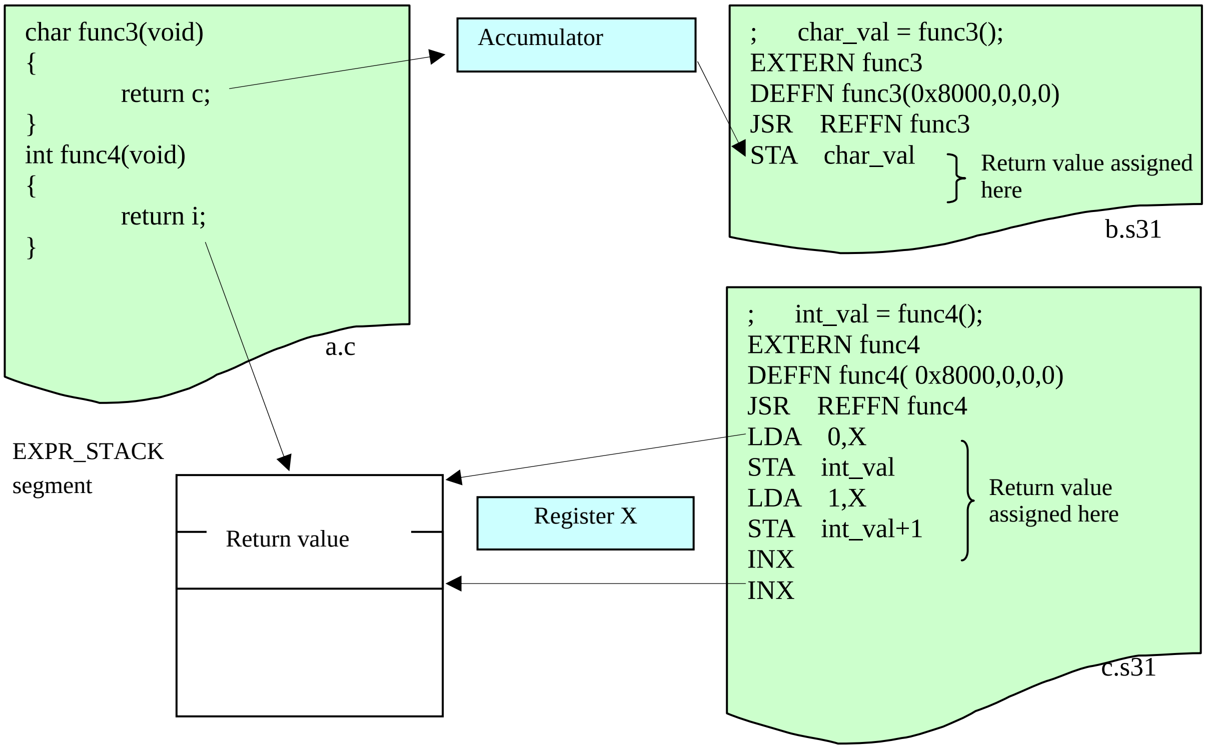
The following shows an example of how to write a program fragment for calling a C language function that has arguments from assembly language.



Use PRMBN to set arguments to a C language function in an assembly language subroutine. The “PRMBN function name + offset” indicates the position of the argument.

3) To call a C language function with return values from assembly language

The following shows an example of how to write a program fragment for calling a C language function that has return values from assembly language.



The return value from a C language function, if *char* type or 1-byte data, is set in the accumulator. The return values of other data types are stored in the EXPR_STACK segment.

Make sure that after a C language function is called in an assembly language subroutine, the return value is assigned to assembly language data according to its type.

If the return value is stored in the EXPR_STACK segment, use the index register X for this assignment because the position of the return value is indicated by the index register X. After assignment, increment the index register X by an amount equal to the size of the return value.

Chapter 5

Linker: XLINK

This chapter describes the options and other features of the linker, XLINK.

5.1 Description of the Basic Options

5.1.1 Outline of the Options

The table below outlines the XLINK options used by the 740 family.

Option	Content
–! comment –!	Comment delimiter.
–C *file*,...	Loads the file as a library. Supplement: To use cstartup.s31, specify –C. (Reason: The library contains a cstartup module, so that unless –C is specified, this module and cstartup.s31 will be used, resulting in a double-module error.)
-c*cpu*	Processor type.
–D *symbol* = *value*	Define symbol.
-d	Disable code generation.
–e *new* = *old* [, *old*]...	Rename external symbols.
–F *format*	Output format.
–f *file*	XCL filename.
-G	No global type checking.
–H *hexa value*	Disable code generation.
–I *path*	Include paths.
–J *size, method* [, *complement*]	Generates checksum.
–L *directory*	Specifies a list file director.
–l *file*	List to named file.
–o *file*	Output file.
–p *number of lines*	Lines/page.
-R[w]	Disable range check.
-S	Silent operation.
–w[*number*|s|t]	Disable warnings.
-x[e][h][i][m][n][s][o]	Cross reference. (Note: Refer to xlink.pdf.)
–Y[*string*]	Format variant.
–y[*string*]	Format variant.
–Z[@] *segment*	Define segments. (Note: Refer to xlink.pdf.)
−z	Segment overlap warnings.

5.2 Description of Option Files

5.2.1 Description of the Link Command File

The link command file has a template prepared in it. To make it adapted for the target system, it must be altered partly.
This section describes the link command file (lnk740.xcl) while at the same time explaining which part of it to change and how, as necessary.

The link command file needs to be changed in the following cases:

1. A microcomputer that has different interrupt vector and memory locations from those of the default microcomputer is used.
2. The C stack area is switched from 1-page memory to zero-page memory.
3. Location of each segment is changed.

Note that the numeric values in the link command file are processed in hexadecimal.

Description of the lnk740.xcl file

lnk740.xcl: Lines 1–30

```
-!                     - 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

            $Id: lnk740.xcl 1.4 2001/07/16 14:14:59Z IJON Exp $

            IMPORTANT:        1. Use a COPY of this file.
                           2. Select a C library at the end of this file
                              that matches the compilation switches.
                           3. If you use the 37600, see note about the
                              INTVEC segment futher down.

            MODIFICATION:     M38034M4

            First: define CPU  -!

-c740


-!   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.
```

A range of lines from this to the second occurrence of -! comprises a comment.

