

To our customers,

---

## Old Company Name in Catalogs and Other Documents

---

On April 1<sup>st</sup>, 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 1<sup>st</sup>, 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.

---

# M16C/80 & M32C/80 Series

## Programming Guidelines <C Language>

---

### Preface

This application note is written for the Renesas M16C/80 and M32C/80 series 16-bit microcomputers. It explains the basics of C language programming and how to put your program into ROM using the NC308 C compiler.

For details about hardware and development support tools available for each type of microcomputer in the M16C/80 and M32C/80 series, please refer to the appropriate hardware manuals, user's manuals and instruction manuals.

### Guide to Using This Application Note

This application note provides programming guidelines for NC308, the C compiler for the M16C/80 and M32C/80 series. Knowledge of the M16C/80 and M32C/80 series microcomputer architectures and the assembly language is helpful in using this manual. The manual contains the following:

- Chapter 1: Introduction to C language
- Chapter 2: Extended Functions of NC308
- Appendix A: Functional Comparison between NC30 and NC308
- Appendix B: NC308 Command Reference
- Appendix C: Questions & Answers

# Table of Contents

<b>Chapter 1 Introduction to C Language .....</b>	<b>5</b>
<b>1.1 Programming in C Language .....</b>	<b>6</b>
1.1.1 Assembly Language and C Language .....	6
1.1.2 Program Development Procedure .....	7
1.1.3 Program Rules and Practices .....	9
<b>1.2 Data Types .....</b>	<b>13</b>
1.2.1 "Constants" in C Language .....	13
1.2.2 Variables .....	15
1.2.3 Data Characteristics .....	17
<b>1.3 Operators .....</b>	<b>19</b>
1.3.1 Operators of NC308 .....	19
1.3.2 Operators for Numeric Calculations .....	20
1.3.3 Operators for Processing Data .....	23
1.3.4 Operators for Examining Condition .....	25
1.3.5 Other Operators .....	26
1.3.6 Priorities of Operators .....	28
1.3.7 Examples for Easily Mistaken Use of Operators .....	29
<b>1.4 Control Statements .....</b>	<b>31</b>
1.4.1 Structuring of Program .....	31
1.4.2 Branching Processing Depending on Condition (Branch Processing) .....	32
1.4.3 Repetition of Same Processing (Repeat Processing) .....	36
1.4.4 Suspending Processing .....	39
<b>1.5 Functions .....</b>	<b>41</b>
1.5.1 Functions and Subroutines .....	41
1.5.2 Creating Functions .....	42
1.5.3 Exchanging Data between Functions .....	44
<b>1.6 Storage Classes .....</b>	<b>45</b>
1.6.1 Effective Range of Variables and Functions .....	45
1.6.2 Storage Classes of Variables .....	46
1.6.3 Storage Classes of Functions .....	48
<b>1.7 Arrays and Pointers .....</b>	<b>51</b>
1.7.1 Arrays .....	51
1.7.2 Creating an Array .....	52
1.7.3 Pointers .....	54
1.7.4 Using Pointers .....	56
1.7.5 Placing Pointers into an Array .....	58
1.7.6 Table Jump Using Function Pointer .....	60

<b>1.8 Struct and Union .....</b>	<b>62</b>
1.8.1 Struct and Union .....	62
1.8.2 Creating New Data Types .....	63
<b>1.9 Preprocess Commands .....</b>	<b>67</b>
1.9.1 Preprocess Commands of NC308 .....	67
1.9.2 Including a File .....	68
1.9.3 Macro Definition .....	69
1.9.4 Conditional Compile .....	71
<b>Chapter 2 Extended Functions of NC308 .....</b>	<b>73</b>
<b>2.1 Memory Mapping .....</b>	<b>74</b>
2.1.1 Types of Code and Data .....	74
2.1.2 Sections Managed by NC308 .....	75
2.1.3 Control of Memory Mapping .....	77
2.1.4 Controlling Memory Mapping of Struct .....	80
<b>2.2 Startup Program .....</b>	<b>82</b>
2.2.1 Roles of Startup Program .....	82
2.2.2 Estimating Stack Sizes Used .....	84
2.2.3 Creating Startup Program .....	87
<b>2.3 Extended Functions for ROM'ing Purposes .....</b>	<b>94</b>
2.3.1 Efficient Addressing .....	94
2.3.2 Handling of Bits .....	99
2.3.3 Control of I/O Interface .....	100
2.3.4 Using Inline Assembly .....	102
2.3.5 Using Assembler Macro Functions .....	104
<b>2.4 Linkage with Assembly Language .....</b>	<b>109</b>
2.4.1 Interface between Functions .....	109
2.4.2 Calling Assembly Language from C Language .....	115
2.4.3 Calling C Language from Assembly Language .....	123
<b>2.5 Interrupt Handling .....</b>	<b>124</b>
2.5.1 Writing Interrupt Handling Functions .....	124
2.5.2 Writing High-speed Interrupt Handling Functions .....	127
2.5.3 Writing Software Interrupt (INT Instruction) Handling Functions .....	129
2.5.4 Registering Interrupt Processing Functions .....	131
2.5.5 Example for Writing Interrupt Processing Function .....	132

<b>Appendices .....</b>	<b>134</b>
<b>Appendix A. Functional Comparison between NC308 and NC30 .....</b>	<b>135</b>
<b>Appendix B. NC308 Command Reference .....</b>	<b>139</b>
<b>Appendix C. Questions &amp; Answers .....</b>	<b>145</b>

---

# Chapter 1

---

## Introduction to C Language

- 1.1 Programming in C Language
- 1.2 Data Types
- 1.3 Operators
- 1.4 Control Statements
- 1.5 Functions
- 1.6 Storage Classes
- 1.7 Arrays and Pointers
- 1.8 Struct and Union
- 1.9 Preprocess Commands

This chapter provides an introduction to the C language for first time users and a reference for more experienced programmers.

## 1.1 Programming in C Language

### 1.1.1 Assembly Language and C Language

As the scale of microcomputer based systems has increased over the years, productivity and maintainability using Assembly language has become an issue. As a result, C language has become a popular alternative.

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 functions have their parameters highly independent of others, a program can easily be made into parts and can easily be reused. Furthermore, modules written in the assembly language can be incorporated into a C language program directly without modification.
- (3) An easily maintainable program can be written.  
For reasons (1) and (2) above, the program after being put into operation can easily be maintained. Furthermore, since the C language is based on standard specifications (ANSI standard<sup>(Note)</sup>), a program written in the C language can be ported into other types of microcomputers after only a minor modification of the source program.

#### Comparison between C and Assembly Languages

Table 1.1.1 outlines the differences between the C and assembly languages with respect to the method for writing a source program.

**Table 1.1.1 Comparison between C and Assembly Languages**

	C language	Assembly language
Basic unit of program (Method of description)	Function (Function name ( ) { })	Subroutine (Subroutine name:)
Format	Free format	1 instruction/line
Discrimination between uppercase and lowercase	Uppercase and lowercase are discriminated (Normally written in lowercase)	Not discriminated
Allocation of data area	Specified by "data type"	Specified by a number of bytes (using pseudo-instruction)
Input/output instruction	No input/output instructions available	Input/output instructions available (However, it depends on hardware and software.)

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.



### 1.1.2 Program Development Procedure

The operation of translating a source program written in "C" into machine language is referred to as "compiling". The software provided for performing this operation is called a "compiler". This section explains the procedure for developing a program by using NC308, the C compiler for the M16C/80 and M32C/80 series of Renesas single-chip microcomputers.

#### NC308 Product List

Figure 1.1.1 lists the products included in NC308, the C compiler for the M16C/80 and M32C/80 series.

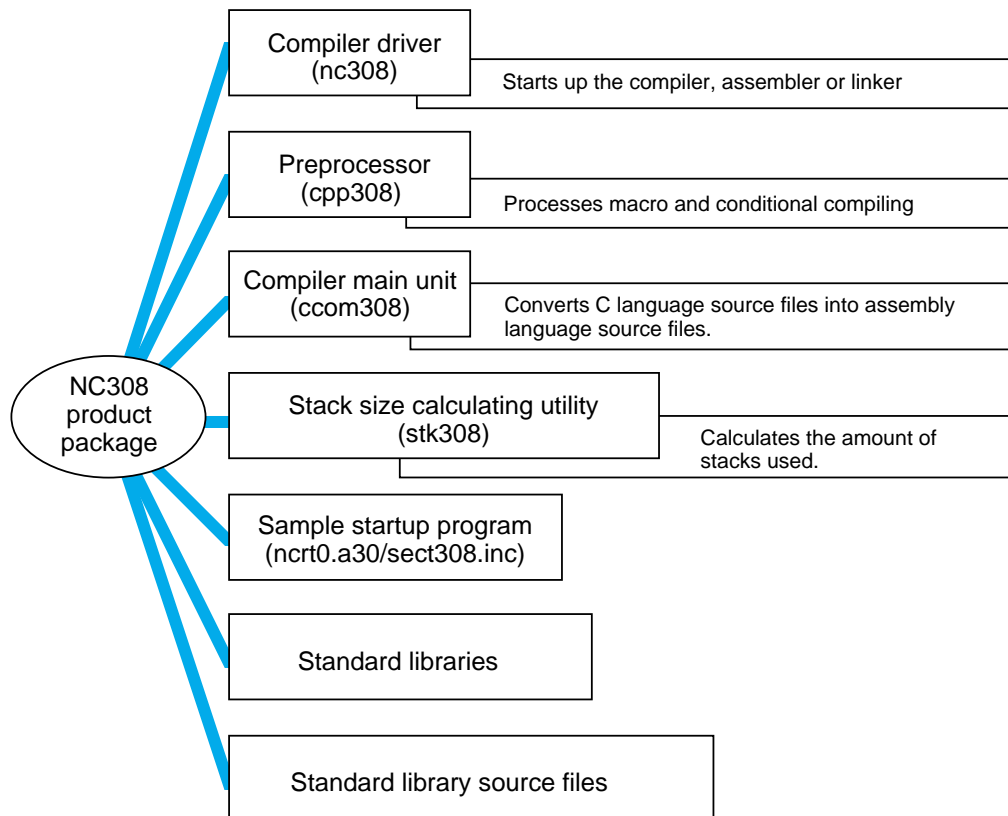
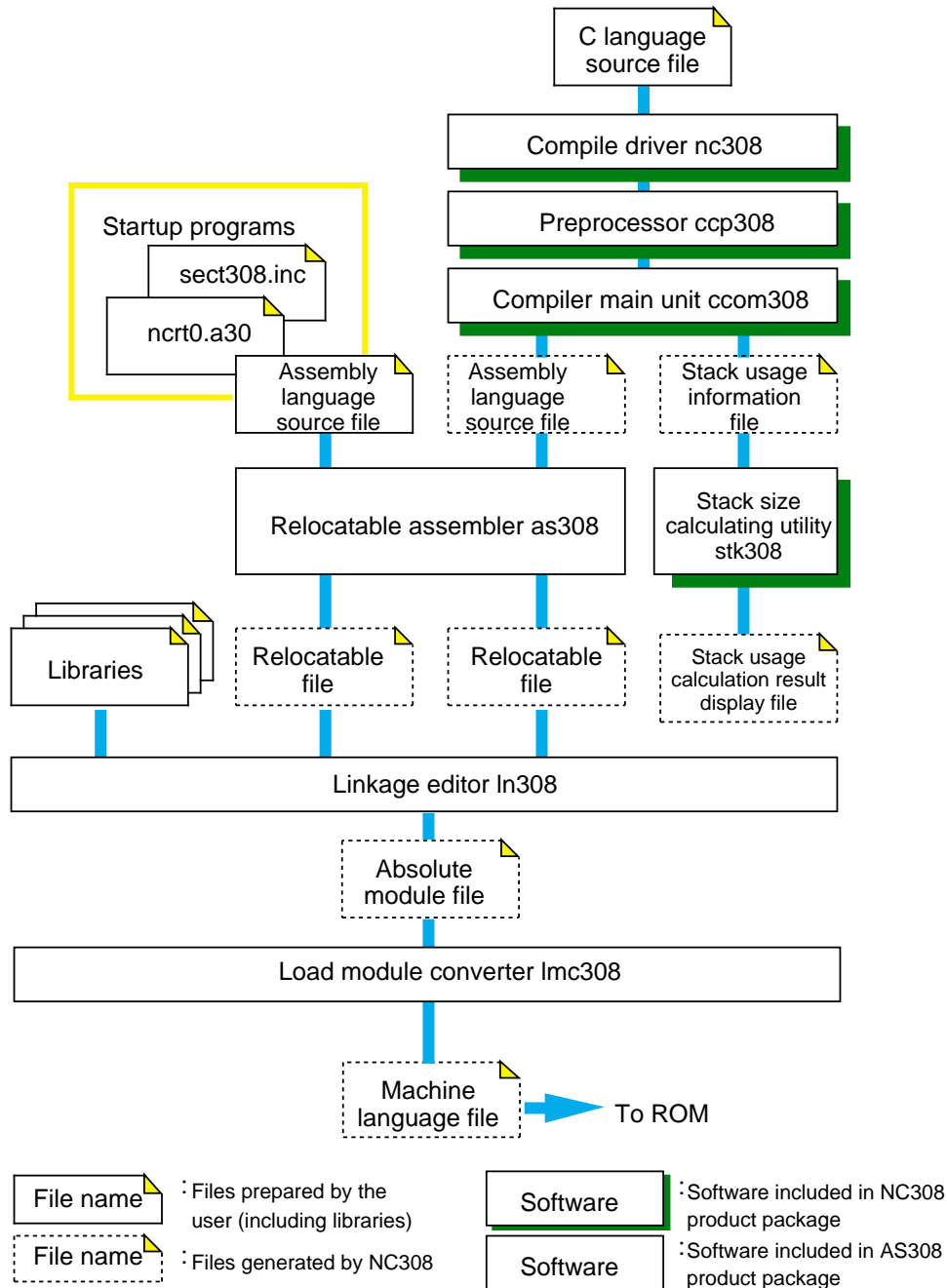


Figure 1.1.1 NC308 Product List

**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.

Figure 1.1.2 shows the tool chain necessary to create a machine language file from a C language source file.



**Figure 1.1.2 Creating Machine Language File from C Language Source File**

### 1.1.3 Program Rules and Practices

Since there is no specific format for C language programs, they can be written in any way desired as long as the stipulated rules of the C language are followed. However in order for a program to be easily read and maintained it should follow some common practices. This section explains some points for creating a well written program.

#### Rules on C Language

The following lists the six items that need to be observed when writing a C language program:

- (1) As a rule, use lowercase English letters to write a program.
- (2) Separate executable statements in a program with a semicolon ";".
- (3) Enclose execution units of functions or control statements with brackets "{" and "}"
- (4) Functions and variables require type declaration.
- (5) Reserved words cannot be used in identifiers (e.g., function names and variable names).
- (6) Write comments between "/\*" and "\*/".

#### Configuration of C Language Source File

Figure 1.1.3 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.

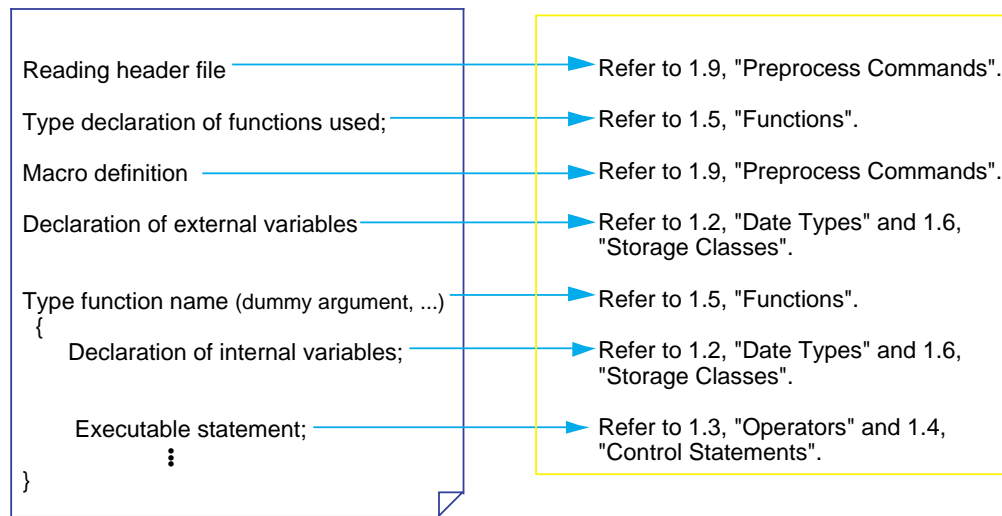
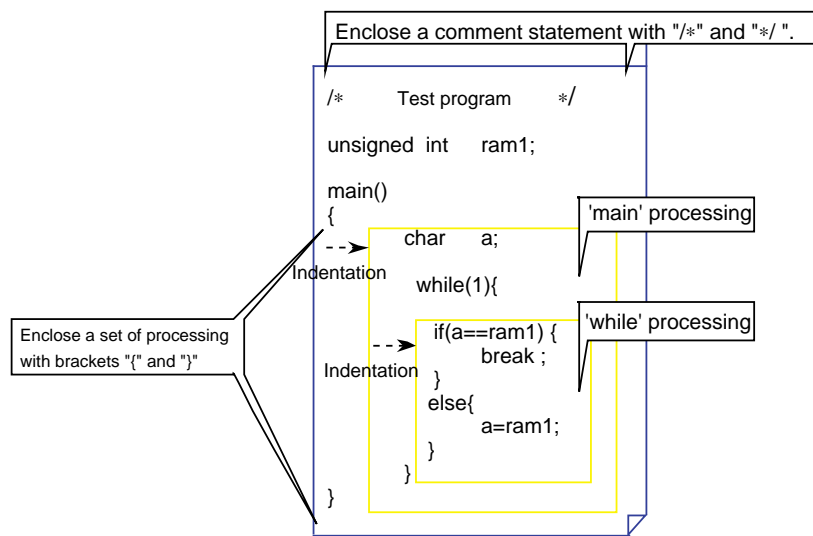


Figure 1.1.3 Configuration of C Language Source File

**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. Figure 1.1.4 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



**Figure 1.1.4 Example of Programming Style of C Language Program**

## 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"" *****
*SystemName   : Test program
* FileName    : TEST.C
* Version     : 1.00
* CPU        : M30800M8-XXXFP
* Compiler    : NC308 (Ver.1.00)
* OS         : Unused
* Programmer  : XXXX
*****
* Copyright, XXXX xxxxxxxxxxxxxxxxxxxx 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
* -----
* Declaration   : void main (void)
* -----
* Functionality : Overall control
* -----
* Argument     : void
* -----
* Return value  : void
* -----
* Input        : None.
* Output       : None.
* -----
* Used functions : voidkey_in ( void )      ; Input function
*                : voidkey_out ( void )    ; Output function
* -----
* Precaution   : Nothing particular.
* ""FUNC COMMENT END"" *****/

void   main ( void )
{
    while(1){           /* Endless loop */
        key_in();      /* Input processing */
        key_out();     /* Output processing */
    }
}
    
```

Figure 1.1.5 Example for Using Comments

**Reserved Words of NC308**

The words listed in Table 1.1.2 are reserved for NC308. Therefore, these words cannot be used in variable or function names.

**Table 1.1.2 Reserved Words of NC308**

_asm	const	far	register	switch
_far	continue	float	return	typedef
_near	default	for	short	union
asm	do	goto	signed	unsigned
auto	double	if	sizeof	void
break	else	int	static	volatile
case	enum	long	struct	while
char	extern	near	inline	

## 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. Table 1.2.1 shows each method for writing integer constants. Constant data are not discriminated between uppercase and lowercase.

**Table 1.2.1 Method for Writing Integer Constants**

Numeration	Method of writing	Example
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

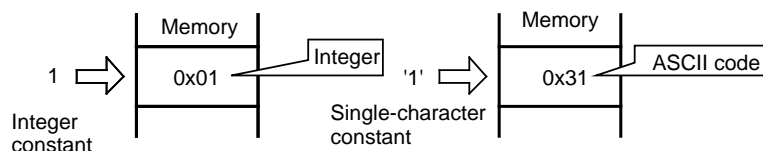
#### Real Constants (Floating-point Constants)

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 in Figure 1.2.1.



**Figure 1.2.1 Difference between 1 and '1'**

### 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!"

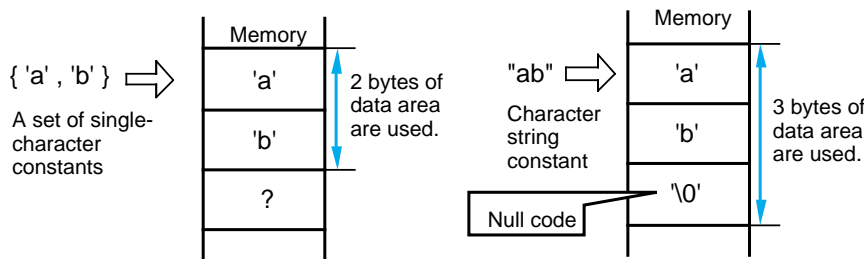


Figure 1.2.2 Difference between { 'a', 'b' } and "ab"

### List of Control Codes (Escape Sequence)

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

Table 1.2.2 Escape Sequence in C Language

Notation	Content	Notation	Content
\f	Form feed (FF)	\'	Single quotation
\n	New line (NL)	\"	Double quotation
\r	Carriage return (CR)	\x constant value	Hexadecimal
\t	Horizontal tab (HT)	\ constant value	Octal
\\	¥symbol	\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 NC308 and how to declare the data types.

### Basic Data Types of NC308

Table 1.2.3 lists the data types that can be handled in NC308. Descriptions enclosed with ( ) in the table below can be omitted when declaring the data type.

**Table 1.2.3 Basic Data Types of NC308**

	Data type	Bit length	Range of values that can be expressed
Integer	(unsigned) char	8 bits	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
	float	32 bits	Number of significant digits: 9
Real	double	64 bits	Number of significant digits: 17
	long double	64 bits	Number of significant digits: 17

**Declaration of Variables**

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

Example: To declare a variable a as char type  
 char a;

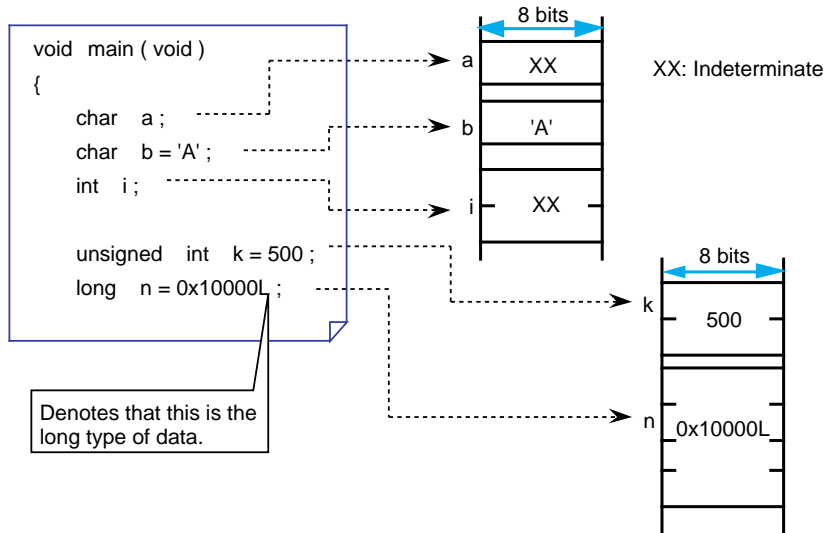
By writing "data type Δ variable name = initial value;", a variable can have its initial value set simultaneously when it is declared.

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

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

Example: int i, j;

Example: inti = 1, j = 2;



**Figure 1.2.3 Declaration of Variables**

### 1.2.3 Data Characteristics

When declaring a variable or constant, NC308 allows its data characteristic to be written along with the data type. The specifier used for this purpose is called the "type qualifier".

This section explains the data characteristics handled by NC308 and how to specify a data characteristic.

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

Write the type qualifier "signed" when the variable or constant to be declared is signed data or "unsigned" when it is unsigned data. If neither of these type specifiers is written when declaring a variable or constant, NC308 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 ;
    signed char s_a ;

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

Synonymous with "unsigned char a";

Synonymous with "signed int b";

Figure 1.2.4 Example for Writing Type Qualifiers "signed" and "unsigned"

#### 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, NC308 outputs a warning.

```
void main ( void )
{
    char a = 10 ;
    constcharc_a = 20 ;

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

Warning is generated.

Figure 1.2.5 Example for writing the type qualifier "const"

### Inhibiting Optimization by Compiler (volatile Qualifier)

NC308 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. NC308 does not optimize the data that is accompanied by this type qualifier and outputs instruction code for it.

```

char port1 ;
volatile char port2 ;

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

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

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

Figure 1.2.6 Example for Writing the Type Qualifier "volatile"

### Syntax of Declaration

When declaring data, write data characteristics using various specifiers or qualifiers along with the data type. Figure 1.2.7 shows the syntax of a declaration.

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

Figure 1.2.7 Syntax of Declaration

## 1.3 Operators

### 1.3.1 Operators of NC308

NC308 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<sup>(Note)</sup>) and the precautions to be noted when using them.

#### NC308 Operators

Table 1.3.1 lists the operators that can be used in NC308.

**Table 1.3.1 NC308 Operators**

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

Note: For address and pointer operators, refer to Section 1.7, "Arrays and Pointers".

### 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.

**Table 1.3.2 Monadic Arithmetic Operators**

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

When using the increment operator (++) or decrement operator (--) in combination with an assignment or relational operator, note that the result of operation may vary depending on which type, prefix or postfix, is used when writing the operator.

<Examples>

Prefix type: The value is increment or decrement before assignment.

$b = ++a; \rightarrow a = a + 1; b = a;$

Postfix type: The value is increment or decrement after assignment.

$b = a++; \rightarrow b = a; a = a + 1;$

#### 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.

**Table 1.3.3 Binary Arithmetic Operators**

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 expressions 1 and 2 after subtracting their values
*	expression 1 * expression 2	Returns the product of expressions 1 and 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

### Division Operator in NC308

In NC308, calculation results are guaranteed in cases where the divide operation results in an overflow. For this reason, when the operation is performed in the following combinations, the compiler by default calls a runtime library (`_i4divU` or `_i4div`).

