To our customers,

Old Company Name in Catalogs and Other Documents

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

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

April 1st, 2010
Renesas Electronics Corporation

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

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

Notice

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

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

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

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

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

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

M16C/60, M16C/20, M16C/Tiny, R8C/Tiny Series

Programming Guidelines <C Language>

Preface

This application note is written for the Renesas M16C/60, M16C/20, M16C/Tiny, R8C/Tiny series 16-bit microcomputers. It explains the basics of C language programming and how to put your program into ROM using the NC30 C compiler.
For details about hardware and development support tools available for each type of microcomputer in the M16C/60, M16C/20, M16C/Tiny, R8C/Tiny 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 NC30, the C compiler for the M16C/60, M16C/20, M16C/Tiny, R8C/Tiny series. Knowledge of the M16C/60, M16C/20, M16C/Tiny, R8C/Tiny 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 NC30
-Appendix A: Functional Comparison between NC30 and NC77
-Appendix B: NC3 Command Reference
-Appendix C: Questions & Answers

Table of contents

Chapter 1 Introduction to C Language 7
1.1 Programming in C Language 8
1.1.1 Assembly Language and C Language 8
1.1.2 Program Development Procedure 9
1.1.3 Program Rules and Practices 11
1.2 Data Types 15
1.2.1 "Constants" in C Language 15
1.2.2 Variables 17
1.2.3 Data Characteristics 19
1.3 Operators 21
1.3.1 Operators of NC30 21
1.3.2 Operators for Numeric Calculations 22
1.3.3 Operators for Processing Data 25
1.3.4 Operators for Examining Condition 28
1.3.5 Other Operators 29
1.3.6 Priorities of Operators 31
1.4 Control Statements 32
1.4.1 Structuring of Program 32
1.4.2 Branch Processing 33
1.4.3 Repeat Processing 37
1.4.4 Suspending Processing 40
1.5 Functions 42
1.5.1 Functions and Subroutines 42
1.5.2 Creating Functions 43
1.5.3 Exchanging Data between Functions 45
1.6 Storage Classes 46
1.6.1 Effective Range of Variables and Functions 46
1.6.2 Storage Classes of Variables 47
1.6.3 Storage Classes of Functions 49
1.7 Arrays and Pointers 51
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

1.8 Struct and Union 62
1.8.1 Struct and Union 62
1.8.2 Creating New Data Types 63
1.9 Preprocess Commands 67
1.9.1 Preprocess Commands of NC30 67
1.9.2 Including a File 68
1.9.3 Macro Definition 69
1.9.4 Conditional Compile 71

Chapter 2 Extended Function of NC30 73
2.1 Memory Mapping 74
2.1.1 Types of Code and Data 74
2.1.2 Sections Managed by NC30 75
2.1.3 Control of Memory Mapping 77
2.1.4 Controlling Memory Mapping of Struct 79
2.2 Startup Program 81
2.2.1 Roles of Startup Program 81
2.2.2 Estimating Stack Sizes Used 83
2.2.3 Creating Startup Program 86
2.3 Extended Functions for ROM'ing Purposes 93
2.3.1 Efficient Addressing 93
2.3.2 Handling of Bits 97
2.3.3 Control of I/O Interface 99
2.3.4 Using Inline Assembly 101
2.4 Linkage with Assembly Language 103
2.4.1 Interface between Functions 103
2.4.2 Calling Assembly Language from C Language 108
2.4.3 Calling C Language from Assembly Language 114
2.5 Interrupt Handling 115
2.5.1 Writing Interrupt Handling Functions 115
2.5.2 Registering Interrupt Processing Functions 118
2.5.3 Example for Writing Interrupt Handling Function 119

Appendices 121

Appendix A. Functional Comparison between NC30 and NC77 122

Appendix B. NC30 Command Reference 125

Appendix C. Questions & Answers 131

Table of contents for example

Chapter 1 Introduction of C Language 7

1.1 Programming in C Language 8
1.2 Data Types 15
1.3 Operators 21
1.4 Control Statements 32
Example 1.4.1 Count Up (if-else Statement) 33
Example 1.4.2 Switchover of Arithmetic Operations (else-if Statement) 34
Example 1.4.3 Switchover of Arithmetic Operations (switch-case Statement) 35
Example 1.4.4 Finding Sum Total (while Statement) 37
Example 1.4.5 Finding Sum Total (for Statement) 38
Example 1.4.6 Finding Sum Total (do-while Statement) 39
1.5 Functions 42
Example 1.5.1 Finding Sum of Integers (Example for a Function) 45
1.6 Storage Classes 46
1.7 Arrays and Pointers 51
Example 1.7.1 Finding Total Age of a Family (1) 51
Example 1.7.2 Finding Total Age of a Family (2) 52
Example 1.7.3 Switching Arithmetic Operations Using Table Jump 61
1.8 Struct and Union 62
1.9 Preprocess Commands 67

Chapter 2 Extended Function of NC30 73

2.1 Memory Mapping 74
2.2 Startup Program 81
2.3 Extended Functions for ROM'ing Purposes 93
Example 2.3.1 Defining SFR Area Using "#pragma ADDRESS" 100
2.4 Linkage with Assembly Language 103
Example 2.4.1 Calling A Subroutine 110
Example 2.4.2 Calling a Subroutine by Table Jump 112
Example 2.4.3 A Slightly Different Way to Use Table Jump 113
2.5 Interrupt Processing 115

Appendices 121

Appendix A. Functional Comparison between NC30 and NC77 122

Appendix B. NC30 Command Reference 125

Appendix C. Questions & Answers 131

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(Note)), a program written in the C language can be ported into other types of microcomputers after only a minor modification of the source program.

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 NC30, the C compiler for the the M16C/60, M16C/20, M16C/Tiny, R8C/Tiny series of Renesas single-chip microcomputers.

NC30 Product List

NC30WA

Figure 1.1.1 lists the products included in NC30, the C compiler for the M16C/60, M16C/20, M16C/Tiny, R8C/Tiny series of Renesas single-chip microcomputers.

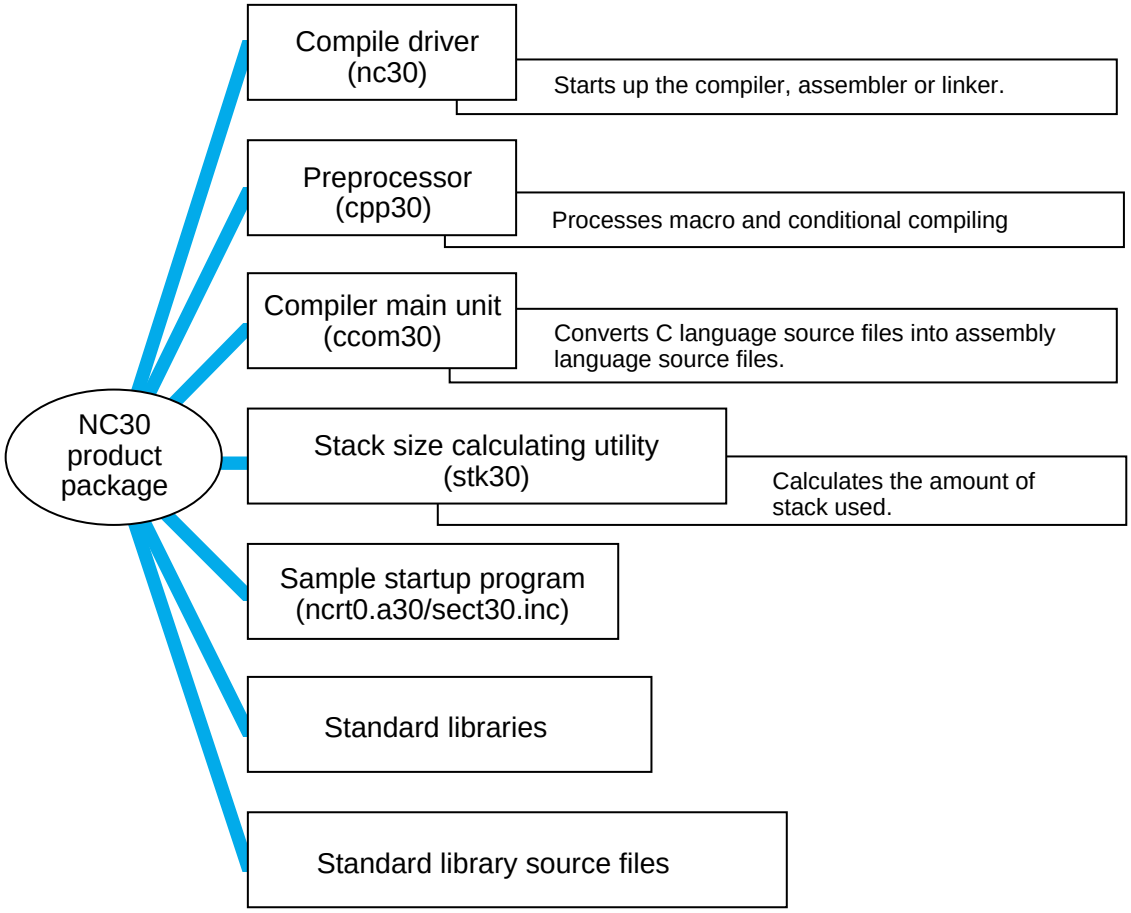


Figure 1.1.1 NC30 Product List

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.

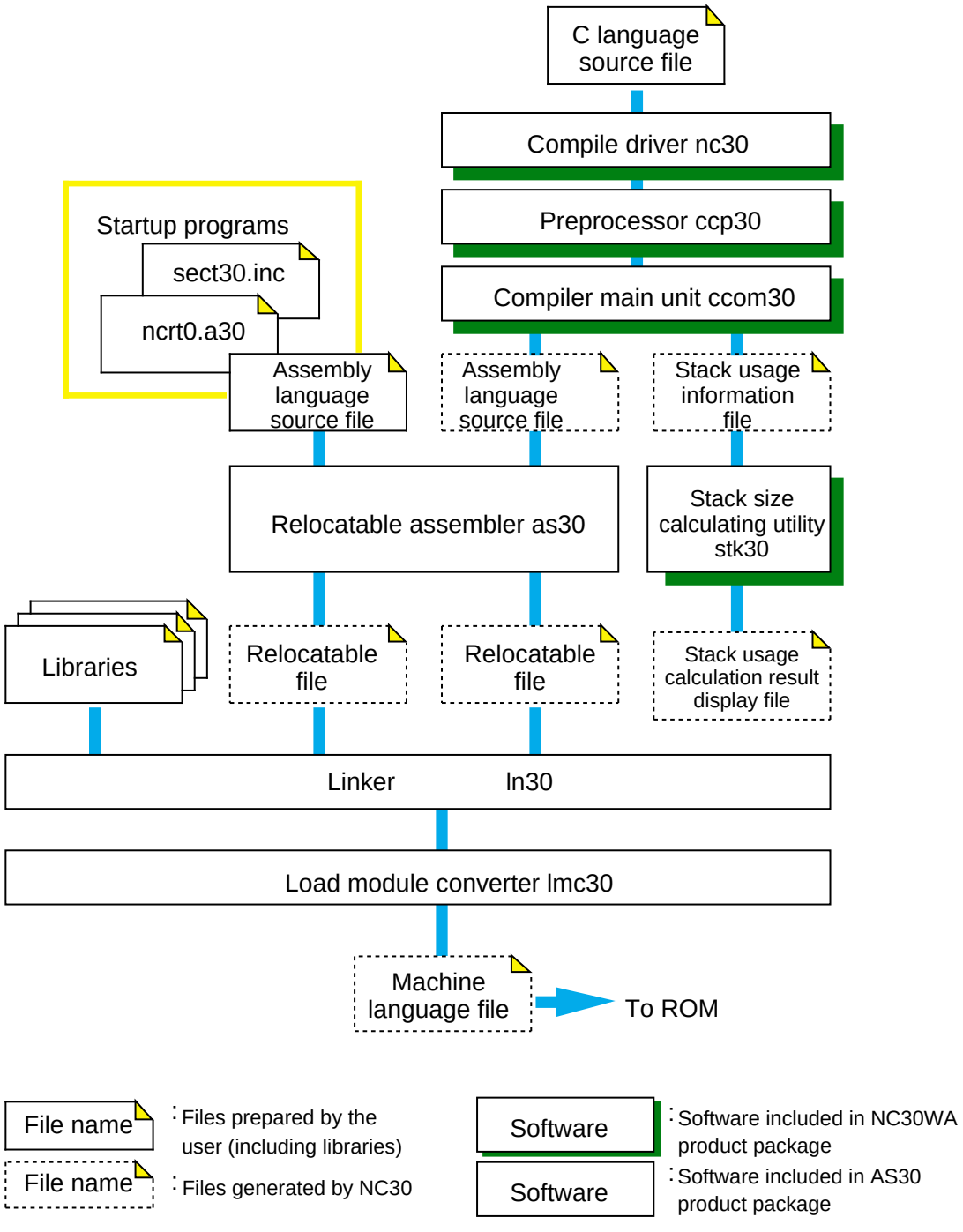


Figure 1.1.2 Creating Machine Language File from C Language Source File

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 convention, use lowercase 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.

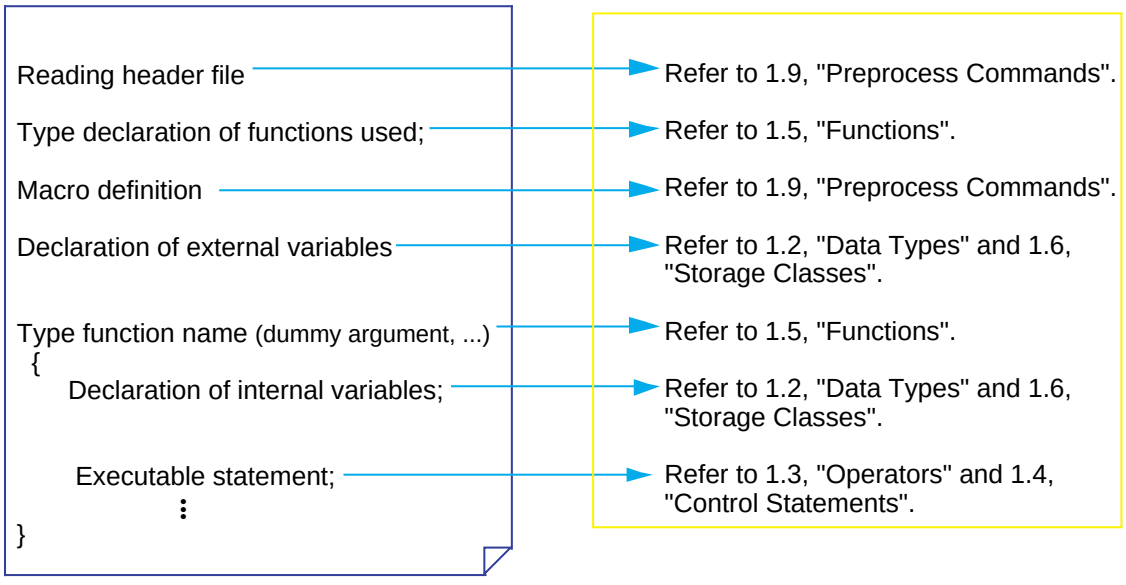


Figure 1.1.3 Configuration of C Language Source File

Programming Style

To improve program maintainability, programming conventions 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.