Defining the CPU

Fixed to the 740 family

Defining the BITVARS segment

BITVARS locates variables whose addresses are undetermined by a bit declaration.
The BITVARS segment is specified by a bit address (address in bit units) from address 0.
For the 3803 group, since the SFR is located at addresses up to 3FH, the next address 40H is indicated by a number of bits, 200H (= 40H x 8).

–Z specifies segment placement.
The word in ( ) assigns type to the segment.
This type affects the manner in which segment overlapping is processed.
The following types are used in this link command file:

BIT	Bit memory
ZPAGE	Zero-page data memory
NPAGE	Data memory accessed by absolute addressing
CODE	Code memory

lnk740.xcl: Lines 31–60

```
    The following segment defintions (EXPR_STACK, INT_EXPR_STACK and
    CSTACK) that do not have an address given must fit inside the
    "41-FF" address range.
    If you have the CSTACK (processor stack) segment outside zero page,
    you have to give it an address and XLINK will no longer try to
    fit it within zero page.                   -!

-Z(ZPAGE)ZPAGE,C_ARGZ,Z_UDATA,Z_IDATA=41-FF


-!  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
```

Setting zero-page RAM

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

The segments separated by a comma (,) are mapped to memory in the order they are written.

Setting an expression stack segment

The " +20" here specifies that the size is 20H. Alter it as necessary.

Segment name + YY: The segment is allocated in such a way that it will have the set memory size (YYH bytes).

Setting an interrupt expression stack segment

The " +0" here specifies that the size is 0. Alter it as necessary.

Setting a C stack segment

Specifies the address range 100H to 13FH for a C stack area.
To locate a C stack area in zero page, alter this line as shown below.
    -Z(ZPAGE) +40
    40H bytes of area will be allocated after INT_EXPR_STACK.
    * cstartup.s31 also needs to be altered.

* Note
The segments listed below must be located in zero page.
    BITVARS, ZPAGE, C_ARGZ, Z_UDATA, Z_IDATA,
    EXPR_STACK and INT_EXPR_STACK

lnk740.xcl: Lines 61–90

Setting nonzero-page RAM

```
-!  Setup "NPAGE" segments at address 1000-7FFF    -!

-Z(NPAGE)NPAGE,C_ARGN,N_UDATA,N_IDATA,ECSTR=100-43F
```

Specifies the segments to be located in N-page memory.
Although this specification begins with address 100H, since CSTACK uses an address range of up to 13FH, NPAGE is 140H.
2 bytes are used to hold the address of RF_STACK.

```
-! Setup "RF_STACK" segment.   This stack is used for recursive function.
   It is by default given a size of 256 bytes by the library.   By giving
   a non-zero size below, you _expand_ the stack by that amount.   -!

-Z(NPAGE)RF_STACK+0
```

When not using recursive functions, specify +0 here.
When using recursive functions, specify any value other than 0 here, so that the stack is assigned 256 + 2 bytes of storage. Two bytes are used to hold the address of RF_STACK for management purpose.

Setting ROM

```
-!  Setup all read-only segments (PROM) at address 8000   -!

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

Sets the segments to be located in ROM and their addresses.
For microcomputers that have a reserved area, make sure these addresses do not include the reserved area.
Make sure also that these addresses do not include the vector area.

```
-!  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                          -!
```

Setting the interrupt vector INTVEC (next page)

Specify an address range from the beginning of the interrupt vector to the end of the reset vector.
The example here applies to the 3803 group. Alter it to suit the MCU used.

Setting the special page C_FNT (next page)

Sets a area for the functions that use extended description tiny_func. If CONST, etc. exceeds FF00H, C_FNT is located following that segment.

lnk740.xcl: Lines 91–122

```
-Z(CODE)INTVEC=FFDC-FFFD
-Z(CODE)C_FNT=FF00-FFDB


-!   See configuration section concerning printf/sprintf -!
-e_small_write=_formatted_write


-!   See configuration section concerning scanf/sscanf -!
-e_medium_read=_formatted_read


-! 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
     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   -!