```
unsigned int = unsigned long / unsigned int
int = long / int
```

If it is desired to forcibly expand a division operation using the `div` or `divu` instruction of the M16C/80 or M32C/80 series, you may specify the command line option "`-fuse_DIV(-fUD)`." However, if the operation results in an overflow, the calculation result is indeterminate.

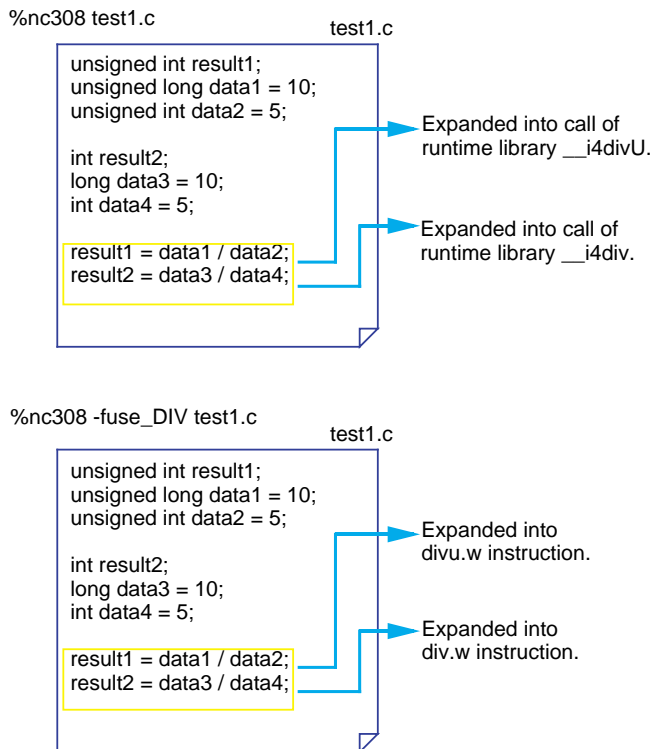


Figure 1.3.1 Division Operator in NC308

## 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.

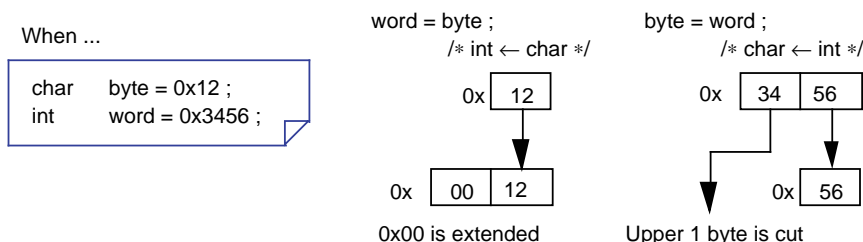
**Table 1.3.4 Substitute Operators**

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.

## Implicit Type Conversion

When performing arithmetic or logic operation on different types of data, NC308 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.



**Figure 1.3.2 Assign Different Types of Data**



### 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.

**Table 1.3.5 Bitwise Operators**

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

#### Shift Operators

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

**Table 1.3.6 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 signed 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)	

Figure 1.3.3 Arithmetic and Logical Shifts

### 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, NC308 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\*3 → (a<<1) +a
  - a\*4 → a<<2
  - a\*7 → (a<<2)+(a<<1) +a
  - a\*8 → a<<3
  - a\*20 → (a<<4) + (a<<2)
- 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.

**Table 1.3.7 Relational Operators**

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.

**Table 1.3.8 Logical Operators**

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

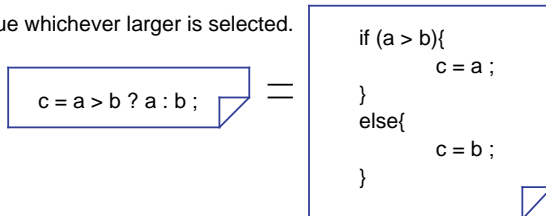
#### Conditional Operator

This operator executes expression 1 if a condition expression is true or expression 2 if the condition expression is false. If this operator is used when the condition expression and expressions 1 and 2 both are short in processing description, coding of conditional branches can be simplified. Table 1.3.9 lists this conditional operator. Figure 1.3.4 shows an example for using this operator.

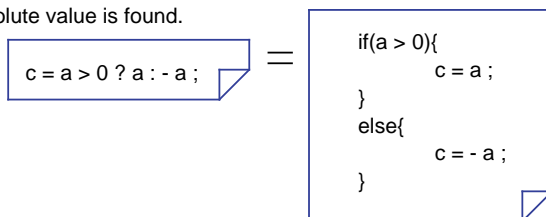
**Table 1.3.9 Conditional Operator**

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.



- Absolute value is found.



**Figure 1.3.4 Example for Using Conditional Operator**

#### sizeof Operator

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

**Table 1.3.10 sizeof Operator**

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.

**Table 1.3.11 Cast Operator**

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

## 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.

**Table 1.3.12 Comma (Sequencing) Operator**

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 1.3.13 Operator Priorities**

Priority resolution	Type of operator	Operator	Rules of combination
<div style="display: flex; align-items: center; justify-content: center;"> <div style="margin-right: 10px;">↑</div> <div style="margin-right: 10px;">High</div> <div style="margin-right: 10px;">↓</div> <div style="margin-right: 10px;">Low</div> </div>	Expression	() [] <sup>(note1)</sup> .->	→
	Monadic arithmetic operators, etc.	! ~ ++ -- - <sup>(note2)</sup> * <sup>(note3)</sup> & sizeof (type)	←
	Multiply/divide operators	<sup>(note4)</sup> * / %	→
	Add/subtract operators	+ -	→
	Shift operator	<< >>	→
	Relational operator (comparison)	< <= > >=	→
	Relational operator (equivalent)	== !=	→
	Bitwise operator (AND)	&	→
	Bitwise operator (EOR)	^	→
	Bitwise operator (OR)		→
	Logical operator (AND)	&&	→
	Logical operator (OR)		→
	Conditional operator	?:	←
	Assignment operator	= += -= *= /= %= <<= >>= &= ^=  =	←
	Comma operator	,	→

Note 1: The dot '.' denotes a member operator that specifies struct and union members.

Note 2: The asterisk '\*' denotes a pointer operator that indicates a pointer variable.

Note 3: The ampersand '&' denotes an address operator that indicates the address of a variable.

Note 4: The asterisk '\*' denotes a multiply operator that indicates multiplication.

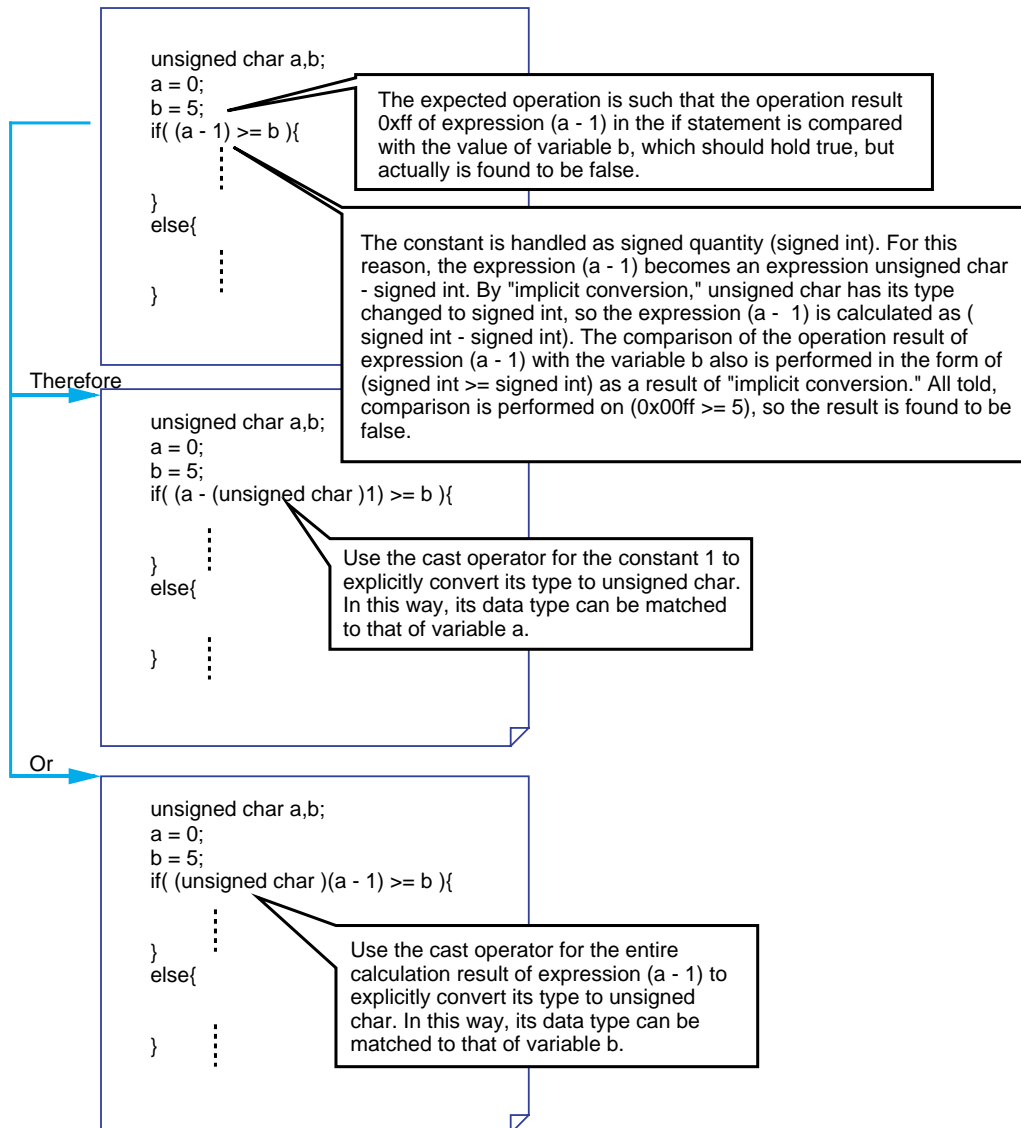
### 1.3.7 Examples for Easily Mistaken Use of Operators

The program may not operate as expected if the "implicit conversion" or "precedence" of operators are incorrectly interpreted.

This section shows examples for easily mistaken use of operators and how to correct.

#### Example 1.3.1 Incorrectly Interpreted "Implicit Conversion" and How to Correct

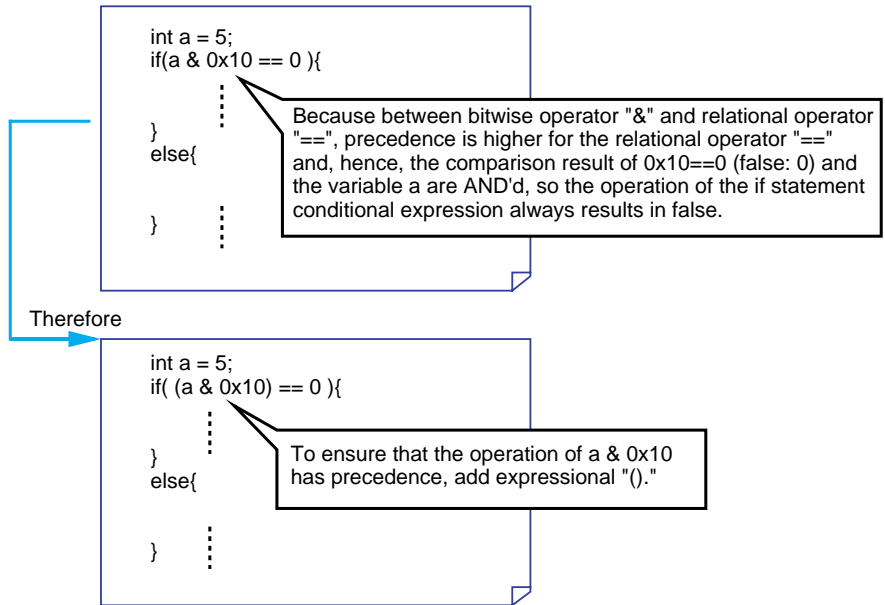
When an operation is performed between different types of data in NC308, the data types are adjusted to that of data which is long in bit length by what is called "implicit conversion" before performing the operation. To ensure that the program will operate as expected, write explicit type conversion using the cast operator.



Example 1.3.1 Incorrectly Interpreted "Implicit Conversion" and How to Correct

**Example 1.3.2 Incorrectly Interpreted "Precedence" of Operators and How to Correct**

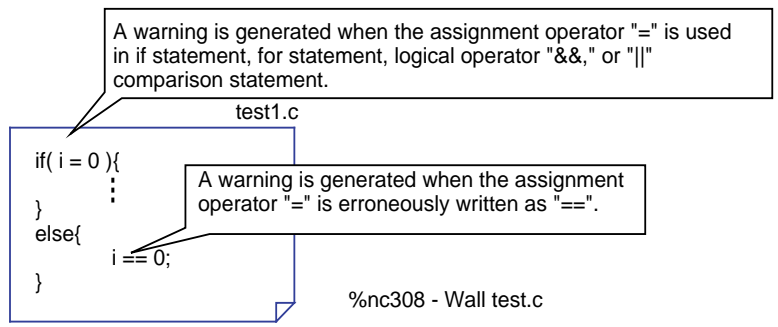
When one expression includes multiple operators, the "precedence" and "associativity" of operators need to be interpreted correctly. Also, to ensure that the program will operate as expected, use expressional "()".



**Example 1.3.2 Incorrectly Interpreted "Precedence" of Operators and How to Correct**

**Detecting the Mistaken Use of Operators (Warning Option "-Wall")**

NC308's warning option "-Wall" can be used to detect the mistaken use of operators<sup>(Note)</sup>. In addition, the warning option "-Wall" indicates other warnings which are equivalent to "-Wnon\_prototype(-WNP)" or "-Wunknown\_pragma(-WUP)".



**Example 1.3.3 Detecting the Mistaken Use of Operators (Warning Option "-Wall")**

Note: Detection is made within the scope that program descriptions are assumed to be incorrect by the compiler.



## 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.

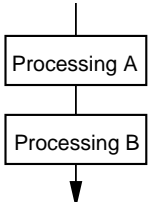
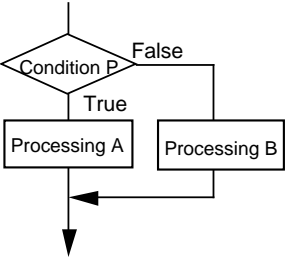
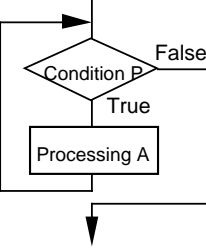
This section describes how to write these control statements and shows some examples of usage.

#### 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".

Table 1.4.1 shows the three basic forms of structured programming.

**Table 1.4.1 Three Basic Forms of Structured Programming**

Sequential processing	 <pre>           graph TD             Start(( )) --&gt; A[Processing A]             A --&gt; B[Processing B]             B --&gt; End(( ))           </pre>	Executed top down, from top to bottom.
Branch processing	 <pre>           graph TD             Start(( )) --&gt; P{Condition P}             P -- True --&gt; A[Processing A]             P -- False --&gt; B[Processing B]             A --&gt; Merge(( ))             B --&gt; Merge             Merge --&gt; End(( ))           </pre>	Branched to processing A or processing B depending on whether condition P is true or false.
Repeat processing	 <pre>           graph TD             Start(( )) --&gt; P{Condition P}             P -- True --&gt; A[Processing A]             A --&gt; P             P -- False --&gt; End(( ))           </pre>	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.

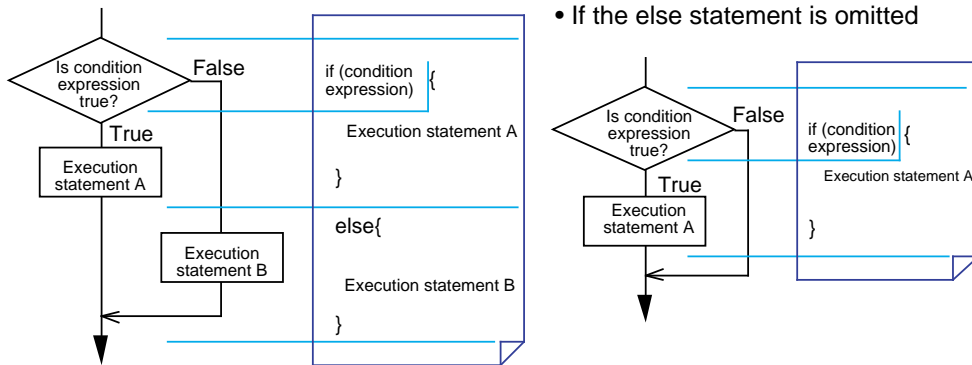


Figure 1.4.1 Example for if-else Statement

#### Example 1.4.1 Count Up (if-else Statement)

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

```

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

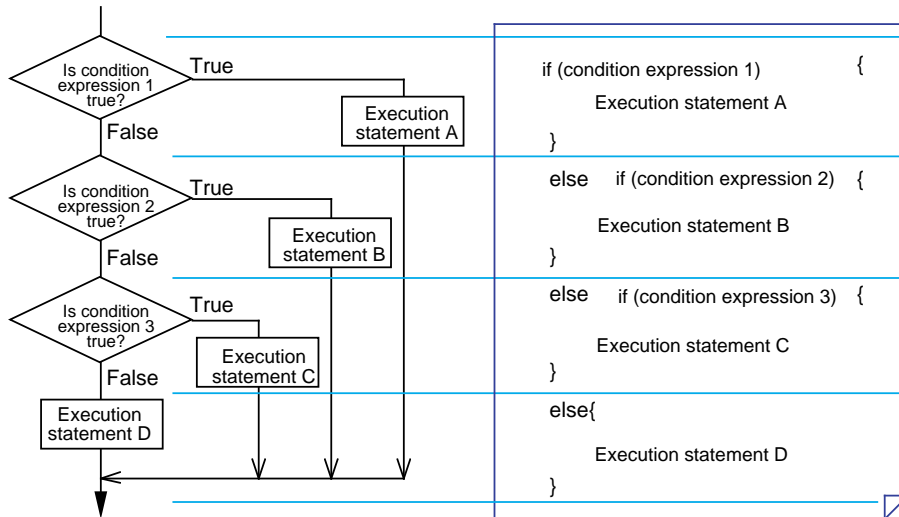
void count_up(void)
{
    if(second >= 59){
        second = 0;
        minute ++;
    }
    else{
        second ++;
    }
}
    
```

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

Example 1.4.1 Count up (if-else Statement)

**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.



**Figure 1.4.2 Example for else-if Statement**

**Example 1.4.2 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".

```

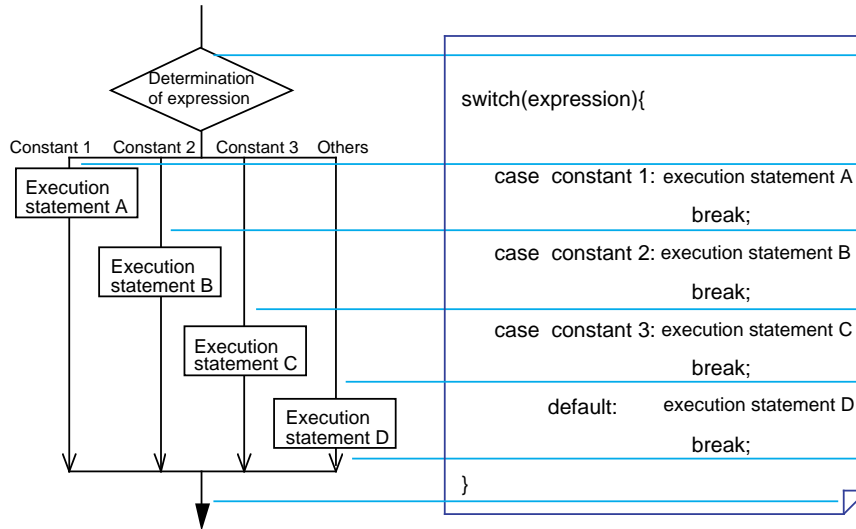
void select(void);           ← Declares "select" function.
                              (Refer to Section 1.5, "Functions".)
int a = 29 , b = 40 ;       ← Declares the variables used.
long int ans ;
char sw ;

void select(void)           ← Defines "select" function.
{
    if(sw == 0){           ← If the content of "sw" is 0,
        ans = a + b ;      ← the program adds data.
    }
    else if(sw == 1){      ← If the content of "sw" is 1,
        ans = a - b ;      ← the program subtracts data.
    }
    else if(sw == 2){      ← If the content of "sw" is 2,
        ans = a*b ;        ← the program multiplies data.
    }
    else if(sw == 3){      ← If the content of "sw" is 3,
        ans = a / b ;      ← the program divides data.
    }
    else{                  ← If the content of "sw" is 4 or greater,
        error();           ← the program performs error
    }                      ← processing.
}
    
```

**Example 1.4.2 Switchover of Arithmetic Operations (else-if Statement)**

**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.



**Figure 1.4.3 Example for switch-case Statement**

**Example 1.4.3 Switchover of Arithmetic Operations (switch-case Statement)**

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

```

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

int a = 29, b = 40;         ← Declares the variables used.
long int ans;
char sw;

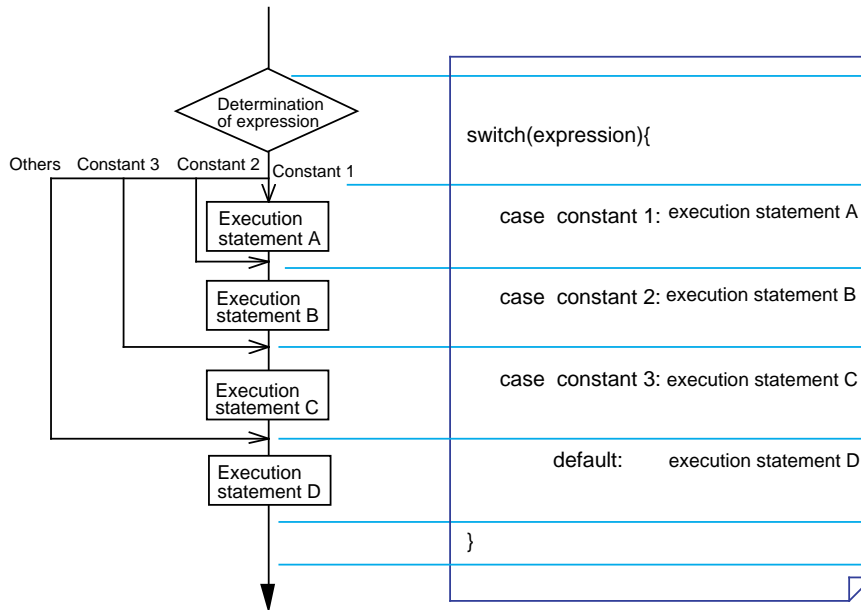
void select(void)           ← Defines "select" function.
{
    switch(sw)              ← Determines the content of "sw".
    {
        case 0: ans = a + b; ← If the content of "sw" is 0, the program
            break;           adds data.
        case 1: ans = a - b; ← If the content of "sw" is 1, the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

**Example 1.4.3 Switchover of Arithmetic Operations (switch-case Statement)**

**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.



**Figure 1.4.4 switch-case Statement without Break**

### 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".

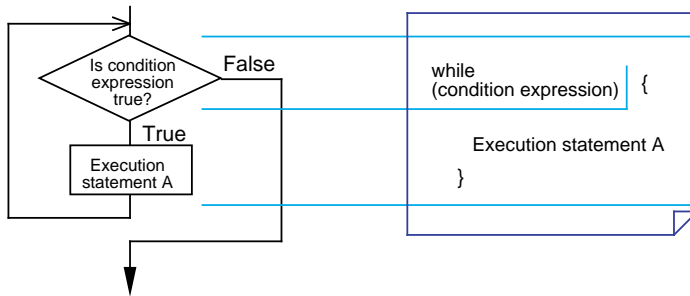


Figure 1.4.5 Example for while Statement

#### Example 1.4.4 Finding Sum Total (while Statement)

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

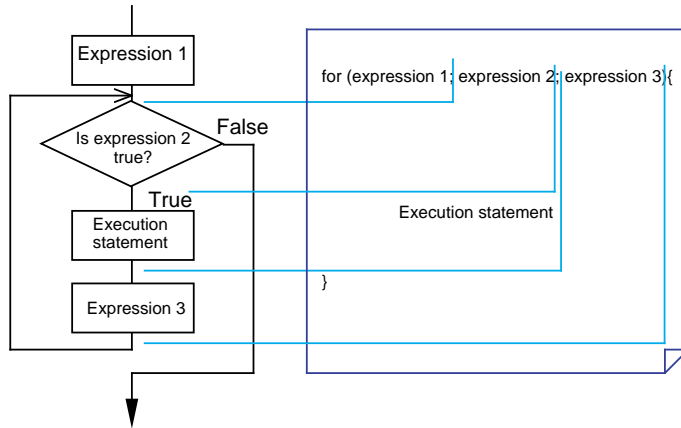
```

void sum(void) ; ← Declares "sum" function. (Refer to Section 1.5,
                    "Functions".)
unsigned int total = 0 ; ← Declares the variables used.
void sum(void) ← Defines "sum" function.
{
    unsigned int i = 1 ; ← Defines and initializes counter variables.
    while(i <= 100){ ← Loops until the counter content reaches 100.
        total += i ;
        i ++ ; ← Changes the counter content.
    }
}
    
```

Example 1.4.4 Finding Sum Total (while Statement)

**for Statement**

The repeat processing that is performed by using a counter like in Example 1.4.4 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. (See Figure 1.4.6.) 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.



**Figure 1.4.6 Example for "for" Statement**

**Example 1.4.5 Finding Sum Total (for Statement)**

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

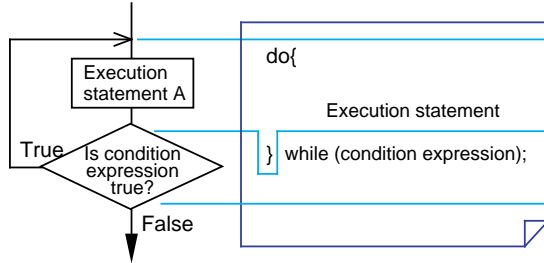
```

void sum(void); // Declares "sum" function. (Refer to Section 1.5, "Functions".)
unsigned int total = 0; // Declares the variables used.
void sum(void) // Defines "sum" function.
{
    unsigned int i; // Defines counter variables.
    for(i = 1 ; i <= 100 ; i++){ // Loops until the counter content increments from 1 to 100.
        total += i;
    }
}
  
```

**Example 1.4.5 Finding Sum Total (for Statement)**

**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.



**Figure 1.4.7 Example for do-while Statement**

**Example 1.4.6 Finding Sum Total (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 = 0 ;
    do{
        i ++ ;
        total += i ;
    }while(i < 100) ;
}
    
```

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

**Example 1.4.6 Finding Sum Total (do-while Statement)**



### 1.4.4 Suspending Processing

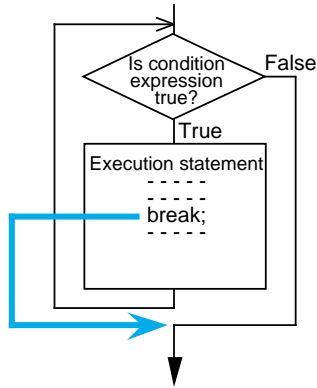
There are control statements (auxiliary control statements) such as break, continue, and goto statements that make it possible to suspend processing and quit.

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



- When used in a for statement

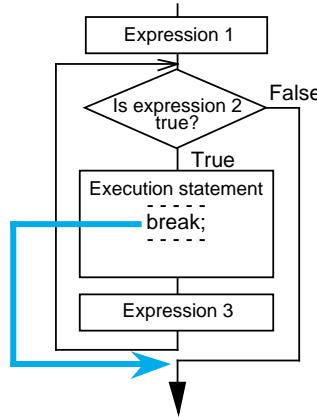
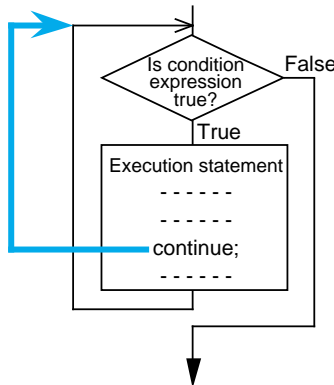


Figure 1.4.8 Example for break Statement

#### 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



- When used in a for statement

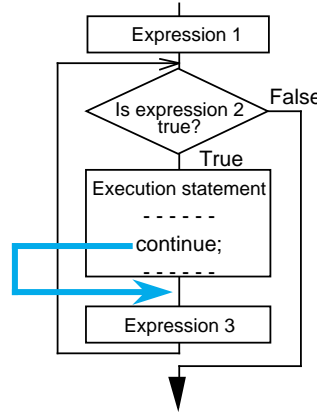
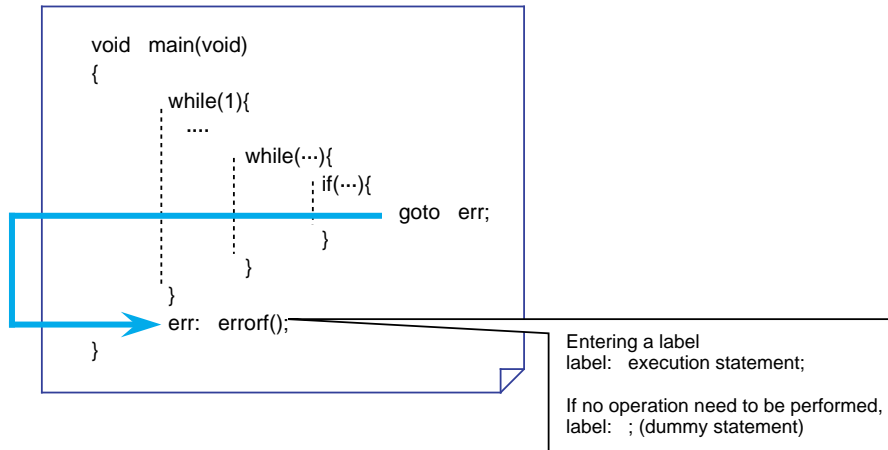


Figure 1.4.9 Example for continue Statement

**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. (See Figure 1.4.10.) 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.



**Figure 1.4.10 Working of goto 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.

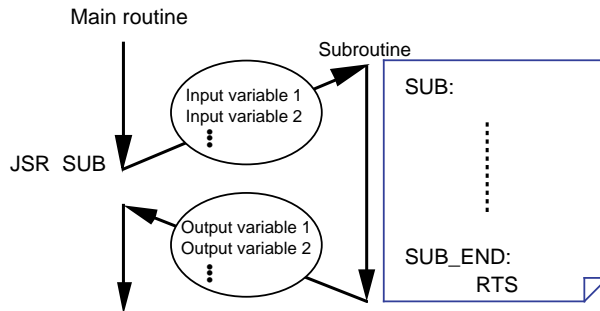
This section explains how to write functions in NC308.

#### 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. No restrictions are imposed on arguments.<sup>(Note)</sup>

- "Subroutine" in assembly language



- "Function" in C language

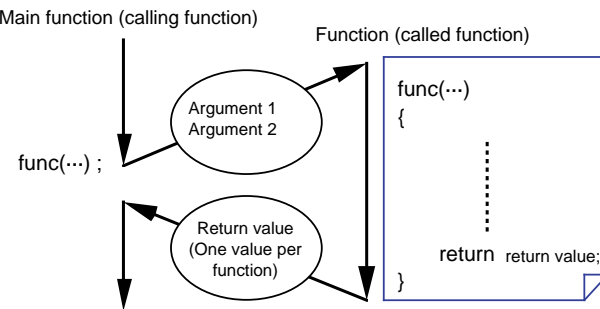


Figure 1.5.1 "Subroutine" vs "Function"

Note: In some compilers designed for writing a finished program into ROM, the number of arguments is limited.

### 1.5.2 Creating Functions

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

#### Function Declaration (Prototype Declaration)

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

The following shows the format of function declaration (prototype declaration):

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

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

#### Function Definition

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

The following shows the format of function definition:

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

#### Function Call

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

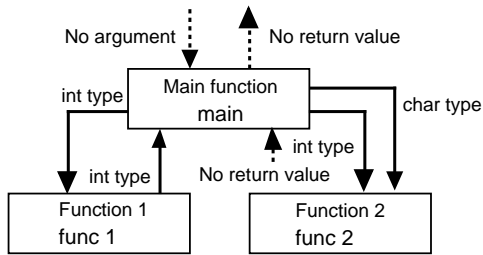
```
function name (argument 1, ...);
```

When there is a return value

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

**Example for a Function**

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



```

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

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

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

/* Definition function 1 */
int func1 ( int x )
{
    int z ;
    :
    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".

**Figure 1.5.2 Example for a Function**

### 1.5.3 Exchanging Data between Functions

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

#### Example 1.5.1 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 ( void )
{
    long int answer ;
    int a = 29 , b = 40 ;

    answer = add ( a , b );
}

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

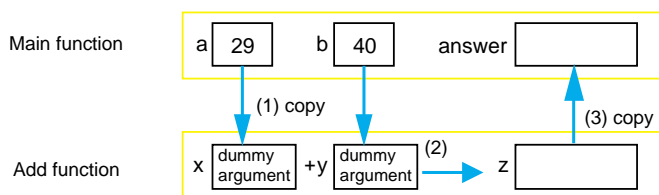
    z = ( long int ) x + y ;
    return z ;
}
    
```

(1) Calls the add function.

(2) Executes addition.

(3) Returns a value for the argument.

<Flow of data>



#### Example 1.5.1 Finding Sum of Integers (a Function)

## 1.6 Storage Classes

### 1.6.1 Effective Range of Variables and Functions

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

This section explains the types of storage classes of variables and functions and how to specify them.

#### Effective Range of Variables and Functions

A C language program consists of multiple source files. Furthermore, each of these source files consists of multiple functions. Therefore, a C language program is hierarchically structured as shown in Figure 1.6.1.

There are following three storage classes for a variable:

- (1) Effective in only a function
- (2) Effective in only a file
- (3) Effective in the entire program

There are following two storage classes for a function:

- (1) Effective in only a file
- (2) Effective in the entire program

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

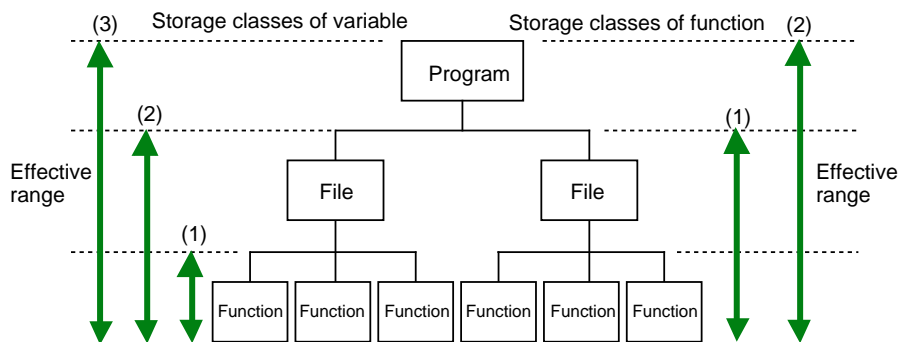


Figure 1.6.1 Hierarchical Structure and Storage Classes of C Language Program

### 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.

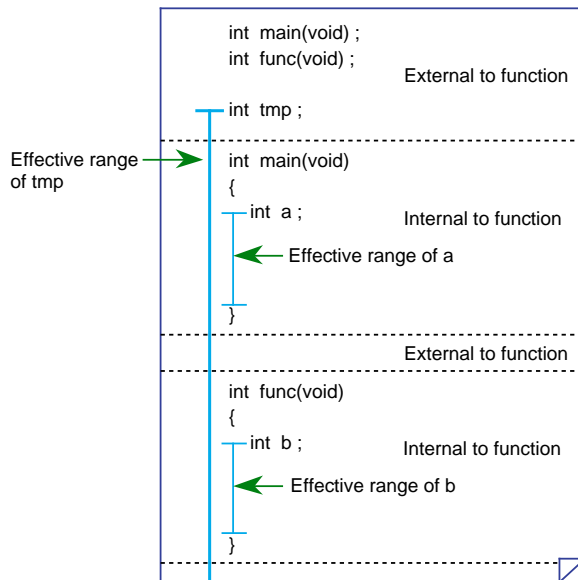


Figure 1.6.2 External and Internal Variables

#### Storage Class Specifiers

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

storage class specifier Δ data type Δ variable name;



### Storage Classes of External Variable

If no storage class specifier is added for an external variable when declaring it, the variable is assumed to be a global variable that is effective in the entire program. On the other hand, if an external variable is specified of its storage class by writing "static" when declaring it, the variable is assumed to be a local variable that is effective in only the file where it is declared.

Write the specifier "extern" when using an external variable that is defined in another file like "mode" in source file 2 of Figure 1.6.3.

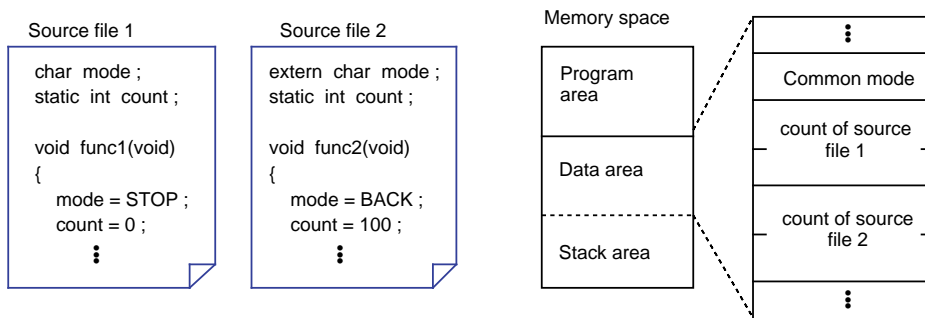


Figure 1.6.3 Storage Classes of External Variable

### Storage Classes of Internal Variable

An internal variable declared without adding any storage class specifier has its area allocated in a stack. Therefore, such a variable is initialized each time the function is called. On the other hand, an internal variable whose storage class is specified to be "static" is allocated in a data area. In this case, therefore, the variable is initialized only once when starting up the program.

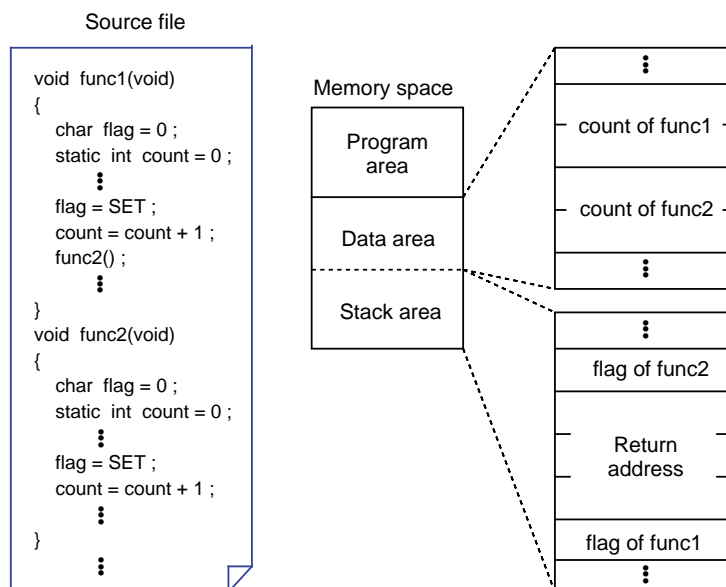


Figure 1.6.4 Storage Classes of Internal Variable

### 1.6.3 Storage Classes of Functions

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

#### Global and Local Functions

- (1) If no storage class is specified for a function when defining it  
 This function is assumed to be a global function that can be called and used from any other source file.
- (2) If a function is declared to be "static" when defining it  
 This function is assumed to be a local function that cannot be called from any other source file.
- (3) If a function is declared to be "extern" in its type declaration  
 This storage class specifier indicates that the declared function is not included in the source file where functions are declared, and that the function in some other source file be called. However, only if a function has its type declared--even though it may not be specified to be "extern", if the function is not found in the source file, the function in some other source file is automatically called in the same way as when explicitly specified to be "extern".

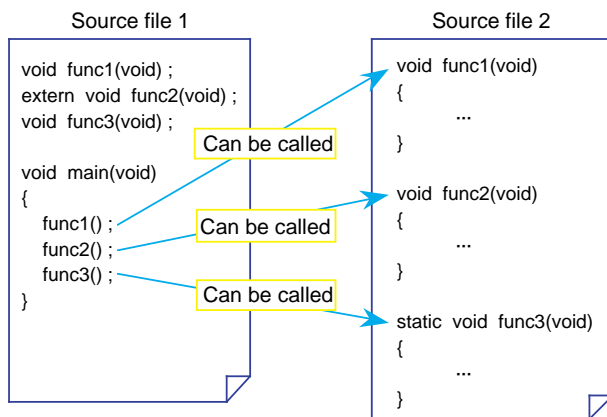


Figure 1.6.5 Storage Classes of Function

**Summary of Storage Classes**

Storage classes of variables are summarized in Table 1.6.1. Storage classes of functions are summarized in Table 1.6.2.

**Table 1.6.1 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 a data area]	Variables that are effective in only the function. [Allocated in a stack when executing the function.]
auto		Variables that are effective in only the function. [Allocated in a stack when executing the function.]
static	Local variables that cannot be referenced from other source files. [Allocated in a data area]	Variables that are effective in only the function. [Allocated in a data area.]
register		Variables that are effective in only the function. [Allocated in the register.] However, this has no effect in NC308 unless the generated code change option "-fenable_register(-fER)" is specified. (ignored when compiled.)
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]

**Table 1.6.2 Storage Classes of Functions**

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

## How to Use Register Variables

In NC308, the register storage class is enabled by specifying the generated code change option "-fenable\_register(-fER)." Note that unless this option is specified, the register storage class specifier written in a program has no effect (ignored when compiled).