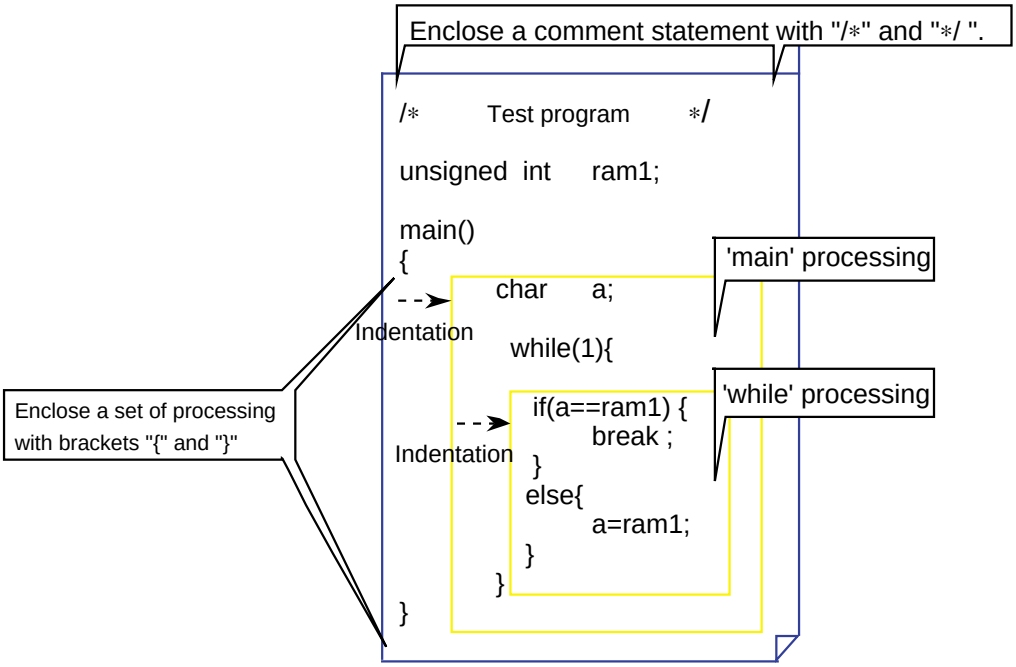


Figure 1.1.4 Example of Programming Style of C Language Program

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.

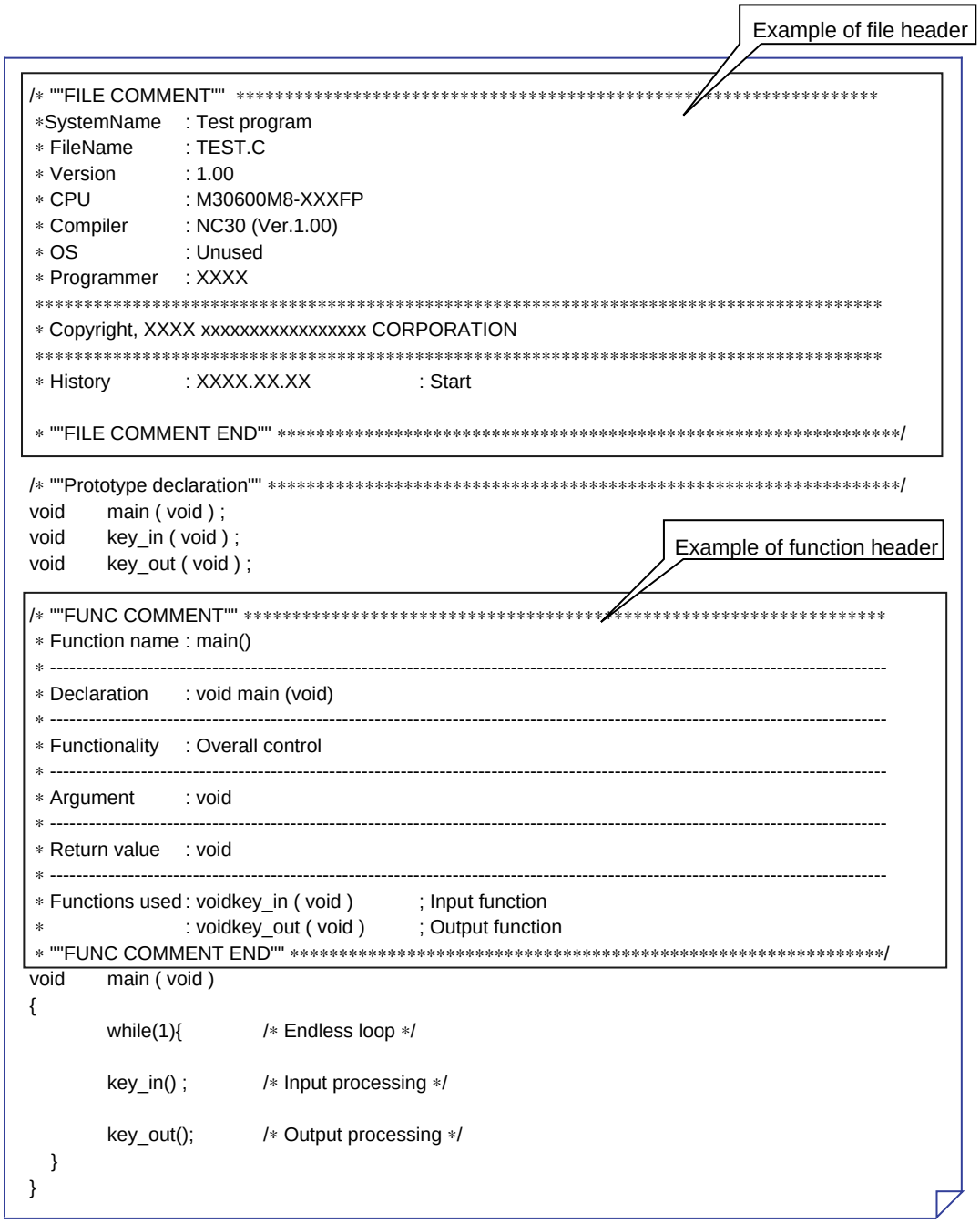


Figure 1.1.5 Example for Using Comments

Reserved Words of NC30

The words listed in Table 1.1.2 are reserved for NC30. Therefore, these words cannot be used in variable or function names.

Table 1.1.2 Reserved Words of NC30

_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		

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.

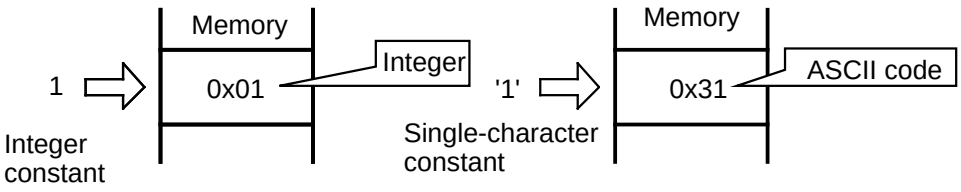


Figure 1.2.1 Difference between 1 and '1'

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!"

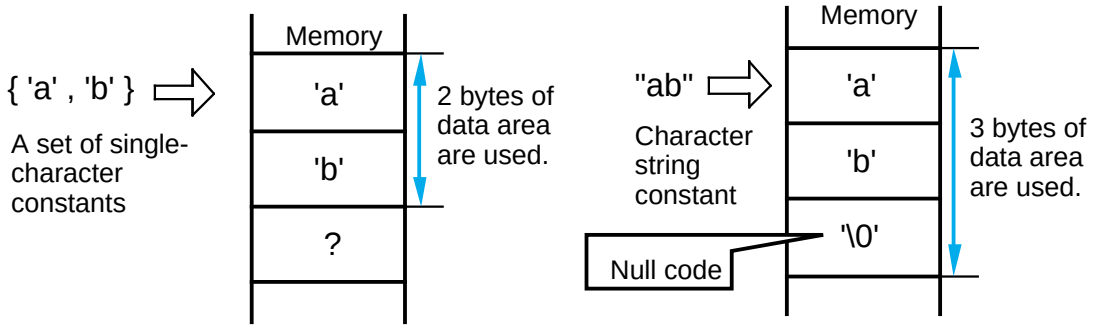


Figure 1.2.2 Difference between {'a', 'b'} and "ab"

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 NC30 and how to declare the data types.

Basic Data Types of NC30

Table 1.2.3 lists the data types that can be handled in NC30. Descriptions enclosed with ( ) in the table below can be omitted when declaring the data type.

Table 1.2.3 Basic Data Types of NC30

	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
Real	float	32 bits	Number of significant digits: 9
	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;

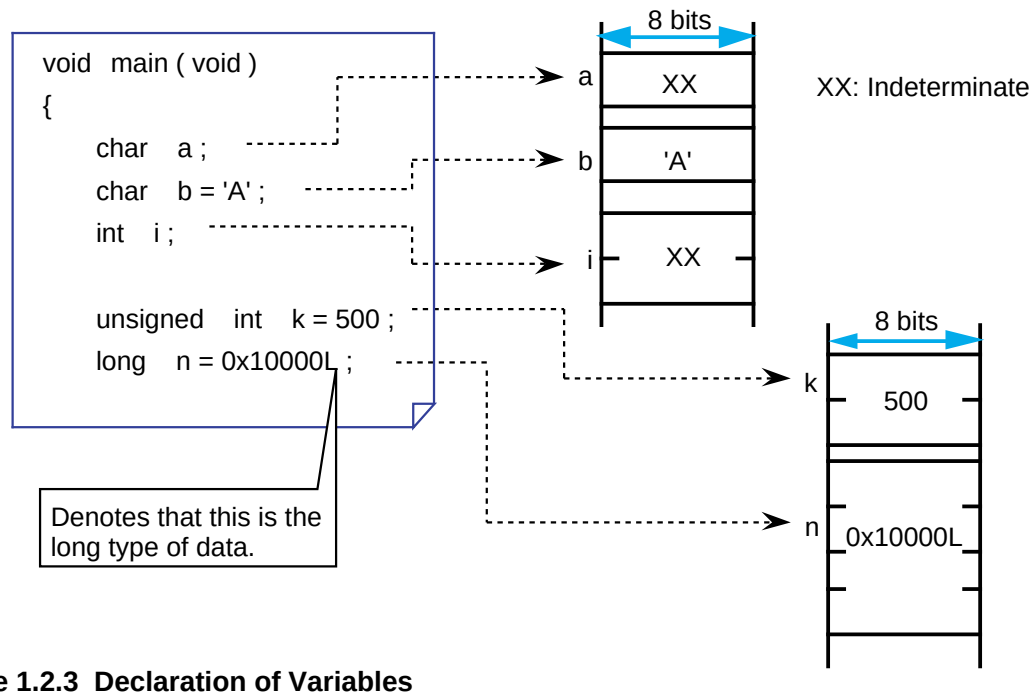


Figure 1.2.3 Declaration of Variables

1.2.3 Data Characteristics

When declaring a variable or constant, NC30 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 NC30 and how to specify a data characteristic.