-!   Code will now reside on file aout.a31 in INTEL-STANDARD format -!
```

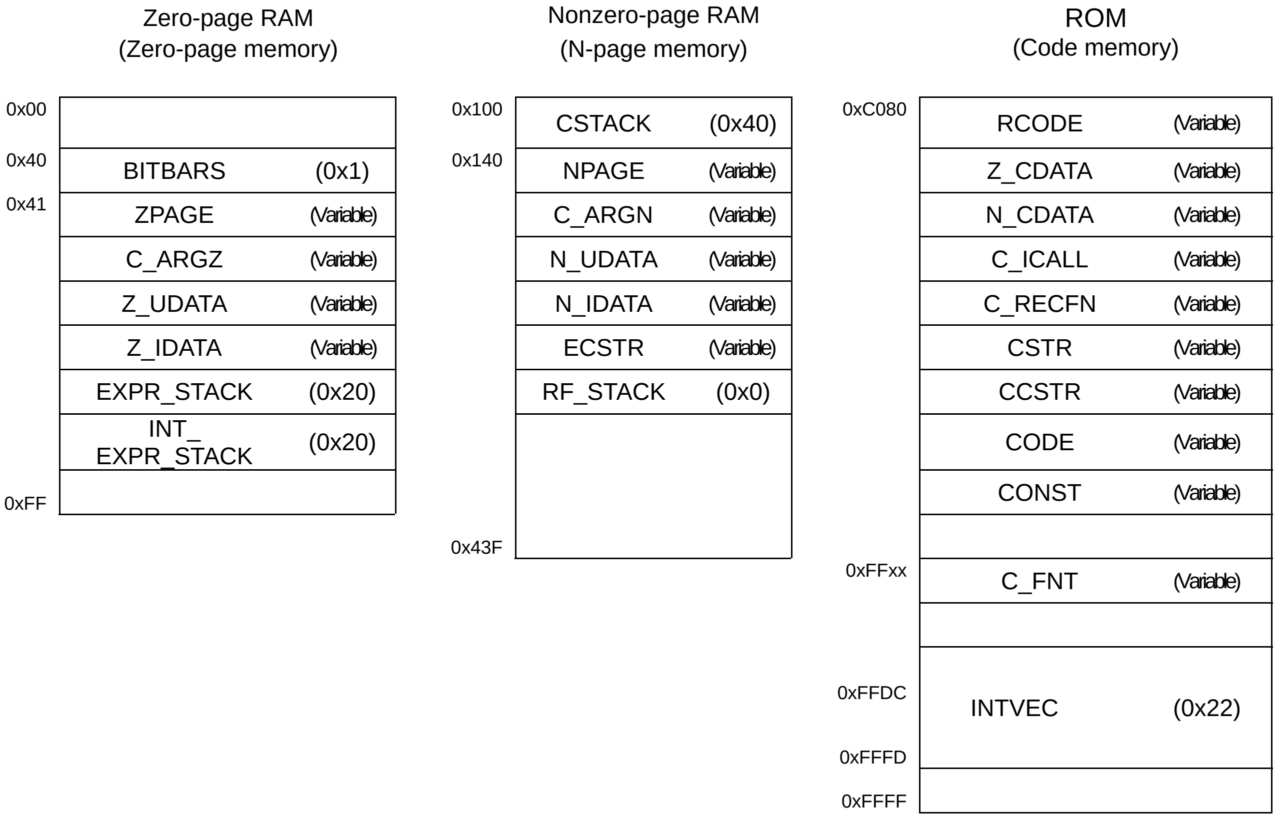
Changing the formats of printf, scanf, etc.

Since the library functions that use printf and other formats are large in size, the necessary size may be reduced by changing the types of variables to be displayed.
When not using printf and other formats, there is no need to change.

-e A=B: Changes the existing external symbol name B to a new name A.

–medium_write	Floating-point numbers unsupported
–small_write	Only %%, %d, %o, %c, %s and %x supported
–medium_read	Floating-point numbers unsupported

The segment placement in the lnk740.xcl file is as shown below.



The variable segment sizes are set by the linker XLINK.
Although C_FNT can be located beginning with 0xFF00, if the CONST segment exceeds 0xFF00, C_FNT is located following that segment.

Chapter 6

Debugger

This chapter describes mainly the functionality of the High-performance Embedded Workshop (HEW) as a “debugger.”

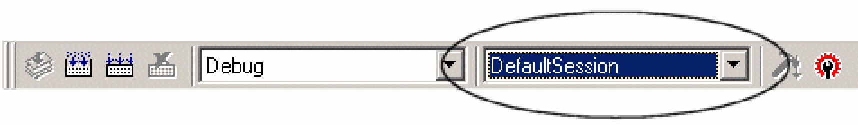
6.1 Starting the Debugger

This section describes how to connect and close the 740 simulator in the debugger.
The debugging can be started by connecting with the simulator.

6.1.1 Connecting the Simulator

Connect the simulator by simply switching the session file to one in which the setting for the simulator use has been registered.

Select “Session740_Simulator” from the drop-down list of the tool bar shown below.



After the session name is selected, the dialog box for setting the debugger is displayed and the simulator will be connected.
For details on setting-up, see section 6.2 and 6.3.
After the setting is finished, the connection will be completed.

6.1.2 Ending the Simulator

The simulator can be exited by using the following methods:

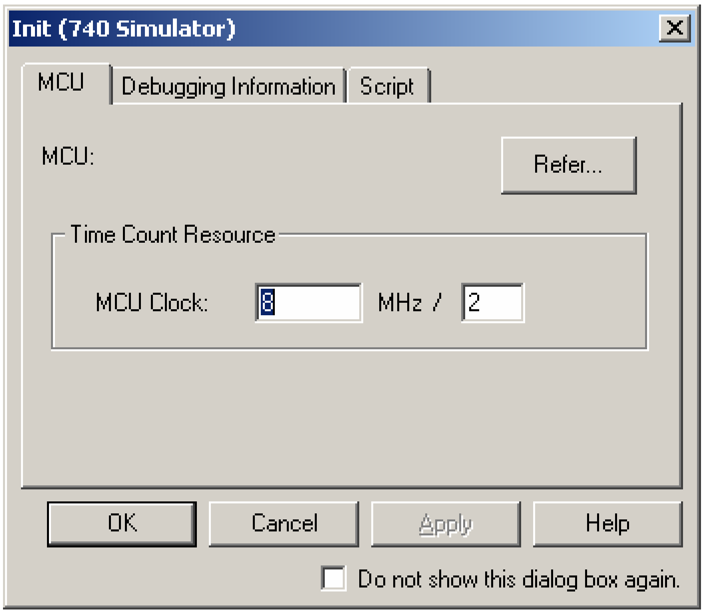
1. Selecting the "DefaultSession"
   Select the "DefaultSession" in the list box that was used at the time of simulator connection.
2. Exiting the High-performance Embedded
   Workshop Select [Exit] from the [File] menu. High-performance Embedded Workshop will be ended.

The message box, that asks whether to save a session, will be displayed when the session is switched or HEW is exited. If necessary to save it, click the [Yes] button. If not necessary, click the [No] button

6.2 Setting Up the Simulator

6.2.1 Init Dialog

The Init dialog box is provided for setting the items that need to be set when the debugger starts up. The contents set from this dialog box are also effective the next time the debugger starts.
In the Init dialog box, specify an MCU file. You can use “Refer…” button and select an MCU file from the ensuing list. If the MCU file for your microcomputer cannot be found, create an MCU file.



You can open the Init dialog using either one of the following methods:
- After the simulator gets started, select Menu - [Setup] → [Simulator] → [System...].
- Start Debugger while holding down the Ctrl key.

6.3 Creating an MCU File for the Simulator

If the MCU file for your microcomputer cannot be found, create an MCU file here.
In the MCU file, write the following contents in the order listed below. For the file name, specify the MCU name (“m3xxxx.mcu). For the extension, specify ".mcu". Write each address in hexadecimal. Do not add the prefix that represents the radix.
Please describe information on 3-6 referring to the data book on MCU used.

1. MCU name
2. Reserved number
3. CPU mode register address and stack page select bit number
4. Reset Vector address
5. BRK vector address
6. Interrupt vector address information

- MCU name and Reserved number
  Always be sure to add a semicolon (;)

- CPU mode register address and stack page select bit number
  Separate the CPU mode register address and the stack page select bit number with a colon (:).

- Reset vector address information
  Add the word ":RST" after the reset vector address.

- BRK vector address information
  Add the word ":BRK" after the BRK vector address.

- Interrupt vector address information
  Separate between the interrupt vector address and interrupt control register address, and between the interrupt control register address and interrupt control bit number with a colon (:). Interrupt vector information can be written for up to 32 points.

Example

The following shows an example (m38000.mcu).