```

void func1(void);
char array[10][10];
void func1(void)
{
  register int i,j;
  for(i = 0,j = 0 ; i < 0, j < 0 ; i++,j++){
    array[i][j] = 0;
  }
}
  
```

When the generated code change option "-fenable\_register(-fER)" is specified, this variable is assigned to a register when executed.

**Figure 1.6.6 How to Use Register Variables**

## 1.7 Arrays and Pointers

### 1.7.1 Arrays

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

#### What is an Array?

The following explains the functionality of an array by using a program to find the total age of family members as an example. The family consists of parents (father = 29 years old, mother = 24 years old), and a child (brother = 4 years old). (See Example 1.7.1.) 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.

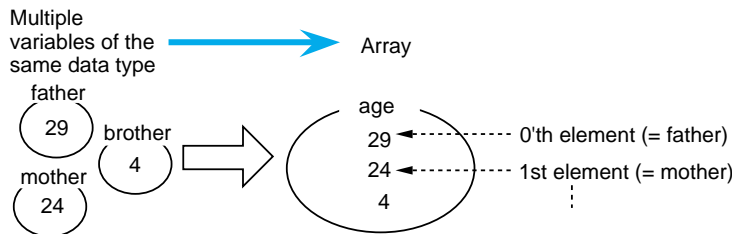
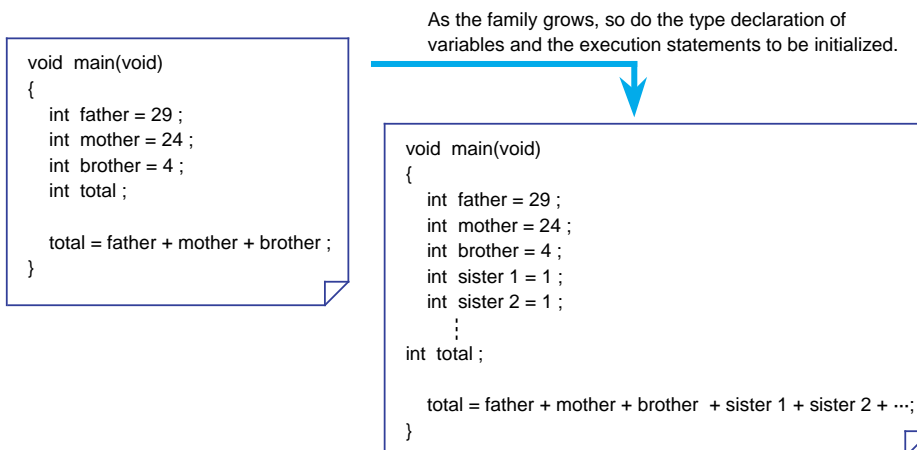


Figure 1.7.1 Concept of an Array

#### Example 1.7.1 Finding Total Age of a Family (1)

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



Example 1.7.1 Finding Total Age of a Family (1)

### 1.7.2 Creating an Array

There are two types of arrays handled in the C language: "one-dimensional array" and "two-dimensional array".

This section describes how to create and reference each type of array.

#### One-dimensional Array

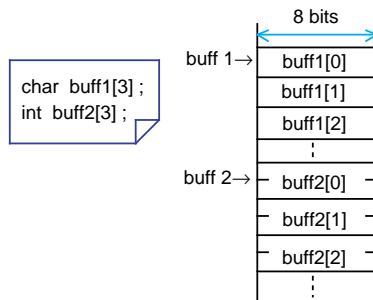
A one-dimensional array has a one-dimensional (linear) expanse. The following shows the declaration format of a one-dimensional array.

Data type    array name [number of elements];

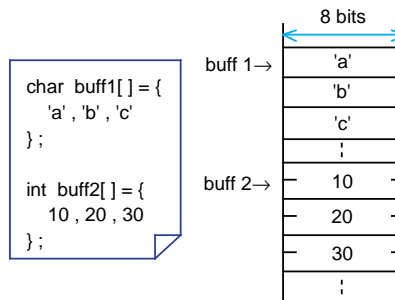
When the above declaration is made, an area is allocated in memory for the number of elements, with the array name used as the beginning label.

To reference a one-dimensional array, add element numbers to the array name as subscript. However, since element numbers begin with 0, the last element number is 1 less than the number of elements.

- Declaration of one-dimensional array



- Declaration and initialization of one-dimensional array



**Figure 1.7.2 Declaration of One-dimensional Array and Memory Mapping**

#### Example 1.7.2 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[] = {
    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".)

#### Example 1.7.2 Finding Total Age of a Family (2)

## Two-dimensional Array

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

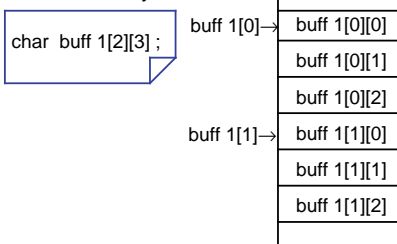
Data type   array name [number of rows] [number of columns];

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

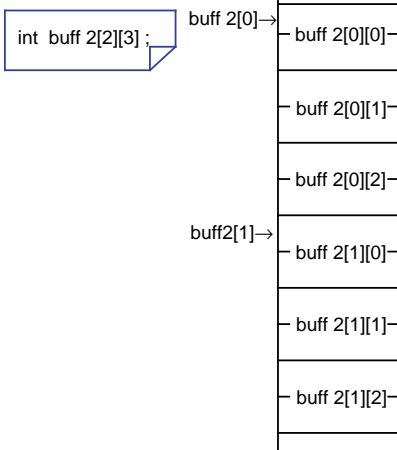
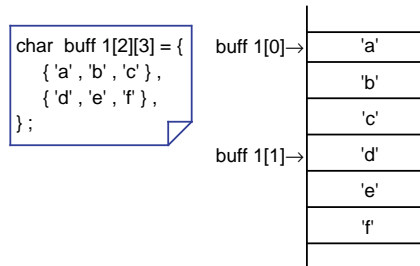
- Concept of two-dimensional array

	Columns→			
Rows ↓	Row 0 column 0	Row 0 column 1	Row 0 column 2	Row 0 column 3
	Row 1 column 0	Row 1 column 1	Row 1 column 2	Row 1 column 3
	Row 2 column 0	Row 2 column 1	Row 2 column 2	Row 2 column 3

- Declaration and initialization of two-dimensional array

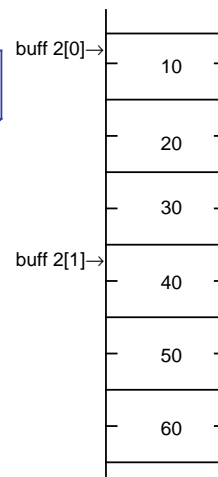


- Declaration and initialization of two-dimensional array



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

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



**Figure 1.7.3 Declaration of Two-dimensional Array and Memory Mapping**

### 1.7.3 Pointers

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

#### Declaring a Pointer Variable

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

```
Pointed data type * pointer variable name;
```

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

• Pointer variable declaration

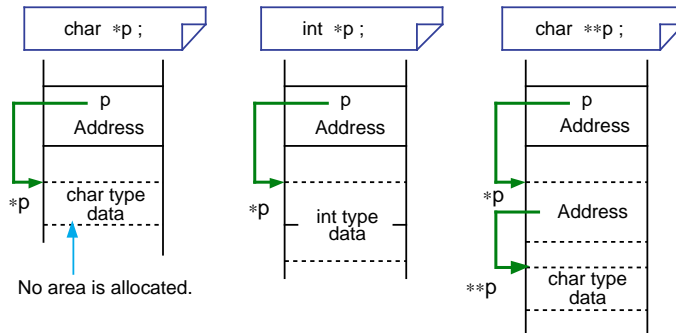


Figure 1.7.4 Pointer Variable Declaration and Memory Mapping



### Relationship between Pointer Variables and Variables

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

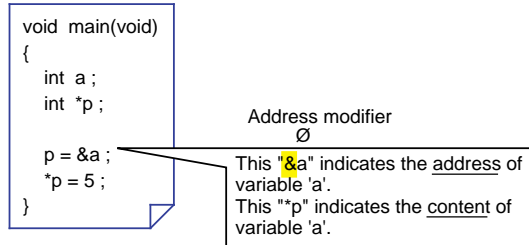


Figure 1.7.5 Relationship between Pointer Variables and Variables

### Operating on Pointer Variables

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

$$\text{Address} + (\text{integer} \times \text{sizeof}(\text{type}))$$

$$\text{Address} - (\text{integer} \times \text{sizeof}(\text{type}))$$

The pointer variable ptr is an int type of variable. When calculated by sizeof(int), the size of the int-type variable is found to be 2 bytes. Therefore, p + 2 points to address 1004H.

```

int *ptr;

ptr = (int *)0x001000;
ptr = p + 2;
    
```

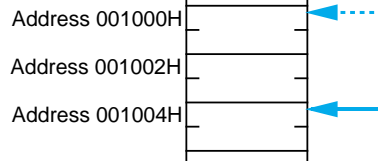


Figure 1.7.6 Operating on Pointer Variables

### Data Length of Pointer Variable

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

Pointer variables in NC308 are 4 bytes in data length because the relevant data by default is assumed to be in the far area. For details, refer to Section 2.3.1, "Efficient Addressing."

### 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.

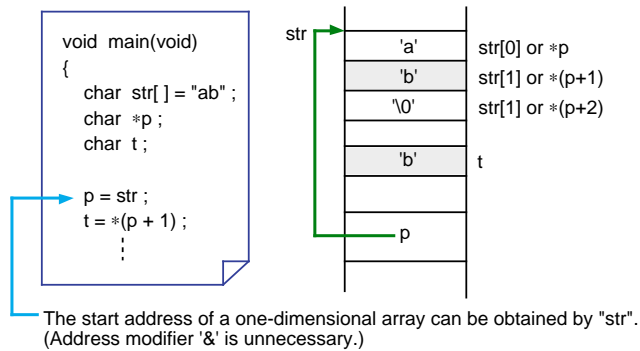


Figure 1.7.7 Pointer Variables and One-dimensional Array

#### 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.

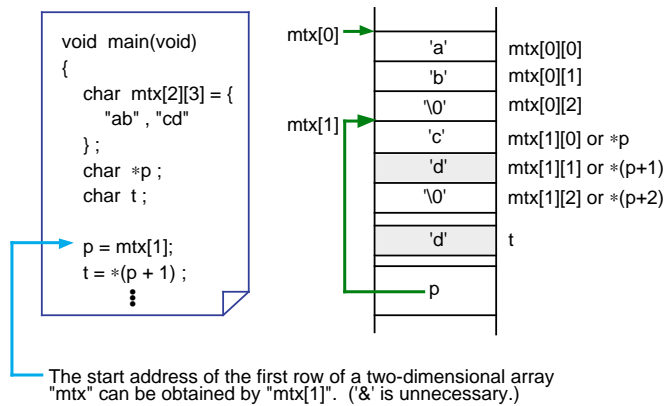


Figure 1.7.8 Pointer Variables and Two-dimensional Array

### Passing Addresses between Functions

The basic method of passing data to and from C language functions is referred to as "Call by Value". With this method, however, arrays and character strings cannot be passed between functions as arguments or returned values. Used to solve this problem is a method, known as "Call by Reference", which uses a pointer variable. In addition to passing the addresses of arrays or character strings between functions, this method can be used when it is necessary to pass multiple data as a returned value. Unlike the Call by Value method, this method has a drawback in that the independency of each function is reduced, because the data in the calling function is rewritten directly. Figure 1.7.9 shows an example where an array is passed between functions using the Call by Reference method.

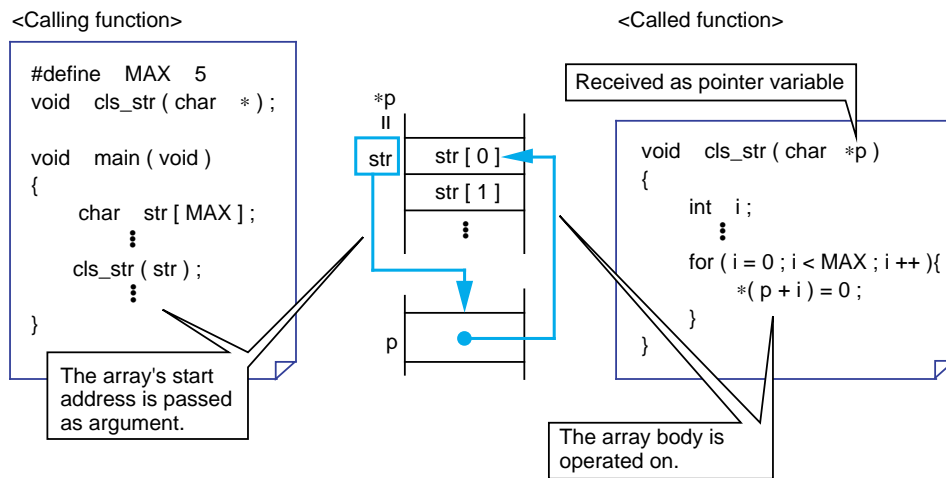


Figure 1.7.9 Example of Call by Reference for Passing an Array

### 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 losing 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.

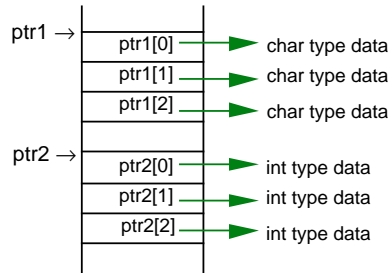
#### Pointer Array Declaration

The following shows how to declare a pointer array.

```
Data type far(Note) * array name [number of elements];
```

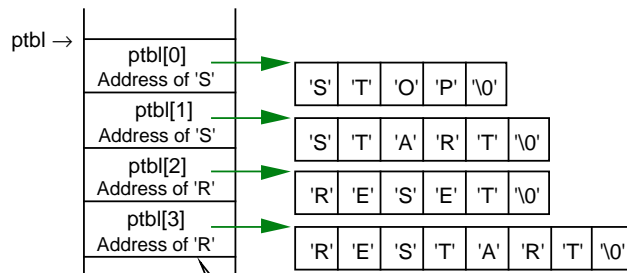
• Pointer array declaration

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



• Pointer array initialization

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



Each character string's start address is stored here.

Figure 1.7.10 Pointer Array Declaration and Initialization

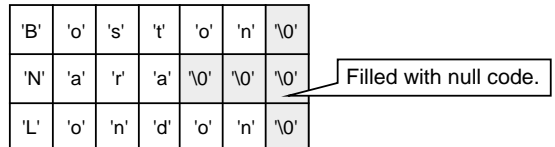
Note: The actual data of pointer arrays in NC308 are located in the far area. Also, pointer-type variables by default are a far type of variable (4 bytes). Therefore, omit the description "far" normally written for pointers. For details, refer to Section 2.3.1, "Efficient Addressing."

### Pointer Array and Two-dimensional Array

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

• Two-dimensional array

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



• Pointer array

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

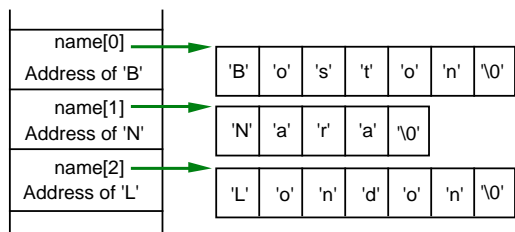


Figure 1.7.11 Difference between Two-dimensional Array and Pointer Array

### **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);

**Example 1.7.3 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[ ] ) ( int , int ) = {
    add_f , sub_f , mul_f , div_f
};

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

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

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

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

Function pointers arranged in an array

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

Setting of jump address

Function call using a function pointer

**Example 1.7.3 Switching Arithmetic Operations Using Table Jump**

## 1.8 Struct and Union

### 1.8.1 Struct and Union

The data types discussed hereto (e.g., char, signed int, and unsigned log int types) are called the "basic data types" stipulated in compiler specifications.

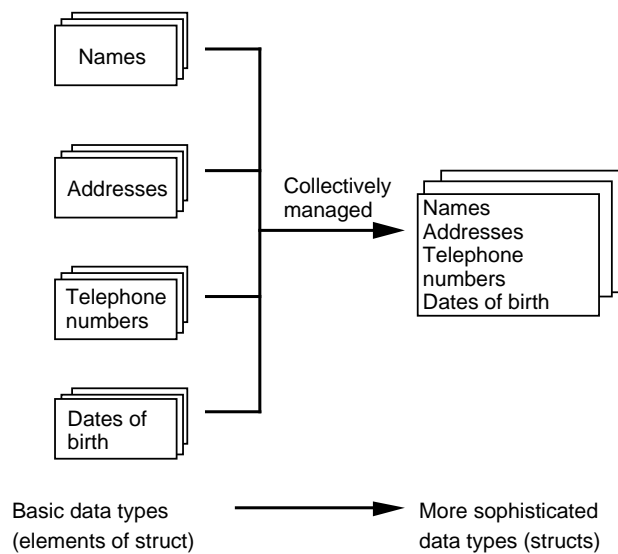
The C language allows the user to create new data types based on these basic data types.

These are "struct" and "union".

The following explains how to declare and reference structs and unions.

#### From Basic Data Types to Structs

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



**Figure 1.8.1 From Basic Data Types to Structs**



**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.)

**Definition and Declaration of Struct**

To define a struct, write "struct".

```

struct struct tag {
    member 1;
    member 2;
    ⋮
};
  
```

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

```

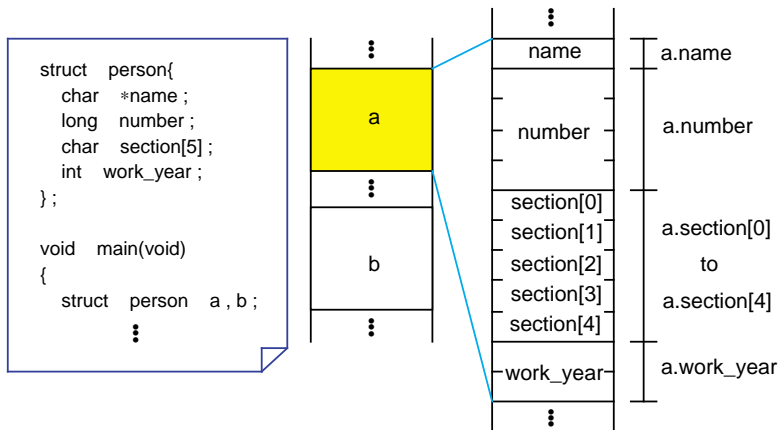
struct Δ struct tag Δ struct variable name;
  
```

## Referencing Struct

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

struct variable name.member name

To initialize a struct variable, list each member's initialization data in the order they are declared, with the types matched.



If the area that contains name is a near area, "struct person" becomes a 13-byte type; if a far area, it becomes a 15-byte type.

\* Initialization of struct variable

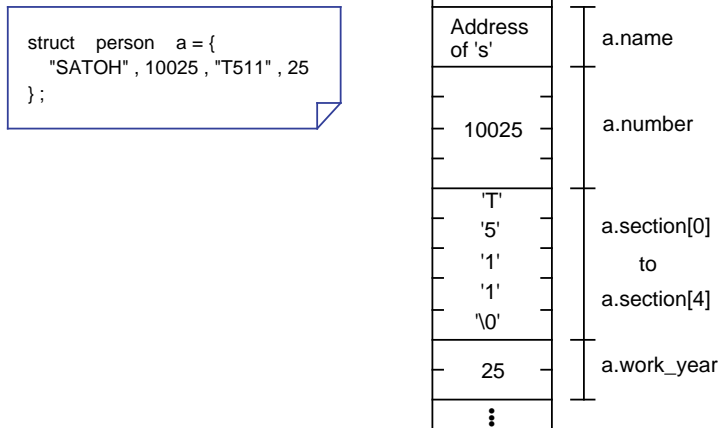


Figure 1.8.2 Struct Declaration and Memory Mapping

**Example for Referencing Members Using a Pointer**

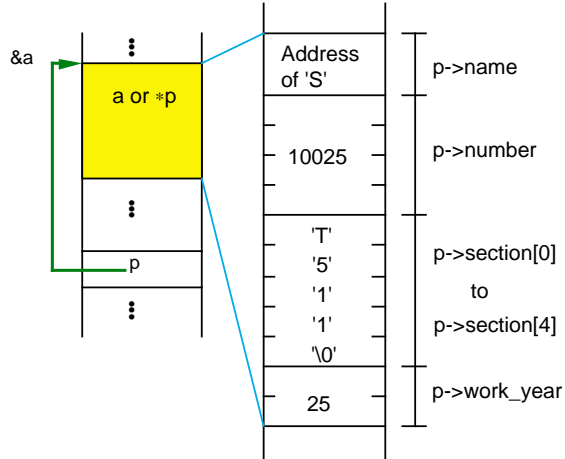
To refer to each member of a struct using a pointer, use an arrow '->'.

Pointer -> member name

```

struct person{
    char far *name;
    long number;
    char section[5];
    int work_year;
};
struct person a = {
    "SATOH", 10025, "T511", 25
};

void main(void)
{
    struct person *p;
    p = &a;
    :
    
```

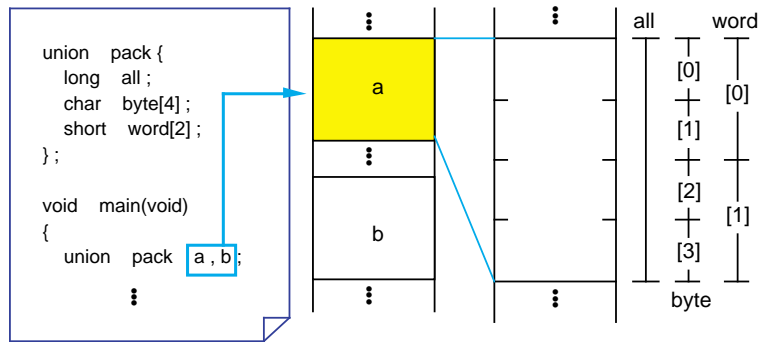


**Figure 1.8.3 Example for Referencing Members Using a Pointer**

## Unions

Unions are characteristic in that an allocated memory area is shared by all members. Therefore, it is possible to save on memory usage by using unions for multiple entries of such data that will never exist simultaneously. Unions also will prove convenient when they are used for data that needs to be handled in different units of data size, e.g., 16 bits or 8 units, depending on situation.

To define a union, write "union". Except this description, the procedures for defining, declaring, and referencing unions all are the same as explained for structs.



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

Figure 1.8.4 Declaring and Referencing a Union

## Type Definition

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

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

When the above description is made, the new type name is assumed to be synonymous with the existing type name and, therefore, either type name can be used in the program. Figure 1.8.5 below shows an example of how "typedef" can actually be used.

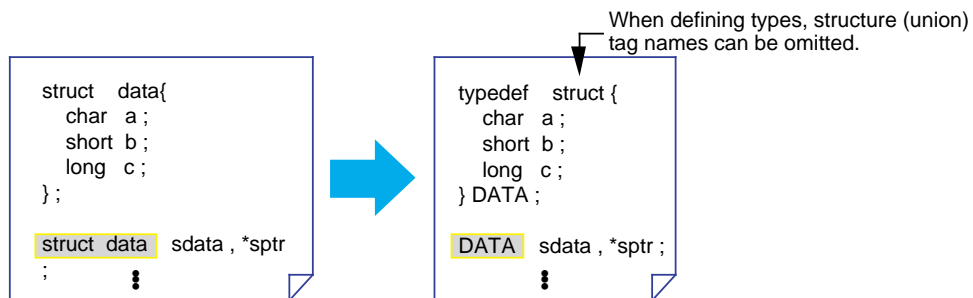


Figure 1.8.5 Example for Using Type Definition "typedef"

## 1.9 Preprocess Commands

### 1.9.1 Preprocess Commands of NC308

The C language supports file inclusion, macro function, conditional compile, and some other functions as "preprocess commands".

The following explains the main preprocess commands available with NC308.

#### Preprocess Command List of NC308

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. Table 1.9.1 lists the main preprocess commands that can be used in NC308.

**Table 1.9.1 Main Preprocess Commands of NC308**

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.
#assert	Outputs alarm when constant expression is false.
#pragma	Instructs processing of NC30's extended function. This is detailed in Chapter 2.

### 1.9.2 Including a File

Use the command "#include" to take in another file. NC308 requires different methods of description depending on the directory to be searched.

This section explains how to write the command "#include" for each purpose of use.

#### Searching for Standard Directory

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

This statement takes in a file from the directory specified with the startup option '-I.' If the specified file does not exist in this directory, NC308 searches the standard directory that is set with NC308's environment variable "INC308" 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, NC308 searches the directory specified with the startup option '-I' and the directory set with NC308's environment variable "INC308" 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"

NC308's command "#include" can be nested in up to 8 levels. If the specified file cannot be found in any directory searched, NC308 outputs an include error.

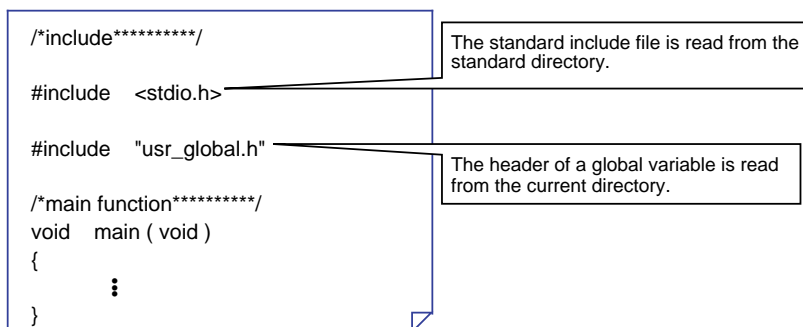


Figure 1.9.1 Typical Description of "#include"

### 1.9.3 Macro Definition

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

#### Defining a Constant

A constant can be assigned a name in the same way as in the assembler "equ statement". 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.

Figure 1.9.2 Example for Defining a Constant

#### Defining a Character String

It is possible to assign a character string a name or, conversely, delete a character string.

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

#define void
void func()
{
    ;
}
```

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

"void" is deleted. For a compiler where "void" is not supported, this definition eliminates the need for modification in the source file.

Figure 1.9.3 Example for Defining a Character String

## 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.

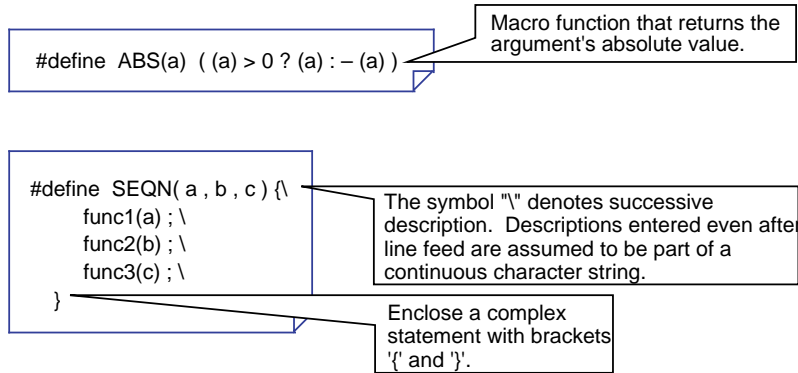


Figure 1.9.4 Example for Defining a Macro Function

## Canceling Definition

```
#undef identifier
```

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

- `_FILE_` Source file name
- `_LINE_` Line number of current source file
- `_DATE_` Compilation date
- `_TIME_` Compilation time



### 1.9.4 Conditional Compile

NC308 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

Table 1.9.2 lists the types of conditional compilation that can be used in NC308.

**Table 1.9.2 Types of Conditional Compile**

Description	Content
<pre>#if condition expression     A #else     B #endif</pre>	If the condition expression is true (not 0), NC308 compiles block A; if false, it compiles block B.
<pre>#ifdef identifier     A #else     B #endif</pre>	If an identifier is defined, NC308 compiles block A; if not defined, it compiles block B.
<pre>#ifndef identifier     A #else     B #endif</pre>	If an identifier is not defined, NC308 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 NC308 startup option '-D'.

`#define identifier` ← Specification of definition by "#define"