Specifying that the Variable or Constant is Signed 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, NC30 assumes that it is unsigned data for only the data type char, and signed 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, NC30 outputs a warning.

```
void   main ( void )
{
    char a = 10 ;
    const char c_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)

NC30 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. NC30 does not optimize the data that is accompanied by this type qualifier and outputs instruction code for it.

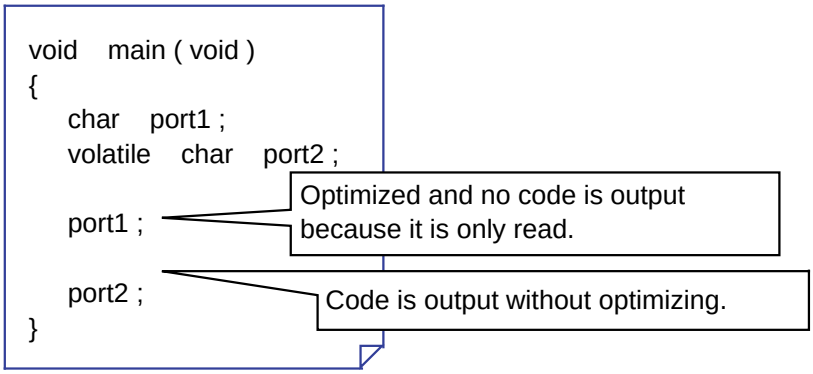


Figure 1.2.6 Example for Writing the Type Qualifier "volatile"

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 register auto extern	unsigned signed const volatile	int char float struct union	dataname

Figure 1.2.7 Syntax of Declaration

1.3 Operators

1.3.1 Operators of NC30

NC30 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(Note)) and the precautions to be noted when using them.

Operators Usable in NC30

Table 1.3.1 lists the operators that can be used in NC30.

Table 1.3.1 NC30 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 a 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 incremented or decremented before assignment.
b = ++a; → a = a + 1; b = a;
Postfix type: The value is incremented or decremented after assignment.
b = a++; → b = a; a = a + 1;

Binary Arithmetic Operators

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

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 diving its value by that of expression 2.
%	expression 1 % expression 2	Returns the remainder of expression 1 after dividing its value by that of expression 2.

Assignment Operators

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

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, NC30 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.

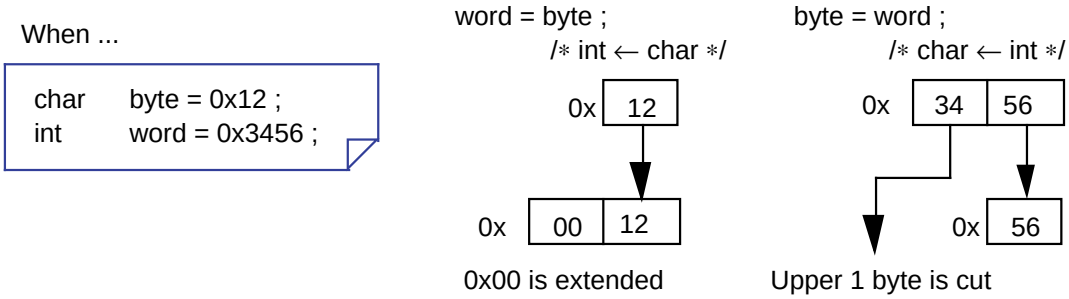


Figure 1.3.1 Assign Different Types of Data

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 Column, "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.

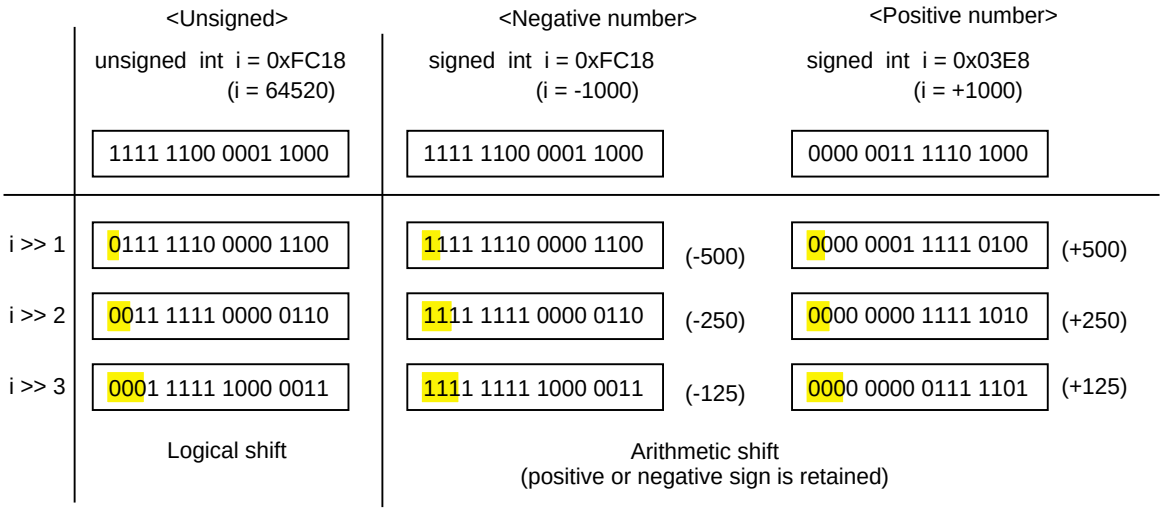


Figure 1.3.2 Arithmetic and Logical Shifts

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, NC30 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 1.3.3 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.

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

=

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

• Absolute value is found.

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

=

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

Figure 1.3.3 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 an operation is performed on data whose types differ from each other, the data used in that operation is 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 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 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

	Type of operator	Operator	Rules of combination
Highest	Expression	( ) [ ] . (Note 1) ->	→
↑	Monadic arithmetic operators, etc.	! ~ ++ -- - * (Note 2) & (Note 3) sizeof( ) (type)	←
	Multiply/divide operators	* (Note 4) / %	→
	Add/subtract operators	+ -	→
	Shift operator	<< >>	→
	Relational operator (comparison)	< <= > >=	→
	Relational operator (equivalent)	== !=	→
	Bitwise operator (AND)	&	→
	Bitwise operator (EOR)	^	→
	Bitwise operator (OR)	|	→
	Logical operator (AND)	&&	→
	Logical operator (OR)	||	→
	Conditional operator	?:	←
↓	Assignment operator	= += - = * = /= %= <<= >>= &= ^= |=	←
Lowest	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.

1.4 Control Statements

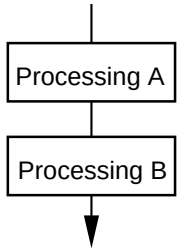
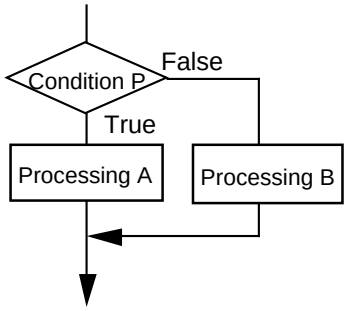
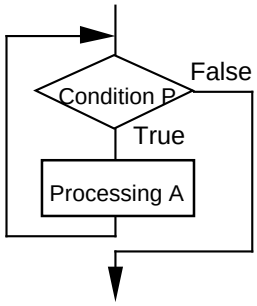
1.4.1 Structuring of Program

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 The three Basic Forms of Structured Programming

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

1.4.2 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.

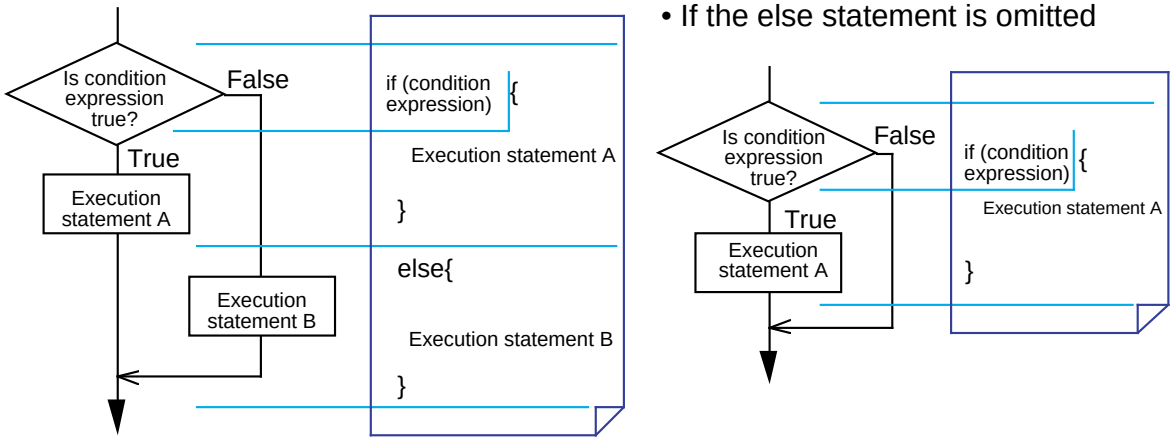


Figure 1.4.1 Example of "if-else Statement

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" (m inutes 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.

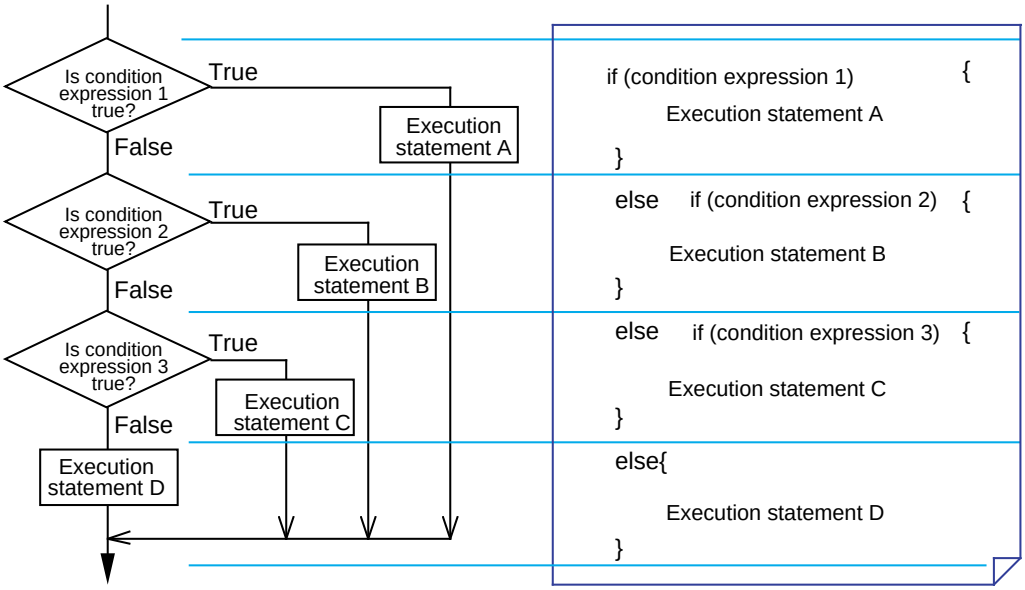


Figure 1.4.2 Example of “else-if” Statement

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);

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

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

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

Declares the variables used.

Defines "select" function.

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

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

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

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

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

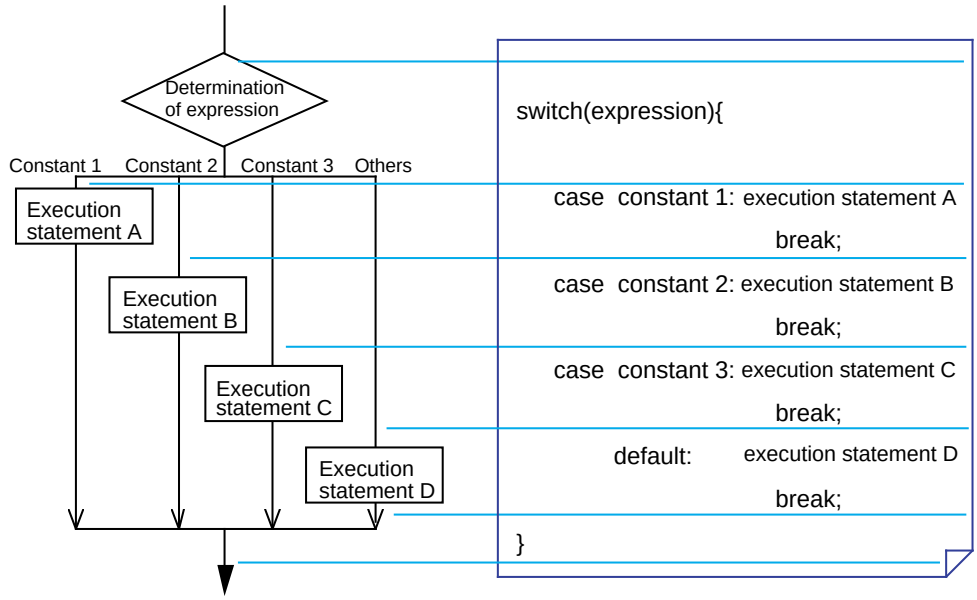


Figure 1.4.3 Example of switch-case Statement

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);

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

void  select(void)
{
  switch(sw){

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

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

Declares the variables used.

Defines "select" function.

Determines the content of "sw".

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

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

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

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

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

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 completing the current 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.

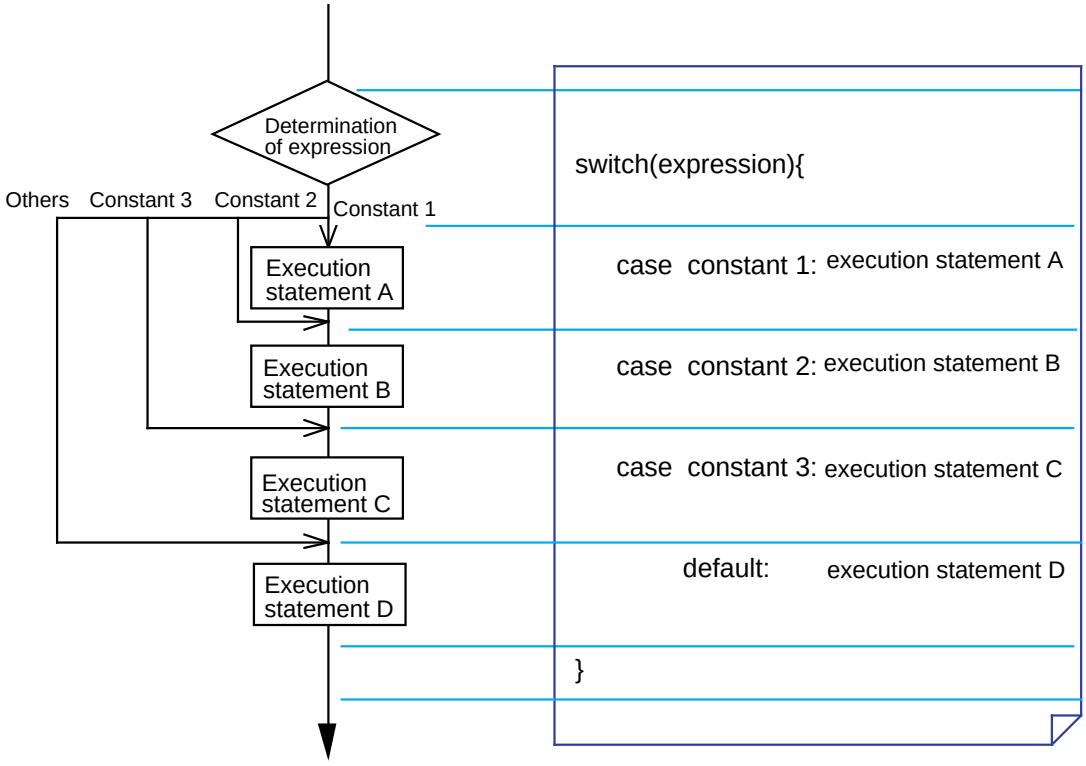


Figure 1.4.4 switch-case Statement without Break

1.4.3 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".

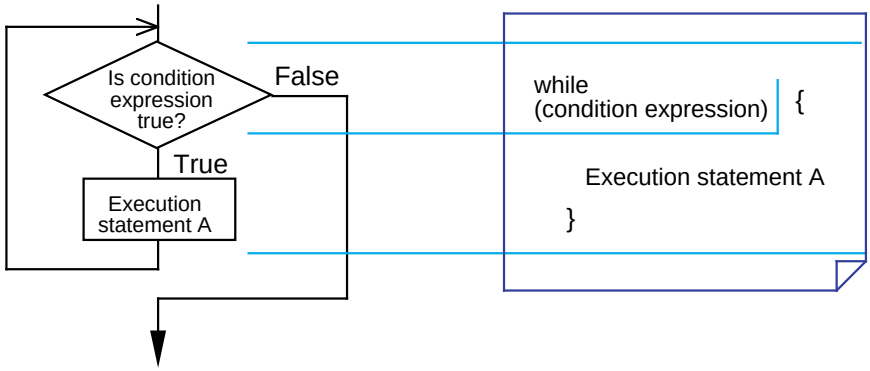


Figure 1.4.5 Example of while Statement

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.

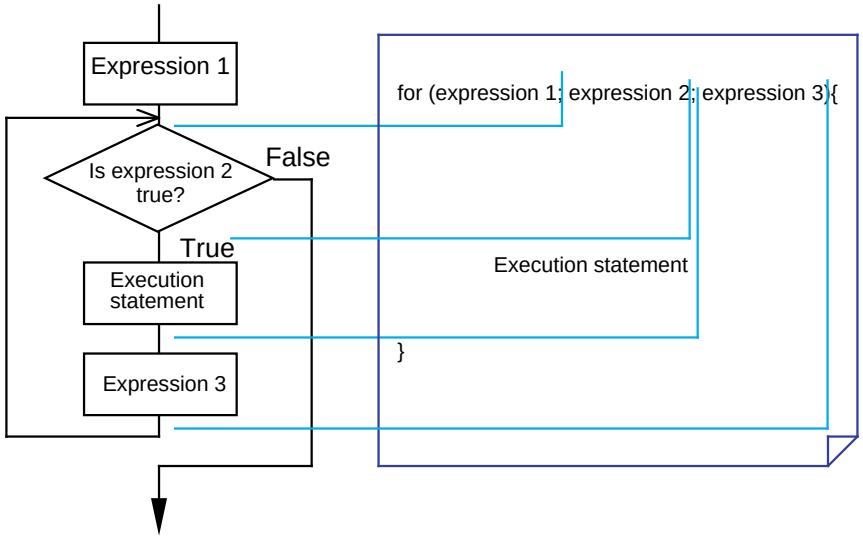


Figure 1.4.6 Example of "for" Statement

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) ;

unsigned  int  total = 0 ;

void  sum(void)
{
   unsigned   int  i ;

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

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

Declares the variables used.

Defines "sum" function.

Defines counter variables.

Loops until the counter content increments from 1 to 100.

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 executed, all processing in a do-while statement is executed at least once.

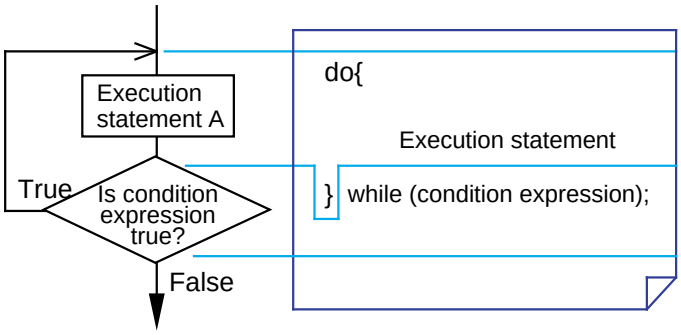


Figure 1.4.7 Example of do-while Statement

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)

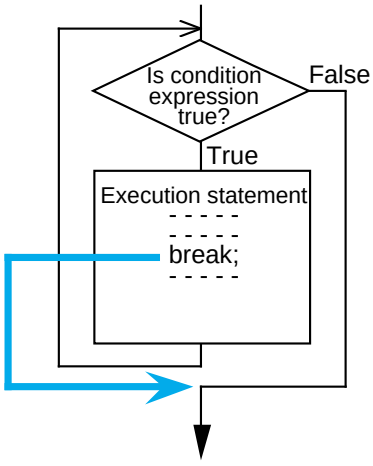
1.4.4 Suspending Processing

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

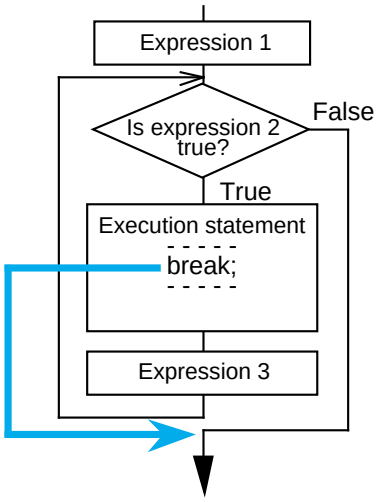
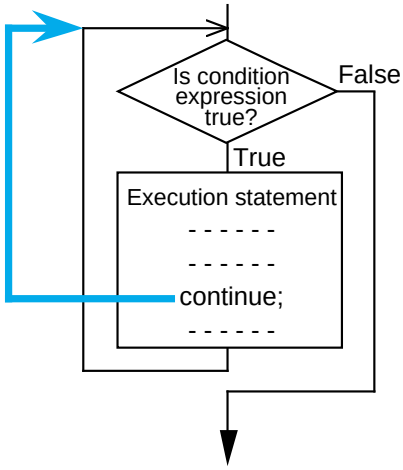


Figure 1.4.8 Example of break Statement

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

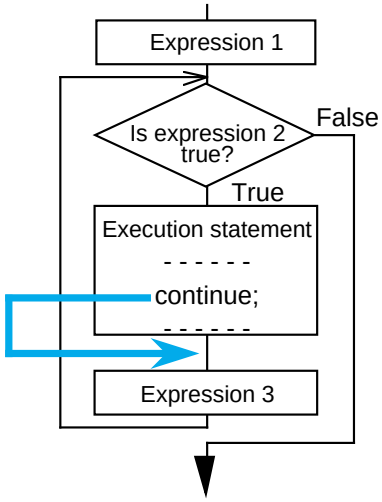


Figure 1.4.9 Example of continue Statement

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.

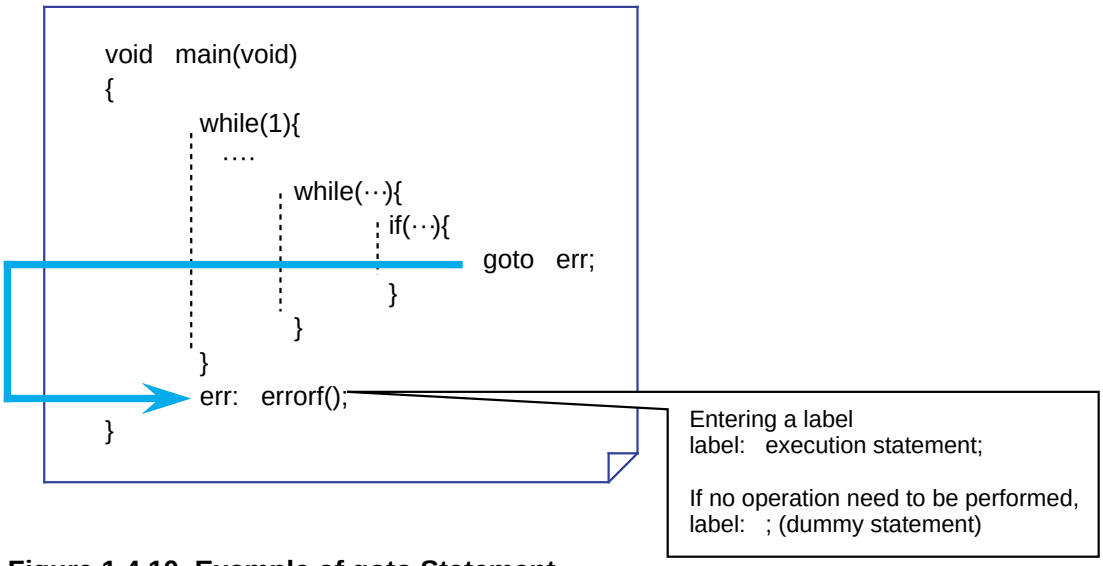


Figure 1.4.10 Example of goto Statement

1.5 Functions

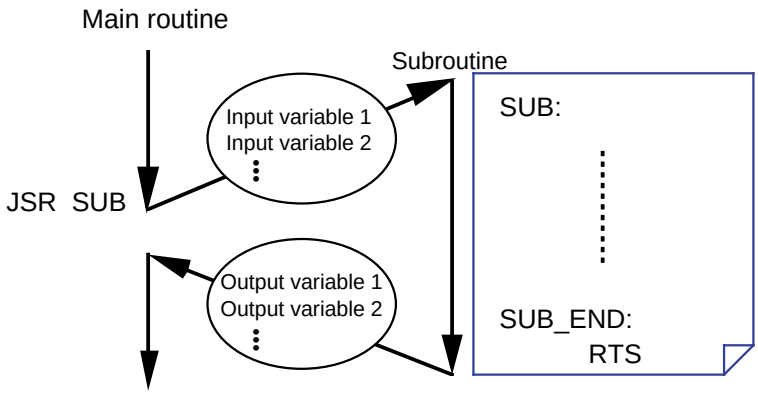
1.5.1 Functions and Subroutines

As subroutines are the basic units of a program in assembly language, so are "functions" in C language.
This section explains how to write functions in NC30.

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 assembly language, no restrictions are imposed on the number of input or output variables. In 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. (Note)

• "Subroutine" in assembly language



• "Function" in C language

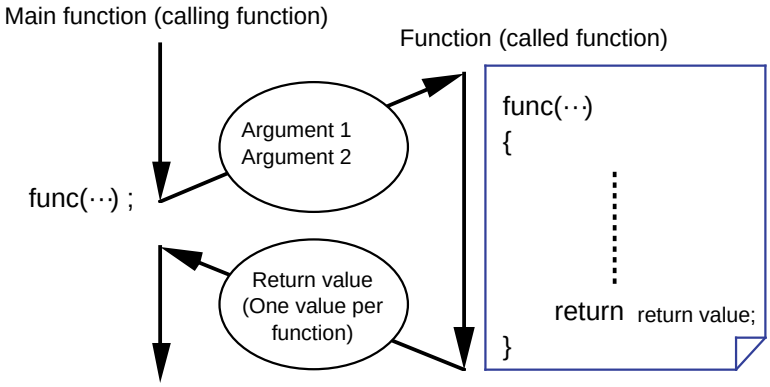


Figure 1.5.1 "Subroutine" vs. "Function"

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

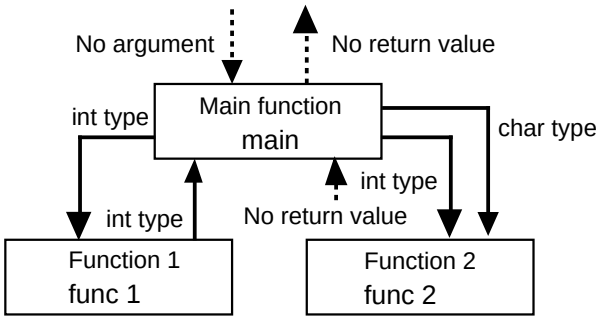
```
function name (argument 1, ...);
```

When there is a return value

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

Example for a Function

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




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

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

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

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

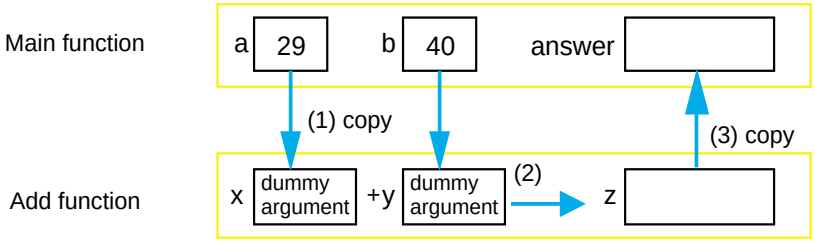
```
/*   Prototype declaration   */
void   main ( void ) ;
long   add ( int , int ) ;