```
;M38000
;1
3B:2
FFFC:RST
FFDC:BRK
FFFA:3E:0
FFF8:3E:1
FFF6:3E:2
FFF4:3E:3
FFF2:3E:4
FFF0:3E:5
FFEE:3E:6
FFEC:3E:7
FFEA:3F:0
FFE8:3F:1
FFE6:3F:2
FFE4:3F:3
FFE2:3F:4
FFE0:3F:5
```

Chapter 7

Tips for Coding

Paying only a little attention during coding in C language is in many cases conducive to creating a program with better code efficiency. This chapter provides tips for the increased efficiency of coding.

1) Use optimization options

Although not a tip for coding, specifying an optimization option helps to improve on code size or execution speed.
There are two types of optimization options, one designed to reduce the code size, and one aiming to increase the execution speed.

–z1 to –z9: Size priority optimization
–s1 to –s9: Speed priority optimization

Value	Level
0	Not optimized
1–3	Fully debuggable
4–6	Some structures undebuggable
7–9	Fully optimized

The –s option cannot be used simultaneously with the –z option.

2) Use the smallest integer type possible

To define variables, try using the smallest integer type possible according to the purpose of use. This will lead to generation of codes efficient in both code size and execution speed.

ex

```
char ch;
short si;
long li;

void main( void )
{
   ch--;
   si--;
   li--;
}
```

```
 7          ch--;
\ 000000 C6..     DEC   zp:ch
 8          si--;
\ 000002 C6..     DEC   zp:si
\ 000004 A5..     LDA   zp:si
\ 000006 3A       INC   A
\ 000007 D002     BNE   ?0000
\ 000009 C6..     DEC   zp:si+1
\                 ?0000:
 9          li--;
\ 00000B 32       SET
\ 00000C CA       DEX
\ 00000D A5..     LDA   zp:li+3
\ 00000F CA       DEX
\ 000010 A5..     LDA   zp:li+2
\ 000012 CA       DEX
\ 000013 A5..     LDA   zp:li+1
\ 000015 CA       DEX
\ 000016 A5..     LDA   zp:li
\ 000018 12       CLT
\ 000019 32       SET
\ 00001A CA       DEX
\ 00001B A9FF     LDA   #255
\ 00001D CA       DEX
\ 00001E A9FF     LDA   #255
\ 000020 CA       DEX
\ 000021 A9FF     LDA   #255
\ 000023 CA       DEX
\ 000024 A9FF     LDA   #255
\ 000026 12       CLT
\ 000027 20.... JSR   np:?L_ADD_L03
\ 00002A B500     LDA   zp:0,X
\ 00002C 85..     STA   zp:li
\ 00002E B501     LDA   zp:1,X
\ 000030 85..     STA   zp:li+1
\ 000032 B502     LDA   zp:2,X
\ 000034 85..     STA   zp:li+2
\ 000036 B503     LDA   zp:3,X
\ 000038 85..     STA   zp:li+3
\ 00003A E8       INX
\ 00003B E8       INX
\ 00003C E8       INX
\ 00003D E8       INX
```

30bytes

3) Use variables of *unsigned int* type

The 740 family is characteristic in that better ROM efficiency is obtained by using variables of *unsigned int* type rather than using variables of *signed int* type. The ROM efficiency is increased especially when type conversion, comparison, array indexing, shift or division are performed.

ex

```
int i;

    if (i<5) i=0;
```

```
unsigned int ui;

    if (ui<5) ui=0;
```

```
\ 000010   32         SET
\ 000011   CA         DEX
\ 000012   AD.... LDA   np:i+1
\ 000015   CA         DEX
\ 000016   AD.... LDA   np:i
\ 000019   CA         DEX
\ 00001A   A900     LDA     #0
\ 00001C   CA         DEX
\ 00001D   A905     LDA   #5
\ 00001F   12         CLT
\ 000020   20....   JSR np:?SS_CMP_L02
\ 000023   B008      BCS   ?0004
\ 000025   A000      LDY   #0
\ 000027   8C....   STY   np:I
\ 00002A   8C....   STY   np:i+1
```

```
\ 00002E   38         SEC
\ 00002F   AD....   LDA   np:ui
\ 000032   E905      SBC   #5
\ 000034   AD....   LDA   np:ui+1
\ 000037   E900      SBC   #0
\ 000039   B008      BCS   ?0006
\ 00003B   A000      LDY   #0
\ 00003D   8C....   STY   np:ui
\ 000040   8C....   STY   np:ui+1
```

4) Use bit fields for bit processing

For bit determination or on/off, better code efficiency is obtained by using bit fields rather than using AND or OR.

ex

```
typedef union {
   unsigned char byte;
   struct {
      unsigned char b0:1;
      unsigned char b1:1;
      unsigned char b2:1;
      unsigned char b3:1;
      unsigned char b4:1;
      unsigned char b5:1;
      unsigned char b6:1;
      unsigned char b7:1;
   } bitf;
} BYTE_BIT;
unsigned char uc;
BYTE_BIT data;
if ( (uc & 0x04) == 0 ) {
     uc |= 0x04;
}
if ( data.bitf.b2 == 0 ) {
     data.bitf.b2 = 1;
}
```

```
000063   A904       LDA   #4
000065   2D.... AND   np:uc
000068   1A          DEC   A
000069   D007       BNE   ?0005
00006B   AD.... LDA   np:uc
00006E   4B          SEB   2,A
00006F   8D.... STA   np:uc
                    ?0005:
```

```
000072   AD....   LDA   np:data
000075   4304      BBS   2,A,?0007
000077   4B         SEB   2,A
000078   8D.... STA   np:data
                   ?0007:
```

5) For *switch* statements, pay attention to the type of condition determination expression and the number of *case* labels

The *switch* statement uses a jump table-based C runtime library according to the type of condition determination expression and the number of *case* labels.
The ROM efficiency can be increased by using a small-size C runtime library.

ex