`%nc308 -D identifier` ← Specification of definition by startup option

**Example for Conditional Compile Description**

Figure 1.9.5 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
  
```

**Figure 1.9.5 Example for Conditional Compile Description**

---

## Chapter 2

---

### Extended Functions of NC308

- 2.1 Memory Mapping
- 2.2 Startup Program
- 2.3 Extended Functions for ROM'ing
- 2.4 Linkage with Assembly Language
- 2.5 Interrupt Processing

This chapter describes precautions to be followed when creating built-in programs by focusing on the extended functions of NC308.

## 2.1 Memory Mapping

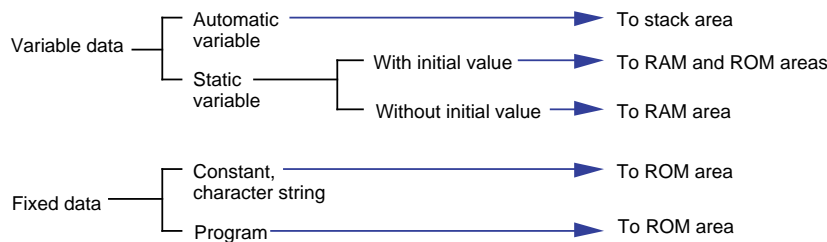
### 2.1.1 Types of Code and Data

There are various types of data and code that constitute a program. Some are rewritable, and some are not. Some have initial values, and some do not. All data and code must be mapped into the ROM, RAM, and stack areas according to their properties.

This section explains the types of data and code that are generated by NC308.

#### Data and Code Generated by NC308

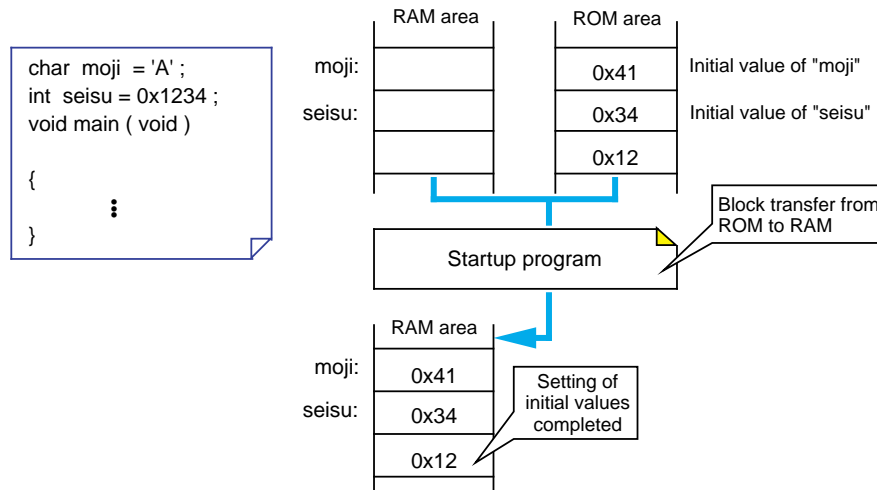
Figure 2.1.1 shows the types of data and code generated by NC308 and their mapped memory areas.



**Figure 2.1.1 Types of Data and Code Generated by NC308 and Their Mapped Areas**

#### Handling of Static Variables with Initial Values

Since "static variables with initial values" are rewritable data, they must reside in RAM. However, if variables are stored in RAM, initial values cannot be set for them. To solve this problem, NC308 allocates an area in RAM for such static variables with initial values and stores initial values in ROM. Then it copies the initial values from ROM into RAM in the startup program.



**Figure 2.1.2 Handling of Static Variables with Initial Values**

### 2.1.2 Sections Managed by NC308

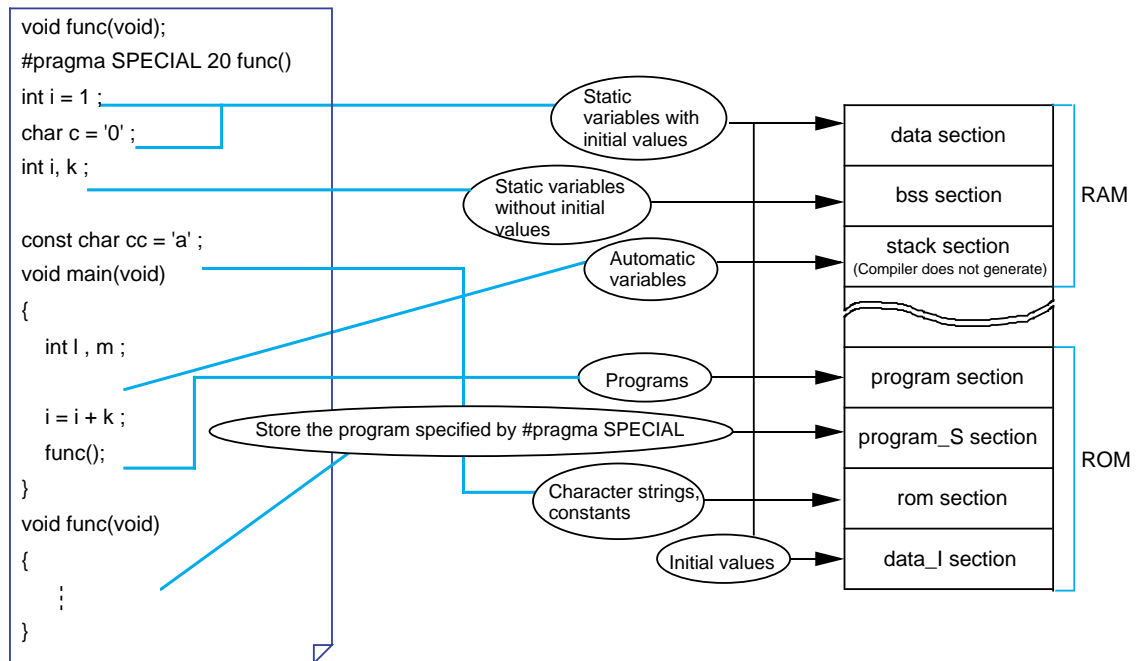
NC308 manages areas in which data and code are located as "sections". This section explains the types of sections generated and managed by NC308 and how they are managed.

#### Sections Types

NC308 classifies data into sections by type for management purposes. (See Figure 2.1.3.) Table 2.1.1 lists the sections types managed by NC308.

**Table 2.1.1 Sections Types Managed by NC308**

Section base name	Content
data	Contains static variables with initial values.
bss	Contains static variables without initial values.
rom	Contains character strings and constants.
program	Contains programs.
program_s	Store the program specified by #pragma SPECIAL.
vector	Variable vector area (compiler does not generate)
fvector	Fixed vector area (compiler does not generate)
stack	Stack area (compiler does not generate)
heap	Heap area (compiler does not generate)



**Figure 2.1.3 Mapping Data into Sections by Type**

**Sections Attributes**

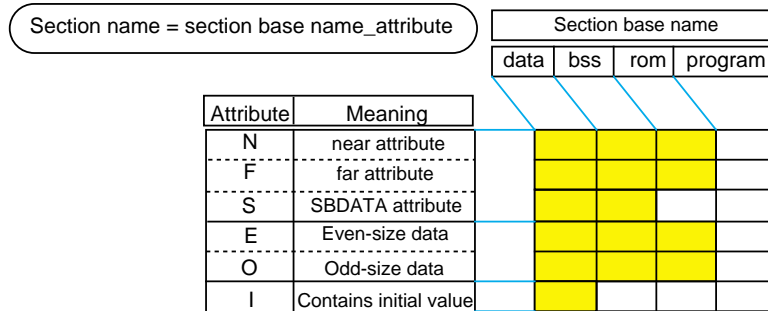
The sections generated by NC308 are further classified into smaller sections by their "attributes", i.e., whether or not they have initial value, in which area they are mapped, and their data size.  
For details on how to specify these attributes, refer to Section 2.3.1, "Efficient Addressing".  
Table 2.1.2 lists the symbols representing each attribute and its contents.

**Table 2.1.2 Sections Attributes**

Attribute	Content	Applicable section name
I	Section to hold data's initial value.	data
N/F/S	N-near attribute (64-Kbyte area at absolute addresses from 000000H to 00FFFFH)	data,bss,rom
	F-far attribute (entire 16-Mbyte memory area from address 000000H to FFFFFFFH)	
	S-SBDATA attribute (area where SB relative addressing can be used)	data,bss
E/O	E-Data size is even.	data,bss,rom
	O-Data size is odd.	

**Rule for Naming Sections**

The sections generated by NC308 are named after their section base name and attributes.  
Figure 2.1.4 shows a combination of each section base name and attributes.



**Figure 2.1.4 Rule for Assigning Section Names**

### 2.1.3 Control of Memory Mapping

NC308 provides extended functions that enable memory mapping to be performed in an efficient way to suit the user's system.

This section explains NC308's extended functions useful for memory mapping.

#### Changing Section Names (#pragma SECTION)

#pragma Δ SECTION Δ designated section base name Δ changed section base name

This function changes section base names generated by NC308. The effective range of a changed name varies between cases when "program" is changed and when some other section base name is changed.

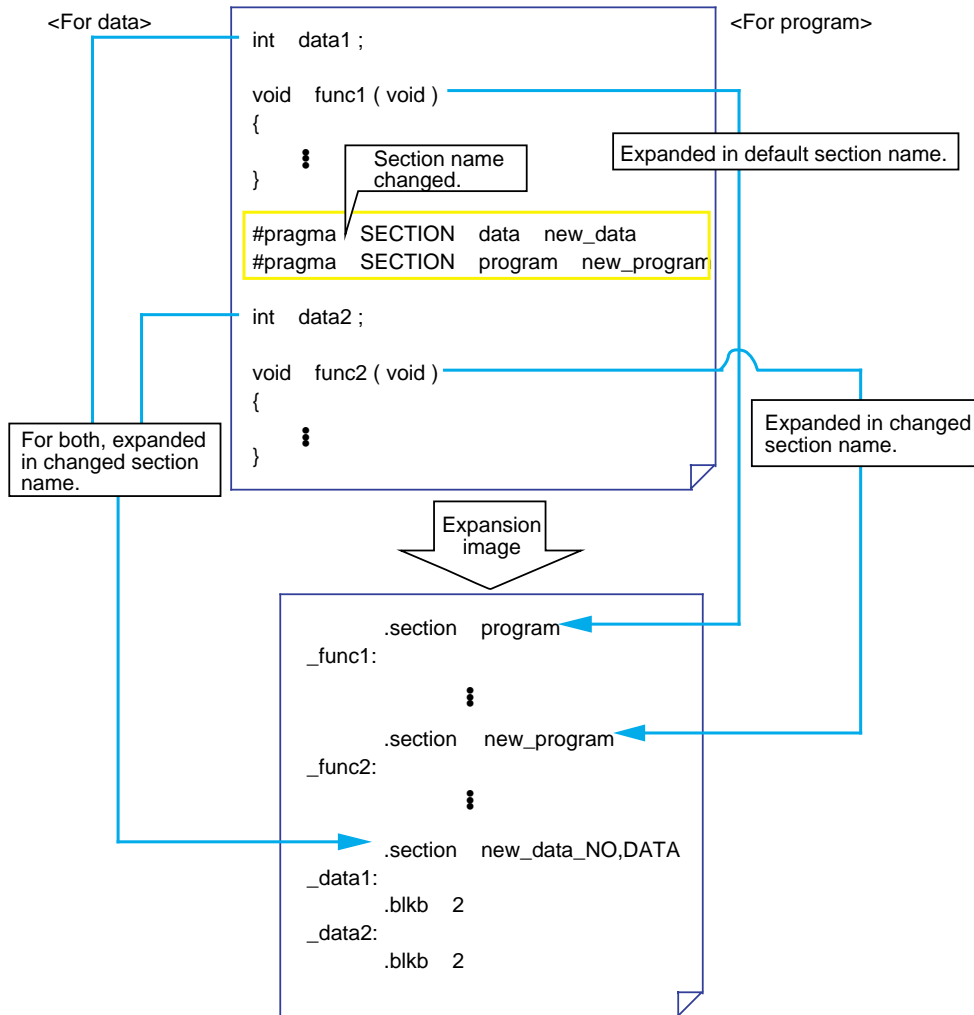


Figure 2.1.5 Typical Description of "#pragma SECTION"

### Adding Section Names ("sect308.inc")

The sections generated by NC308 are defined in the section definition file "sect308.inc."<sup>(Note)</sup> Changing a section name with #pragma SECTION means that a section base name to be generated by NC308 has been added. Therefore, when you've changed section names, always be sure to define them in the section definition file "sect308.inc."

```

;-----
; Arrangement of section
;-----
; Near RAM data area
;-----
; SBDATA area
    .section      data_SE,DATA
    .org          400H
data_SE_top:
;
    .section      bss_SE,DATA
bss_E_top:
;
    .section      data_NO,DATA
data_NO_top:
;
    .section      new_data_NO,DATA
new_data_NO_top:
;
;-----
; code area
;-----
    .section      interrupt
;
    .section      program
;
    .section      new_program
;
;-----

```

Define the new section name after being changed by #pragma SECTION .

Define the new section name after being changed by #pragma SECTION .

Figure 2.1.6 Adding Section Names ("sect308.inc")

Note: For details about the section definition file "sect308.inc," refer to Section 2.2, "Startup Program."



### Forcible Mapping into ROM (const Modifier)

Both RAM and ROM areas are allocated by writing the initial data when declaring the type of a variable. However, if this data is a fixed data that does not change during program execution, write the "const" modifier when declaring the type. Because only a ROM area is allocated and no RAM area is used, this method helps to save the amount of memory used. Furthermore, since explicit substitutions are checked when compiling the program, it is possible to check rewrite errors.

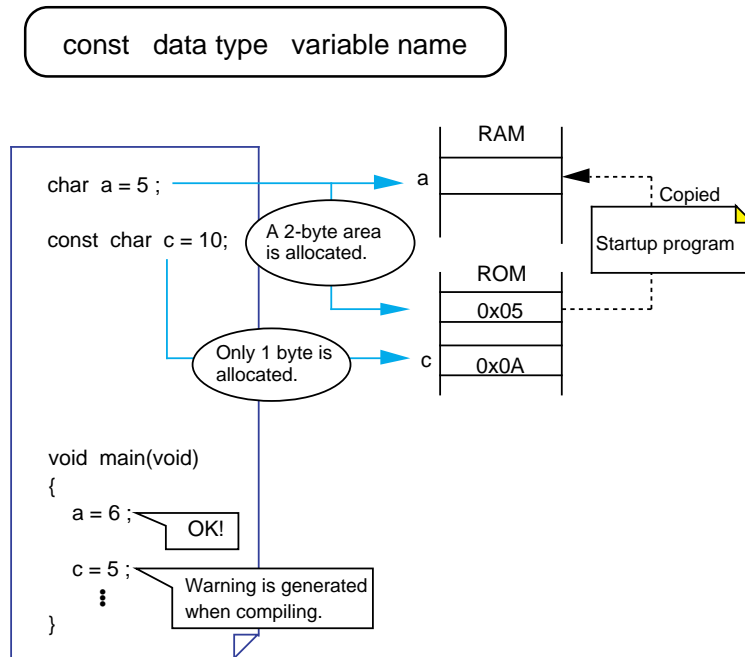


Figure 2.1.7 const Modifier and Memory Mapping

### Optimization by Replacing Referenced External Variables with Constants

When the optimization option "-Oconst" is added, referenced external variables declared by the const qualifier are replaced with constants for optimization when compiled. The external variables optimized in this way are any external variables except structures, unions, and arrays. Also, this optimization is limited to external variables for which initialization is written in the same C language source file.

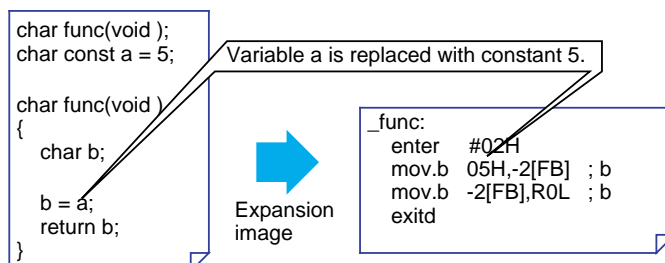


Figure 2.1.8 Optimization by Replacing Referenced External Variables with Constants

**2.1.4 Controlling Memory Mapping of Struct**

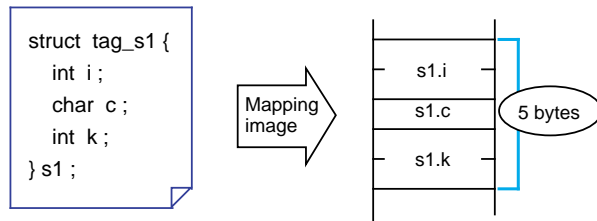
When allocating memory for structs, NC308 packs them in the order they are declared in order to minimize the amount of memory used. However, if the processing speed is more important than saving memory usage, write a statement "#pragma STRUCT" to control the method of mapping structs into memory.

This section explains NC308's specific extended functions used for mapping structs into memory.

**NC308 Rules for Mapping Structs into Memory**

NC308 follows the rules below as it maps struct members into memory.

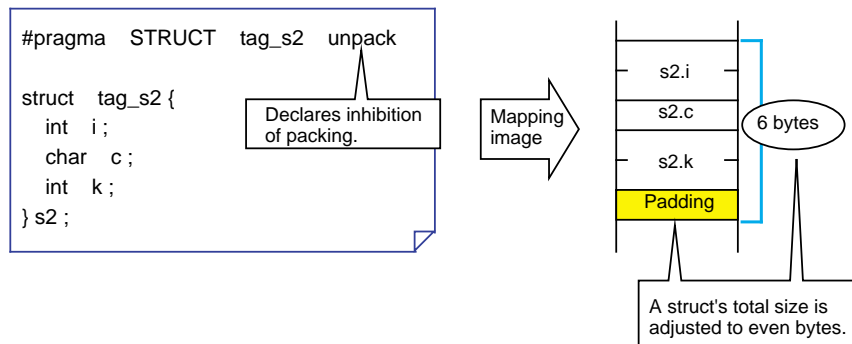
- (1) Structs are packed. No padding occurs inside the struct.
- (2) Members are mapped into memory in the order they are declared.



**Figure 2.1.9 Image Depicting How NC308's Default Struct is Mapped into Memory**

**Inhibiting Struct Members from Being Packed (#pragma STRUCT tag name unpack)**

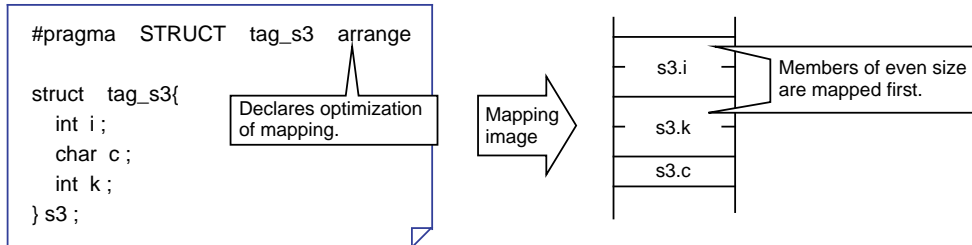
This command statement inserts pads into a struct so that its total size of struct members equals even bytes. Use this specification when the access speed has priority.



**Figure 2.1.10 Inhibiting Struct Members from Being Packed**

**Optimizing Mapping of Struct Members (#pragma STRUCT tag\_name arrange)**

This command statement allocates memory for the members of an even size before other members no matter in which order they are declared. If this statement is used in combination with the "#pragma STRUCT unpack" statement described above, each member of an even size is mapped into memory beginning with an even address. Therefore, this method helps to accomplish an efficient memory access.



**Figure 2.1.11 Optimizing Memory Allocation for Struct Members**

## 2.2 Startup Program

### 2.2.1 Roles of Startup Program

For a built-in program to operate properly, it is necessary to initialize the microprocessor and set up the stack area before executing the program. This processing normally cannot be written in the C language. Therefore, an initial setup program is written in the assembly language separately from the C language source program. This is the startup program.

The following explains the startup programs supplied with NC308, "n crt0.a30" and "sect308.inc".

### Roles of Startup Program

The following lists the roles performed by the startup program:

- (1) Allocate a stack area.
- (2) Initialize the microprocessor.
- (3) Initialize a static variable area.
- (4) Set the interrupt table register "INTB".
- (5) Call the main function.
- (6) Set the interrupt vector table.

### Structure of Sample Startup Programs

NC308's startup program consists of two files: "ncrt0.a30" and "sect308.inc".

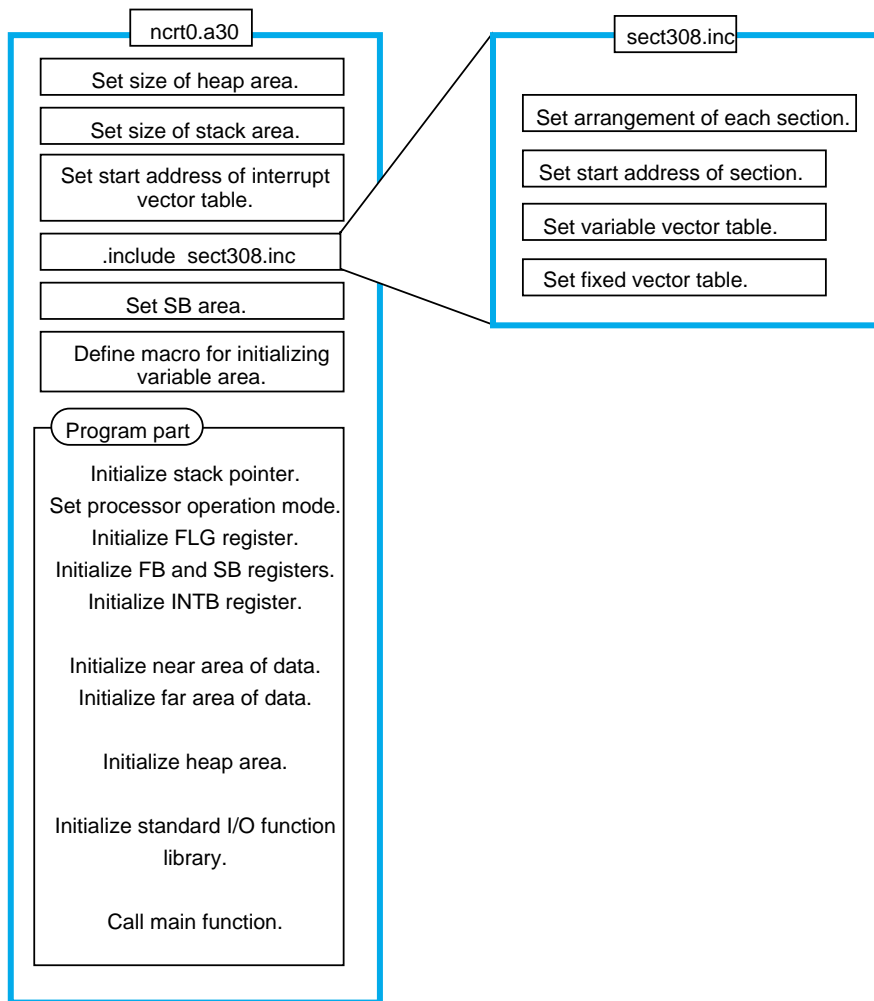


Figure 2.2.1 Structure of Sample Startup Program

### 2.2.2 Estimating Stack Sizes Used

Set an appropriate stack size in the startup program. If the stack size is excessively small, the system could run out of control. Conversely, if excessively large, it means wasting memory. This section explains how to estimate an appropriate stack size.

#### Items that Use a Stack

The following items use a stack:

- (1) Automatic variable area
- (2) Temporary area used for complex calculation
- (3) Return address
- (4) Old frame pointer
- (5) Arguments to function
- (6) Storage address when the return value is a structure or union.

#### File for Displaying Stack Sizes Used

Calculate the stack sizes used by each function. Although it can be estimated from program lists, there is a more convenient way to do it. Specify a startup option "- fshow\_stack\_usage(-fSSU)" when starting up NC308. It generates a file "xxx.stk" that contains information about the stack sizes used. However, this information does not include the stacks used by assembly language subroutine call and inline assembler. Calculate the stack sizes used for these purposes from program lists.

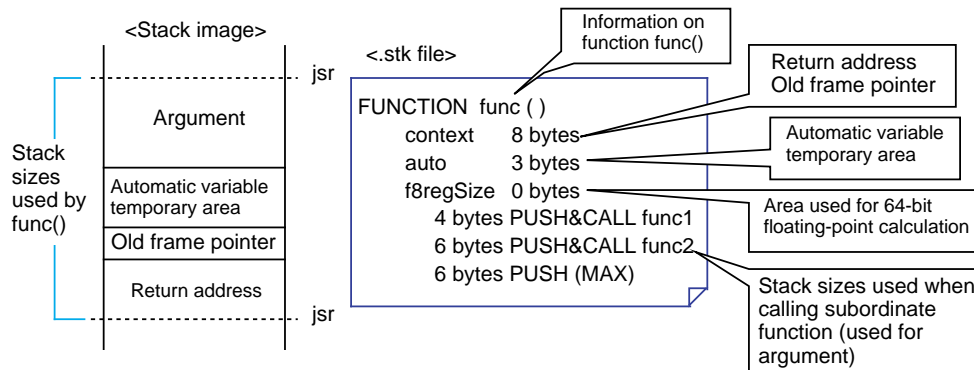


Figure 2.2.2 Stack Size Usage Information File

### Calculating the Maximum Size of Stacks Used

Find the maximum size of stacks used from the stack sizes used by each individual function after considering the relationship of function calls and handling of interrupts. Figure 2.2.3 shows by using a sample program an example of how to calculate the maximum size of stacks used.

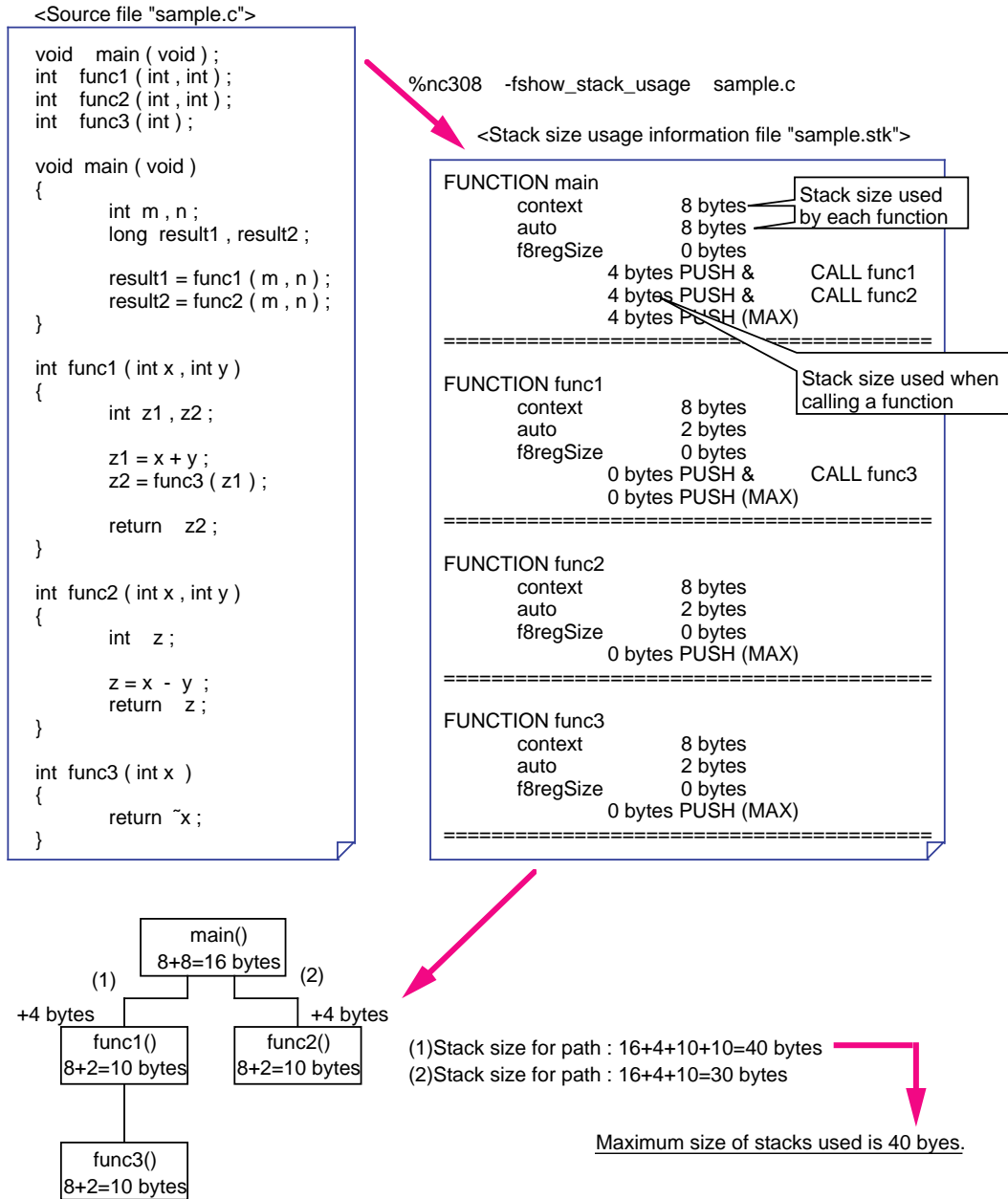


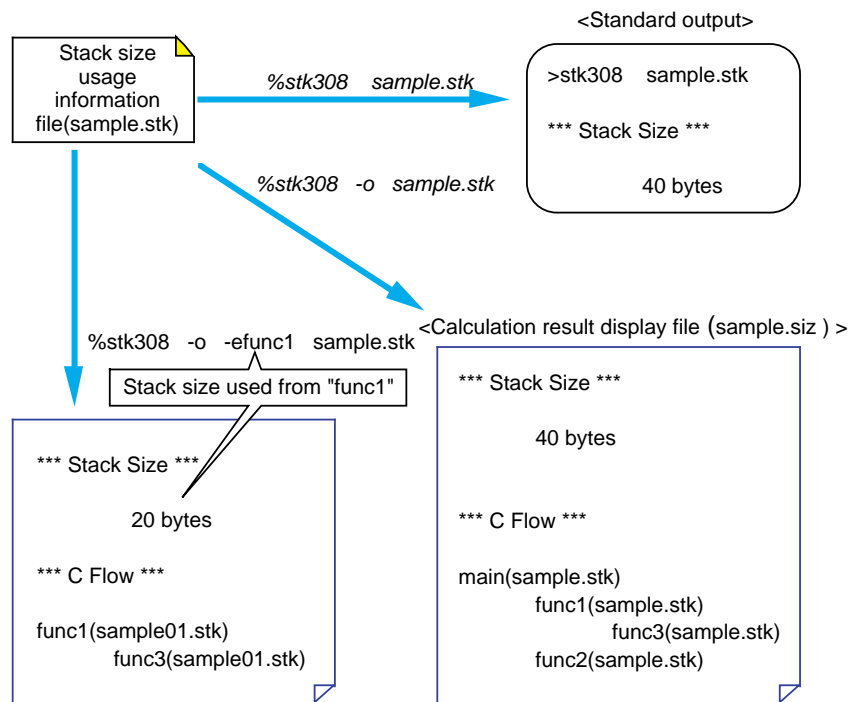
Figure 2.2.3 Method for Calculating the Maximum Size of Stacks Used

**Automatically Calculating the Maximum Size of Stacks Used**

If the program structure is simple, it is possible to estimate the stack sizes used by following the method described above. However, if the program structure is complicated or when the program uses internal functions, calculations require time and labor. In such a case, Renesas recommends using the "stack size calculating utility, stk308" that is included with NC308. It automatically calculates the maximum size of stacks used from the stack size usage information file "xxx.stk" that is made at compiling and outputs the result to standard output devices. Furthermore, if a startup option '-o' is added, it outputs the relationship of function calls along with the calculation result to a "calculation result display file ,xxx.siz".

To estimate an interrupt stack size, it is necessary to calculate the stack sizes used by each interrupt function and those used by the functions called by the interrupt function. In this case, use a startup option '-e function name'. If this startup option is used along with '-o', the stk308 utility outputs the stack sizes used below a specified function and the relationship of function calls.

Figure 2.2.4 shows the processing results of stk308 by using the sample program described above.



**Figure 2.2.4 Stack Size Calculating Utility "stk308"**



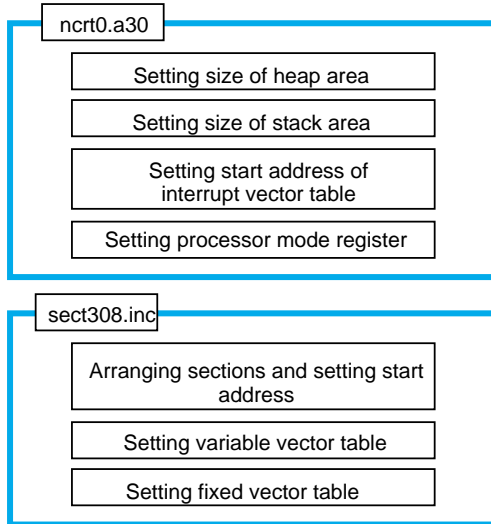
**2.2.3 Creating Startup Program**

The sample startup program shown above must be modified to suit a C language program to be created.

This section describes details on how to modify the sample startup program.

**Modifying Sample Startup Program**

Modify the following points to suit a C language program to be created:



**Figure 2.2.5 Points to Be Modified in Sample Startup Program**

### Setting the Size of Heap Area ("ncrt0.a30")

Set the required memory size to be allocated when using memory management functions (calloc, malloc). Set '0' when not using memory management functions. In this case, it is possible to prevent unwanted libraries from being linked and reduce ROM sizes by turning lines of statements initializing the heap area in "ncrt0.a30" into comments.

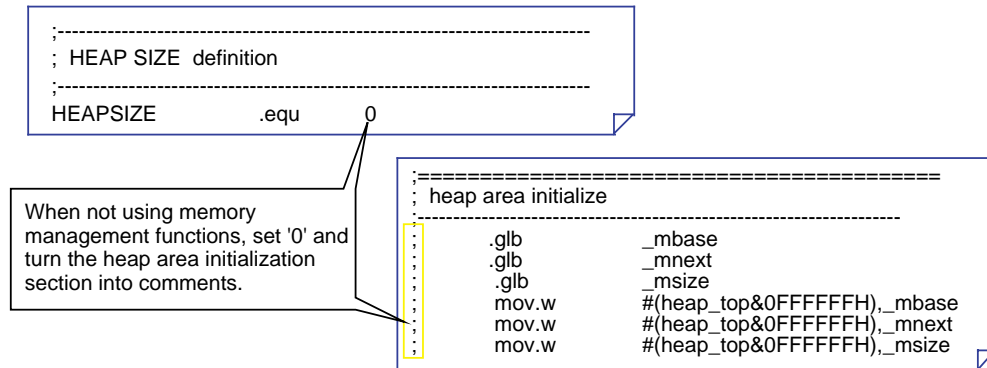


Figure 2.2.6 Setting the Heap Area

### Setting the Size of Stack Area ("ncrt0.a30")

By using the results obtained by the stack size calculating utility "stk308", etc., set the user stack and the interrupt stack sizes.

When using multiple interrupts, find the total size of interrupt stacks used for them and set it as the interrupt stack size.

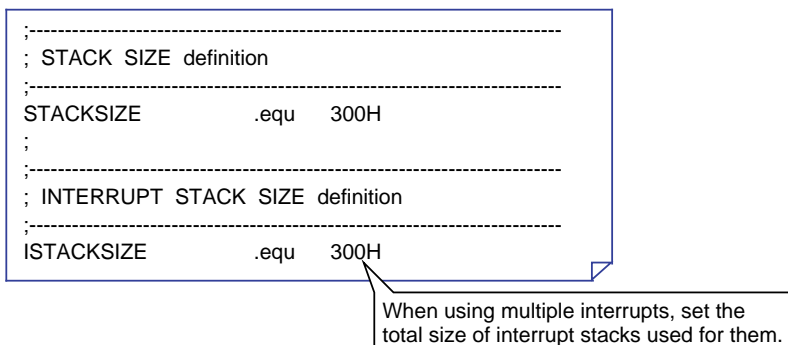


Figure 2.2.7 Setting the Stack Size

### Setting the Start Address of Interrupt Vector Table ("nrt0.a30")

Set the start address of the interrupt vector table. The value set here is set in the interrupt table register "INTB" within "nrt0.a30".

```

;-----
; INTERRUPT VECTOR ADDRESS definition
;-----
VECTOR_ADR      .equ   0FFFD00H
;-----
; interrupt section start
;-----
.glb            start
.section       interrupt
start:
;-----
; after reset , this program will start
;-----
;-----
ldc            #VECTOR_ADR,intb
;-----

```

Set in interrupt table register "INTB".

Figure 2.2.8 Setting the Start Address of Interrupt Vector Table

### Setting the Processor Operation Mode ("nrt0.a30")

Set the processor operation mode. In the same way, add the instructions here that directly controls the M16C/80 or M32C/80 operation, such as one that sets the system clock. Figure 2.2.9 shows locations where to add these instructions and how to write the instruction statements.

```

;-----
; interrupt section start
;-----
.glb            start
.section       interrupt
start:
;-----
; after reset , this program will start
;-----
;-----
ldc            #istack_top-1,isp
;-----
mov.b          #00000011B,000AH ; disable register protect
mov.b          #10000111B,0004H ; processor mode register 0
mov.b          #00001000B,0006H ; system clock control register 0
mov.b          #00100000B,0007H ; system clock control register 1
mov.b          #00000000B,000AH ; enable register protect
;-----
ldc            #0080H,flg
ldc            #stack_top-1,sp

ldc            #stack_top-1,fb
ldc            #data_SE_top,sb

ldc            #VECTOR_ADR,intb
;-----

```

After a reset, the program starts from this label.

Add settings matched to the system.

Figure 2.2.9 Setting the Processor Operation Mode

**Arranging Each Section and Setting Start Address ("sect308.inc")**

Arrange the sections generated by NC308 and set their start addresses. Use the pseudo-instruction ".org" to specify the start address of each section. If any section does not have a specified start address, memory for it is allocated in a contiguous location following the previously defined section.

```

;-----
; Arrangement of section
;-----
; Near RAM data area
;-----
; SBDATA area
    .section    data_SE,DATA
    .org       400H
data_SE_top:
;
    .section    bss_SE,DATA
bss_E_top:
    ;
;-----
; Far RAM data area
;-----
    .section    data_FE,DATA
    .org       10000H
data_FE_top:
    ;
;-----
; Far ROM data area
;-----
    .section    rom_FE,ROMDATA
    .org       0FF0000H
data_FE_top:
    ;
  
```

Specify the start address of each area in conformity with memory map.

**Figure 2.2.10 Setting the Start Address of Each Section**

### Setting the Variable VectorTable ("sect308.inc")

Add the setup items related to the variable vector table to the section definition file "sect308.inc".

Figure 2.2.11 shows an example of how to set.