/*   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.

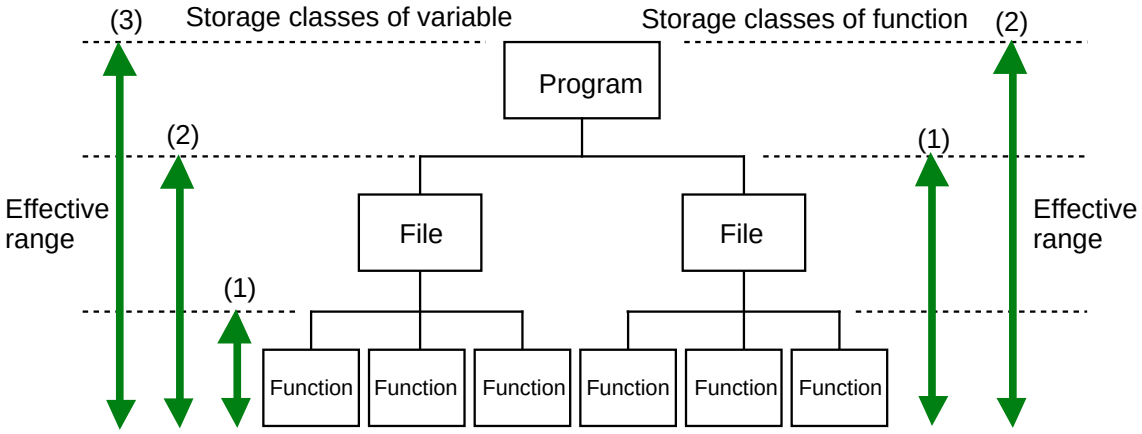


Figure 1.6.1 Hierarchical Structure and Storage Classes of C Language Program

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.

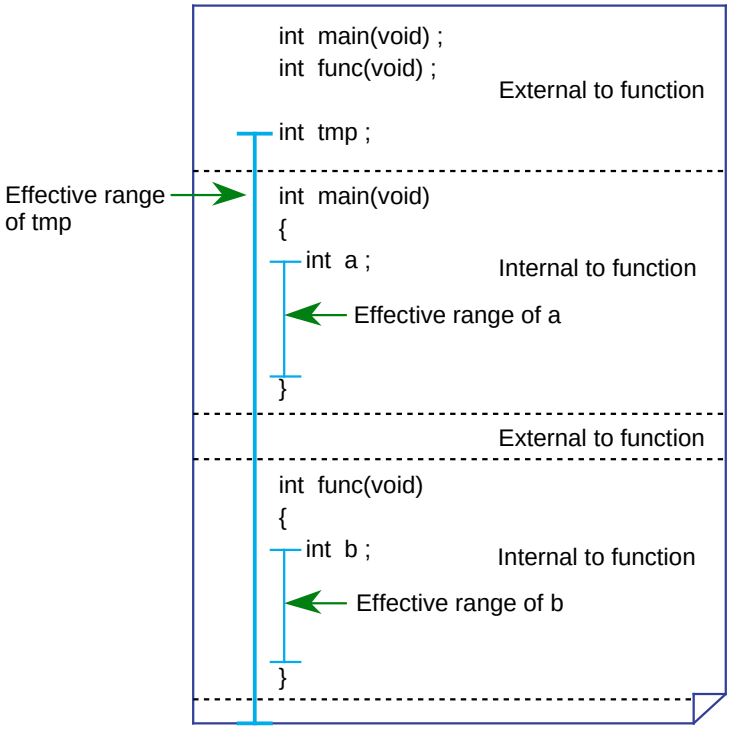


Figure 1.6.2 External and Internal Variables

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.

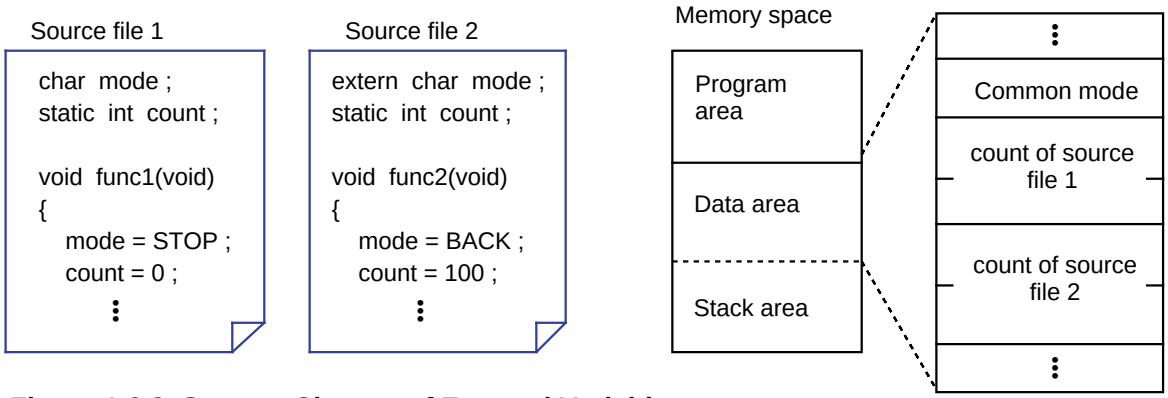


Figure 1.6.3 Storage Classes of External Variable

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.

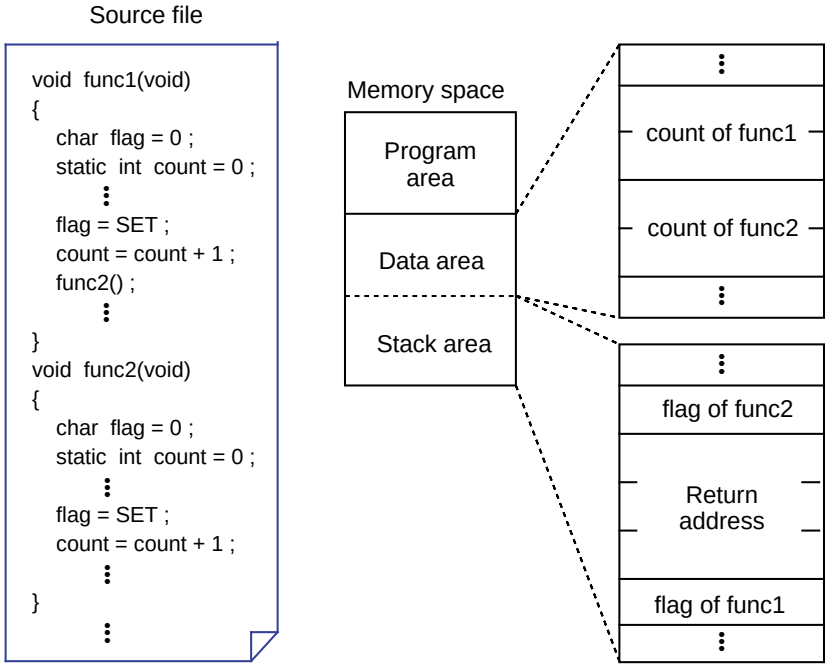


Figure 1.6.4 Storage Classes of Internal Variable

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".

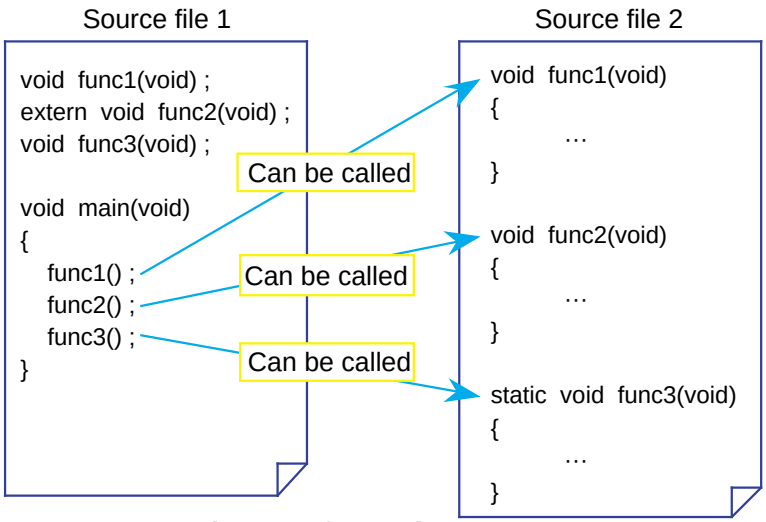


Figure 1.6.5 Storage Classes of Function

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 a register when executing the function] However, they do not have any effect in NC30 (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 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]

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 boy.

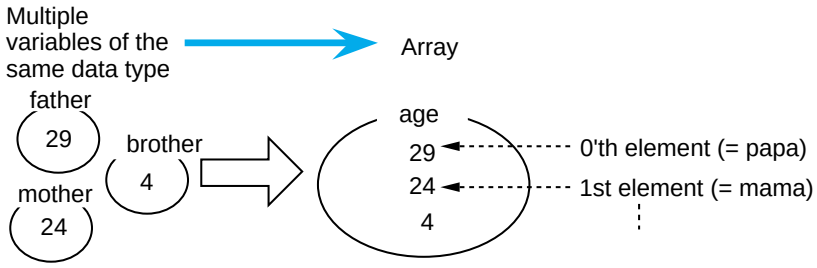


Figure 1.7.1 Concept of an Array

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).