```
switch( type ) {
case 1:
        ...
        ...
case xx:
}
```

Type	Number of *case* labels	C runtime library	Library size
signed char	**4**	**–**	**small**
	5	**?C_S_SWITCH_L06**	
unsigned char	**4**	**–**	
	5	**?C_S_SWITCH_L06**	
signed short	**1**	**?S_V_SWITCH_L06**	
unsigned short	**1**	**?S_V_SWITCH_L06**	
signed int	**1**	**?S_V_SWITCH_L06**	
unsigned int	**1**	**?S_V_SWITCH_L06**	
signed long	**1**	**?L_V_SWITCH_L06**	**big**
unsigned long	**1**	**?L_V_SWITCH_L06**	

6) Declare the function prototype

If any function is called without making a prototype declaration, the compiler assumes that the called function is one that returns a value of *int* type.
This will result in unnecessary codes being generated. Therefore, always be sure to write a function prototype declaration before calling the function.

ex

```
void xxx( void )
{
    func1();
```

```
void func1( void );
void xxx( void )
{
    func1();
```

```
Warning[52]:

740 specific: 'No prototype for function "func1",

assuming that it returns int'
```

```
\ 000000   20....   JSR   np:REFFN func1
\ 000003   E8          INX
\ 000004   E8          INX
```

```
\ 000000   20....   JSR   np:REFFN func1
```

7) Use an explicit cast

Use an explicit cast.

ex

```
char c1; int i1,i2;
c1 = i1 + i2;
```

→

```
char c1; int i1,i2;
c1 = (char)i1 + (char)i2;
```

8) Write expressions simply

Rather than writing one complex expression, divide it into two or more simple expressions.

ex

```
((6+ch1)*ch2)*ch1;
```

→

```
tmp = 6+ch1;
tmp = tmp*ch2;      /* char tmp; */
return tmp*ch1;
```

9) Avoid a complicated use of suffix operators

Use of suffix operators requires full knowledge about the priority and associativity of operators. An inconsiderate use of suffix operators may produce unexpected results.

ex

```
char *src ;
char *dest ;
while( *src ){
      *dest++ = *src++ ;
}
```

→

```
char *src ;
char *dest ;
while( *src ){
      *dest = *src ;
       dest++ ;
       src++ ;
}
```

Chapter 8

Estimating the Stack Size

This chapter explains how to estimate the necessary stack size.

8.1 Default Stack Size

In the C compiler package for the 740, when a project is created the respective stack sizes are set by default, as shown below.

EXPR_STACK segment	32 bytes
INT_EXPR_STACK segment	32 bytes (for Tiny model, 16 bytes)
CSTACK segment	64 bytes
C_ARGN segment	Set by the linker
C_ARGZ segment	Set by the linker
RF_STACK segment	Set by the linker

To change the default stack sizes, estimate the necessary size of each stack as described below.

8.2 EXPR_STACK and INT_EXPR_STACK Segments

Estimation method: Use the size (a) or (b) whichever is larger.

(a) Amount of stack used by runtime functions
  (a-1) Simple estimate
  - 4 bytes for arithmetic operations of *short* and *int* types
  - 8 bytes for arithmetic operations of *long* type
  - 16 bytes for arithmetic operations (additions) of *float* and *double* types

  (a-2) Detail estimate
  - Maximum value of the runtime functions used

(b) Maximum size out of the return value sizes of user-defined functions

*** If return values are used directly in arithmetic operations without being assigned to variables, add the sizes of return values.**

8.3 CSTACK Segment

Estimation method: Add up the sizes (a) through (f).

(a) Maximum number of nested levels of function 2 bytes (return address)
(b) Maximum number of nested levels of interrupt function 2 bytes (return address)
(c) Amount of stack used during interrupt (2 bytes for return address + 4 bytes for register)
(d) Amount of stack used when MUL/DIV instructions are executed
(e) Amount of stack manipulated in an assembly language statement
(f) Amount of CSTACK used by runtime functions

*** The nested structures of functions can be confirmed by specifying the option -xmso to the linker.**

8.4 C_ARGN and C_ARGZ Segments

Estimation method: No stack sizes can be set because the linker sets the stack size.

8.5 RF_STACK Segment

Estimation method: If an area larger than 256 bytes set by the linker needs to be set, set the desired additional size.