```

;-----
;
;   variable vector section
;-----
;
.section      vector          ; variable vector table
.org         VECTOR_ADR
;
;
.lword      dummy_int        ; vector 0 ( BRK )
.org        ( VECTOR_ADR + 32 )
.lword      dummy_int        ; DMA0 ( software int 8 )
.lword      dummy_int        ; DMA1 ( software int 9 )
.lword      dummy_int        ; DMA2( software int 10 )
.lword      dummy_int        ; DMA3 (software int 11)
.lword      dummy_int        ; TIMER A0 ( software int 12 )
.lword      dummy_int        ; TIMER A1 (software int 13 )
.lword      dummy_int        ; TIMER A2 (software int 14 )
.lword      dummy_int        ; TIMER A3 (software int 15)
.lword      dummy_int        ; TIMER A4 (software int 16)
.lword      dummy_int        ; UART0 trance (software int 17)
.lword      dummy_int        ; UART0 receive (software int 18 )
.lword      dummy_int        ; UART1 trance ( software int 19 )
.lword      dummy_int        ; UART1 receive (software int 20 )
.lword      dummy_int        ; TIMER B0 (software int 21)
.lword      dummy_int        ; TIMER B1 ( software int 22)
.lword      dummy_int        ; TIMER B2 (software int 23)
.lword      dummy_int        ; TIMER B3 (software int 24)
.lword      dummy_int        ; TIMER B4(software int 25)
.lword      dummy_int        ; INT5 (software int 26)
.lword      dummy_int        ; INT4 (software int 27)
.lword      dummy_int        ; INT3 ( software int 28)
.lword      dummy_int        ; INT2(software int 29)
.lword      dummy_int        ; INT1 (software int 30)
.lword      dummy_int        ; INT0 ( software int 31)
.lword      dummy_int        ; TIMER B5(software int 32)
.lword      dummy_int        ; uart2 trance/NACK(software int 33)
.lword      dummy_int        ; uart2 receive/ACK(software int 34)
.lword      dummy_int        ; uart3 trance/NACK(software int 35)
.lword      dummy_int        ; uart3 receive/ACK(software int 36)
.lword      dummy_int        ; uart4 trance/NACK(software int 37)
.lword      dummy_int        ; uart4 receive/ACK(software int 38)
.lword      dummy_int        ; uart2 bus collision(software int 39)
.lword      dummy_int        ; uart3 bus collision(software int 40)
.lword      dummy_int        ; uart4 bus collision(software int 41)
.lword      dummy_int        ; AD Convert ( software int 42)
.lword      dummy_int        ; input key ( software int 43 )
.lword      dummy_int        ; software int (software int 44 )
.lword      dummy_int        ; software int (software int 45 )
.lword      dummy_int        ; software int (software int 46 )
.lword      dummy_int        ; software int (software int 47 )
.lword      dummy_int        ; software int (software int 48 )
.lword      dummy_int        ; software int (software int 49 )
.lword      dummy_int        ; software int (software int 50 )
.lword      dummy_int        ; software int (software int 51 )
.lword      dummy_int        ; software int (software int 52 )
.lword      dummy_int        ; software int (software int 53 )
.lword      dummy_int        ; software int (software int 54 )
.lword      dummy_int        ; software int (software int 55 )
.lword      dummy_int        ; software int (software int 56 )
.lword      dummy_int        ; software int (software int 57 )
.lword      dummy_int        ; software int (software int 58 )
.lword      dummy_int        ; software int (software int 59 )
.lword      dummy_int        ; software int (software int 60 )
.lword      dummy_int        ; software int (software int 61 )
.lword      dummy_int        ; software int (software int 62 )
.lword      dummy_int        ; software int (software int 63 )
; to vector 63 from vector 44 is used for MR308

```

Figure 2.2.11 Setting Variable Vector Table



**Precautions for Operating in Single-chip Mode**

When operating M16C/80 or M32C/80 in single-chip mode, note that the "near ROM" and the "far ROM" areas are not used. Delete the "nrcrt0.a30" and the "sect308.inc" blocks shown in Figure 2.2.13 or turn them into comment statements.

nrcrt0.a30: far area initialization program ("FAR area initialize")  
sect308.inc: near ROM area allocation ("Near ROM data area")  
far RAM area allocation ("Far RAM data area")

```

(" nrcrt0.a30 ")
;
;=====
; FAR area initialize.
;-----
; bss_FE & bss_FO zero clear
;-----
; BZERO      bss_FE_top,bss_FE
; BZERO      bss_FO_top,bss_FO
;-----
; Copy data_FE(FO) section from data_IFE(IFO) section
;-----
; BCOPY      data_FEI_top,data_FE_top,data_FE
; BCOPY      data_FOI,data_FO_top,data_FO
;
; ldc        #stack_top-1,sp
;
(" sect308.inc ")
;-----
; Near ROM data area
;-----
; .section    rom_NE,ROMDATA,ALIGN
; rom_NE_top:
;
; .section    rom_NO,ROMDATA
; rom_NO_top:
;-----
; Far RAM data area
;-----
; .section    data_EI,DATA
; .org        10000H
; data_FE_top:
;
; .section    bss_FE,DATA,ALIGH
; bss_FE_top:
;
; .section    data_FO,DATA
; data_FE_top:
;
; .section    bss_FO,DATA
; bss_FO_top:
;

```

Leave these lines as comments.

**Figure 2.2.13 Example for Writing Program when Operating in Single-chip Mode**

## 2.3 Extended Functions for ROM'ing Purposes

### 2.3.1 Efficient Addressing

The maximum area accessible by the M16C/80 or M32C/80 series is 16 Mbytes. NC308 divides this area into a "near area" in addresses from 000000H to 00FFFFH and a "far area" in addresses from 000000H to FFFFFFFH for management purposes.

This section explains how to arrange and access variables and functions in these areas.

#### The near and the far Areas

NC308 divides a maximum 16 Mbytes of accessible space into the "near area" and the "far area" for management purposes. Table 2.3.1 lists the features of each area.

**Table 2.3.1 near Area and far Area**

Area name	Feature
near area	This space is where the M16C/80 and M32C/80 series can access data efficiently. It is a 64-Kbyte area in absolute addresses from 000000H to 00FFFFH, in which stacks and internal RAM are located.
far area	This is the entire 1-Mbyte memory space in absolute addresses from 000000H to FFFFFFFH that can be accessed by M16C/80 and M32C/80. Internal ROM, etc. are located in this area.

#### Default near/far Attributes

NC308 discriminates the variables and functions located in the near area as belonging to the "near attribute" from those located in the far area as belonging to the "far attribute". Table 2.3.2 lists the default attributes of variables and functions.

**Table 2.3.2 Default near/far Attributes**

Classification	Attribute
Program	far, fixed
RAM data	near (However, the pointer type is far <sup>(note)</sup> )
ROM data	far
Stack data	near, fixed

If any of these default near/far attributes needs to be modified, specify the following startup options when starting up NC308:

-ffar\_RAM (-fFRAM) : Changes the default attribute of RAM data to "far".

-fnear\_ROM (-fNROM) : Changes the default attribute of ROM data to "near".

-fnear\_pointer (-fNP) : Changes the default attribute of the pointer type to "near".

Note: The size of pointer-type variables in NC308 by default are the far type of variable (4 bytes). The size of pointer-type variables in NC30 by default are the near type of variable (2 bytes).



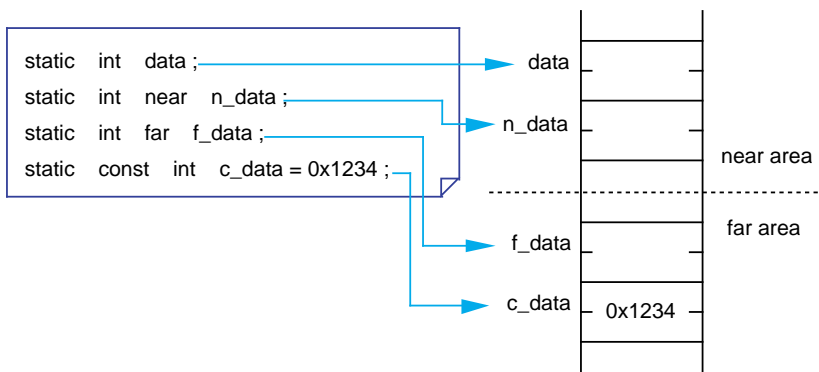
**near/far of Functions**

The attributes of NC308 functions are fixed to the far area for reasons of the M16C/80 and M32C/80 series architecture. If near is specified for an NC308 function, NC308 outputs a warning when compiling the program and forcibly locates it in the far area.

**near/far of Variables**

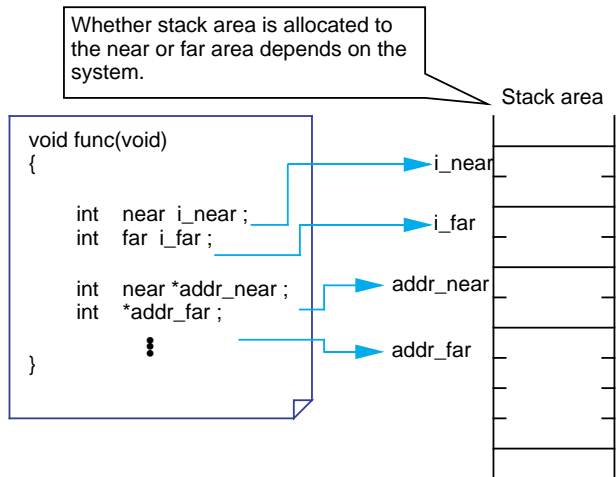
[storage class] Δ type specifier Δ near/far Δ variable name;

Unless near/far is specified when declaring type, RAM data is located in the near area, and RAM data with the const modifier specified and ROM data are located in the far area.



**Figure 2.3.1 near/far of Static Variables**

Specification of near/far for automatic variables does not have any effect at all. (All automatic variables are located in the stack area.) What is affected by this specification is only the result of the address operator '&'.



**Figure 2.3.2 near/far of Automatic Variables**

**near/far of Pointers**

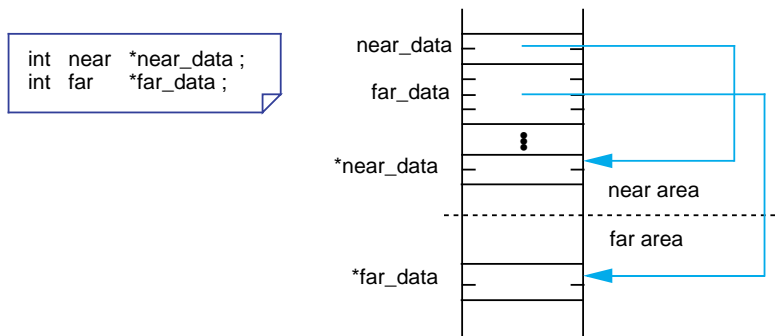
By specifying near/far for a pointer, it is possible to specify the size of addresses stored in the pointer and an area where to locate the pointer itself. If nothing is specified, all pointers are handled as belonging to the near attribute.

(1) Specify the size of addresses stored in the pointer

Handled as a 32 bits long (4 bytes) pointer variable pointing to a variable in the far area unless otherwise specified.

```
[storage class] Δ type specifier Δ near/far Δ * variable name;
```

- near → The address size stored in pointer variable is 16 bits long.
- far → The address size stored in pointer variable is 32 bits long.



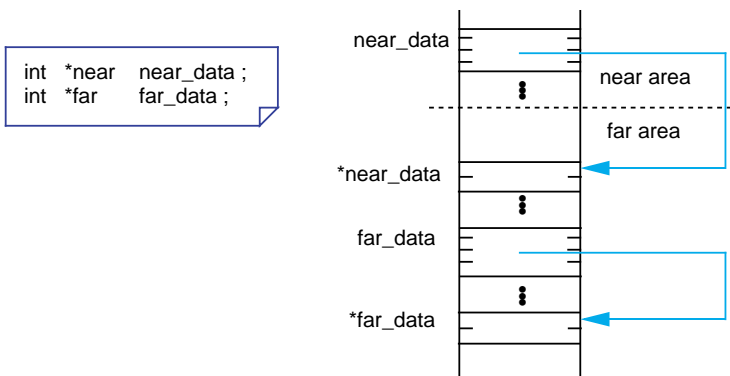
**Figure 2.3.3 Specifying Address Size Stored in Pointer**

(2) Specify the area in which to locate the pointer itself

The pointer variable itself is located in the near area unless otherwise specified.

```
[storage class] Δ type specifier Δ * near/far Δ variable name;
```

- near → Area for the pointer variable itself is located in the near area
- far → Area for the pointer variable itself is located in the far area



**Figure 2.3.4 Specifying Area to Locate the Pointer**

### Differences in near/far Specification of Pointers between NC308 and NC30

In C compiler NC30 for the M16C/60 and M16C/20 series, all pointers are handled as having the near attribute unless they are explicitly specified to be near or far. In NC308, unless pointer attribute is explicitly specified when specifying the address size to be stored in the pointer, the size of pointer variable is assumed to be 32 bits long (4 bytes) and the pointer is handled as pointing to a variable in the far area.

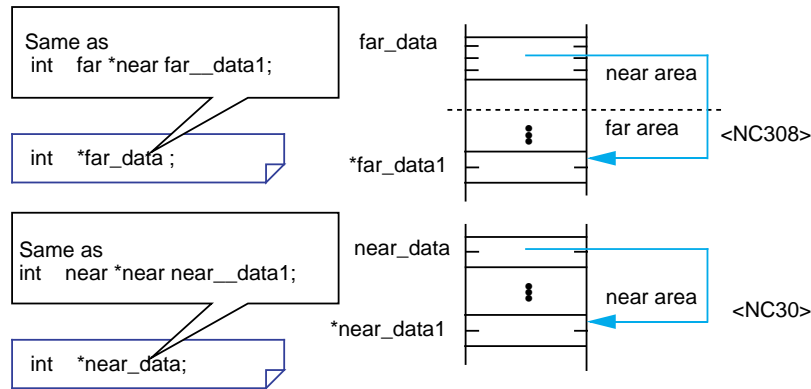


Figure 2.3.5 Differences in near/far Specification of Pointers between NC308 and NC30

### Storing the Address of a Variable Located in the far Area in a near Pointer for Address Assignment

When an attempt is made to store the address of a variable located in the far area in a near pointer for address assignment, NC308 outputs a warning message to the effect that the assignment will be performed by ignoring the upper bytes of the address. Also, the compiler outputs a warning message to the effect that the far pointer has been changed explicitly or implicitly to a near pointer.

```

void func(int near *);
int near i_near;
int far i_far;
int *addr_far;
int near *addr_near;

void main(void)
{
    addr_near = &i_far;
    addr_far = &i_far;
    addr_far = &i_near;
    addr_near = addr_far;
    addr_near = (near *)addr_far;
    func(addr_far);
}

void func(int near *ptr)
{
    ;
}
    
```

Annotations for the code above:

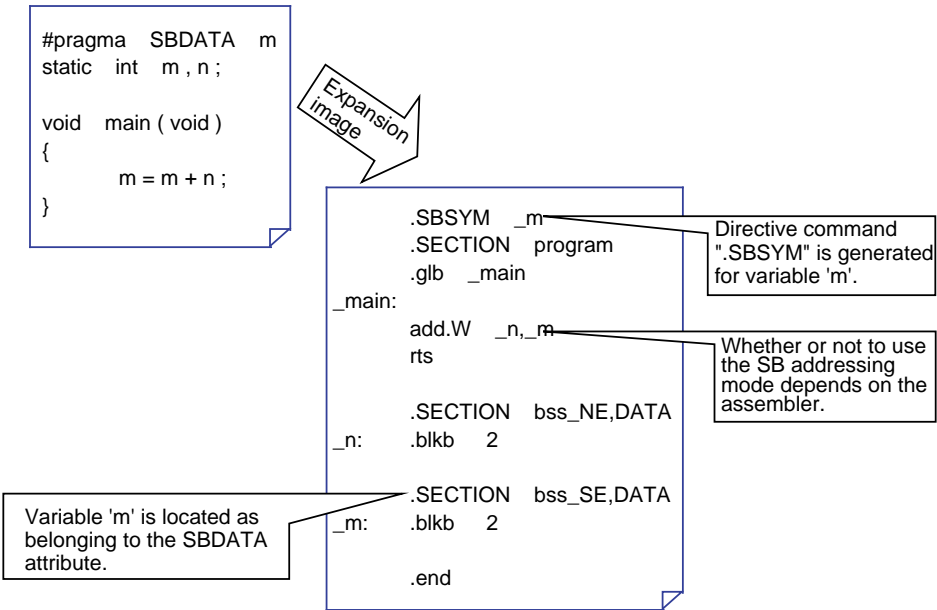
- Function that receives near pointer as parameter.
- Prepares pointer variable for far area.
- Prepares pointer variable for near area.
- Warning generated. Stored by ignoring upper address.
- OK!
- OK!
- Warning generated. Stored by ignoring upper address.
- Warning generated. far pointer explicitly changed to near pointer.
- Warning generated. far pointer implicitly changed to near pointer.

Figure 2.3.6 Storing the Address of a Variable Located in the far Area in a near Pointer for Address Assignment

**Using SB Relative Addressing (#pragma SBDATA)**

```
#pragma SBDATA variable name
```

For the variables declared in this way, NC308 generates AS308 directive command ".SBSYM" and uses the SB relative addressing mode when referencing them. This makes it possible to generate highly ROM-efficient code.



**Figure 2.3.7 Image Depicting Expansion of "#pragma SBDATA"**

### 2.3.2 Handling of Bits

NC308 allows the user to handle data in units of bits. There are two methods to use data in such a way: "bit field", an application of structs, and an extended function of NC308. This section explains each method of use.

#### Bit Field

NC308 supports a bit field as a method to handle bits. A bit field refers to using structs to assign bit symbols. The following shows the format of bit symbol assignment.

```
struct tag {
    type specifier Δ bit symbol : number of bits;
    :
};
```

When referencing a bit symbol, separate it with a period '.' when specifying it, as in the case of structs and unions.

```
variable name.bit symbol
```

Memory allocation for a declared bit field varies with the compiler used. NC308 has two rules according to which memory is allocated for bit fields. Figure 2.3.8 shows an example of actually how memory is allocated.

- (1) Allocated sequentially beginning with the LSB.
  - (2) Different type of data is located in the next address.
- (The size of the allocated area varies with each data type.)

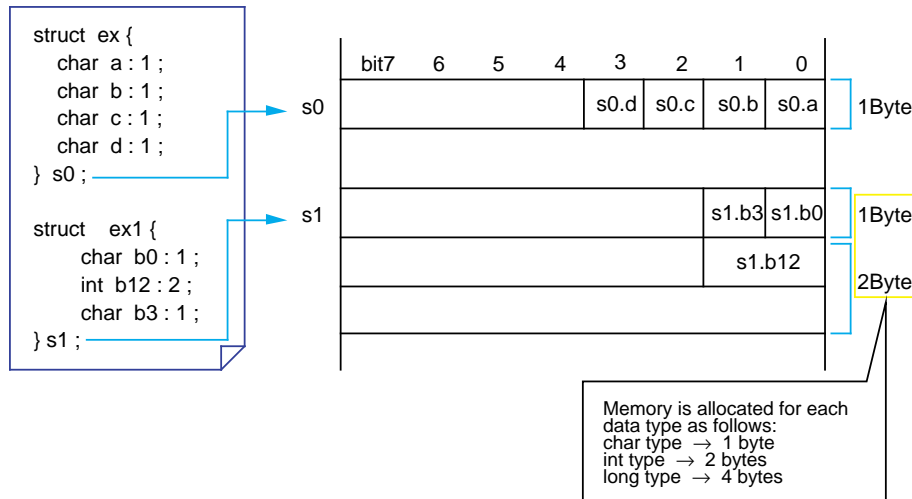


Figure 2.3.8 Example of Memory Allocation for Bit Fields

### 2.3.3 Control of I/O Interface

When controlling the I/O interface in a built-in system, specify absolute addresses for variables. There are two methods for specifying absolute addresses in NC308: one by using a pointer, and one by using an extended function of NC308. This section explains each method of specification.

#### Specifying Absolute Addresses Using a Pointer

Use of a pointer allows you to specify absolute addresses. Figure 2.3.9 shows a description example.

Example: Substituting 0xef for address 00000aH

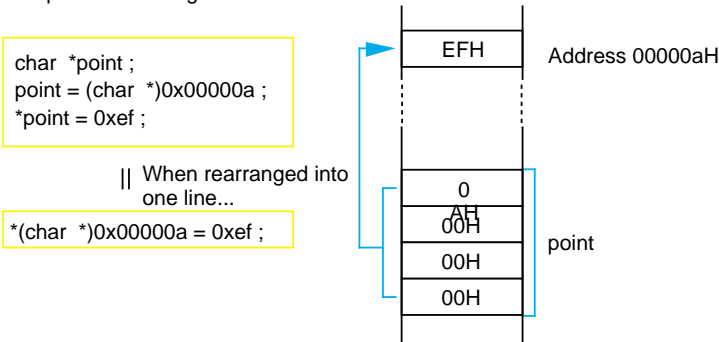
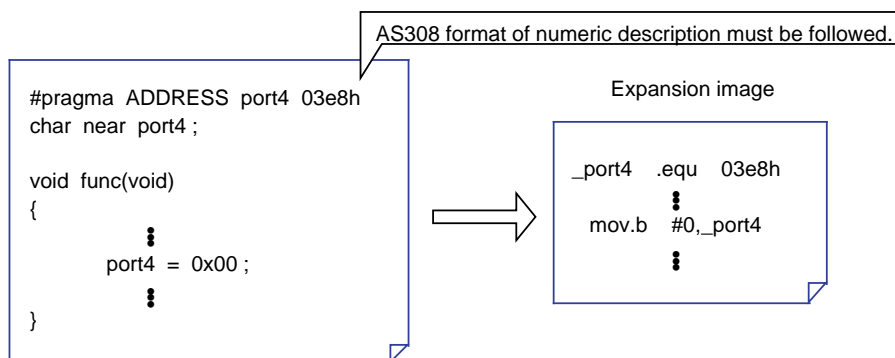


Figure 2.3.9 Specifying Absolute Addresses Using a Pointer

#### Specifying Absolute Addresses Using an Extended Function (#pragma ADDRESS)

#pragma Δ ADDRESS Δ variable name Δ absolute address

The above declaration causes a variable name to be located at an absolute address. Since this method defines a variable name as synonymous with an absolute address, there is no need to allocate a pointer variable area as required for the above method. Therefore, this method helps to save memory usage.



"#pragma ADDRESS" is effective for only variables defined outside a function.

Figure 2.3.10 Specifying Absolute Addresses Using "#pragma ADDRESS"

**Example 2.3.1 Defining SFR Area Using "#pragma ADDRESS"**

The extended function "#pragma ADDRESS" can be used to set the SFR area. For this method of SFR setting, normally prepare a separate file and include it in the source program.

The following shows one example of an SFR area definition file.

SFR area definition file <M30800.h >

```

/*-----*/
/*  type definition  */
/*-----*/
union {
    struct {
        char b0 : 1;
        char b1 : 1;
        char b2 : 1;
        char b3 : 1;
        char b4 : 1;
        char b5 : 1;
        char b6 : 1;
        char b7 : 1;
    }bit;
    unsigned char all;
}SFR;

;
/*-----*/
/*  input/output port  */
/*-----*/
#pragma ADDRESS P6 03c0h
static SFR P6;
#pragma ADDRESS P7 03c1h
static SFR P7;
#pragma ADDRESS PD6 03c2h
static SFR PD6;
#pragma ADDRESS PD7 03c3h
static SFR PD7;
#pragma ADDRESS P8 03c4h
static SFR P8;
#pragma ADDRESS P9 03c5h
static SFR P9;
#pragma ADDRESS PD8 03c6h
static SFR PD8;
#pragma ADDRESS PD9 03c7h
static SFR PD9;

;
/*-----*/
/*  Port P6 direction register bit  */
/*-----*/
#define PD6_0 PD6.bit.b0 /* P6 direction register bit0 */
#define PD6_1 PD6.bit.b1 /* P6 direction register bit1 */
#define PD6_2 PD6.bit.b2 /* P6 direction register bit2 */
#define PD6_3 PD6.bit.b3 /* P6 direction register bit3 */
#define PD6_4 PD6.bit.b4 /* P6 direction register bit4 */
#define PD6_5 PD6.bit.b5 /* P6 direction register bit5 */
#define PD6_6 PD6.bit.b6 /* P6 direction register bit6 */
#define PD6_7 PD6.bit.b7 /* P6 direction register bit7 */
;
    
```

Type declaration for bit operation.

Sets absolute addresses.

Reads in the SFR area definition file

< Source file >

```

#include "M30800.h"
;
void main ( void )
{
;
    P6.all = 0x00 ;
    PD6_0 =1;
;
}
    
```

Reference SFR area,for bitwise access.

Reference SFR area,for bitwise access.

**Example 2.3.1 Defining SFR Area Using "#pragma ADDRESS"**

### 2.3.4 Using Inline Assembly

There are some cases where hardware-related processing cannot be written in the C language. This occurs when, for example, processing cannot be finished in time or when one wishes to control the C flag directly. To solve this problem, NC308 allows you to write the assembly language directly in C language source programs ("inline assemble" function). There are two inline assemble methods: one using the "asm" function, and one using "#pragma ASM". This section explains each method.

#### Writing Only One Line in Assembly Language (asm Function)

asm ("character string")

When the above line is entered, the character string enclosed with double quotations (") is expanded directly (including spaces and tabs) into the assembly language source program. Since this line can be written both in and outside a function, it will prove useful when one wishes to manipulate flags and registers directly or when high speed processing is required.

Figure 2.3.11 shows a description example.

```
void main ( void )
{
    initialize() ;
    asm("  FSET  I");
    ;
}
```

Sets interrupt enable flag.

Figure 2.3.11 Typical Description of asm Function

#### Accessing Automatic Variables in Assembly Language (asm Function)

When it is necessary to access automatic variables inside the function, write a statement using "\$\${FB}" as shown in Figure 2.3.12. Since the compiler replaces "\$\$" with the FB register's offset value, automatic variable names in the C language can be used in assembly language programs.

```
void main ( void )
{
    unsigned int m ;
    m = 0x07 ;
    asm("  MOV.W  $$[FB],R0",m) ;
}
```

Defines automatic variable 'm'.

FB offset value of 'm' is -2.

FB relative addressing is used

<Expansion image>

```
_main:
    enter    #02H
    mov.w   #0007H,-2[FB] ; m
    ;### ASM START
    MOV.W   -2[FB],R0
    ;### ASM END
    exitd
```

<Format>

asm ( "assembly language", automatic variable name );

Figure 2.3.12 Using Automatic Variables in asm Function



### Writing Entire Module in Assembly Language (#pragma ASM)

If the embedded assembly language consists of multiple lines, use an extended function "#pragma ASM". With this extended function, NC308 determines a section enclosed with "#pragma ASM" and "#pragma ENDASM" to be an area written in the assembly language and outputs it to the assembly language source program directly as it is.

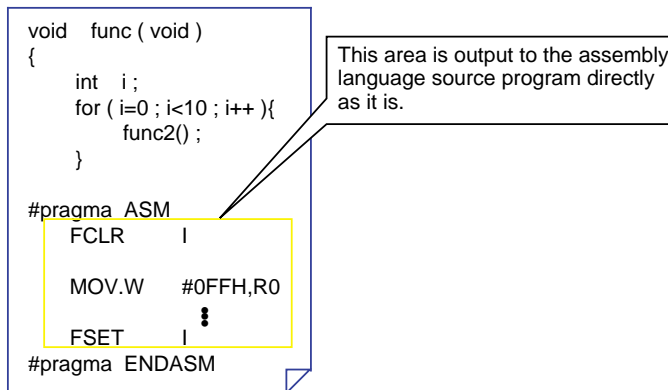


Figure 2.3.13 Example for Using "#pragma ASM" Function

### Suppressing Optimization Partially by Using asm Function

When the startup option '-O' is added, NC308 optimizes generated code when compiling the program. However, if this optimization causes inconveniences such as when an interrupt occurs, NC308 allows you to suppress optimization partially by using the asm function. Similarly, by specifying the optimization option "-One\_bit-(ONB)," it is possible to suppress optimizing bit manipulations into a single instruction. Figure 2.3.14 shows an example for using the asm function for this purpose.

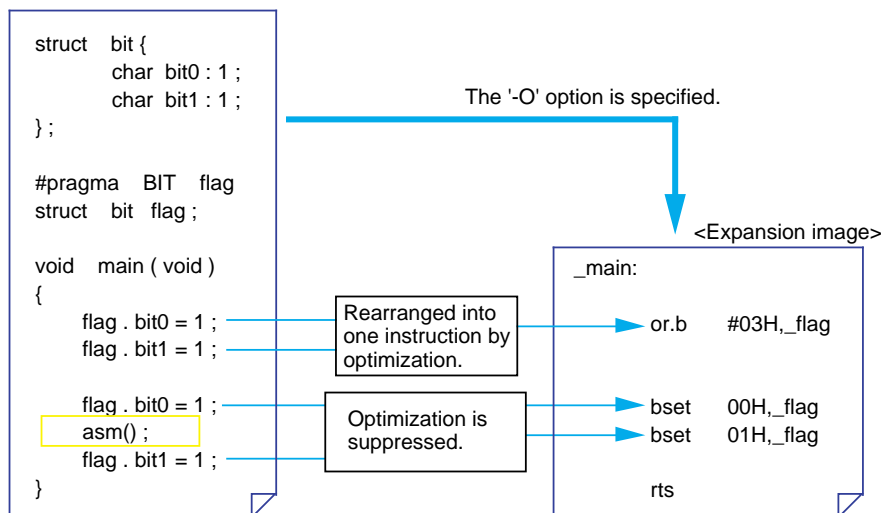


Figure 2.3.14 Suppressing Optimization Partially by Using asm Function

### 2.3.5 Using Assembler Macro Functions

NC308 allows part of assembly language instructions to be written as C language functions, which are known as "assembler macro functions."

In ordinary C language description, the assembly language instructions that NC308 does not expand can be written directly in a C language program. This helps the program to be tuned up easily.

This section explains how to write assembler macro functions and an example for using the assembler macro functions.

#### Assembly Language Instructions that Can Be Written Using Assembler Macro Functions (1)

In NC308, a total of 18 assembly language instructions can be written using assembler macro functions.

Assembler macro function names represent assembly language instructions in lowercase letters. The bit lengths referenced during operation are expressed by "\_b," "\_w," and "\_l." Tables 2.3.3 and 2.3.4 list the assembly language instructions that can be written using assembler macro functions.

**Table 2.3.3 Assembly Language Instructions that Can Be Written Using Assembler Macro Functions (1)**

Assembly language instruction	Assembler macro function name	Function	Format
DADD	dadd_b	Returns result of decimal addition of val1 and val2.	char dadd_b(char val1, char val2);
	dadd_w		int dadd_w(int val1, int val2 );
DADC	dadc_b	Returns result of decimal addition of val1 and val2 with carry.	char dadc_b(char val1, char val2);
	dadc_w		int dadc_w(int val1, int val2);
DSUB	dsub_b	Returns result of decimal subtraction of val1 and val2.	char dsub_b(char val1, char val2 );
	dsub_w		int dsub_w(int val1, int val2 );
DSBB	dsbb_b	Returns result of decimal subtraction of val1 and val2 with borrow.	char dsbb_b(char val1, char val2 );
	dsbb_w		int dsbb_w(int val1, int val2 );
RMPA	rmpa_b	Returns result of multiply/accumulate operation indicating initial value by int, count by count, and start addresses of multiplier locations by p1 and p2.	long rmpa_b(long init, int count, char *p1, char *p2);
	rmpa_w		long rmpa_w(long init, int count,int *p1, int *p2);
MAX	max_b	Returns selected maximum values of val1 and val2 as the result.	char max_b(char val1, char val2 );
	max_w		int max_w(int val1, int val2 );
MIN	min_b	Returns selected minimum values of val1 and val2 as the result.	char min_b(char val1, char val2 );
	min_w		int min_w(int val1, int val2 );

## Assembly Language Instructions that Can Be Written Using Assembler Macro Functions (2)

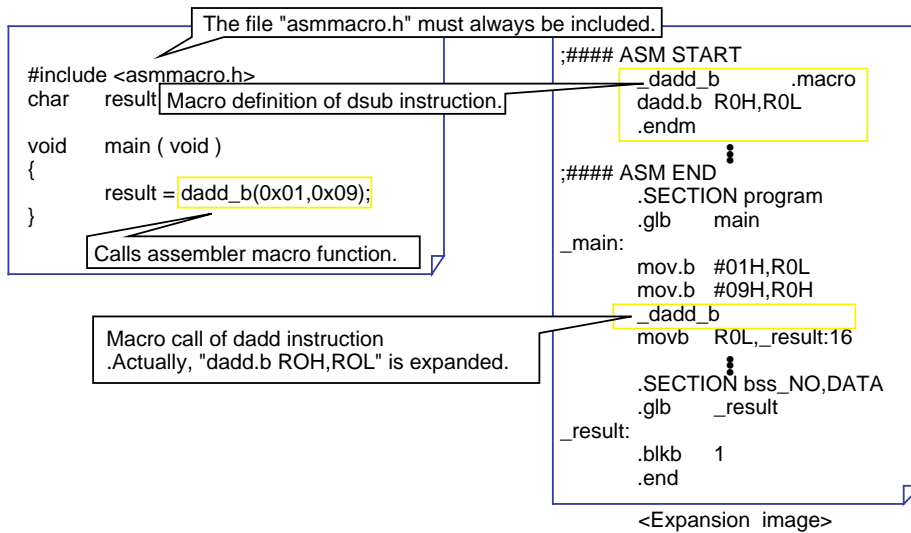
**Table 2.3.4 Assembly Language Instructions that Can Be Written Using Assembler Macro Functions (2)**

Assembly language instruction	Assembler macro function name	Function	Format
SMOVB	smovb_b	Transfers string in reverse direction from transfer address p1 to transfer address p2 as many times as indicated by count.	void smovb_b(char *p1, char *p2, unsigned int count );
	smovb_w		void smovb_w(int *p1, int *p2, unsigned int count );
SMOVF	smovf_b	Transfers string in forward direction from transfer address p1 to transfer address p2 as many times as indicated by count.	void smovf_b(char *p1, char *p2, unsigned int count );
	smovf_w		void smovf_w(int *p1, int *p2, unsigned int count );
SMOVU	smovu_b	Transfers string in forward direction from transfer address p1 to transfer address p2 as many times as indicated by count until 0 is detected.	void smovu_b(char *p1, char *p2 );?
	smovu_w		void smovu_w(int *p1, int *p2 );?
SIN	sin_b	Transfers string in forward direction from fixed transfer address p1 to transfer address p2 as many times as indicated by count.	void sin_b(char *p1, char *p2, unsigned int count );
	sin_w		void sin_w(int *p1, int *p2, unsigned int count );
SOUT	sout_b	Transfers string in forward direction from transfer address p1 to transfer address p2 as many times as indicated by count .	void sout_b(char *p1, char *p2, unsigned int count );
	sout_w		void sout_w(int *p1, int *p2, unsigned int count );
SSTR	sstr_b	Stores string with the data to be stored indicated by val, transfer address by p, and transfer count by count.	void sstr_b(char val, char *p, unsigned int count );
	sstr_w		void sstr_w(int val, int *p, unsigned int count );
ROLC	rolc_b	Returns val after rotating it left by 1 bit including carry as the result.	unsigned char rolc_b(unsigned char val );
	rolc_w		unsigned int rolc_w(unsigned int val );
RORC	rorc_b	Returns val after rotating it right by 1 bit including carry as the result.	unsigned char rorc_b(unsigned char val );
	rorc_w		unsigned int rorc_w(unsigned int val );
ROT	rot_b	Returns val after rotating it as many times as indicated by count as the result.	unsigned char rot_b(signed char val, unsigned int count );
	rot_w		unsigned int rot_w(signed char val, unsigned int count );
SHA	sha_b	Returns val after arithmetically rotating it as many times as indicated by count as the result.	unsigned char sha_b(signed int count, unsigned char val );
	sha_w		unsigned int sha_w(signed int count, unsigned int val );
	sha_l		unsigned long sha_l(signed int count, unsigned long val );
SHL	shl_b	Returns val after logically rotating it as many times as indicated by count as the result.	unsigned char shl_b(signed int count, unsigned char val );
	shl_w		unsigned int shl_w(signed int count, unsigned int val );
	shl_l		unsigned long shl_l(signed int count, unsigned long val );

**Decimal Additions Using Assembler Macro Function "dadd\_b"**

When using the assembler macro functions of NC308 after calling them in your program, make sure the assembler macro function definition file "asmmacro.h" is included in the program.

Example 2.3.2 below shows an example of a decimal addition performed by using assembler macro function "dadd\_b."



**Example 2.3.2 Decimal Additions Using Assembler Macro Function "dadd\_b"**

String Transfers Using Assembler Macro Function "smovf\_b"

An example of a string transfer performed by using assembler macro function "smovf\_b" is shown in Example 2.3.3 below.