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

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

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

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

    total = father + mother + brother  + sister 1 + sister 2 + ⋯;
}
```

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.

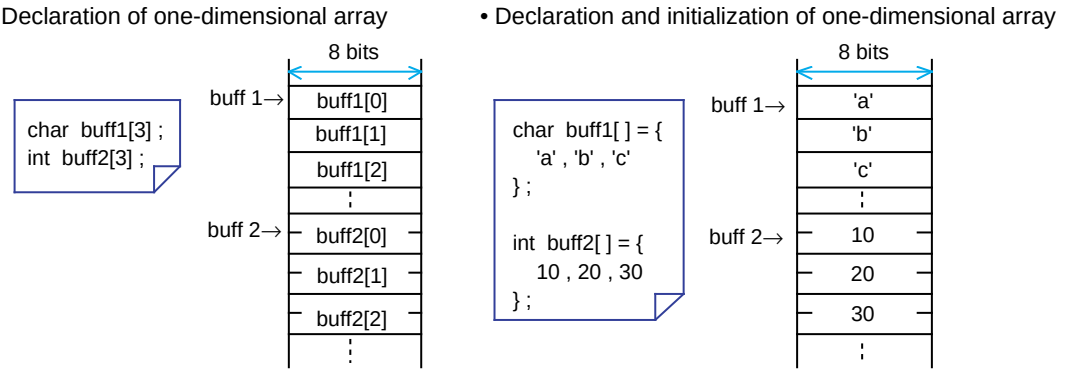


Figure 1.7.2 Declaration of One-dimensional Array and Memory Mapping

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).

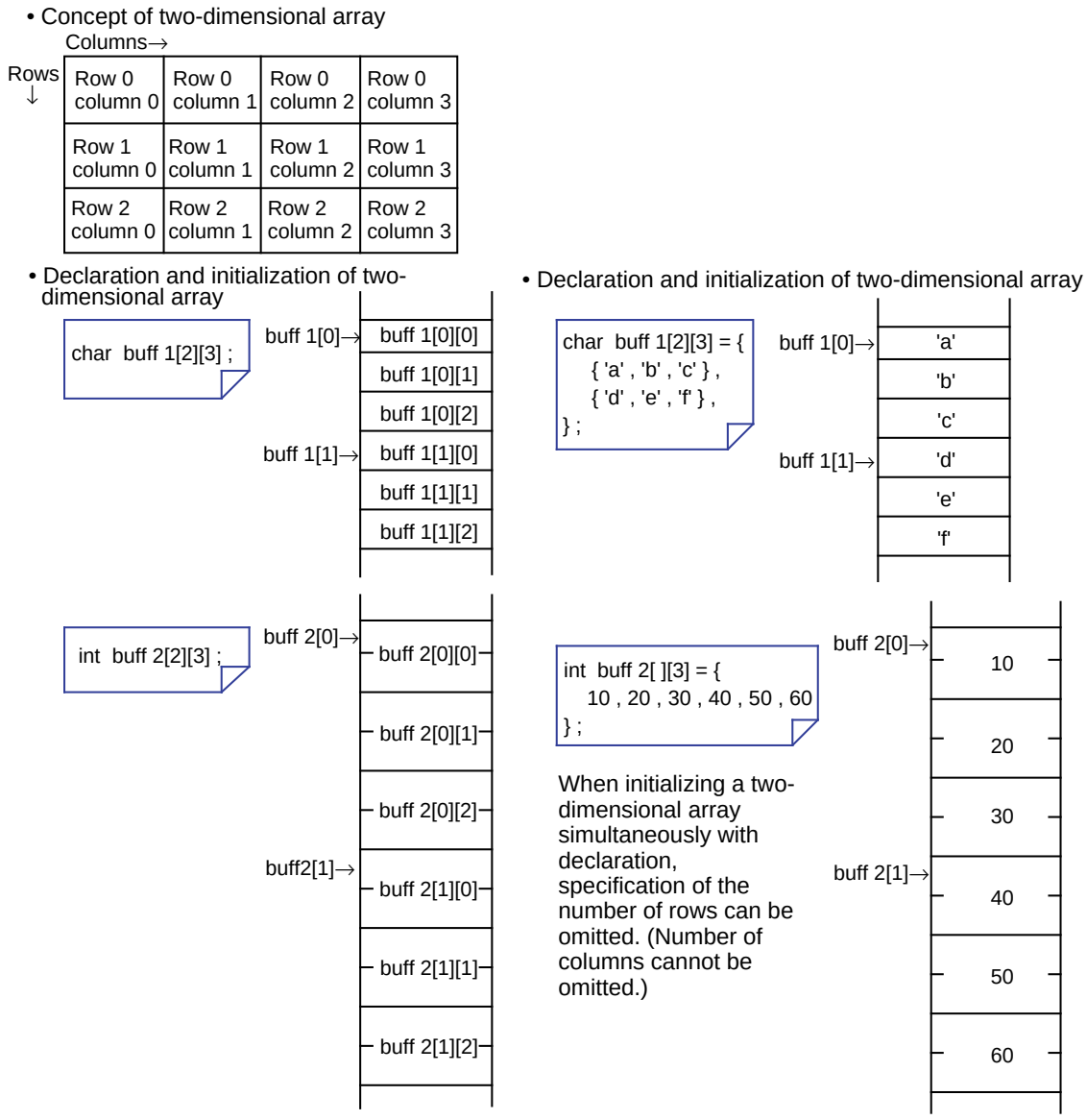


Figure 1.7.3 Declaration of Two-dimensional Array and Memory Mapping

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

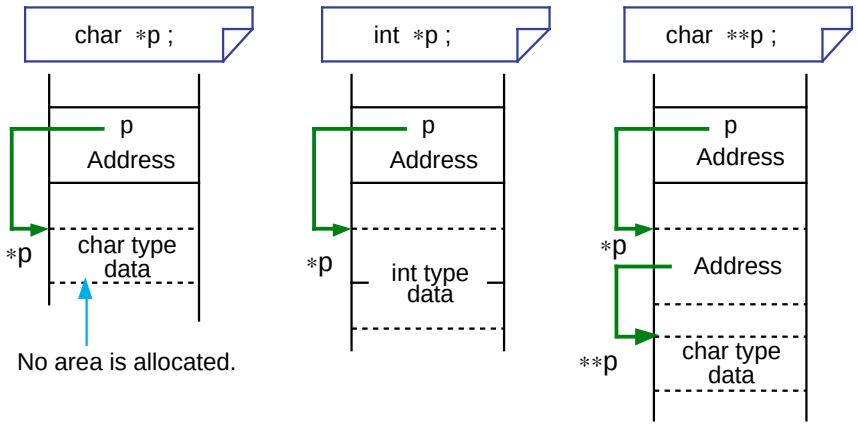
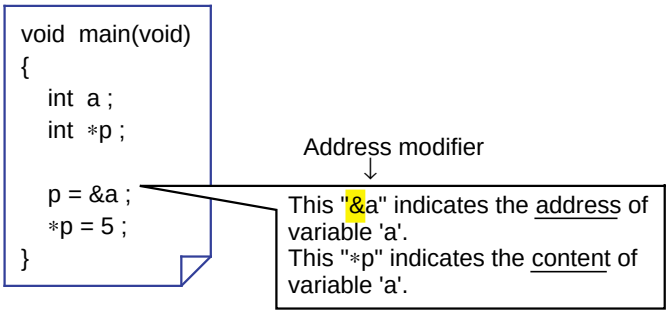


Figure 1.7.4 Pointer Variable Declaration and Memory Mapping

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.



The result is a = 5.

Figure 1.7.5 Relationship Between Pointer Variables and 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 NC30 are two or four bytes in data length depending on the location (near or far area) where the corresponding data is stored. For details about this, refer to Section 2.1, "Memory Mapping".

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.

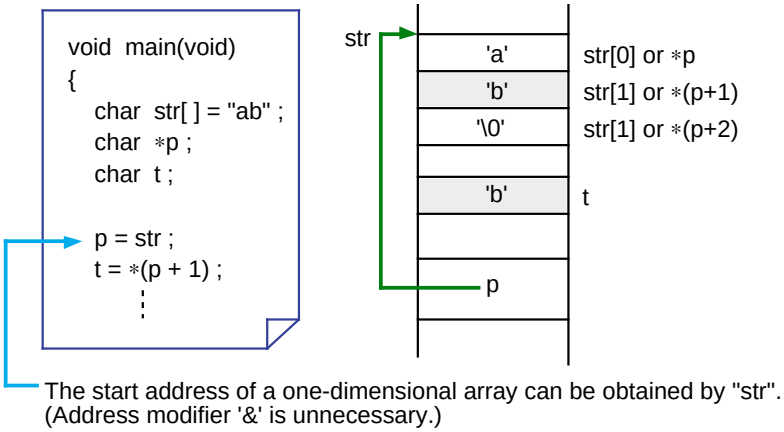


Figure 1.7.6 Pointer Variables and One-dimensional Array

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.

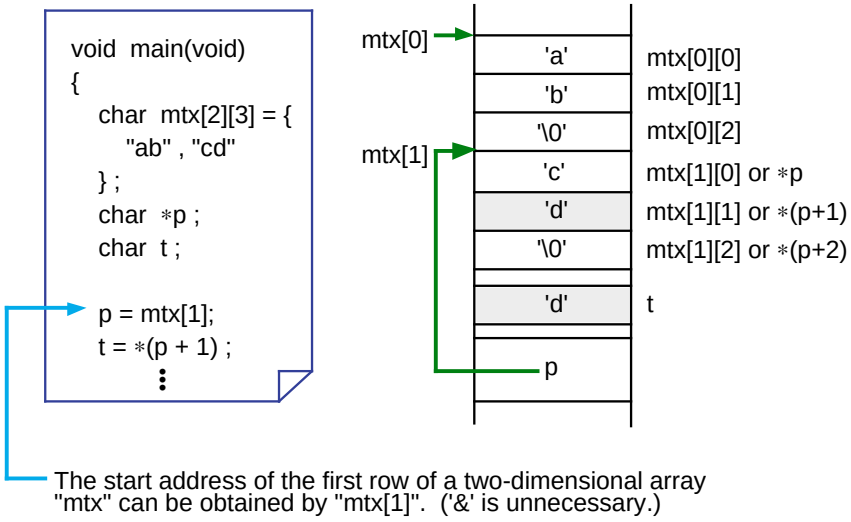


Figure 1.7.7 Pointer Variables and Two-dimensional Array

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 1.7.8 shows an example where an array is passed between functions using the Call by Reference method.

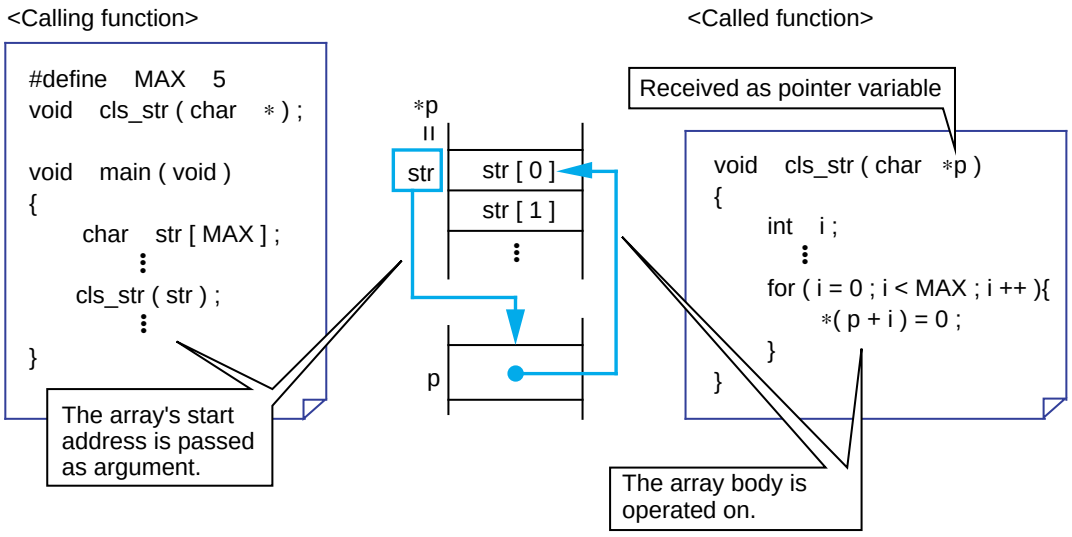


Figure 1.7.8 Example of Call by Reference for Passing an Array

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 loosing the independency of functions and, hence, is not recommended for use in C language programs. Yet, it has the advantage that functions can be called at high speed because entry and exit processing (argument and return value transfers) normally required when calling a function are unnecessary. Therefore, this method is frequently used in ROM'ed programs where general-purpose capability is not an important requirement and the primary concern is high-speed processing.

1.7.5 Placing Pointers into an Array

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

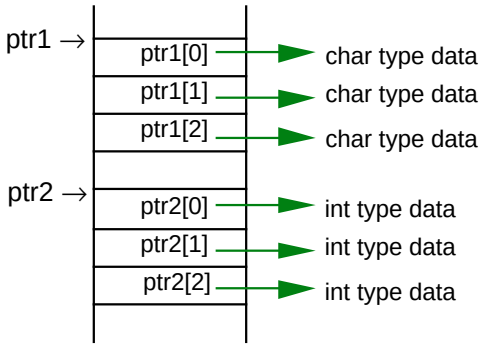
Pointer Array Declaration

The following shows how to declare a pointer array.

Data type 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"
} ;
```

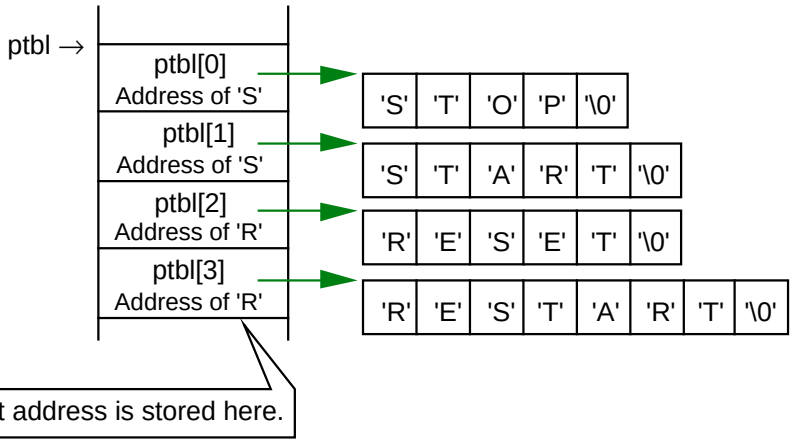


Figure 1.7.9 Pointer Array Declaration and Initialization

Note: In NC30, the body data of a pointer array is located in the far area. Consequently, be sure to write "far" for the pointer. (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"
} ;
```

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

Filled with null code.

• Pointer array

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

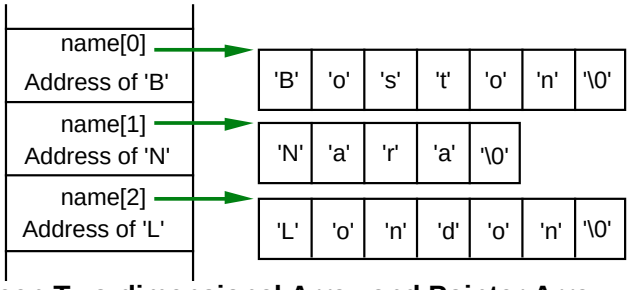


Figure 1.7.10 Difference between Two-dimensional Array and Pointer Array

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.

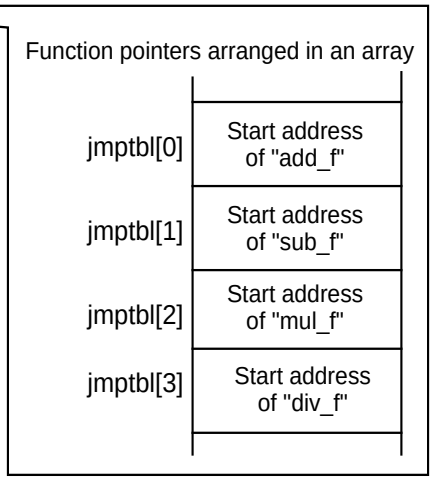
<Declaration format> Type of return value (* function pointer name) (data type of argument);

<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 ;
}
```



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.