```
-Z(NPAGE) RF_STACK+10
```

8.6 Amount of Stack Used by ICC740 Runtime Functions

RunTime	Function	EXPR_STACK	CSTACK	Call
8-bit integer				
?C_ADD_L01	add	2	2	
?C_SUB_L01	sub	2	2	
?C_MUL_L01	multiplication	2	3	
?C_FIND_SIGN_L01	character sign finding	2	3	
?C_DIVMOD_L01	character division and modulo	3	2	
?SC_DIV_L01	signed character division	3	5	?C_DIVMOD_L01 ?C_FIND_SIGN_L01
?UC_DIV_L01	unsigned division	2	3	
?SC_MOD_L01	signed character modulo	3	6	?C_DIVMOD_L01 ?C_FIND_SIGN_L01
?UC_MOD_L01	unsigned modulo	2	4	
?C_SHL_L01	left_shift_A_Y_steps	0	2	
?SC_SHR_L01	right_signed_shift_A_Y_steps	0	2	
?UC_SHR_L01	right_unsigned_shift_A_Y_steps	0	2	
?UC_CMP_L01	unsigned_compare	2	2	
?SC_CMP_L01	signed_compare	2	2	
16-bit integer				
?S_ADD_L02	add	4	2	
?S_AND_L02	and	4	2	
?S_ORA_L02	or	4	2	
?S_EOR_L02	xor	4	2	
?S_SUB_L02	sub	4	2	
?S_MUL_L02	multiplication	6	3	
?S_FIND_SIGN_L02	character sign finding	4	3	
?US_DIV_L02	unsigned division	7	2	
?S_DIVMOD_L02	character division and modulo	7	2	
?SS_DIV_L02	signed word division	7	5	?S_DIVMOD_L02 ?S_FIND_SIGN_L02
?SS_MOD_L02	signed word modulo	7	6	?S_DIVMOD_L02 ?S_FIND_SIGN_L02
?US_MOD_L02	unsigned modulo	7	4	?S_DIVMOD_L02
?S_SHL_L02	left_shift_NOS_TOS:8_steps	3	2	
?SS_SHR_L02	right_signed_shift_NOS_TOS:8_steps	3	2	(?US_SHR_L02)
?US_SHR_L02	right_unsigned_shift_NOS_TOS:8_steps	3	2	
?SS_CMP_L02	signed_compare	4	2	
?US_CMP_L02	unsigned_compare	4	2	
?US_ZERO_L02	is_zero	2	2	
32-bit integer				
?L_ADD_L03	add	8	2	
?L_AND_L03	and	8	2	
?L_OR_L03	or	8	2	
?L_XOR_L03	xor	8	2	
?L_SUB_L03	sub	8	2	
?L_MUL_L03	multiplication	12	3	
?L_FIND_SIGN_L03	character sign finding	8	3	
?UL_DIV_L03	unsigned division	13	2	
?L_DIVMOD_L03	long division and modulo	13	2	

?SL_DIV_L03	signed long division	13	5	?L_DIVMOD_L03 ?L_FIND_SIGN_L03 ?L_NOT_L03 ?L_INC_L03
?SL_MOD_L03	signed long modulo	13	6	?L_DIVMOD_L03 ?L_FIND_SIGN_L03 ?L_NOT_L03 ?L_INC_L03
?UL_MOD_L03	unsigned modulo	13	4	?L_DIVMOD_L03
?L_SHL_L03	left_shift_NOS_TOS:8_steps	5	2	
?SL_SHR_L03	right_signed_shift_NOS_TOS:8_steps	5	2	(?UL_SHR_L03)
?UL_SHR_L03	right_unsigned_shift_NOS_TOS:8_steps	5	2	
?SL_CMP_L03	signed_compare	8	2	
?UL_CMP_L03	unsigned_compare	8	2	
?L_ZERO_L03	is_zero	4	2	
?L_TEST_L03	test	8	4	?L_SUB_L03, (?L_ZERO_L03)
?L_NOT_L03	not	4	2	
?L_INC_L03	increment	4	2	
32-bit floating-point				
?F_MUL_L04	Floating point Multiplication	12	4	?F_UNPACK_L04 (?F_PACK_2_L04) (?F_ROUND_L04) (?F_OVERFLOW_TEST_L04) (?F_OVERFLOW_TEST1_L04) (?F_UNDERFLOW_L04) (?F_EXIT_L04)
?F_DIV_L04	Floating point division	12	4	?F_UNPACK_L04 (?F_PACK_L04) (?F_PACK_2_L04) (?F_UNDERFLOW_L04) (?F_OVERFLOW_TEST1_L04) (?F_UP_ROUND_L04)
?F_ADD_L04	Floating point addition	12	4	?F_UNPACK_L04 (?F_PACK_2_L04) (?F_ROUND_L04) (?F_EXIT_L04) (?F_UNDERFLOW_L04) (?F_OVERFLOW_L04)
?F_SUB_L04	Floating point subtraction	12	4	?F_UNPACK_L04 (?F_PACK_2_L04) (?F_ROUND_L04) (?F_EXIT_L04) (?F_UNDERFLOW_L04) (?F_OVERFLOW_L04)
?SL_TO_F_L04	Cast a signed long integer	4	5	?F_0_SUB_L04
?UL_TO_F_L04	Cast a unsigned long integer	4	3	
?F_TO_L_L04	Cast a floating point	4	3	(?F_0_SUB_L04)
?F_CMP_L04	Float compare	8	2	
?F_UNPACK_L04	Internal entry	0	0	
?F_ROUND_L04	Internal entry	0	0	(?F_PACK_L04) (?F_UP_ROUND_L04)
?F_UP_ROUND_L04	Internal entry	0	0	(?F_PACK_L04)

?F_PACK_L04	Internal entry	0	0	(?F_UNDERFLOW_L04) (?F_OVERFLOW_TEST_L04) (?F_EXIT_L04)
?F_PACK_2_L04	Internal entry	0	0	?F_PACK_L04)
?F_UNDERFLOW_L04	Internal entry	0	0	(?F_EXIT_L04)
?F_OVERFLOW_L04	Internal entry	0	0	(?F_EXIT_L04)
?F_OVERFLOW_TEST_L04	Internal entry	0	0	(?F_EXIT_L04)
?F_OVERFLOW_TEST1_L04	Internal entry	0	0	(?F_EXIT_L04)
?F_NEG_OVERFLOW_L04	Internal entry	0	0	(?F_EXIT_L04)
?F_0_SUB_L04	Internal entry	0	0	
?F_EXIT_L04	Internal entry	0	0	
switch				
?C_S_SWITCH_L06	switch (char):series	1	2	
?S_S_SWITCH_L06	switch (short):series	2	2	
?L_S_SWITCH_L06	switch (long):series	6	2	
?C_V_SWITCH_L06	switch (char)	2	2	
?S_V_SWITCH_L06	switch (short)	4	2	
?L_V_SWITCH_L06	switch (long)	6	2	
Function enter/leave				
?ENTER_L08	Function enter, save DP0 and DP1	0	4	
?LEAVE_L08	Function leave, restore DP0 and DP1	0	0	
?ENTER_FP_L08	Function enter, save DP0, DP1 and FP	0	6	
?LEAVE_FP_L08	Function leave, restore DP0, DP1 and FP	0	0	