```

#include <asmmacro.h>
char src_string[] = "ABCDEFGH";
char dest_string[8];

void main ( void )
{
    smovf_b( src_string, dest_string, sizeof(src_string));
}

##### ASM START
    .smovf_b .macro
    pushm R3,A0,A1
    smovf.b
    popm R3,A0,A1
    .endm
##### ASM END
    .SECTION program
    .glob main
    _main:
    pushm R3,A0,A1
    mov.w #(_dest_string&0FFFFH),A1
    mov.w #(_src_string&0FFFFH),A0
    .smovf_b
    popm R3,A0,A1
    rts
#####
    .SECTION data_NE,DATA
    .glob _src_string
    _src_string:
    .blkb 8

    .SECTION data_NE1,ROMDATA
    .byte 41H ;'A'
    .byte 42H ;'B'
    .byte 43H ;'C'
#####
    .SECTION bss_NE,DATA
    .glob _dest_string
    _dest_string:
    .blkb 8
    .end
    
```

The file "asmmacro.h" must always be included.

Macro definition of smovf instruction.

Calls assembler macro function.

Actually, the following  
pushm R3,A0,A1  
smovf.b  
popm R3,A0,A1  
is expanded.

< Expansion image >

Example 2.3.3 String Transfers Using Assembler Macro Function "smovf\_b"

**Multiply/accumulate Operations Using Assembler Macro Function "rmpa\_w"**

An example of a multiply/accumulate operation performed by using assembler macro function "rmpa\_w" is shown in Example 2.3.4 below.

```

#include <asmmacro.h>
int str1[10] = {0,1,2,3,4,5,6,7,8,9};
int str2[10] = {0,1,2,3,4,5,6,7,8,9};
long result;

void main ( void )
{
    result = rmpa_w(0, 9, str1, str2);
}

##### ASM START
rmpa_w .macro
pushm R1,R3,A1,A0
mov.w #00H,R1
rmpa.w
popm R1,R3,A1,A0
.endm
##### ASM END
.SECTION program
.glob main
_main:
mov.w #(_str2&0FFFFH),A1
mov.w #(_str1&0FFFFH),A0
mov.w #0009H,R3
mov.l #00000000H,R2R0
rmpa_w
mov.l R2R0,_result:16
#####
.SECTION data_NE,DATA
.glob _str1
_str1:
.blkb 20
.glob _str2
_str2:
.blkb 20
#####
.SECTION data_NEI,ROMDATA
.word 0000H
.word 0001H
.word 0002H
.word 0003H
#####
.SECTION bss_NO,DATA
.glob _result
_result:
.blkb 4
.end
    
```

The file "asmmacro.h" must always be included.

Macro definition of rmpa instruction.

Initial value.

Number of times operation is performed.

Calls assembler macro function.

Macro call of rmpa instruction. Actually, pushm R1,R3,A1,A0 mov.w #00H,R1 rmpa.w popm R1,R3,A1,A0 is expanded.

< Expansion image

**Example 2.3.4 Multiply/accumulate Operations Using Assembler Macro Function "rmpa\_w"**

Note: The macro assembler function definition file "asmmacro.h" is stored in the standard directory that has been set by NC308's environment variable "INC308."

## 2.4 Linkage with Assembly Language

### 2.4.1 Interface between Functions

When the module size is small, inline assemble is sufficient to solve the problem. However, if the module size is large or when using an existing module in the program, NC308 allows you to call an assembly language subroutine from the C language program or vice versa.

This section explains interfacing between functions in NC308.

### Entry and Exit Processing of Functions

The following lists the three primary processings performed in NC308 when calling a function:

- (1) Construct and free stack frame
- (2) Transfer argument
- (3) Transfer return value

Figure 2.4.1 shows a procedure for these operations.

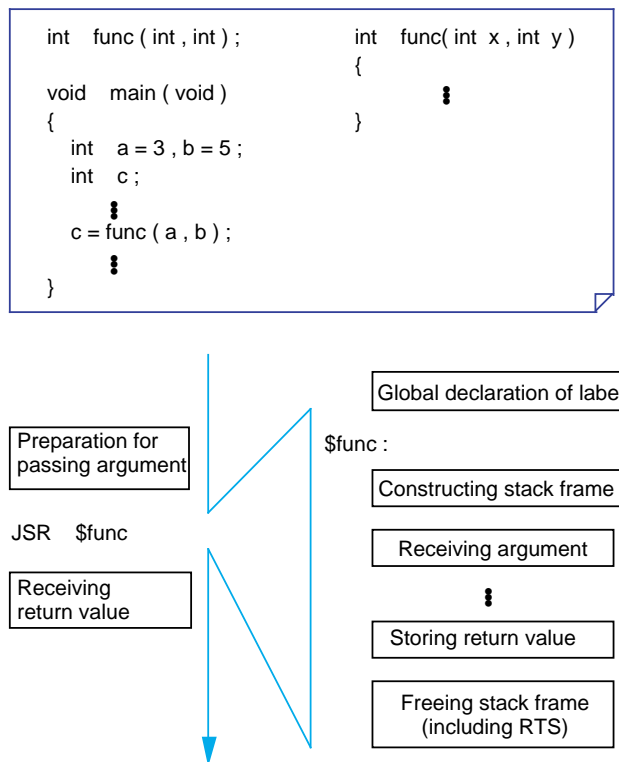


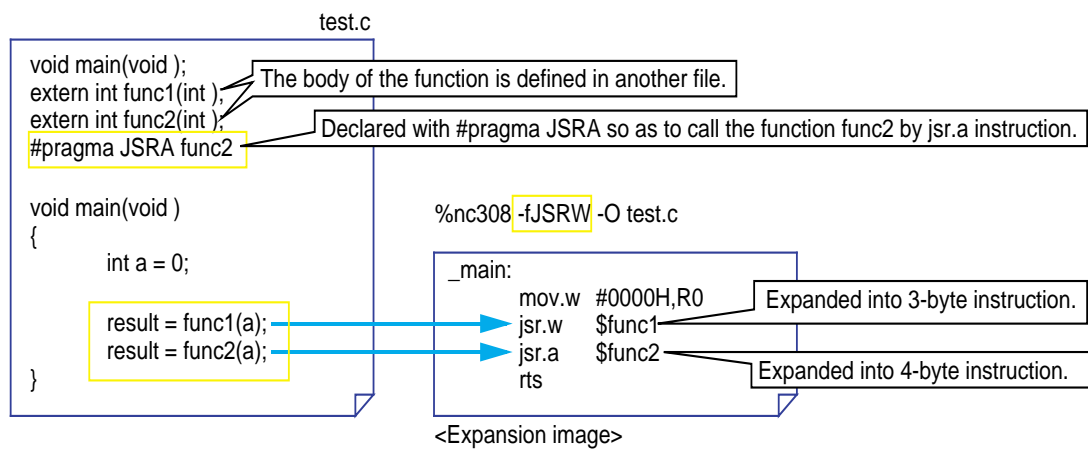
Figure 2.4.1 Operations for Calling a Function

### Method for Compressing ROM Size during Function Call

When calling a function defined in another file from NC308, the instruction is expanded into a jsr.a instruction (comprised of 4 bytes) before calling the function. Therefore, even when calling the function that exists within 64 Kbytes (-32768 to +32767), the instruction is expanded in full address specification (3 bytes of operand). This results in poor ROM efficiency.

If the generated code change option "-fJSRW" is specified when compiling, the function call instruction can be expanded into a jsr.w instruction (comprised of 3 bytes) before calling the function. This helps to compress the ROM size.

If an error occurs when linking (could not be reached by 16-bit relative), the function can be declared with "#pragma JSRA" so that the instruction is expanded into a jsr.a instruction before calling the function. This helps to handle function calls in a large program exceeding 64 Kbytes. Therefore, if your program is within 64 Kbytes, we recommend using the generated code change option "-fJSRW."



**Figure 2.4.2 Method for Compressing ROM Size during Function Call**



### Structure of a Stack Frame

When a function is called, an area like the one shown below is created in a stack. This area is called a "stack frame".  
The stack frame is freed when control returns from the called function.

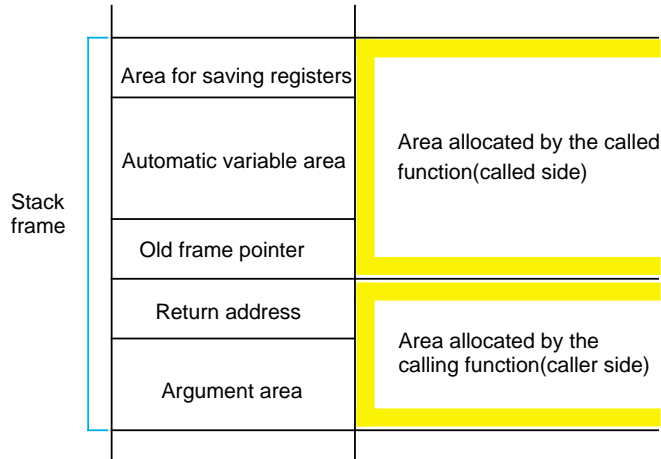


Figure 2.4.3 Structure of a Stack Frame

### Constructing a Stack Frame

Figure 2.4.4 shows how a stack frame is constructed by tracing the flow of a C language program.

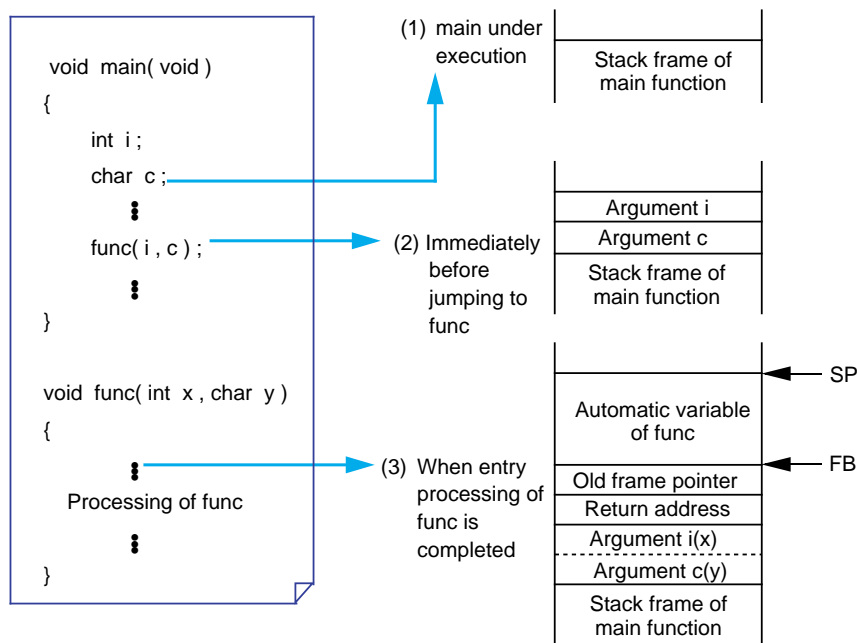


Figure 2.4.4 Constructing a Stack Frame

### Rules for Passing Arguments

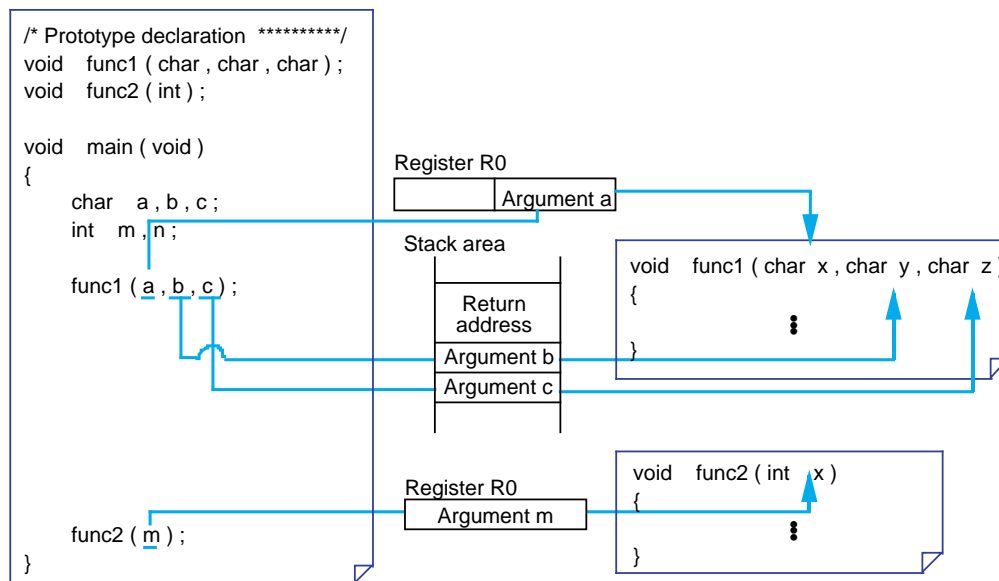
NC308 has two methods for passing arguments to a function: "via a register" and "via a stack".

When the following three conditions are met, arguments are passed via a register; otherwise, arguments are passed via a stack.

- (1) The types of the function's arguments are prototype declared.
- (2) The types of argument match those of the arguments listed in Table 2.4.1.
- (3) No variable arguments are used in the argument part of prototype declaration.

**Table 2.4.1 Rules for Passing Arguments**

Type of argument	First argument	Second and following arguments
char type	R0L	Stack
short, int types near pointer type	R0	Stack
Other types	Stack	Stack



**Figure 2.4.5 Example for Passing Arguments to Functions**

### Rules for Passing Return Values

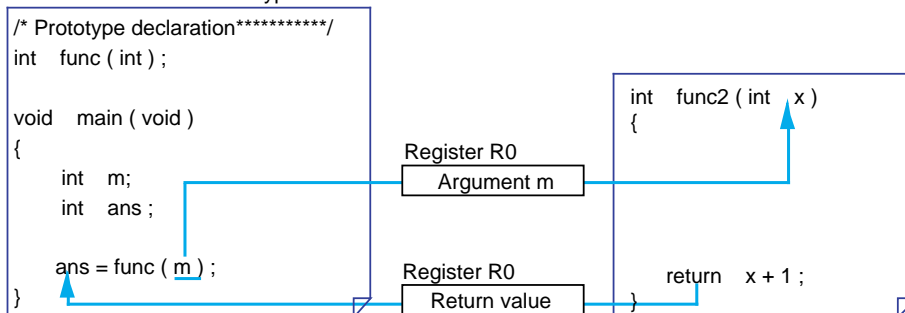
All return values except those expressed by a struct or union, are stored in registers. However, different registers are used to store the return values depending on their data types.

The return values represented by a struct or union are passed via "stored address and stack". Namely, an area to store a return value is prepared when calling a function, and this address is passed via a stack as a hidden argument. The called function writes its return value to the area indicated by the address placed in the stack when control returns from it.

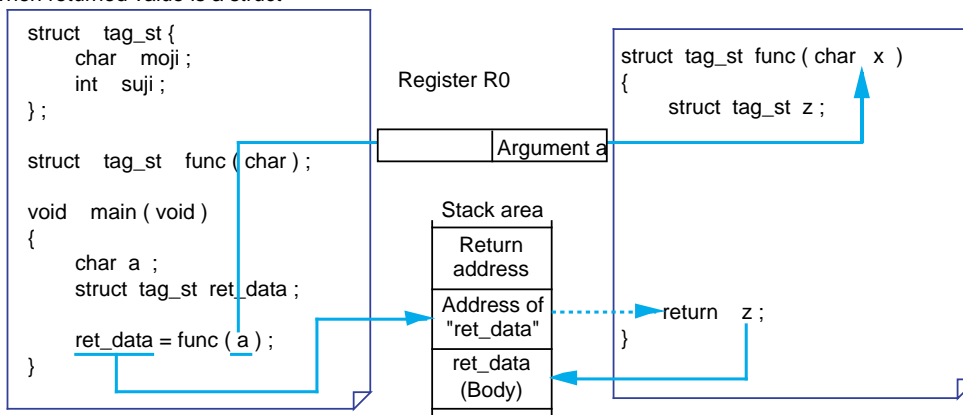
**Table 2.4.2 Rules for Passing Return Value**

Data type	Returning method
char	R0L
int short	R0
long float	R2R0
double	R3R2R1R0
near pointer	R0
far pointer	R2R0
struct union	Store address is passed via a stack

- When returned value is a int type



- When returned value is a struct



**Figure 2.4.6 Example for Passing Return Value**

## Rules for Symbol Conversion of Functions into Assembly Language

In NC308, the converted symbols differ depending on the properties of functions. Table 2.4.3 lists the rules for symbol conversion.

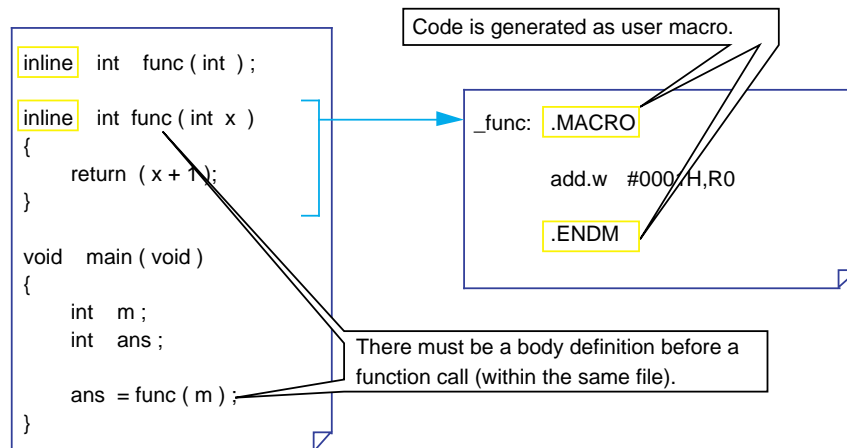
**Table 2.4.3 Rules for Symbol Conversion**

Function type	Conversion method
Arguments passed via register	Functions are prefixed with "\$".
Arguments passed via stack No argument #pragma INTERRUPT[B/E/F] #pragma PARAMETER[C]	Functions are prefixed with "_".

## A Measure for Calling Functions Faster

A function call requires stack manipulation for the return values and arguments to be passed from a function to another. This takes time before the actual processing can be performed. Consequently, the via-register transfer reduces the time required for procedures from calling to processing, because it involves less stack manipulation than the other method.

To reduce this time difference further, NC308 has an "inline storage class" available. For functions that have been specified to be of the inline storage class, code is generated as macro functions when compiling, and code is expanded directly in the spot from which the function is called. As a result, the function call time and ordinary stack operation are eliminated, helping to increase the execution speed<sup>(note)</sup>.



**Figure 2.4.7 Example for Writing Inline Storage Class**

Note: Although the execution speed increases, the ROM efficiency degrades if this is used for frequently called functions, because code is generated in all spots from which the function is called.

**2.4.2 Calling Assembly Language from C Language**

This section explains details on how to write command statements for calling an assembly language subroutine as a C language function.

**Passing Arguments to Assembly Language (#pragma PARAMETER)**

```
#pragma Δ PARAMETER Δ function name (register name, ...)
```

A function that is written as shown above sets arguments in specified registers without following the ordinary transfer rules as it performs call-up operation. Use of this facility helps to reduce the overhead during function call because it does not require stack manipulation for argument transfers. However, the following precautions must be observed when using this facility:

- (1) Before writing "#pragma PARAMETER", be sure to prototype declare the specified function.
- (2) Observe the following in prototype declaration:
  - Always make sure that function arguments are the char type, int type, long type, float type, near pointer type, or far pointer type. The structure type, union type, and double type cannot be declared.
  - Structs and unions cannot be declared as a function return value.
  - Make sure the register sizes and argument sizes are matched.
  - Register names are not discriminated between uppercase and lowercase.
  - If the body of a function specified with this #pragma command is defined in the C language, an error results.