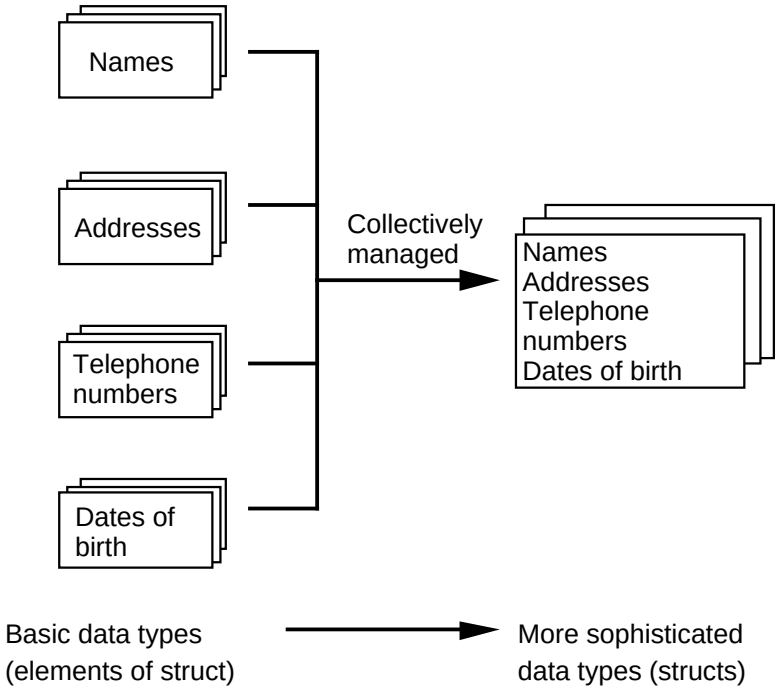


Figure 1.8.1 From Basic Data Types to Structs

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.

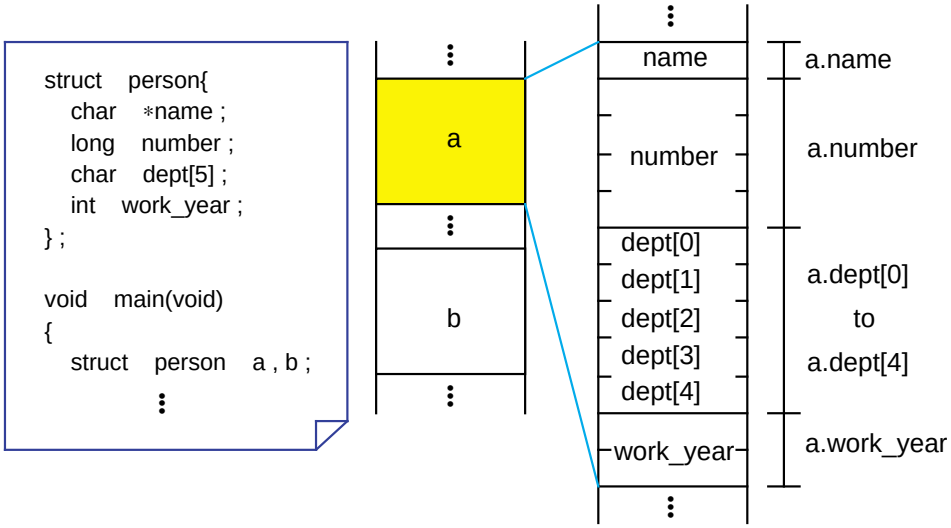
```
struct Δ struct tag Δ struct variable name;
```

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

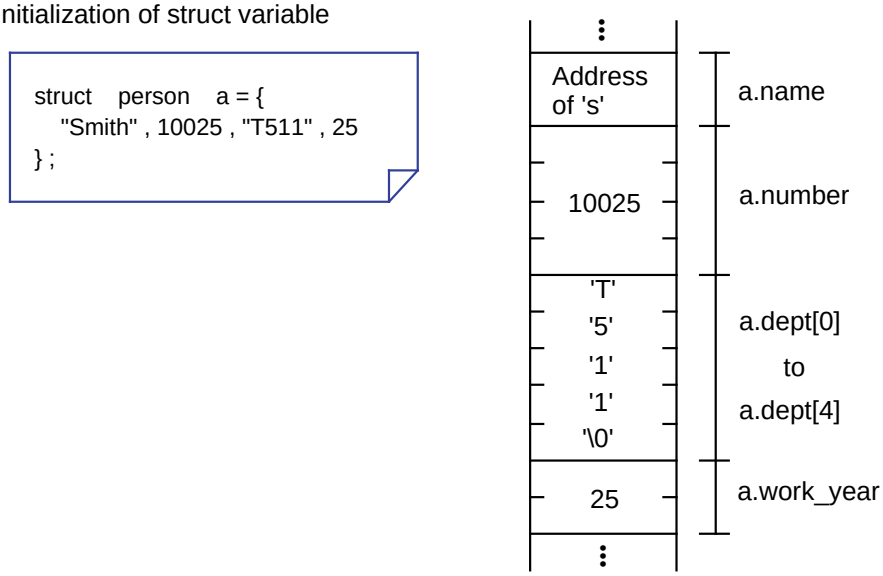


Figure 1.8.2 Struct Declaration and Memory Mapping

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    dept[5] ;
    int     work_year ;
} ;
struct    person    a = {
    "Smith" , 10025 , "T511" , 25
} ;

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

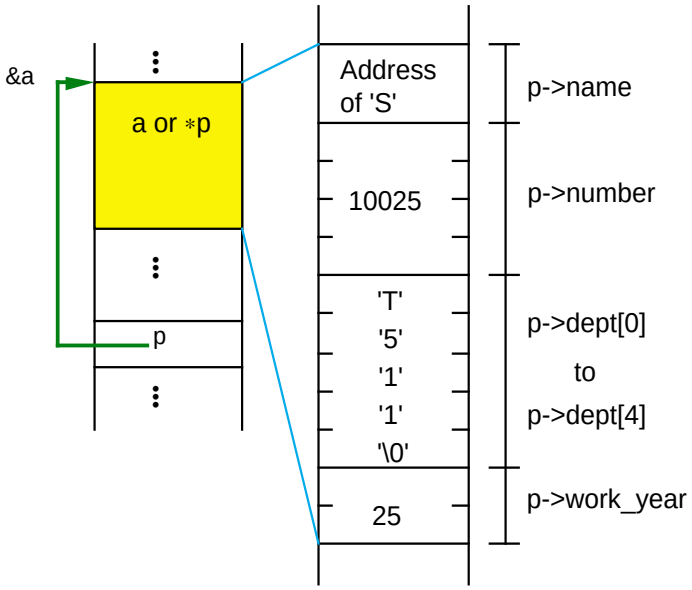
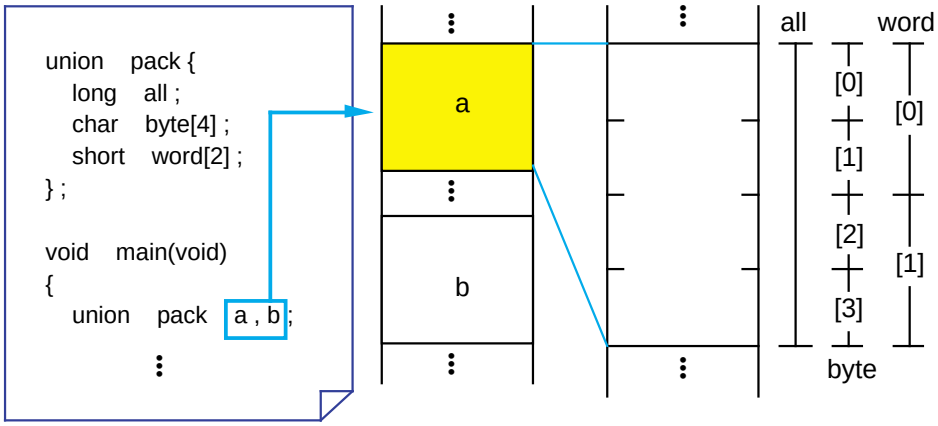


Figure 1.8.3 Example for Referencing Members Using a Pointer

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.

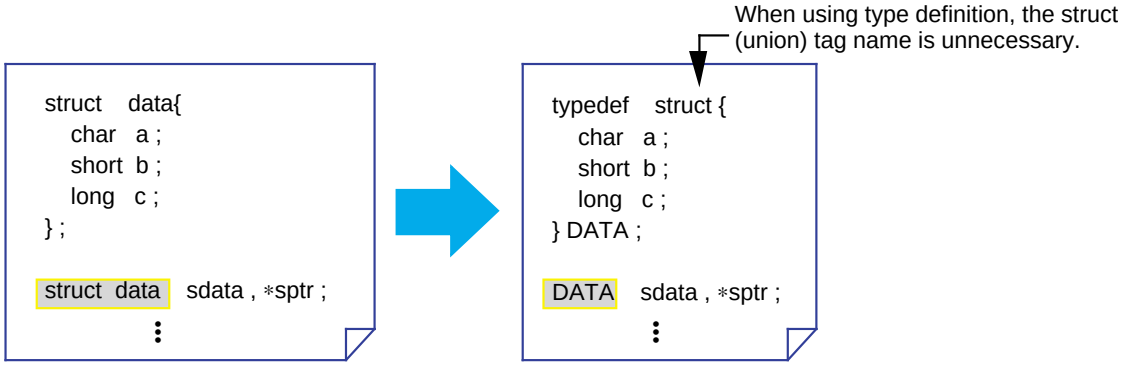


Figure 1.8.5 Example for Using Type Definition "typedef"

1.9 Preprocess Commands

1.9.1 Preprocess Commands of NC30

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 NC30.

Preprocess Command List of NC30

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 NC30.

Table 1.9.1 Main Preprocess Commands of NC30

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. NC30 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, NC30 searches the standard directory that is set with NC30's environment variable "INC30" 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, NC30 searches the directory specified with the startup option '–I' and the directory set with NC30's environment variable "INC30" 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"

NC30's command "#include" can be nested in up to 8 levels. If the specified file cannot be found in any directory searched, NC30 outputs an include error.

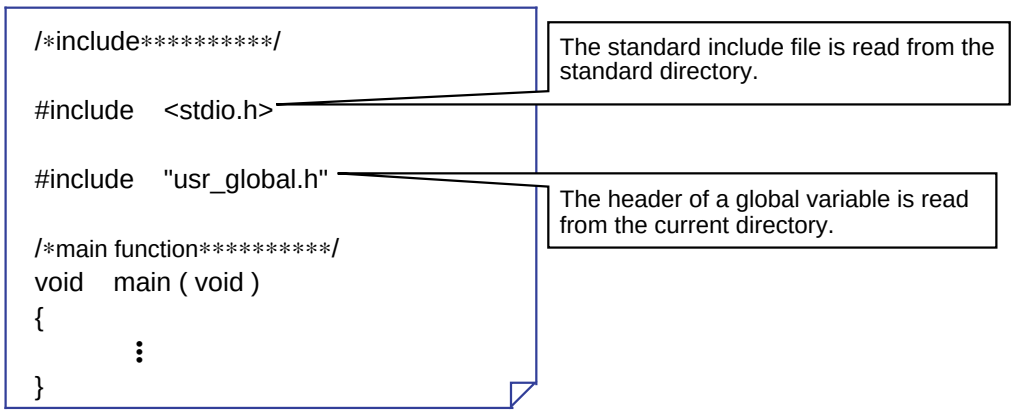


Figure 1.9.1 Typical Description of "#include"

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 (immediates with unknown meanings) in the program.

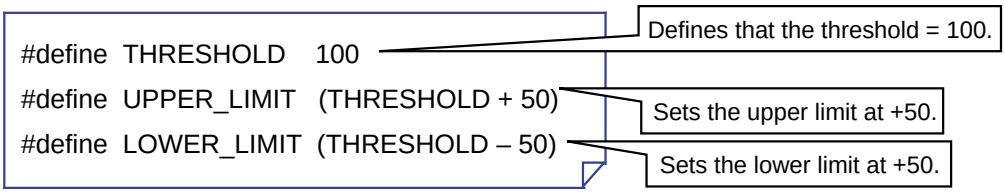


Figure 1.9.2 Example for Defining a Constant

Defining a Character String

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

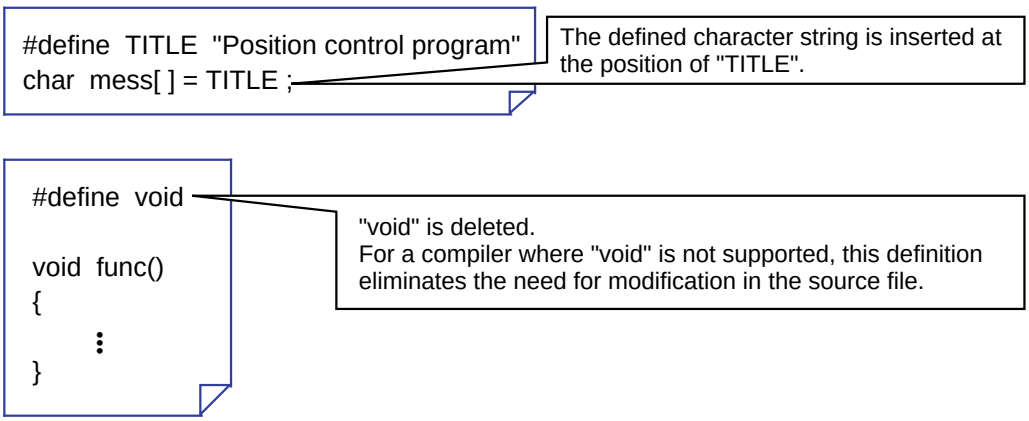


Figure 1.9.3 Example for Defining a Character String

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.

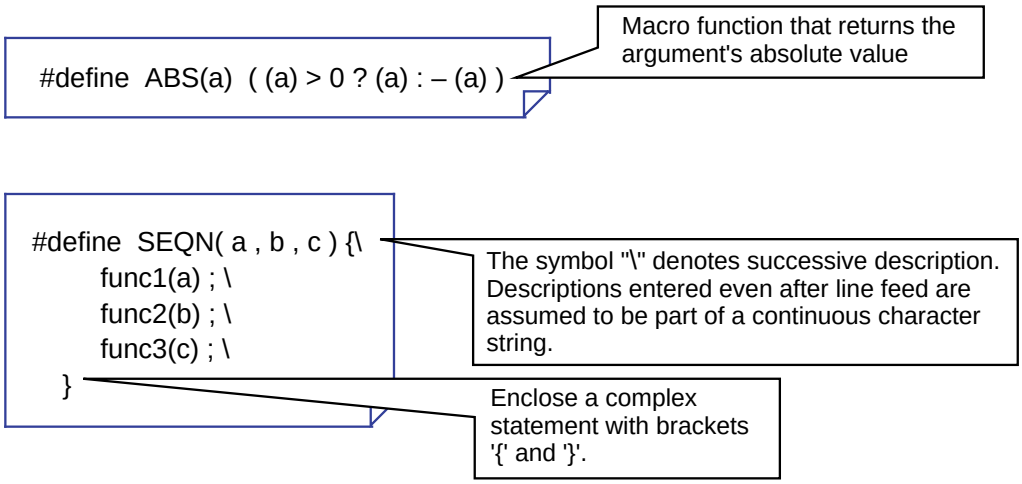


Figure 1.9.4 Example for Defining a Macro Function

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

NC30 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 NC30.