Stack				
?IND_STK_16_DP0_L09	(tos) -> tos	2	3	
?IND_STK_16_DP1_L09	(tos) -> tos	2	3	
?IND_STK_16_L09	(tos) -> tos	2	5	
?IND_DP0_DP1_L09	(dp0) -> dp1	0	3	
?IND_DP1_DP0_L09	(dp1) -> dp0	0	3	
?STK_TO_DP0_L09	TOS -> dp0	2	3	
?STK_TO_DP1_L09	TOS -> dp1	2	3	
?DP0_TO_STK_L09	DP0 -> TOS	0	2	
?DP1_TO_STK_L09	DP1 -> TOS	0	2	
?IND_DP0_L09	(dp0) -> dp1	0	4	
?IND_DP1_L09	(dp1) -> dp0	0	4	
?PUSH_A_16_L09	0 A -> TOS	0	3	
?MOVE_LONG_L09	Block move (dp1) -> (dp0)	2	3	
?PUSH_DP0_L09	dp0 -> cpu stack	0	4	
?POP_DP0_L09	cpu stack --> dp0	0	0	
?PUSH_DP1_L09	dp1 -> cpu stack	0	4	
?POP_DP1_L09	cpu stack --> dp1	0	0	

For the functions such as four rules of arithmetic that have a return value, the return value is set in the area of EXPR_STACK.
For ( ) of Call, the *jmp* instruction is used.
If a call to any lower-level function is involved, the stack size of the lower-level function is included.
For CSTACK, the return address size of lower-level function 1 is included.
0 for ?POP_DP0_L09 includes a subtracted value (–2) from 2 for ?PUSH_DP0_L09. The same applies to ?POP_DP1_L09 too.
The Internal entry value under the heading “32-bit floating-point” is included in the upper-level function, so that the stack size is 0.
?LEAVE_L08 is a pair function of ?ENTER_L08. LEAVE_FP_L08 is a pair function of ?ENTER_FP_L08.

Chapter 9
Interrupt Handling

This chapter describes interrupt handling of ICC740.

9.1 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 the interrupt vector area
(4) Setting up the interrupt vector segment

9.1.1 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      "sfr_3803h.h"          /* SFR header file for the 3803H group */
unsigned char     counter ;

interrupt [30] void INT0_TimerZ( void )     /* Interrupt handling function */
{
      cld_instruction() ;          /* CLD instruction to initialize decimal mode flag */
      counter = 0 ;
}

interrupt [8] 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 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 */

      /* (2) 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 */

      /* (3) 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 "sfr_3803h.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.

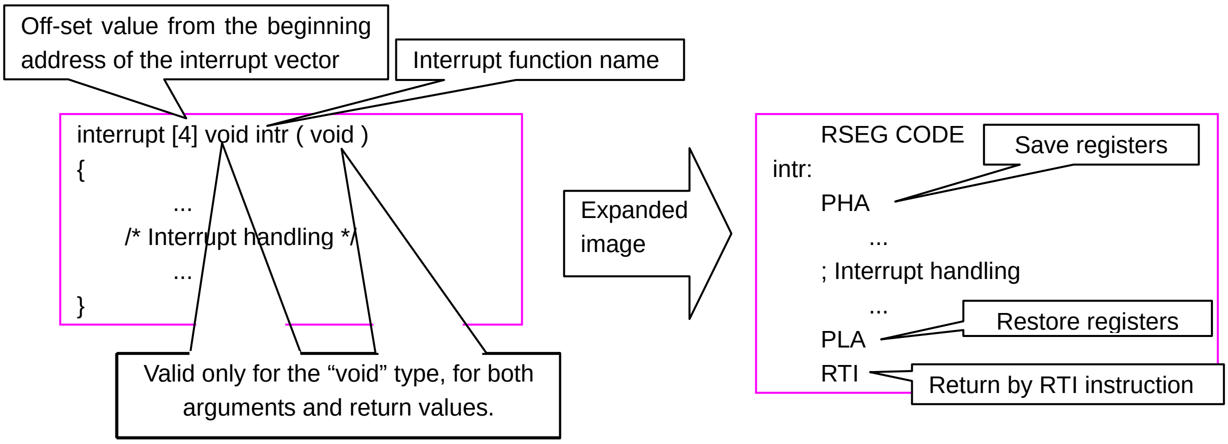
Processor Status Register
(Automatically saved to the stack during interrupt)
- D and T flags
  Initialized by init_C in cstartup.s31.
- I flag
  Set to 1 (disabled) immediately after the microcomputer is reset.

9.1.2 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.

By specifying the off-set value with a bracket from the starting address of the interrupt vector immediately after “interrupt”, an address of the interrupt handling routine is inserted into the vector and this function is called when an interrupt is generated. When the off-set value is not specified, the vector table of the interrupt function is defined in the cstartup.s31 file (declare the interrupt handling function by the simulated instruction “EXTERN” as an external reference and register it to the interrupt vector).

* 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.

Notes on Interrupt

Do not call the function which is normally called in the interrupt function.

<Reason>
Since dynamic variable is located statically, the automatic variable of the function is rewritten when an interrupt is generated while the function is normally called.

To call the function which is normally called in the interrupt function, provide two same functions for normal and interrupts.

The indirect calling function has the same limitations as above.

Column　How to jump to init_C (startup) from interrupt handling function

The unfilled functions which do not describe sentences are left for the unused interrupt functions as follows.

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

Only the RTI instruction is generated by describing as above. Therefore, the RTI instruction is executed and program continues when an unexpected interrupt is generated.

When the unexpected interrupt is generated as another description besides the above-mentioned, describe as follows to execute from init_C as at reset:

c source file