```
void asm_func ( int , int );
#pragma PARAMETER asm_func ( R0 , R1 )
void main ( void )
{
  int i , j ;
  asm_func ( i , j );
}
```

Be sure to declare the assembler function's prototype before declaring #pragma PARAMETER.

Following can be used as register names:  
 R0, R1, R2, R3,  
 R0L, R0H, R1L, R1H,  
 A0, A1  
 Note, however, that arguments are passed to a function via these registers.

Argument i and argument j are stored in R0 and R1, respectively when calling a function.

**Figure 2.4.8 Example for Writing #pragma PARAMETER**

### Calling Assembly Language Subroutine

Follow the rules described below when calling an assembly language subroutine from a C language program.

- (1) Write the subroutine in a file separately from the C language program.
- (2) Follow symbol conversion rules for the subroutine name.
- (3) In the C language program on the calling side, declare the prototype of subroutine (assembly language function). At this time, declare it to be externally referenced by using the storage class specifier "extern."
- (4) When accessing automatic variables using FB relative addressing in the subroutine (assembly language function), declare them with ".fb" and set the value 0.
- (5) In the subroutine (assembly language function), do not normally change the values of the B flag, U flag, SB register, and FB register that are used exclusively by NC308. If any of these values need to be changed, save the values to the stack at entry to the function and restore them from the stack at exit from the function.

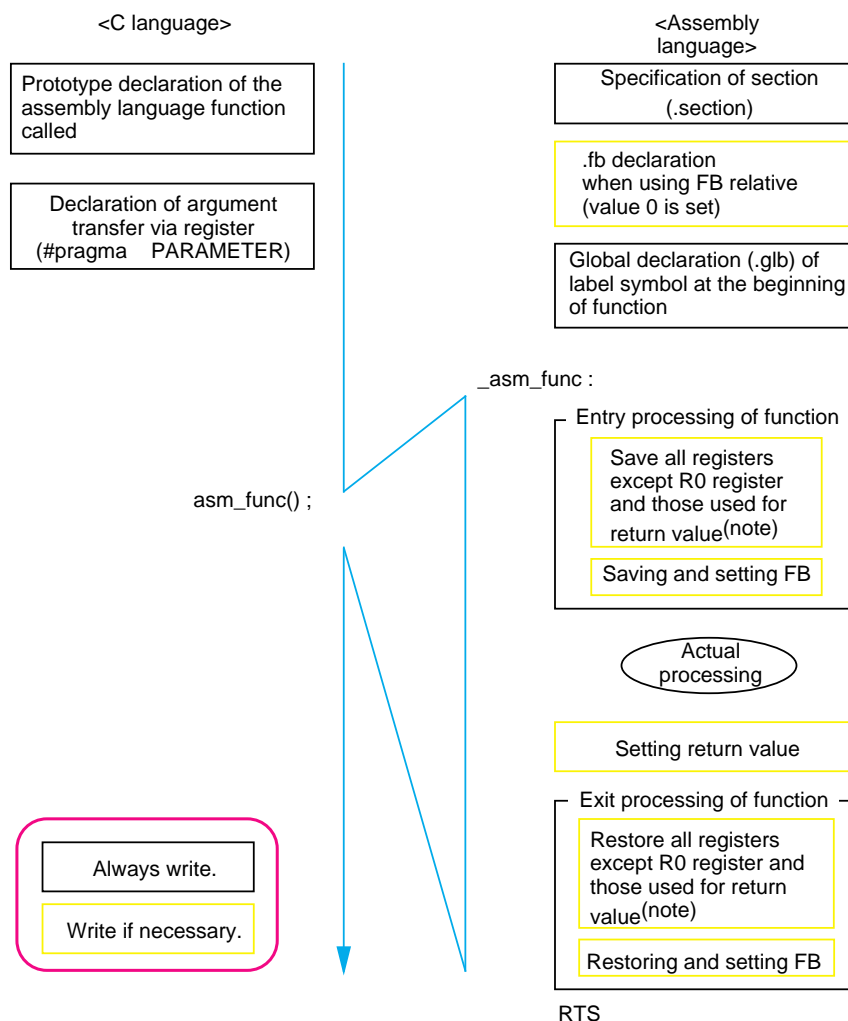


Figure 2.4.9 Calling Assembly Language Subroutine

Note: If R0 register or the register used for return value need to be saved, NC308 generates code necessary to save them on the function calling side (caller side). Therefore, R0 register and the register used for return value do not need to be saved or restored in a program.

### Difference in Calling Conventions between NC308 and NC30

When calling functions, NC30 saves registers on the function calling side (by the caller), while NC308 does this on the called function side. For this reason, when calling assembly language functions from a C language function, care must be taken not to destroy the contents of registers used in the C language function. To this end, always be sure to write processing statements to save the registers which will be destroyed in the function (e.g., any registers other than R0 register and the register used for return value) at entry to the assembly language function by using the PUSHM instruction and restore the saved registers at exit from the assembly language function by using the POPM instruction. To call functions following the same calling conventions as in NC30, write #pragma PARAMETER/C (with the switch "/C" added) in your program to ensure that when calling functions, registers are saved on the function calling side (by the caller). The same switch "/C" is also available for #pragma SPECIAL, which can be used to have the calling sequence in NC308 matched to that of NC30.

Figure 2.4.10 shows an example of a #pragma PARAMETER/C description.

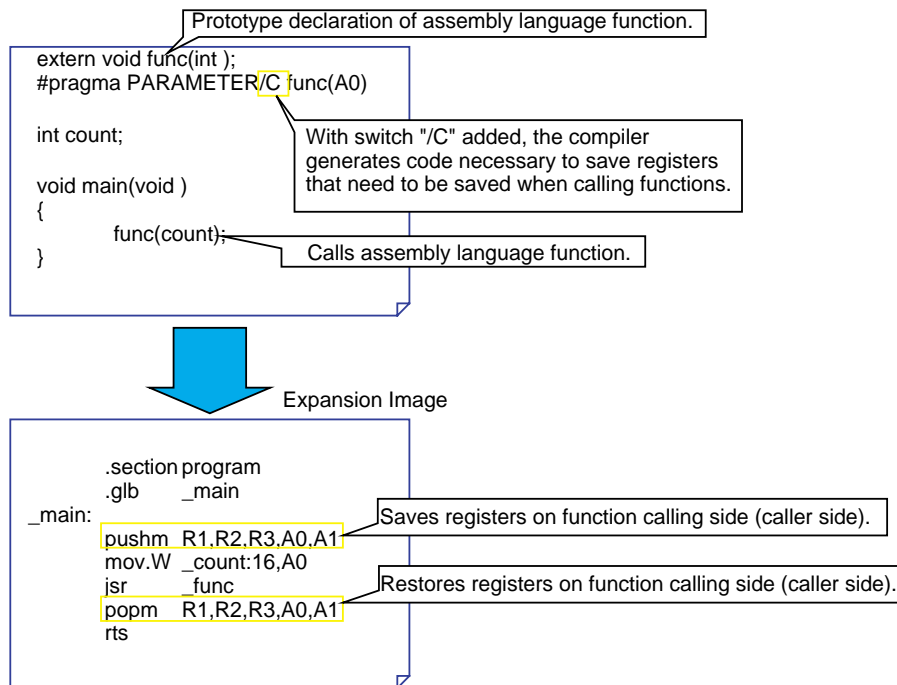
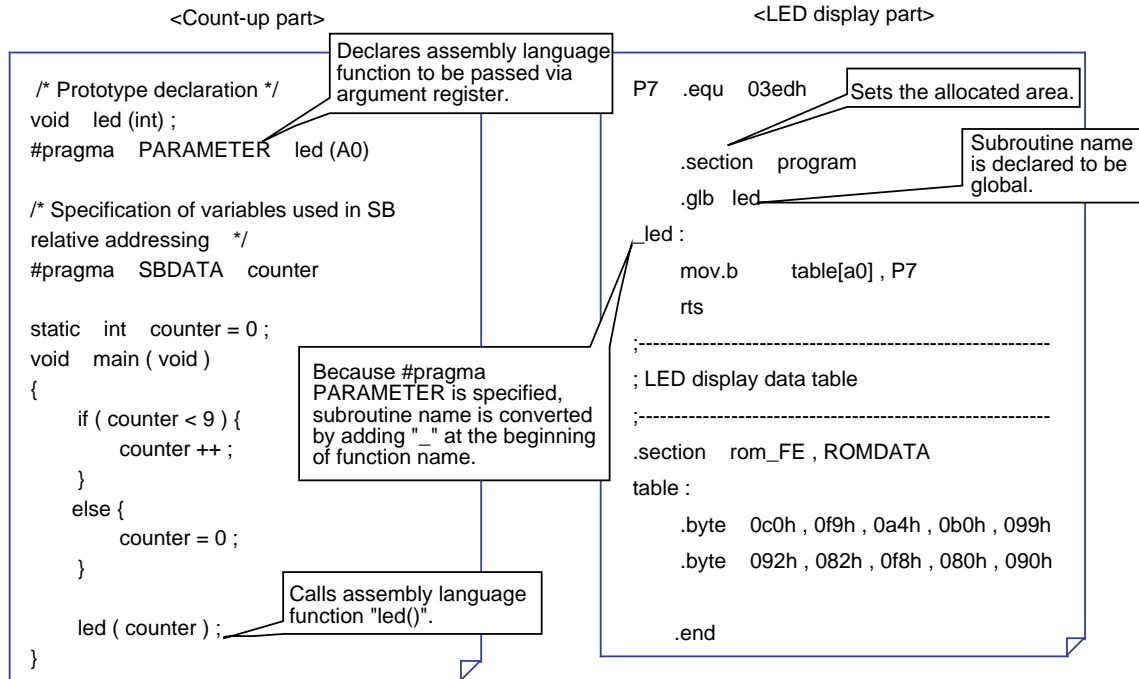


Figure 2.4.10 Example of #pragma PARAMETER/C Description

**Example 2.4.1 Calling Subroutine**

The program in this example displays count-up results using LEDs. The LED display part is written in the assembly language and the count-up part is written in the C language. Then the two parts are linked.



**Example 2.4.1 Calling Subroutine**



### Calling Special Page Subroutine (#pragma SPECIAL[/C])

#pragma Δ SPECIAL[/C] Δ calling number Δ function name ()

A function call written like the above is expanded into a jsrs instruction (special page subroutine call instruction). Use of this facility helps to reduce the number of instruction bytes when calling frequently called functions<sup>(Note)</sup>. Following precautions must be observed when using this facility:

- (1) Before writing #pragma SPECIAL, be sure to declare the prototype of function.
- (2) The functions declared by #pragma SPECIAL are located in the program\_S section. Therefore, in the startup program, locate the program\_S section in an area from address F00000H to address FFFFFFFH.
- (3) Calling numbers can only be specified in decimal, in the range of 18 to 255.
- (4) A label "\_\_SPECIAL\_calling number:" is output for the start address of a function declared by #pragma SPECIAL. Therefore, the output label name of this function must be set in the special page vector table.

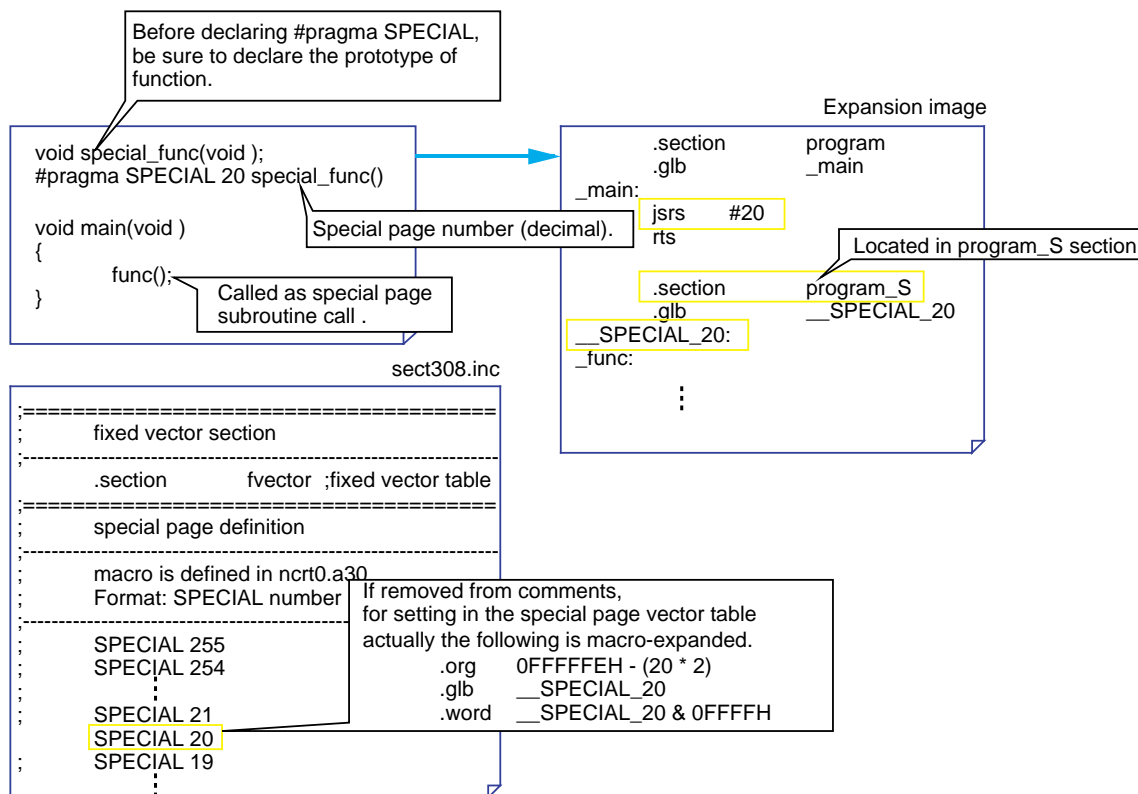


Figure 2.4.11 Calling Special Page Subroutine (#pragma SPECIAL[/C])

Note: When using the jsrs instruction, note that the number of instruction bytes required is 2. Because another 2 bytes are required for the special page vector table, it may prove effective to use #pragma SPECIAL declaration for functions that are called three times or more.

### Calling a Subroutine by Indirect Addressing

Normally an instruction "jsr" is generated for calling an assembly language subroutine from the C language. To call a subroutine by indirect addressing using "jsri", use a "function pointer". However, when using a function pointer, note that no registers can be specified for argument transfers by "#pragma PARAMETER". Figure 2.4.12 shows a description example.

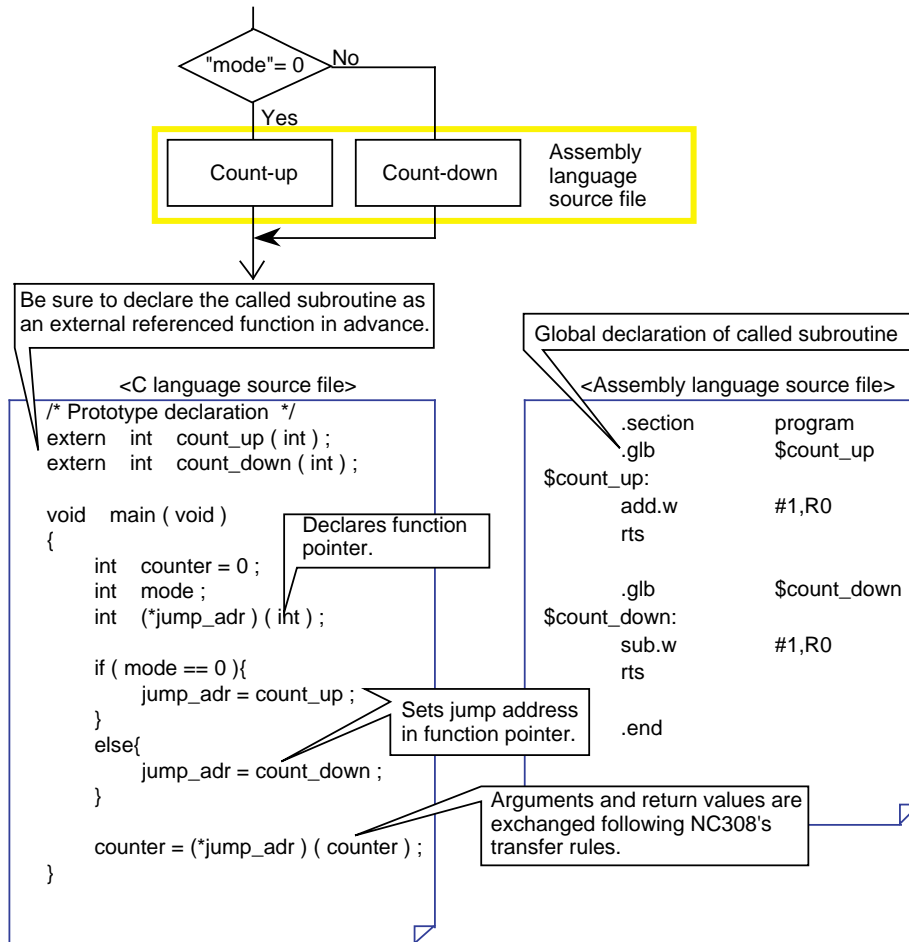
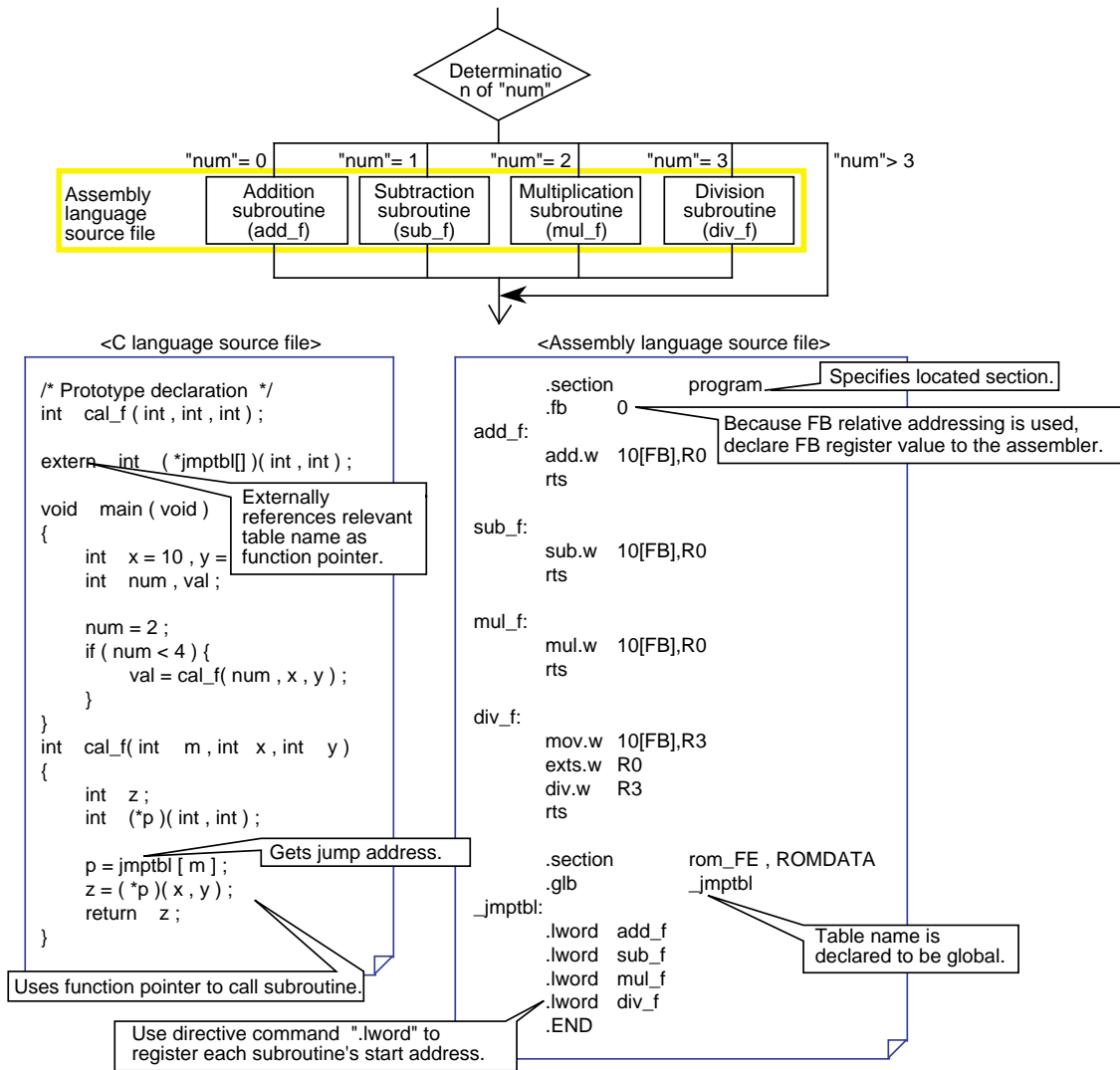


Figure 2.4.12 Calling a Subroutine by Indirect Addressing

**Example 2.4.2 Calling a Subroutine by Table Jump**

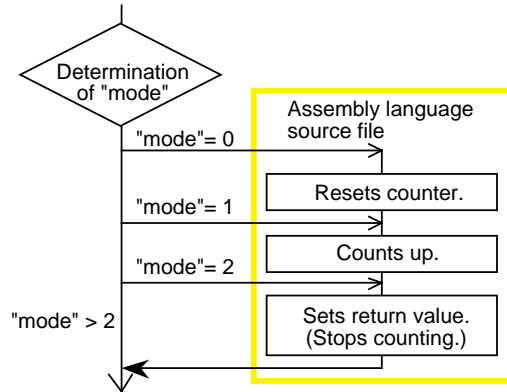
The program in this example calls different subroutines from a C language program according to the value of "num". In cases where multiple branches are involved like in this example, use of table jump makes it possible to call any desired subroutine in the same processing time. However, no registers can be specified for argument transfers by "#pragma PARAMETER".



**Example 2.4.2 Calling a Subroutine by Table Jump**

**Example 2.4.3 A Little Different Way to Use Table Jump**

Once the internal labels of a subroutine are registered in a jump table, NC308 allows you to change the start address of the subroutine depending on the mode. Since multiple processings can be implemented by a single subroutine, this method helps to save ROM capacity.



```

<C language source file>

/* Prototype declaration */
int clock ( int , int );

extern int ( *clock_mode [ ] ) ( int );

void main ( void )
{
    int mode ;
    int counter = 0 ;

    mode = 2 ;
    if ( mode < 3 ) {
        counter = clock( mode , counter ) ;
    }
}

int clock( int m , int x )
{
    int z ;
    int ( *p ) ( int ) ;

    p = clock_mode [ m ] ;
    z = ( *p ) ( x ) ;
    return z ;
}
    
```

```

<Assembly language source file>

.section    program
reset:
    mov.w #0FFFFH,R0

count:
    add.w #1,R0
stop:
    rts

.section    rom_FE,ROMDATA
.glb
_clock_mode:
.lword    reset
.lword    count
.lword    stop
.END

Registers internal labels of
subroutine in jump table.
    
```

**Example 2.4.3 A little Different Way to Use Table Jump**

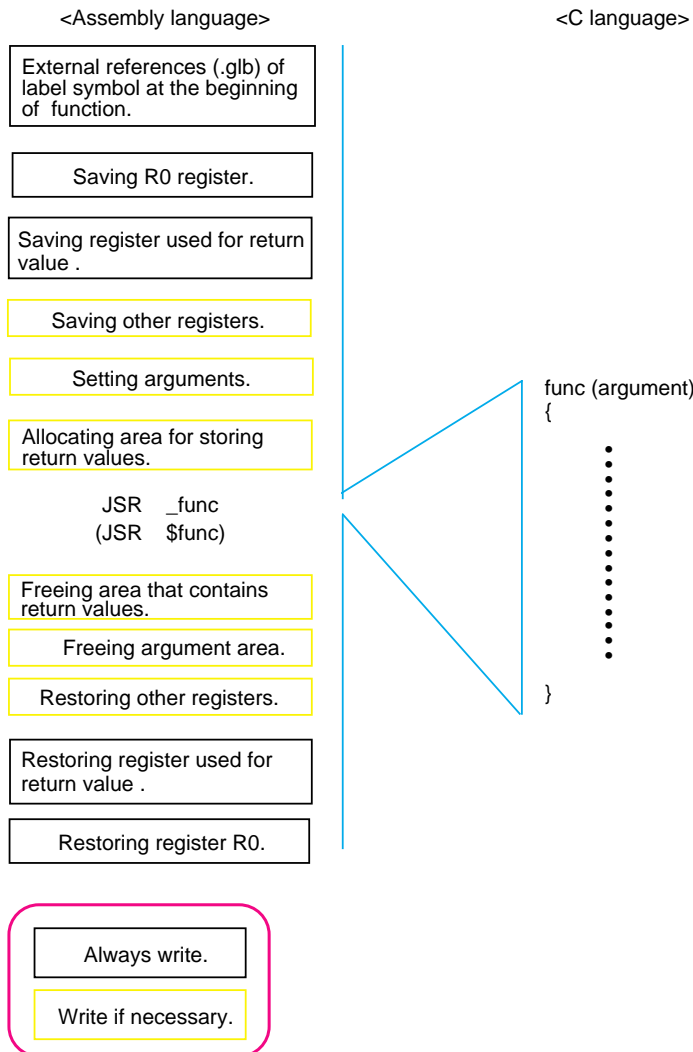
**2.4.3 Calling C Language from Assembly Language**

This section explains how to call a C language function from an assembly language program.

**Calling a C Language Function**

Follow the rules described below when calling a C language function from an assembly language program.

- (1) Follow NC308's symbol conversion rules for the labels of the called subroutine.
- (2) Write the C language function in a file separately from the assembly language program.
- (3) In the assembly language file, declare external references using AS308's directive command ".glb" before calling the C language function.
- (4) C language functions do not save R0 register and the register used for return value as part of processing at entry to the function. For this reason, always be sure to save R0 register and the register used for return value before calling C language functions.



**Figure 2.4.13 Calling C Language Function**

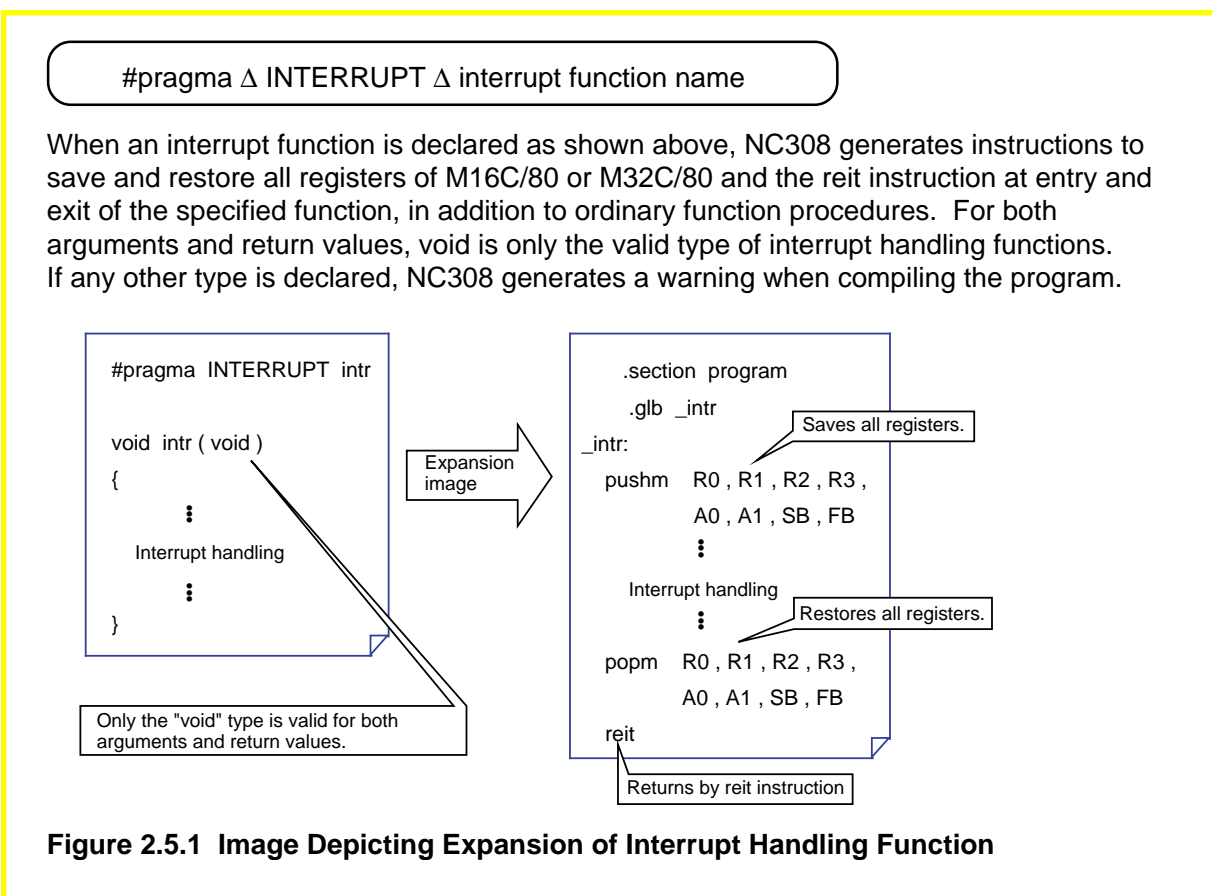
## 2.5 Interrupt Handling

### 2.5.1 Writing Interrupt Handling Functions

NC308 allows you to write interrupt handling as C language functions. There are four procedures to be followed<sup>(Note)</sup>:

- (1) Write interrupt handling functions.
  - (2) Register them in an interrupt vector table.
  - (3) Set interrupt enable flag (I flag)  
Use the inline assembly facility to do this.
  - (4) Set the interrupt priority level of the interrupt used.  
Set this priority before enabling the interrupt as when using the assembler.
- This section explains how to write C language functions for each type of interrupt handling.

#### Writing Hardware Interrupts (#pragma INTERRUPT)



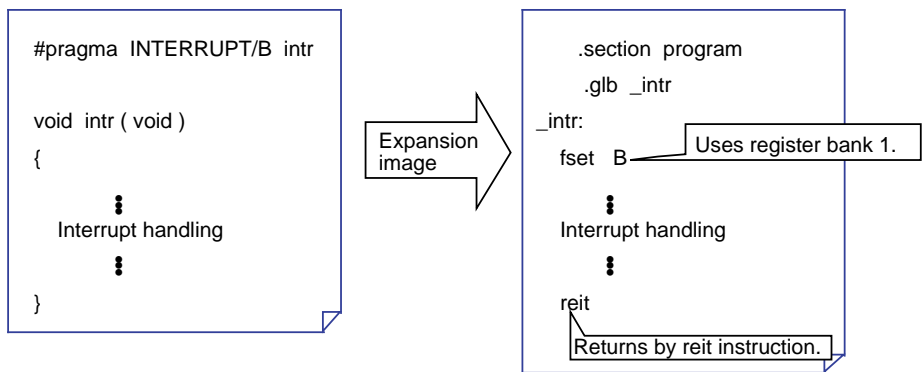
Note: For details, refer to Section 2.5.5, "Description Example of Interrupt Handling Functions."

### Writing Interrupt Handling Using Register Banks (#pragma INTERRUPT/B)

In the M16C/80 and M32C/80 series, by switching over register banks, it is possible to reduce the time required for interrupt handling to start, while at the same time protecting register contents, etc. If you want to use this facility, write the line shown below.

```
#pragma Δ INTERRUPT/B Δ interrupt function name
```

When written like the above, an instruction to switch over register banks is generated, in place of instructions to save and restore registers. However, since the register banks available in the M16C/80 and M32C/80 series are only register banks 0 and 1, only one interrupt can be specified<sup>(Note)</sup>. Use this facility for an interrupt that needs to be started in a short time.



**Figure 2.5.2 Image Depicting Expansion of Interrupt Handling Function Using Register Banks**

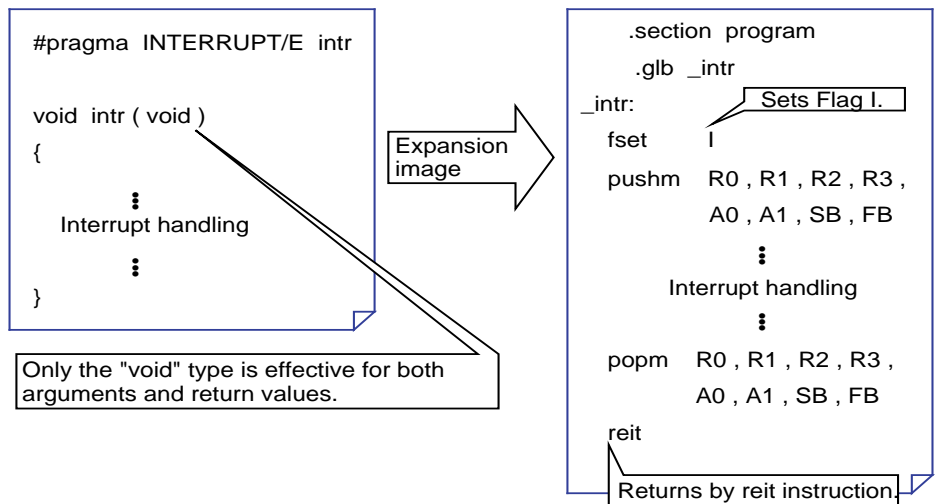
Note: When not using multiple interrupts, this facility can be used in all interrupts.

**Writing Interrupt Handling to Enable Multiple Interrupts (#pragma INTERRUPT/E)**

In the M16C/80 and M32C/80 series, once an interrupt request is accepted, the interrupt enable flag (I flag) is reset to 0 causing other interrupts to be disabled. Therefore, if necessary, the interrupt enable flag (I flag) can be set back to 1 at entry to the interrupt handling function (immediately after entering the interrupt handling function) to enable multiple interrupts, for improved interrupt response. If you want to use this facility, write the line shown below.

```
#pragma Δ INTERRUPT/E Δ interrupt function name
```

When written like the above, an instruction to set the interrupt enable flag (I flag) to 1 is generated at entry to the interrupt handling function (immediately after entering the interrupt handling function). However, if multiple interrupts need to be enabled, i.e., another interrupt request is accepted while executing an interrupt handling function, write a "#pragma INTERRUPT" declaration and set the interrupt enable flag (I flag) to 1 using the asm() function from within the interrupt handling function.



**Figure 2.5.3 Image Depicting Expansion of Interrupt Handling Function to Enable Multiple Interrupts**



### 2.5.2 Writing High-speed Interrupt Handling Functions

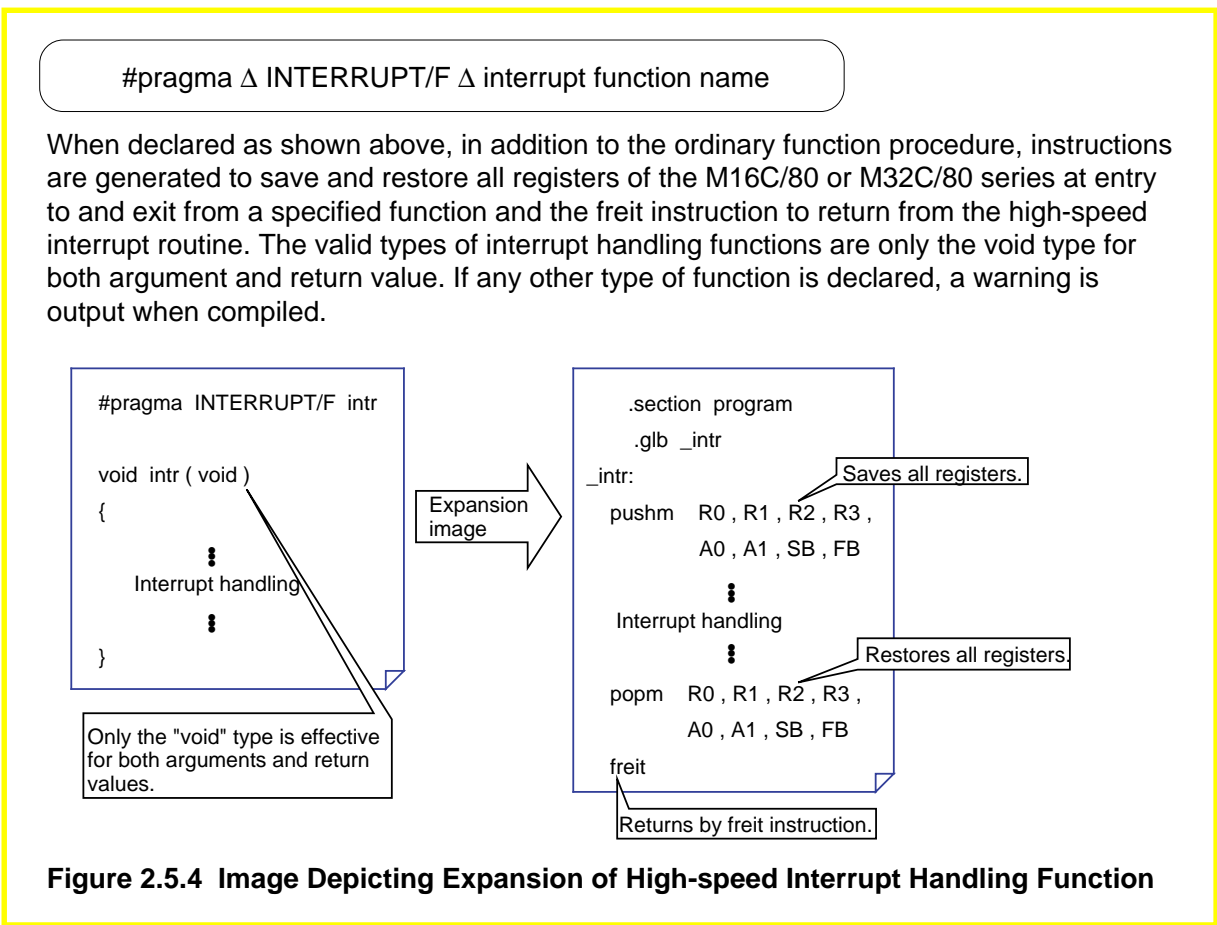
In NC308, high-speed interrupt handling where an interrupt can be acknowledged in 5 cycles and control returned from it in 3 cycles can be written as C language function. However, interrupts that can be set to be a high-speed interrupt are only one interrupt whose interrupt priority level = 7.

The procedure consists of the following five steps<sup>(note)</sup>:

- (1) Write a high-speed interrupt handling function.
- (2) Set the interrupt priority level of the high-speed interrupt used.  
Do this setting before enabling interrupts as when using the assembler.
- (3) Set the high-speed interrupt set bit.
- (4) Set the vector register (VCT).
- (5) Set the interrupt enable flag (I flag).  
Use the inline assemble facility to do this setting.

The following explains how to write functions for each type of high-speed interrupt handling.

#### Writing High-speed Hardware Interrupt Handling (#pragma INTERRUPT/F)



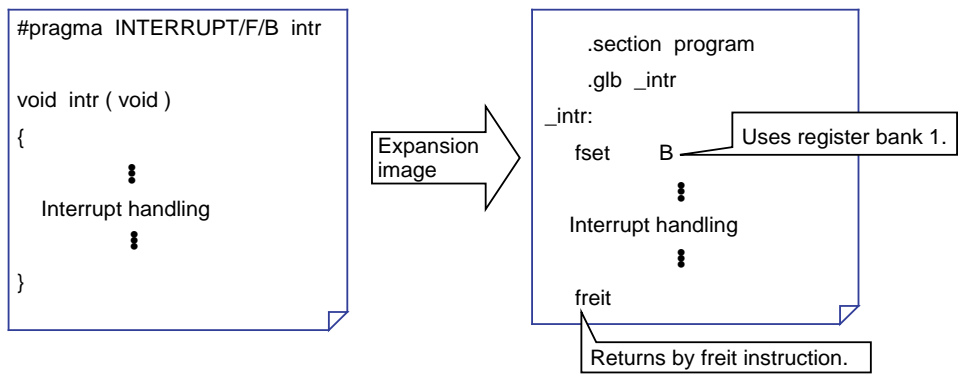
Note: For details, refer to Section 2.5.5, "Description Example of Interrupt Handling Functions."

**Writing High-speed Interrupt Functions Using Register Banks (#pragma INTERRUPT/F/B)**

In the M16C/80 and M32C/80 series, by switching over register banks, it is possible to reduce the time required for high-speed interrupt handling to start, while at the same time protecting register contents, etc. If you want to use this facility, write the line shown below.

```
#pragma Δ INTERRUPT/F/B Δ interrupt function name
```

When declared as shown above, an instruction to switch over register banks is generated, in place of instructions to save and restore registers. However, since the register banks available in the M16C/80 or M32C/80 series are only register banks 0 and 1, only one interrupt can be specified<sup>(Note)</sup>. Use this facility for an interrupt that needs to be started in the shortest time possible.



**Figure 2.5.5 Image Depicting Expansion of High-speed Interrupt Handling Function Using Register Banks**

Note: Unless multiple interrupts are used, register bank switchover can be used for all interrupts.

### 2.5.3 Writing Software Interrupt (INT Instruction) Handling Functions

#### Writing Software Interrupt to Call an Assembly Language Function

To use the software interrupt (INT instruction) of the M16C/80 and M32C/80 series, write #pragma INTCALL. Use of this facility makes it possible to generate interrupts in a simulated manner when debugging. Furthermore, because the number of bytes required for the INT instruction is only 2, this facility helps to increase the ROM efficiency<sup>(Note)</sup>. The method of writing #pragma INTCALL differs depending on whether the body of the function called from a software interrupt is written in assembly language or written in C language.

If the body of the function called from a software interrupt is written in the assembly language, write #pragma INTCALL as shown below.

```
#pragma Δ INTCALL Δ software interrupt number Δ assembly language function
name (register name, register name,...)
```

When the body of the function called from a software interrupt is written in assembly language, arguments can be passed via registers. Also, return values in any other types than structure or union can be received.

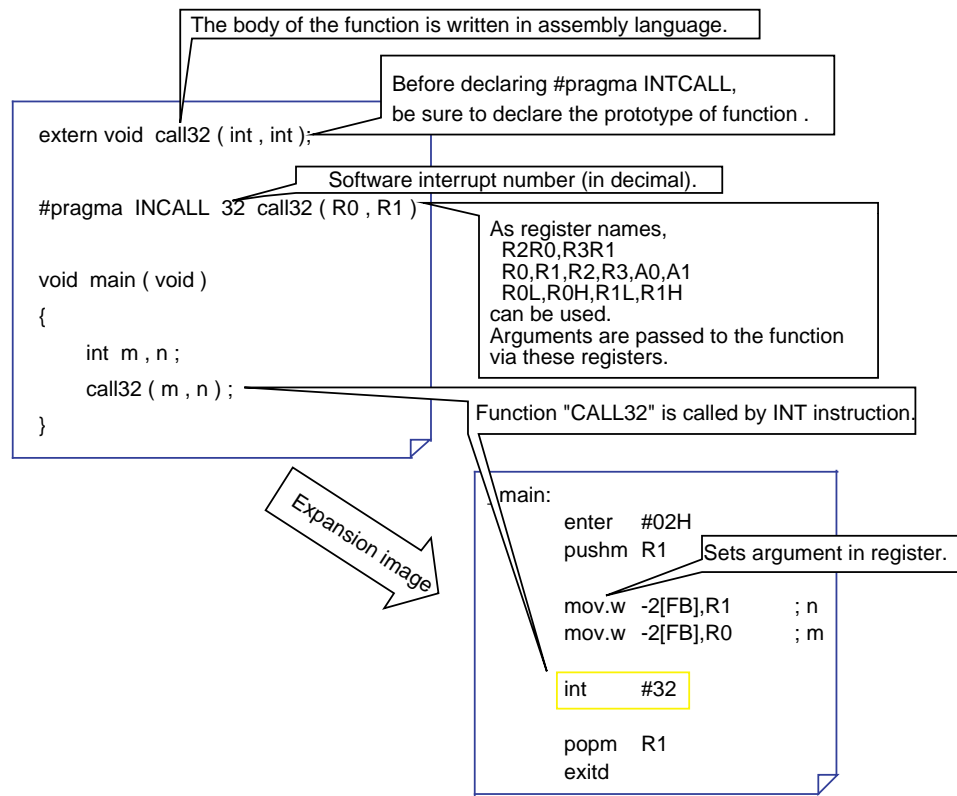


Figure 2.5.6 Example for Writing "#pragma INTCALL" to Call an Assembly Language Function

Note: Because 12 execution cycles of the INT instruction + interrupt sequence execution time are required, it takes some time before the function is executed.

### Writing Software Interrupt to Call a C Language Function (#pragma INTCALL)

When the body of the function to be called from a software interrupt (INT instruction) is written in C language, write #pragma INTCALL as shown below.