Table 1.9.2 Types of Conditional Compile

Description	Content
#if condition expression A #else B #endif	If the condition expression is true (not 0), NC30 compiles block A; if false, it compiles block B.
#ifdef identifier A #else B #endif	If an identifier is defined, NC30 compiles block A; if not defined, it compiles block B.
#ifndef identifier A #else B #endif	If an identifier is not defined, NC30 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 NC30 startup option '-D'.

```
#define identifier          ←Specification of definition by "#define"
%nc30 -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.

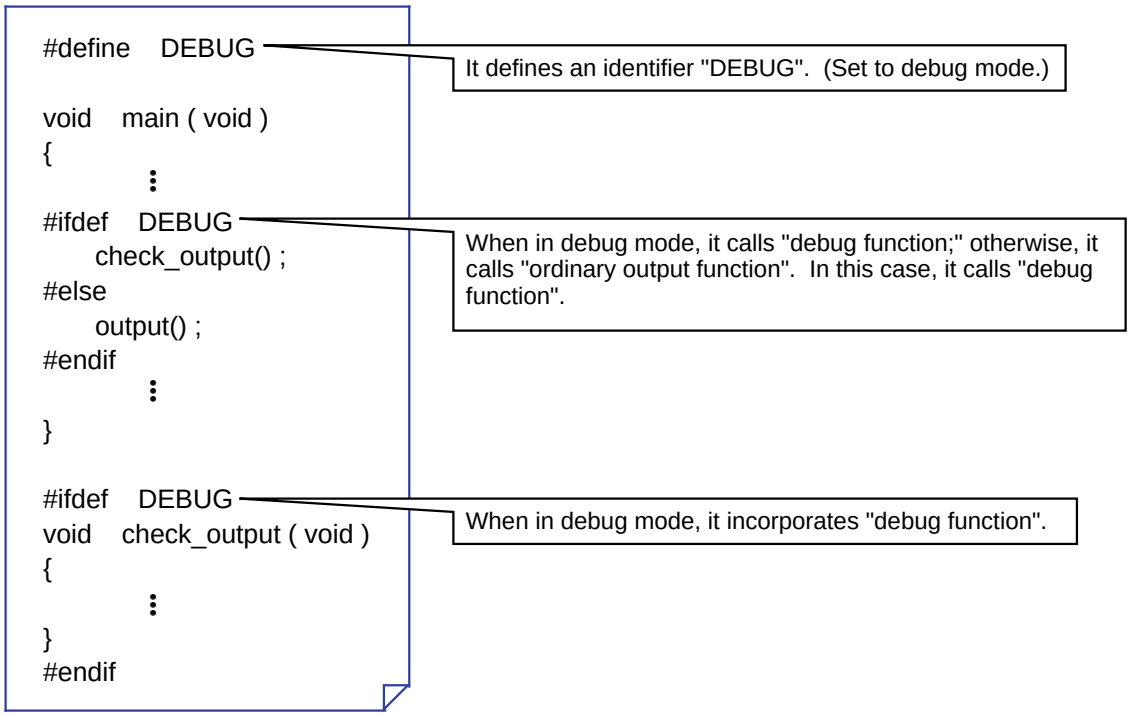


Figure 1.9.5 Example for Conditional Compile Description

Chapter 2

Extended Functions of NC30

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 NC30.

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 NC30.

Data and Code Generated by NC30

Figure 2.1.1 shows the types of data and code generated by NC30 and their mapped memory areas.

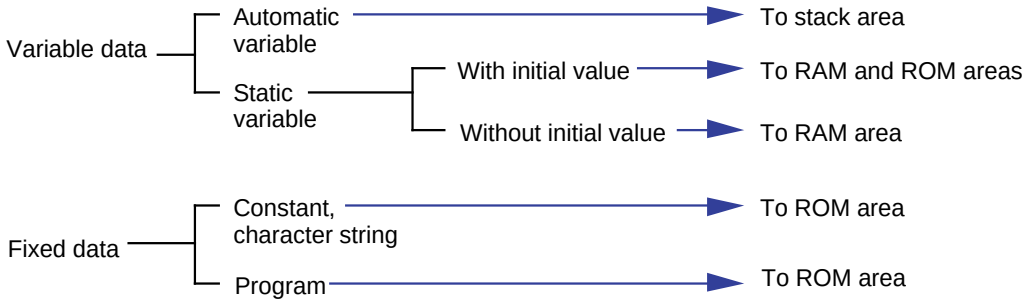


Figure 2.1.1 Types of Data and Code Generated by NC30 and Their Mapped Areas

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, NC30 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.

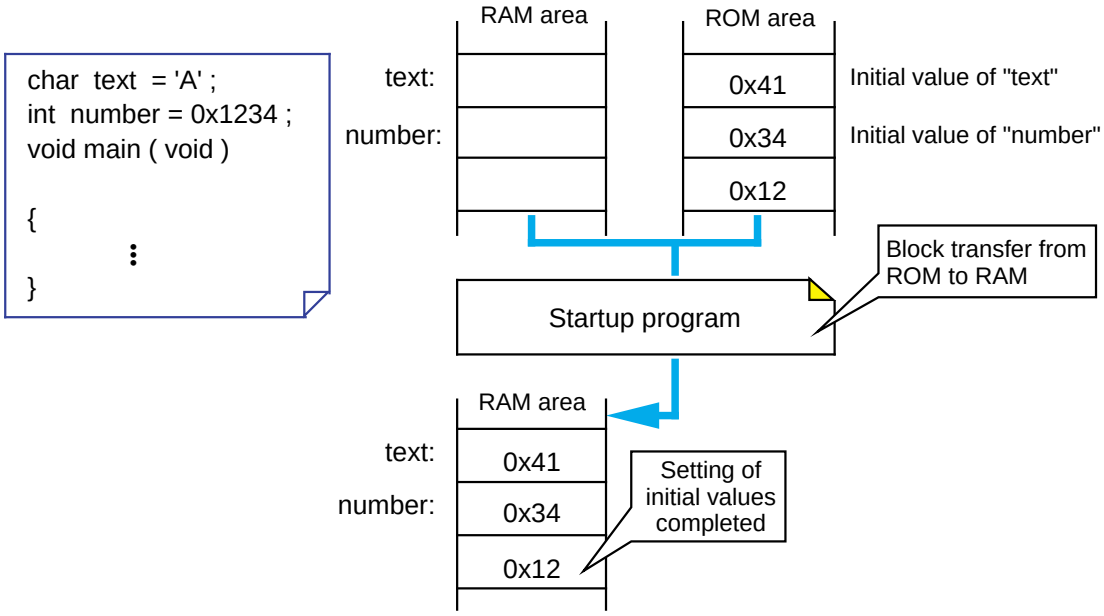


Figure 2.1.2 Handling of Static Variables with Initial Values

2.1.2 Sections Managed by NC30

NC30 manages areas in which data and code are located as "sections".
This section explains the types of sections generated and managed by NC30 and how they are managed.

Sections Types

NC30 classifies data into sections by type for management purposes. (See Figure 2.1.3.)
Table 2.1.1 lists the sections types managed by NC30.

Table 2.1.1 Sections types Managed by NC30

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.
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)

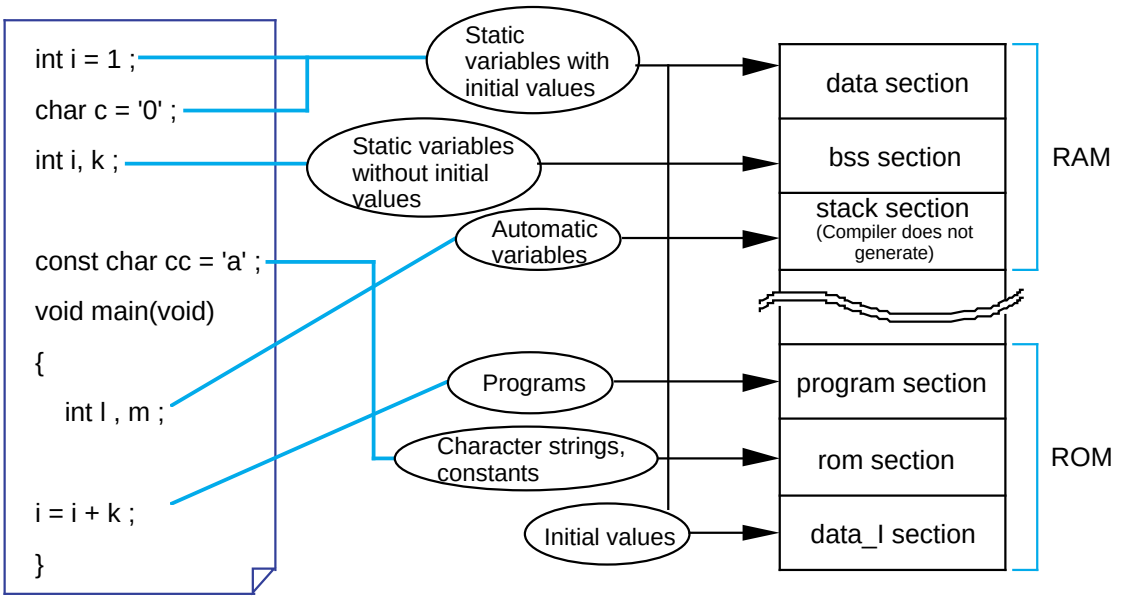


Figure 2.1.3 Mapping Data into Sections by Type

Sections Attributes

The sections generated by NC30 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.
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-byte area at absolute addresses from 0 to 0FFFF) F-far attribute (entire 1-Mbyte memory area from address 0 to FFFFF) S-SBDATA attribute (area where SB relative addressing can be used)	data,bss,rom
E/O	E-Data size is even. O-Data size is odd.	data,bss,rom

For details on how to specify these attributes, refer to Section 2.3.1, "Efficient Addressing".

Rule for Naming Sections

The sections generated by NC30 are named after their section base name and attributes.
Figure 2.1.4 shows a combination of each section base name and attributes.

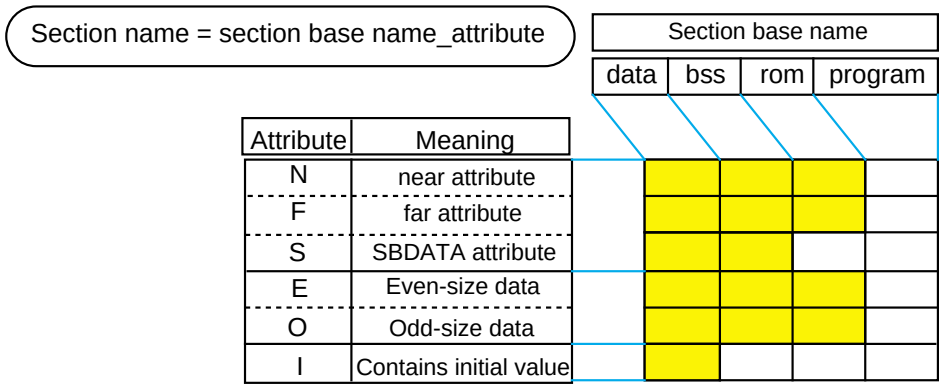


Figure 2.1.4 Rule for Assigning Section Names

2.1.3 Control of Memory Mapping

NC30 provides extended functions that enable memory mapping to be performed in an efficient way to suit the user's system.
This section explains NC30'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 NC30. The effective range of a changed name varies between cases when "program" is changed and when some other section base name is changed.

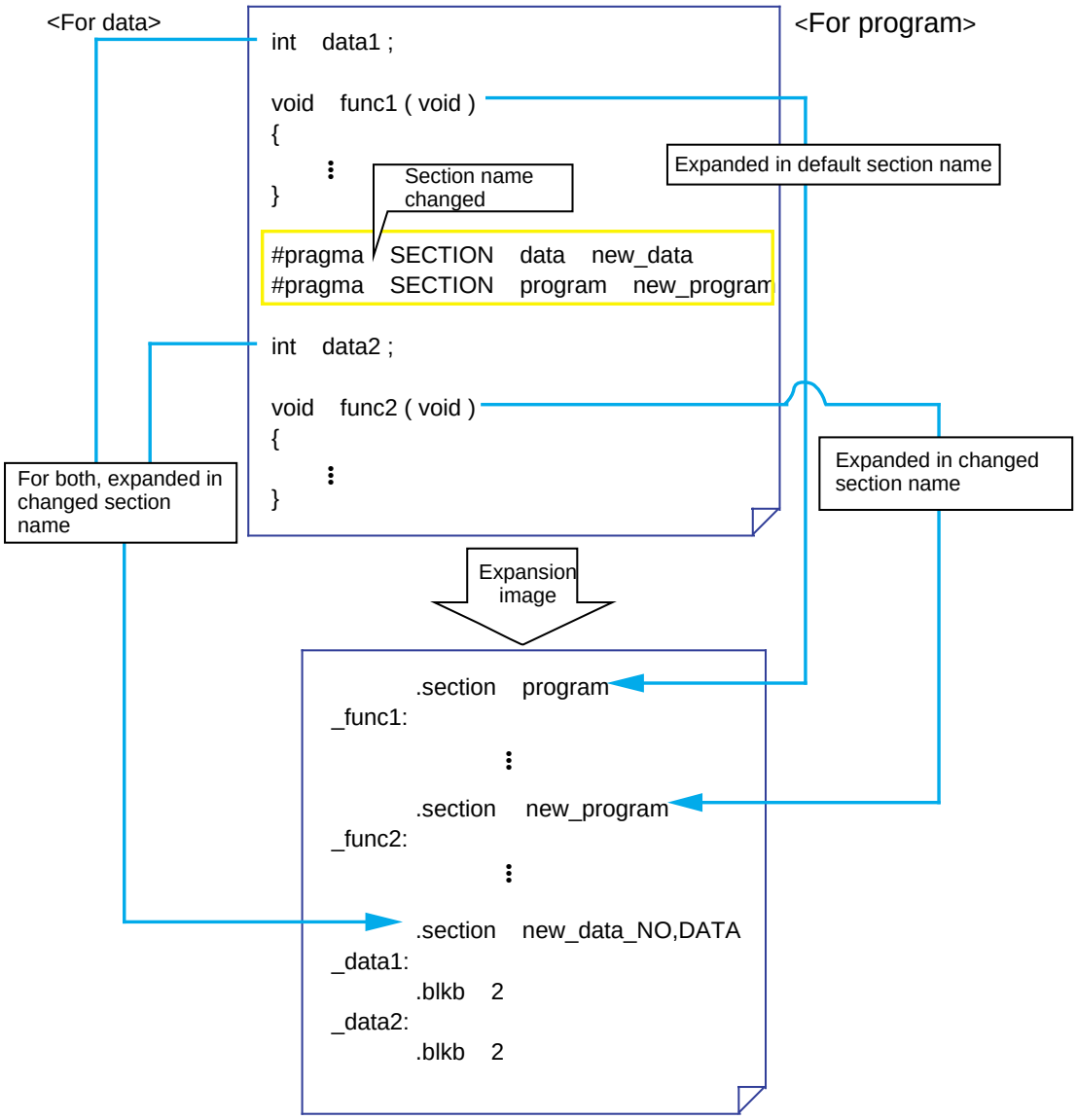


Figure 2.1.5 Typical Description of "#pragma SECTION"

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.

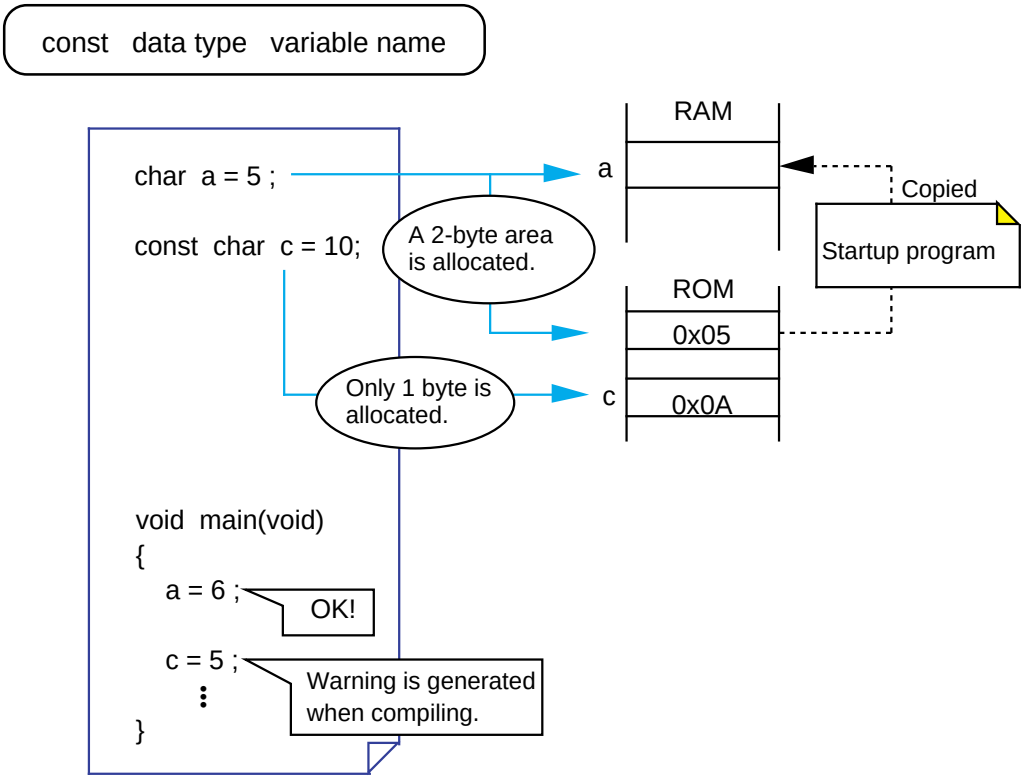


Figure 2.1.6 const Modifier and Memory Mapping

2.1.4 Controlling Memory Mapping of Struct

When allocating memory for structs, NC30 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 NC30's specific extended functions used for mapping structs into memory.

NC30 Rules for Mapping Structs into Memory

NC30 follow 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.

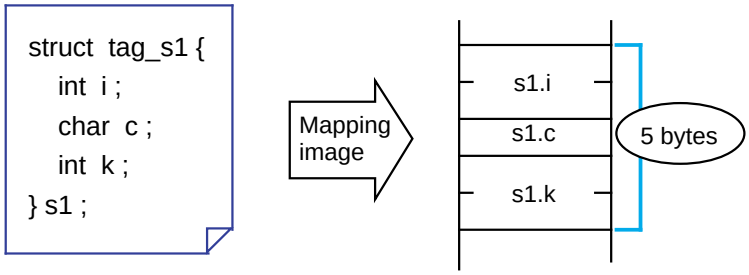


Figure 2.1.7 Image Depicting How NC30's Default Struct is Mapped into Memory

Inhibiting Struct Members from Being Packed (#pragmaΔSTRUCTΔtag nameΔunpack)

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.

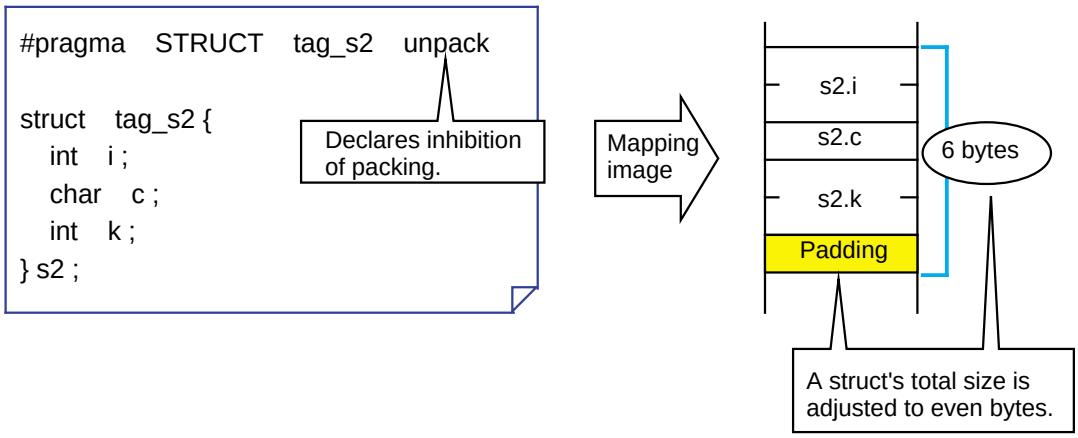


Figure 2.1.8 Inhibiting Struct Members from Being Packed

Optimizing Mapping of Struct Members (#pragmaΔSTRUCTΔtag nameΔarrange)

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.

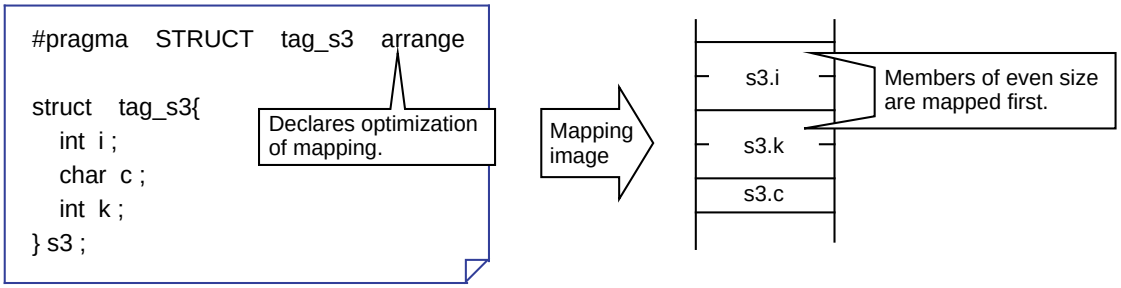


Figure 2.1.9 Optimizing Memory Allocation for Struct Members

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 NC30, "ncrt0.a30" and "sect30.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

NC30's startup program consists of two files: "ncrt0.a30" and "sect30.inc".

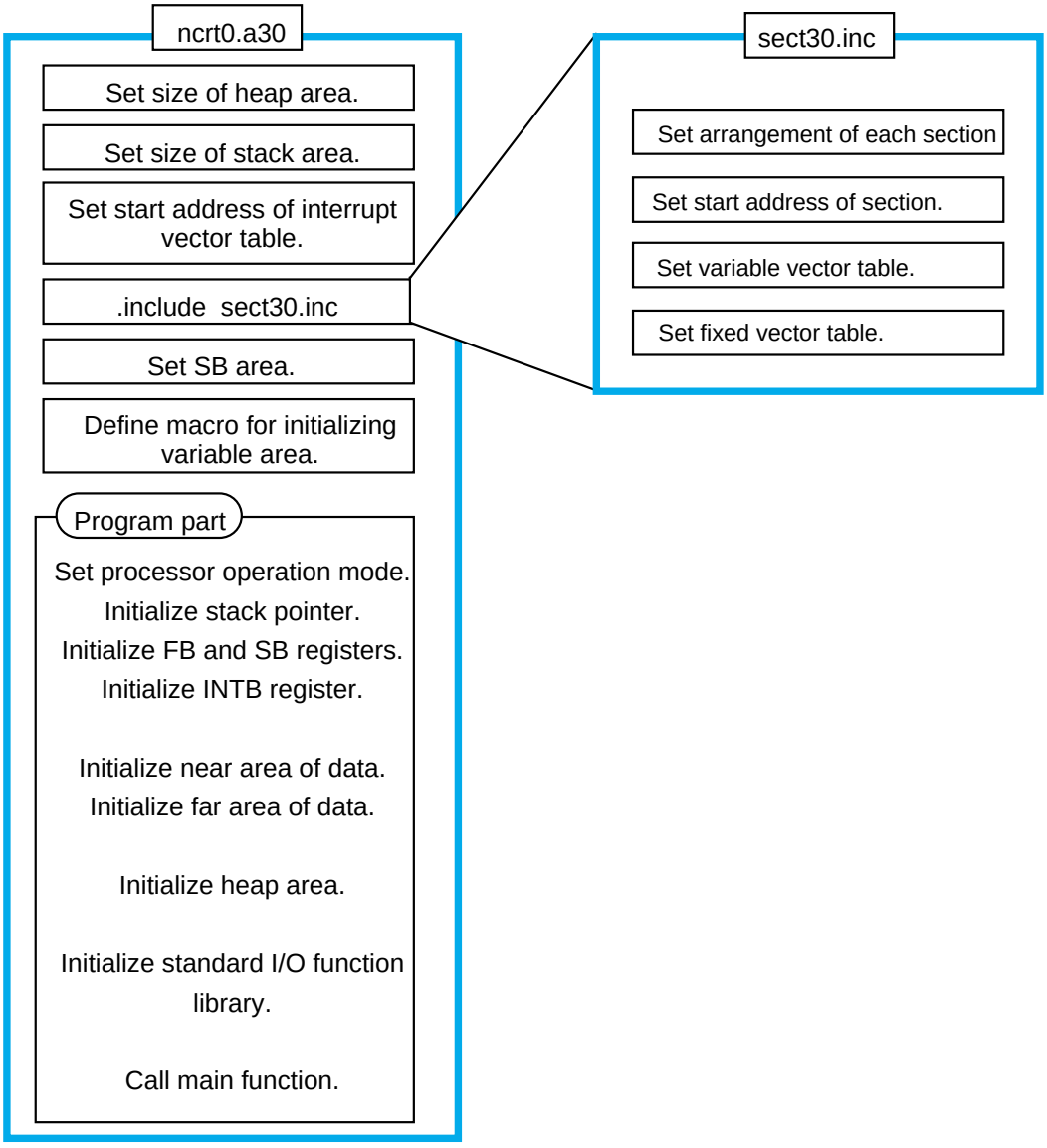


Figure 2.2.1 Structure of Sample Startup Program

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

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" when starting up NC30. 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.

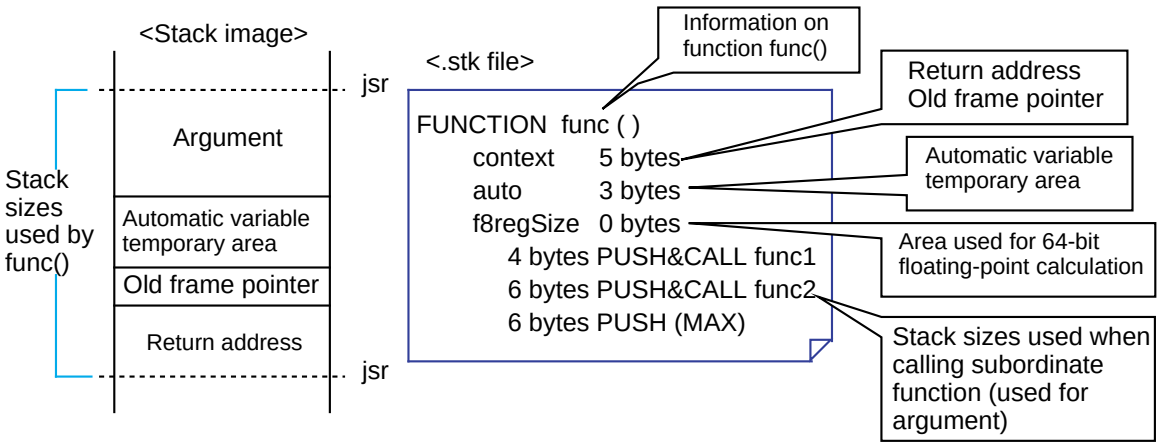


Figure 2.2.2 Stack Size Usage Information File

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.

<Source file "sample.c">