```
extern void init_C( void ) ;
    .
    .
    .
interrupt [28] void Int1( void )
{
    __asm( "JMP init_C" ) ;
}
```

Add one line to declare init_C (assembly language function) as extern

Interrupt function which is not normally called

Describe an instruction to jump to init_C using asm function

startup.s31 file

```
DEFFN init_C( 0, 0, 0, 0, 0x8000, 0, 0, 0 )
PUBLIC init_C
```

Add these two lines to the point before init_C (beginning of the file)

9.1.3 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) by the CLI instruction. 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
?0003:
      BRA      ?0003
      SEI
      RTS
```

enable_interrupt()

disable_interrupt()

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.

9.1.4 Registering the Interrupt Vector Area

Registering interrupt vector area

Alter the content of the INTVEC segment in the cstartup.s31 included with the ICC740 according to the interrupt area of the microcomputer used.

```
      COMMON    INTVEC

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

Assign the size of the interrupt vector area with the BLKB simulated instruction. At this time, since reset is not an interrupt to be generated on purpose, specify the size in which two bytes are subtracted.

The reset vector is set at the bottom. At reset, the program starts from init_C that is registered in the reset vector.

9.1.5 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."

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

Beginning and ending addresses of the interrupt vector area

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

9.2 Multiple Interrupts

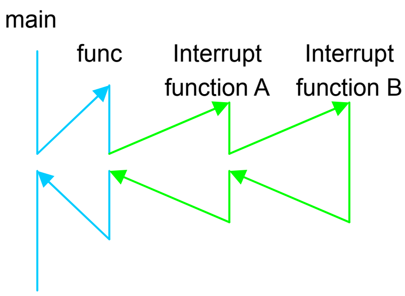
9.2.1 How to Use Multiple Interrupts

This section describes necessary setting, notes, and how to describe functions to use multiple interrupts.

Notes to use multiple interrupts

Program the interrupt function B with the following points in the program such as the figure below
- Simple handling such as the flag setting or counter update
- Do not call C run time library.
- Do not use a function as a multiple interrupt function when it has a local variable used for an interrupt function which enables multiple interrupts.

<Reason>
To call the C run time library and functions but the char type return value, an interrupt expression stack is used.
The interrupt expression stack is an area to hold the result temporarily while the expression is evaluated by the interrupt handling.
This area is commonly used for all interrupts. Therefore, they are used for the interrupt function A will be overwritten by using the interrupt expression stack for the interrupt function B.



* The C run time library is to be automatically called according to the need of a compiler, but it cannot be expressly called on a program. The C run time library can be called by multiple calculations such as signed char type division.

Compile option

To use multiple interrupts, it is necessary to add -h to the compile option of the ICC740.

Column Application example to use interrupt expression stack by multiple interrupts

There are the following ways to use the interrupt expression stack for the interrupt function B in the above diagram.
- An interrupt is disabled while the interrupt expression stack is used for the interrupt function A. However, real time program will be lost.
- Describe the calculation parts of the interrupt function B in the assembly language and try not to use the interrupt expression stack.
- If the interrupt expression stack has not been used for the interrupt function A, there is no problem to use the interrupt expression stack for the interrupt function B.
  (The content of the interrupt expression stack will not be overwritten if either the interrupt function A or B is used.)

9.2.2 Definition on multiple interrupts

Setting is performed to use multiple interrupts in the following cstartup.s31.

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

To use (multiple interrupts), do not leave ';' (semicolon).

"Use" is selected for the default. To use multiple interrupts, leave this line. When multiple interrupts are not used, delete ; at the beginning of the last line and use as not a comment line but a valid line.

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

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

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

The settings are performed for multiple interrupts at these 2 points. To use multiple interrupts (NO_INTERRUPTABLE_ISR is not defined), this point is assembled. When multiple interrupts are not used (NO_INTERRUPTABLE_ISR is defined), this point is not assembled.

9.2.3 Descriptions of Multiple Interrupt Handling Functions

Program descriptions with the following specifications are listed below based on the notes described above.

Function	Content	Interrupt Priority	State
INT1 interrupt handling function	unusual handling (emergency halt)	1	Multiple interrupts disabled
Serial I/O1 receive interrupt handling function	communication received	2	Multiple interrupts for only one interrupt enabled
Timer 2 interrupt handling function	general handling	3	Multiple interrupts enabled

For (3) of the main function, set the enable bit of the interrupts to be generated to enable. For the interrupt handling function of timer 2, all interrupts are enabled by executing the CLI instruction at the front. This allows multiple interrupts. However, multiple interrupts to the executed interrupt are disabled. For the interrupt handling function of the serial I/O 1 reception, setting the interrupts other than INT1 to disable and executing the CLI instruction allows multiple interrupts for only the INT1 interrupt. Multiple interrupts are not generated while the functions are executed and the interrupts are disabled for the INT1 interrupt handling function.

Also, the interrupts are disabled (I flag = 1) immediately after reset. The interrupts are disabled (I flag = 1) to enter the interrupt function, and they are enabled (I flag = 0) to exit the interrupt function (by the RTI instruction).

```
#include    <intr740.h>      /* Header file for inline function */
#include    "sfr_3803h.h"    /* SFR header file for the 3803H Group */

void main( void )
{
        .
        .
        .
    /* (1) Set the interrupt edge select bit (and interrupt source bit) */
    INTEDGE.1 = 1 ;             /* INT1 rising edge active */

    /* (2) Set the appropriate interrupt request bit to 0 (no request) after waiting more than one
instruction. */
    nop_instruction() ;         /* Wait one instruction */
    IREQ1.1 = 0 ;               /* INT1 interrupt request bit → clear */
    IREQ1.2 = 0 ;               /* Serial I/O1 receive interrupt request bit → clear */
    IREQ1.7 = 0 ;               /* Timer 2 interrupt request bit → clear */

    /* (3) Set the appropriate interrupt enable bit to 1(enabled)    */
    ICON1.1 = 1 ;               /* INT1 interrupt enable bit → enabled */
    ICON1.2 = 1 ;               /* Serial I/O1 receive interrupt enable bit → enabled */
    ICON1.7 = 1 ;               /* Timer 2 interrupt enable bit → enabled */

    enable_interrupt() ;        /* Interrupt enabled CLI instruction */

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

set the enable bit of the interrupts to be generated to enable.

```
#include      <intr740.h>      /* Header file for inline function */
#include      "sfr_3803h.h"    /* SFR header file for the 3803H Group */

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

interrupt [28] void Int1( void )          /* INT1 interrupt handling function [emergency halt] */
{
      Cntr = 0 ;
}
```

- Multiple interrupts disabled
- Interrupt expression stack cannot be used (refer to notes)

```
interrupt [26] void SIO1R( void )         /* Serial I/O1 receive interrupt handling function */
{
      ICON1.2 = 0 ;              /* Serial I/O1 receive interrupt enable bit → disabled */
      ICON1.7 = 0 ;              /* Timer 2 interrupt enable bit → disabled */
      enable_interrupt() ;       /* Interrupt enabled CLI instruction */
             ¦
      T_flg = 1 ;
             ¦
      disable_interrupt() ;      /* Interrupt disabled SEI instruction */
      ICON1.2 = 1 ;              /* Serial I/O1 receive interrupt enable bit → enabled */
      ICON1.7 = 1 ;              /* Timer 2 interrupt enable bit → enabled */
}
```

- Multiple interrupts enabled for only emergency halt
- Interrupt expression stack cannot be used (refer to notes)

```
interrupt [16] void Timer2( void )        /* Timer 2 interrupt handling function */
{
      ICON1.7 = 0 ;              /* Timer 2 interrupt enable bit → disabled */
      enable_interrupt() ;       /* Interrupt enabled CLI instruction */
```

- Multiple interrupts enabled
- Interrupt expression stack can be used

Interrupt to timer 2 disabled

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

      disable_interrupt() ;      /* Interrupt disabled SEI instruction */
      ICON1.7 = 1 ;              /* Timer 2 interrupt enable bit → enabled */
}
```

Interrupt to timer 2 enabled

C Compiler Package for 740 Family M3T-ICC740 Application Notes

Publication Date: Sep. 16, 2006 Rev.1.00

Published by: Sales Strategic Planning Div.
Renesas Technology Corp.

Edited by: Microcomputer Tool Development Department
Renesas Solutions Corp.

C Compiler Package for 740 Family
Application Notes

Renesas Electronics Corporation
1753, Shimonumabe, Nakahara-ku, Kawasaki-shi, Kanagawa 211-8668 Japan

REJ06J0005-0100