```
#pragma Δ INTCALL Δ software interrupt number Δ C language function name ( )
```

When the body of the function to be called from a software interrupt (INT instruction) is written in C language, note that only the functions all arguments of which are passed via registers can be specified. No arguments can be written for functions that declare #pragma INTCALL. Return values in any other types than structure or union can be received.

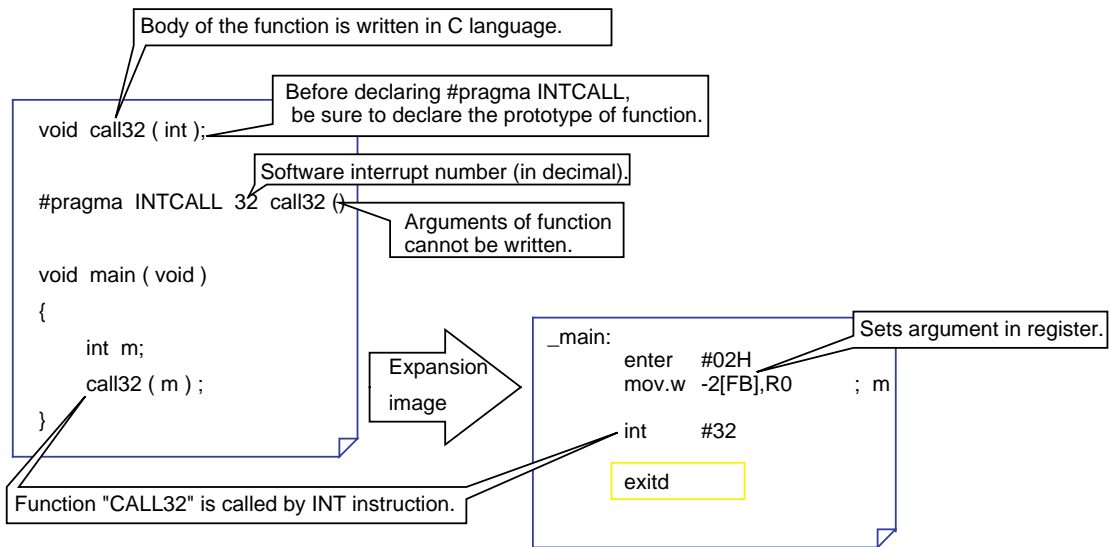


Figure 2.5.7 Example for Writing "#pragma INTCALL" to Call an C Language Function

### 2.5.4 Registering Interrupt Processing Functions

For interrupts to be serviced correctly, in addition to writing interrupt handling functions, it is necessary to register them in an interrupt vector table.

This section explains how to register interrupt handling functions in an interrupt vector table.

#### Registering in Interrupt Vector Table

When interrupt handling functions are written, they must be registered in an interrupt vector table. This can be accomplished by modifying the interrupt vector table in the sample startup program "sect30.inc".

Follow the procedure described below to modify the interrupt vector table.

- (1) Externally define the interrupt handling function names using the directive command ".glb".
- (2) Change the dummy function names "dummy\_int" of the interrupts used to interrupt handling function names.

```

;-----
; variable vector section
;-----
.section      vector          ; variable vector table
.org         VECTOR_ADR
;
.lword      dummy_int        ; vector (BRK)
.org        ( VECTOR_ADR + 32 )
.lword      dummy_int        ; DMA0 (software int 8)
.lword      dummy_int        ; DMA1 (software int 9)
.lword      dummy_int        ; DMA2 (software int 10)
.lword      dummy_int        ; DMA3 (software int 11)
.glb       _ta0
.lword      _ta0             ; TIMER A0 (software int 12)
.lword      dummy_int        ; TIMER A1 (software int 13)
.lword      dummy_int        ; TIMER A2 (software int 14)
.lword      dummy_int        ; TIMER A3 (software int 15)
.lword      dummy_int        ; TIMER A4 (software int 16)
;

```

Figure 2.5.8 Interrupt Vector Table ("sect308.inc")

### 2.5.5 Example for Writing Interrupt Processing Function

The program shown in this description example counts up the content of "counter" each time an INTO interrupt occurs.

#### Writing Interrupt Handling Function

Figure 2.5.9 shows an example of source file description.

```

/* Prototype declaration *****/
void int0 ( void );
void int1 ( void );
#pragma INTERRUPT/F int0
#pragma INTERRUPT int1
/* *****/

unsigned int counter;

void int0 ( void ) /* High-speed Interrupt function */
{
    counter = 0 ;
}

void int1 ( void ) /* Interrupt function */
{
    if ( counter < 9 ) {
        counter ++ ;
    }
    else {
        coutter = 0 ;
    }
}

void main ( void )
{
    INT0IC = 0x07 ; /* Setting high-speed interrupt priority level */
    RLV L = 0x08 ; /* Setting Exit priority register */
    asm ( " LDC #_int0,VCT " ) ; /* Setting Vector register */
    INT1IC = 0x01 ; /* Setting interrupt priority level */

    asm ( " fset i " ) ; /* Enabling interrupt */

    while ( 1 ) ; /* Interrupt waiting loop */
}

```

Figure 2.5.9 Example for Writing Interrupt Handling Function

## Registering in Interrupt Vector Table

Figure 2.5.10 shows an example for registering the interrupt handling functions in an interrupt vector table.

```

;-----
;   variable  vector  section
;-----
;
;   .section      vector          ; variable vector table
;   .org          VECTOR_ADR
;
;               ;
;               ;
;   .org          ( VECTOR_ADR + 32 )
;   .lword        dummy_int       ; DMA0 (software int 8)
;   .lword        dummy_int       ; DMA1 (software int 9)
;   .lword        dummy_int       ; DMA2 (software int 10)
;   .lword        dummy_int       ; DMA3 (software int 11)
;   .lword        dummy_int       ; TIMER A0 (software int 12)
;   .lword        dummy_int       ; TIMER A1 (software int 13)
;   .lword        dummy_int       ; TIMER A2 (software int 14)
;   .lword        dummy_int       ; TIMER A3 (software int 15)
;   .lword        dummy_int       ; TIMER A4 (software int 16)
;   .lword        dummy_int       ; uart0 trance (software int17)
;   .lword        dummy_int       ; uart0 receive (software int18)
;   .lword        dummy_int       ; uart1 trance (software int19)
;   .lword        dummy_int       ; uart1 receive (software int 20)
;   .lword        dummy_int       ; TIMER B0 (software int 21)
;   .lword        dummy_int       ; TIMER B1 (software int 22)
;   .lword        dummy_int       ; TIMER B2 (software int 23)
;   .lword        dummy_int       ; TIMER B3 (software int 24)
;   .lword        dummy_int       ; TIMER B4 (software int 25)
;   .lword        dummy_int       ; INT5 (software int 26)
;   .lword        dummy_int       ; INT4 (software int 27)
;   .lword        dummy_int       ; INT3 (software int 28)
;   .lword        dummy_int       ; INT2 (software int 29)
;
;   .glb          _int1
;   .lword        _int1           ; INT1 (software int 30)
;   .lword        dummy_int       ; INT0 (software int 31)
;   .lword        dummy_int       ; TIMER B5 (software int 32)
;
;               ;

```

Figure 2.5.10 Example for Registering in Interrupt Vector Table

# Appendices

---

- Appendix A. Functional Comparison between  
NC308 and NC30
- Appendix B. NC308 Command Reference
- Appendix C. Questions & Answers



## Appendix A. Functional Comparison between NC308 and NC30

### Calling Conventions when Calling Functions

When calling a function in NC30, registers are saved on the function calling side (caller side), whereas in NC308 registers are saved on the called function side. For this reason, care must be taken not to destroy the contents of registers used in the C language function. To this end, always be sure to write processing statements to save the registers which will be destroyed in the function at entry to the assembly language function (using the PUSHM instruction) and restore the saved registers at exit from the assembly language function (using the POPM instruction).

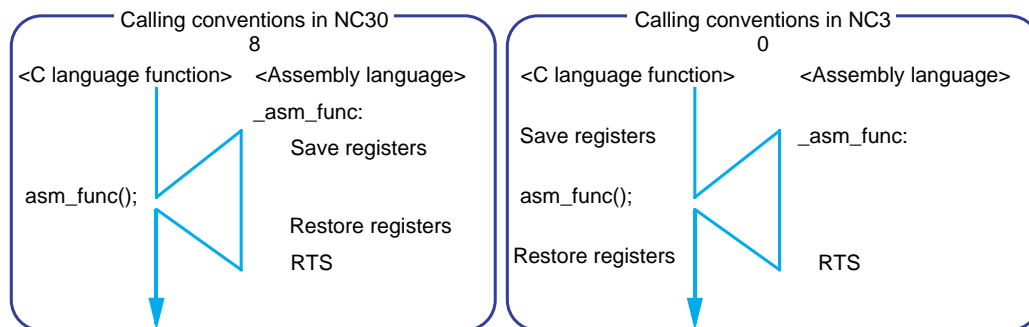


Figure A.1 Calling Conventions when Calling Functions

### Rules for Passing Parameters

Rules for passing parameters in NC308 and those in NC30 are different, as summarized in Tables A.1 and A.2 below.

Table A.1 Rules for Passing Parameters in NC308

Type of argument	First argument	Second argument	Third and following arguments
char type	R0L	Stack	Stack
short, int type near pointer type	R0	Stack	Stack
Other types	Stack	Stack	Stack

Table A.2 Rules for Passing Parameters in NC30

Type of argument	First argument	Second argument	Third and following arguments
char type	R1L	Stack	Stack
short, int type near pointer type	R1	R2	Stack
Other types	Stack	Stack	Stack

**Assembler Macro Functions**

NC308 supports the facility called "assembler macro functions" which allows part of assembly language instructions to be written as C language functions. Use of this facility makes it possible to write part of assembly language instructions directly in a C language program. The effect is that the program can be tuned up easily. For details, refer to Section 2.3.5, "Using Assembler Macro Functions."

**Table A.3 Assembly language Instructions that Can Be Written Using Assembler Macro Functions**

DADD	DADC	DSUB	DSBB
RMPA	MAX	MIN	SMOVB
SMOVF	SMOVU	SIN	SOUT
SSTR	ROLC	RORC	ROT
SHA	SHL		

## Modified Extended Functions

Definitions of near/far qualifiers and the default address sizes of pointer variables have been changed.

**Table A.4 Modified Extended Functions**

Item	Function in NC308	Function in NC30
near/far qualifiers	Specify addressing mode in which to access data. near: Access data at addresses 000000H to 00FFFFH far : Access data at addresses 000000H to FFFFFFFH	Specify addressing mode in which to access data. near: Access data at addresses 00000H to 0FFFFH far : Access data at addresses 00000H to FFFFFH
Default address size of pointer variable	far pointer (4 bytes)	near pointer (2 bytes)

## Added Extended Functions

Pursuant to the addition of a new function "high-speed interrupt" beginning with the M16C/80 and M32C/80 series, extended functions have been added to NC308 .

**Table A.5 Added Extended Functions**

Item	Function
#pragma INTERRUPT /F	When calling a high-speed interrupt function, saves (and restores) all registers and generates a FREIT instruction to return from the high-speed interrupt routine.
#pragma INTERRUPT /F/B	When calling a high-speed interrupt function, switches over register banks instead of saving registers to the stack. It also generates a FREIT instruction to return from the high-speed interrupt routine.
#pragma INTERRUPT /F/E	When calling a high-speed interrupt function, sets the interrupt enable flag (I flag) to 1 at entry to the interrupt handling function (immediately after entering the interrupt handling function) to allow multiple interrupts to be accepted. It also generates a FREIT instruction to return from the high-speed interrupt routine.

**Deleted Extended Functions**

The extended functions of NC30 listed in Table A.6 are not supported by NC308.

**Table A.6 Extended Functions Not Supported by NC308**

Item	Function
#pragma BIT	Declares that the variable is in an area where 1-bit manipulating instruction in 16-bit absolute addressing mode can be used.
#pragma EXT4MPTR	A functional extension which shows a variables is a pointer accessing 4-Mbyte expanded ROM.

## Appendix B. NC308 Command Reference

### NC308 Command Input Format

%nc308 Δ [startup option] Δ [assembly language source file name] Δ [relocatable object file name] Δ <C language source file name>

% : Indicates the prompt.

< > : Indicates an essential item.

[ ] : Indicates items that can be written as necessary.

Δ : Indicates a space.

When writing multiple options, separate them with the space key.

### Options for Controlling Compile Driver

**Table B.1 Options for Controlling Compile Driver**

Option	Function
-c	Creates relocatable file (attribute .r30) and ends processing.
-D <i>identifier</i>	Defines an identifier. Same function as #define.
-I <i>directory name</i>	Specifies the directory name where file specified by "#include" exists. Up to 8 directories can be specified.
-E	Invokes only preprocess commands and outputs result to standard output device.
-P	Invokes only preprocess commands and creates a file (attribute .i).
-S	Creates an assembly language source file (attribute .a30) and ends processing.
-U <i>predefined macro name</i>	Undefines the specified predefined macro.
-silent	Suppresses the copyright message display at startup.

If startup options -c, -E, -P, and -S are not specified, NC308 controls the compile driver up to ln308 until it creates the absolute module file (attribute .x30).

## Options for Specifying Output Files

**Table B.2 Options for Specifying Output Files**

Option	Function
-o <i>file name</i>	Specifies the name(s) of the file(s) (e.g., absolute module file, map file) generated by In308. This option can also be used to specify the destination directory name. Do not specify the file name extension.
-dir <i>directory name</i>	Specifies the destination directory of the file(s) (e.g., absolute module file, map file) generated by In308.

## Options for Displaying Version Information

**Table B.3 Options for Displaying Version Information**

Option	Function
-v	Displays the name of the command program and the command line during execution.
-V	Displays the startup message of the compiler programs, then finished processing (without compiling).

## Options for Debugging

**Table B.4 Options for Debugging**

Option	Function
-g	Outputs debugging information to an assembly language source file (attribute. a30).
-genter	When calling function, it always outputs an enter instruction. Be sure to specify this option when using debugger's stack trace function.
-gno_reg	Suppresses the output of debugging information for register variables.

## Optimization Options

**Table B.5 Optimization Options**

Option	Abbreviation	Function
-O	None.	Maximum optimization of speed and ROM size.
-OR	None.	Maximum optimization of ROM size flowed by speed.
-OS	None.	Maximum optimization of speed flowed by ROM size.
-Oconst	-OC	Performs optimization by replacing references to the const-qualified external variables with constants.
-Ono_bit	-ONB	Suppresses optimization based on grouping of bit manipulations.
-Ono_break_source_debug	-ONBSD	Suppresses optimization that affects source line information.
-Ono_float_const_fold	-ONFCF	Supresses the constant folding processing of floating point numbers.
-Ono_stdlib	-ONS	Suppresses inline embedding of standard library functions or modification of library functions.
-Osp_adjust	-OSA	Optimizes to remove stack correction code. This allows the necessary ROM capacity to be reduced. However, this may result in an increased amount of stack being used.

## Library Specifying Options

**Table B.6 Library specifying options**

Option	Function
<i>-library file name</i>	Specifies a library file that is used by In308 when linking files.

## Generated Code Modification Options

**Table B.7 Generated Code Modification Options**

Option	Abbreviation	Function
-fansic	None.	Enables -fnot_reserve_asm, -fnot_reserve_far_and_near, -fnot_reserve_inline, and -fextend_to_int.
-fnot_reserve_asm	-fNRA	Excludes "asm" from reserved words. (Only _asm is valid.)
-fnot_reserve_far_and_near	-fNRFAN	Excludes "far" and "near" from reserved words. (Only _far and _near are valid.)
-fnot_reserve_inline	-fNRI	Excludes "inline" from reserved words. (Only _inline is valid.)
-fextend_to_int	-fETI	Performs operation after extending the char-type data to the int type.(Extended according to ANSI standards.) <sup>(Note)</sup>
-fchar_enumerator	-fCE	Handles the enumerator type as an unsigned char type, not as an int type.
-fno_even	-fNE	Allocates all data to the odd attribute section without separating them between odd and even when outputting data.
-fshow_stack_usage	-fSSU	Outputs the usage condition of the stack pointer to a file(extension .stk).
-ffar_RAM	-fFRAM	Changes the default attribute of RAM data to far.
-fnear_ROM	-fNROM	Changes the default attribute of ROM data to near.
-fnear_pointer	-fNP	Specifies the default attribute of the pointer type variables to near.
-fconst_not_ROM	-fCNR	Does not handle the types specified by const as ROM data.
-fnot_address_volatile	-fNAV	Does not recognize the variables specified by #pragma ADDRESS (#pragma EQU) as those specified by volatile.
-fsmall_array	-fSA	When referencing a far-type array, if its total size is within 64 Kbytes, this option calculates subscripts in 16 bits.
-fenable_register	-fER	Enables register storage class.
-align	None.	Maps starting address of functions to even address.
-fJSRW	None.	Changes the default instruction for calling functions to "JSR.W" instruction.
-fuse_DIV	-fUD	This option changes generated code for divide operation.

Note: Although char-type data in NC308 are, by default, evaluated without being extended, char-type or int-type data under ANSI standards are always extended into int type before evaluation.



## Warning Options

**Table B.8 Warning Options**

Option	Abbreviation	Function
-Wnon_prototype	-WNP	Outputs the warning messages for the functions without the prototype declarations.
-Wunknown_pragma	-WUP	Outputs the warning messages for non-supported "#pragma".
-Wno_stop	-WNS	Prevents the compiler stopping when an error occurs.
-Wstdout	None.	Outputs the error messages to the host machine's standard output device (stdout).
-Werror_filetagfilename	-WEF	Outputs the error messages to the specified tag file.
-Wstop_at_waring	-WSAW	Stops the compiling process when a warning occurs.
-Wnesting_comment	-WNC	Outputs a warning for a comment including /*.
-Wccom_max_warnings	-WCMW	This option allows you to specify an upper limit for the number of warnings output by ccom308.
-Wall	None.	Displays message for all detectable warnings.
-Wmake_tagfile	-WMT	Outputs error message to the tag file of source-file by source-file.

## Assemble and Link Options

**Table B.9 Assemble and Link Options**

Option	Function
-as308 <i>"option"</i>	Specifies options for the "as308" assemble command. If you specify two or more options, be sure to enclose them with double quotations ("").
-ln308 <i>"option"</i>	Specifies options for the "ln308" link command. If you specify two or more options, be sure to enclose them with double quotations ("").

## Other Options

**Table B.10 Other Options**

Option	Abbreviation	Function
-dsorce	-dS	Outputs C language source listing as comment in assembly language source file list to be output.

### Command Input Example

- 1 Link the startup program (ncrt0.a30) and a C language source program (c\_src.c) to create an absolute module file (test.x30).

```
%nc308 -otest ncr0.a30 c_src.c
→Specifies the output file name.
```

- 2 Generate an assembler list file and a map file.

```
%nc308 -as308 "-l" -ln3088 "-M" c_src.c
→Specifies the options of "as308" and "ln308".
```

- 3 Output debug information to an assembly language source file (attribute.a30).

```
%nc308 -g -S ncr0.a30 c_src.c
```

## Appendix C. Questions & Answers

### Transferring (Copying) Structs

<Question> \_\_\_\_\_  
 What method can be used to transfer (copy) structs?

<Answer>

- (1) When transferring structs of the same definition  
 →Use a struct vs. variable name and a assignment operator to transfer the structs.
- (2) When transferring structs of different definitions  
 →Use a assignment operator for each member to transfer the structs.

```

struct tag1 {                /*Definition of struct */
    int    mem1 ;
    char   mem2 ;
    int    mem3 ;
};

struct tag2 {
    int    mem1 ;
    char   mem2 ;
    int    mem3 ;
};

near struct tag1 near_s1t1,near_s2t1 ;
near struct tag2 near_s1t2 ;
far struct tag1 far_s1t1,far_s2t1 ;

main()
{
    near_s1t1.mem1 = 0x1234 ;
    near_s1t1.mem2 = 'A' ;
    near_s1t1.mem3 = 0x5678 ;

    /* Transferring structs of the same definition----- */
    near_s2t1 = near_s1t1 ;    /* near -> near */
    far_s1t1 = near_s1t1 ;    /* near -> far */
    near_s2t1 = far_s1t1 ;    /* far -> near */
    far_s2t1 = far_s1t1 ;    /* far -> far */

    /*Transferring structs of different definitions ----- */
    near_s1t2.mem1 = near_s1t1.mem1 ;
    near_s1t2.mem2 = near_s1t1.mem2 ;
    near_s1t2.mem3 = near_s1t1.mem3 ;
}
    
```

(1) For structs of the same definition  
 Can be transferred using a struct vs.  
 variable name and a assignment operator  
 irrespective of allocated areas.

(2) For structs of different definitions  
 Transfer the structs, one member at a time.

Figure C.1 Example for Writing Transfers of Struct

Reducing Generated Code (1)

<Question> \_\_\_\_\_  
We wish to reduce the amount of generated code. What points should we check?

<Answer>  
Check the following points:

[When declaring data...]

- (1) Among the data declared to be the int type, is there data that falls within the following range? If any, correct its data type. Designations in ( ) can be omitted.  
Unsigned int type that falls within 0 to 255 → Correct it to the (unsigned) char type.  
(signed) int type that falls within -128 to 127 → Correct it to the signed char type.
- (2) Among the data other than the int type where the unsigned/signed modifiers are omitted, is there data that does not have a negative value? If any, add the unsigned modifier.  
(In NC30, data other than the int type assumes the "signed" modifier by default.)

[When declaring far-type array...]

- (1) Isn't the number of elements omitted for any far-type array whose size is within 64 Kbytes and which is extern declared?  
-> Explicitly write the number of array elements. Or specify the generated code change option "-fsmall\_array(-fSA)".  
(In NC308, when the generated code change option "-fsmall\_array(-fSA)" is specified, the number of elements is calculated in 16 bits.

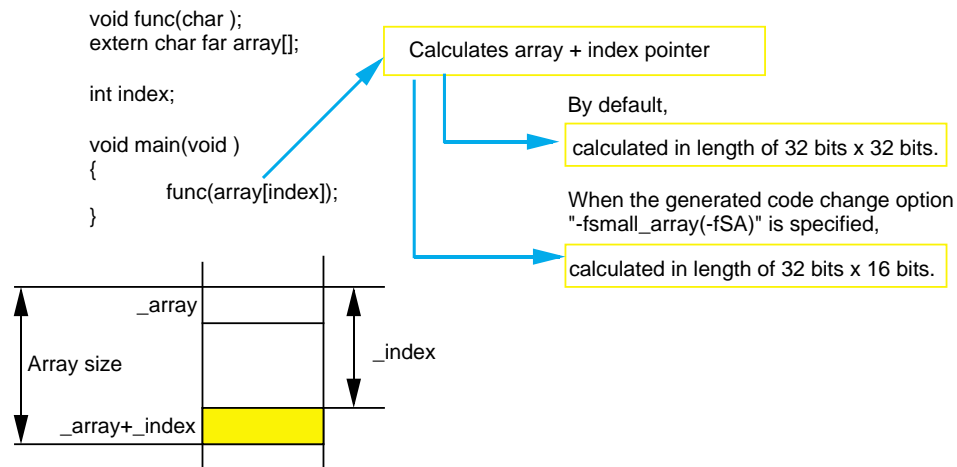
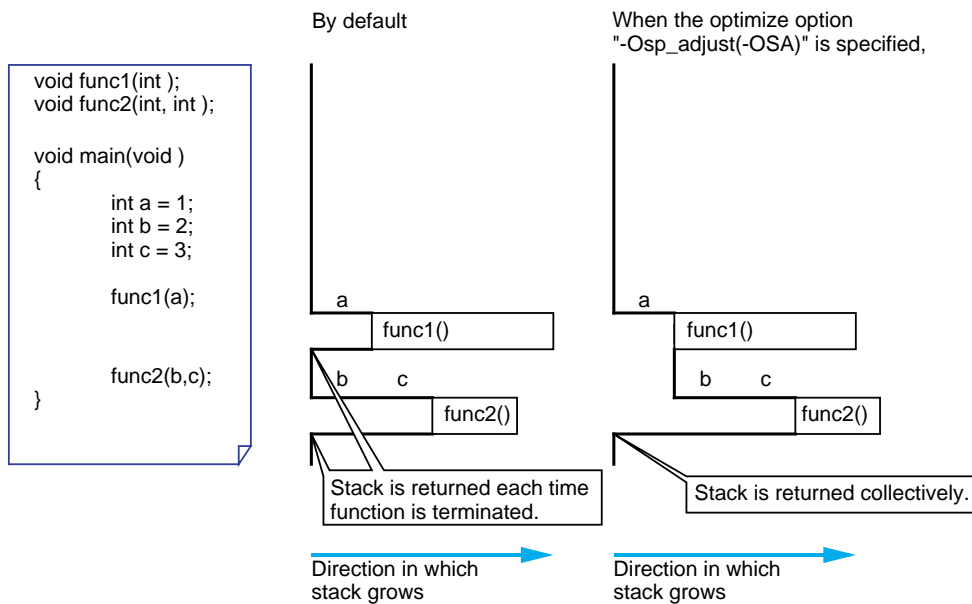


Figure C.2 Example for Using the Generated Code Change Option "-fsmall\_array(-fSA)"

[When compiling...]

- (1) Is the optimization option "-OR" specified? If not, specify this option.  
(When the optimization option "-OR" is specified in NC308, it optimizes code generation by placing emphasis on ROM efficiency.)
- (2) Is the optimize option "-Osp\_adjust(-OSA)" specified? -> If not, specify "-Osp\_adjust(-OSA)".  
(In NC308, when the optimize option "-Osp\_adjust(-OSA)" is specified, stack correction code is removed for optimization. This helps to reduce the ROM size. However, this may lead to an increased amount of stack used.)



**Figure C.3 Example for Using the Optimize Option "-Osp\_adjust(-OSA)"**

Reducing Generated Code (2)

<Question>  
Our program consists of multiple files. What points should we consider in order to reduce the generated code?

<Answer>  
Pay attention to the following:

[When referencing data located in SB relative addressing...]

- (1) When referencing data located in an SB relative addressing area, always be sure to declare "#pragma SBDATA".

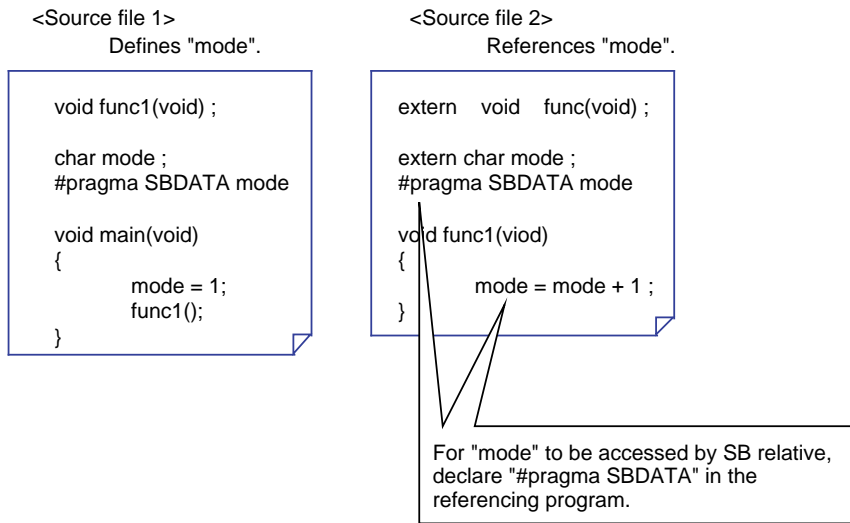


Figure C.4 Example for Writing "#pragma SBDATA"

[For programs whose generated code is 64 Kbytes or less...]

- (1) Specify the generated code change option "-fJSRW."  
When an error occurs while linking, declare #pragma JSRA for only the functions in which the error occurred.

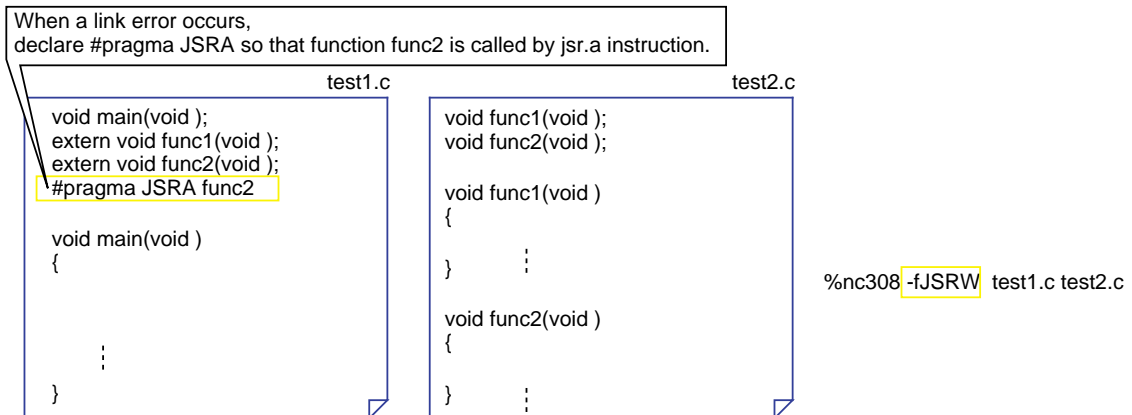


Figure C.5 Example for Using -fJSRW and Writing #pragma JSRA

REVISION HISTORY	Programming Guidelines <C Language> Application Note
------------------	---

Rev.	Date	Description	
		Page	Summary
1.00	Oct.30, 2003	-	New Document
1.10	Jun.25, 2004		<b>M32C/80 series</b> added
		All pages	" <b>Column</b> " removed from titles
		1	<b>Preface</b> Description modified <b>Guide to Using This Application Note</b> Description modified
		6	<b>Chapter 1</b> • <b>1.1.1 Assembly Language and C Language</b> Description modified • <b>Table 1.1.1 Comparison between C and Assembly Languages</b> "Format" row modified
		7	• <b>1.1.2 Program Development Procedure</b> Description modified • <b>Figure 1.1.1 NC308 Product List</b> Figure modified
		8	• <b>Creating Machine Language File from C Language Source File</b> Description modified
		9	• <b>1.1.3 Program Rules and Practices</b> Title and description modified
		10	• <b>Programming Style</b> Description modified
		11	• <b>Method for Writing Comments</b> Title and description modified
		13	• <b>1.2.1 "Constants" in C Language</b> Title modified • <b>Table 1.2.1 Method for Writing Integer Constants</b> "Decimal" and "Hexadecimal" rows modified • <b>Table 1.2.2 Escape Sequence in C Language</b> Symbol modified
		19	• <b>Table 1.3.1 NC308 Operators</b> Title modified
		21	• <b>Division Operator in NC308</b> Description modified
		31	• <b>1.4.1 Structuring of Program</b> Description modified
		51	• <b>What is an Array?</b> and <b>Example 1.7.1 Finding Total Age of a Family (1)</b> Example name modified
54	• <b>1.7.3 Pointers</b> Description modified		
59	• <b>Figure 1.7.11 Difference between Two-dimensional Array and Pointer Array</b> Example names modified		
68	• <b>1.9.2 Including a File</b> Title modified		
		<b>Chapter 2</b>	
102	• <b>2.3.4 Using Inline Assembly</b> Title modified		
104	• <b>2.3.5 Using Assembler Macro Functions</b> Title modified		
		<b>Chapter 3</b> Deleted	
		<b>Appendices</b>	
148	• <b>Reducing Generated Code (2)</b> Question modified		

---

Keep safety first in your circuit designs!

---

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

---

Notes regarding these materials

---

- These materials are intended as a reference to assist our customers in the selection of the Renesas Technology Corporation product best suited to the customer's application; they do not convey any license under any intellectual property rights, or any other rights, belonging to Renesas Technology Corporation or a third party.
- Renesas Technology Corporation assumes no responsibility for any damage, or infringement of any third-party's rights, originating in the use of any product data, diagrams, charts, programs, algorithms, or circuit application examples contained in these materials.
- All information contained in these materials, including product data, diagrams, charts, programs and algorithms represents information on products at the time of publication of these materials, and are subject to change by Renesas Technology Corporation without notice due to product improvements or other reasons. It is therefore recommended that customers contact Renesas Technology Corporation or an authorized Renesas Technology Corporation product distributor for the latest product information before purchasing a product listed herein.

The information described here may contain technical inaccuracies or typographical errors. Renesas Technology Corporation assumes no responsibility for any damage, liability, or other loss rising from these inaccuracies or errors.

Please also pay attention to information published by Renesas Technology Corporation by various means, including the Renesas Technology Corporation Semiconductor home page (<http://www.renesas.com>).

- When using any or all of the information contained in these materials, including product data, diagrams, charts, programs, and algorithms, please be sure to evaluate all information as a total system before making a final decision on the applicability of the information and products. Renesas Technology Corporation assumes no responsibility for any damage, liability or other loss resulting from the information contained herein.
- Renesas Technology Corporation semiconductors are not designed or manufactured for use in a device or system that is used under circumstances in which human life is potentially at stake. Please contact Renesas Technology Corporation or an authorized Renesas Technology Corporation product distributor when considering the use of a product contained herein for any specific purposes, such as apparatus or systems for transportation, vehicular, medical, aerospace, nuclear, or undersea repeater use.
- The prior written approval of Renesas Technology Corporation is necessary to reprint or reproduce in whole or in part these materials.
- If these products or technologies are subject to the Japanese export control restrictions, they must be exported under a license from the Japanese government and cannot be imported into a country other than the approved destination.  
Any diversion or reexport contrary to the export control laws and regulations of Japan and/or the country of destination is prohibited.
- Please contact Renesas Technology Corporation for further details on these materials or the products contained therein.