```
void    main ( void ) ;
int     func1 ( int , int ) ;
int     func2 ( char , char ) ;
int     func3 ( int ) ;

void  main ( void )
{
        int  m , n ;
        long  kekka1 , kekka2 ;

        kekka1 = func1 ( m , n ) ;
        kekka2 = func2 ( m , n ) ;
}

int  func1 ( int x , int y )
{
        int  z1 , z2 ;

        z1 = x + y ;
        z2 = func3 ( z1 ) ;

        return    z2 ;
}

int  func2 ( char x , char y )
{
        int    z ;

        z = x  *  y  ;
        return    z ;
}
        int  func3 ( int x  )
{
        return  ~x ;
}
```

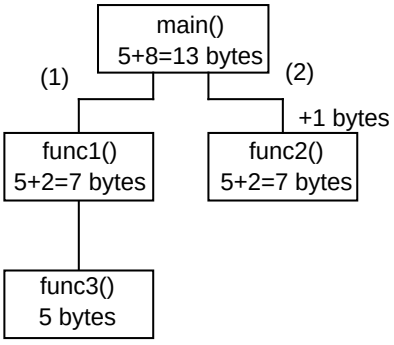
%nc30 -fshow_stack_usage sample.c

<Stack size usage information file "sample.stk">

```
FUNCTION main
        context         5 bytes
        auto            8 bytes
        f8regSize       0 bytes
                0 bytes PUSH &      CALL func1
                1 bytes PUSH &      CALL func2
                1 bytes PUSH (MAX)
=======================================

FUNCTION func1
        context         5 bytes
        auto            2 bytes
        f8regSize       0 bytes
                0 bytes PUSH &      CALL func3
                0 bytes PUSH (MAX)
=======================================

FUNCTION func2
        context         5 bytes
        auto            2 bytes
        f8regSize       0 bytes
                0 bytes PUSH (MAX)
=======================================

FUNCTION func3
        context         5 bytes
        auto            0 bytes
        f8regSize       0 bytes
                0 bytes PUSH (MAX)
=======================================
```

Stack size used by each function

Stack size used when calling a function



(1)Stack size for path : 13+7+5=25 bytes
(2)Stack size for path : 13+1+7=21 bytes

Maximum size of stacks used is 25 byes.

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, stk30" that is included with NC30. 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 stk30 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 stk30 by using the sample program described above.

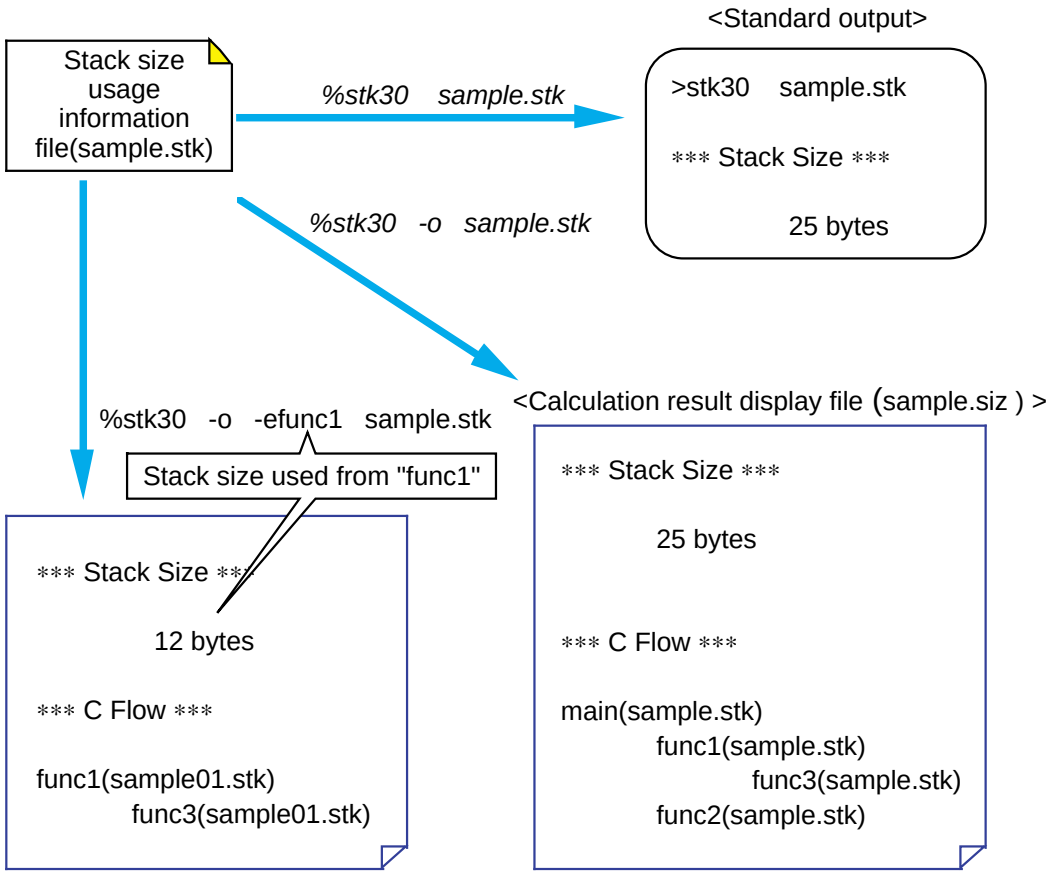


Figure 2.2.4 Stack Size Calculating Utility "stk30"

2.2.3 Creating Startup Program

The sample startup program shown above must be modified to suit the 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 the C language program to be created:

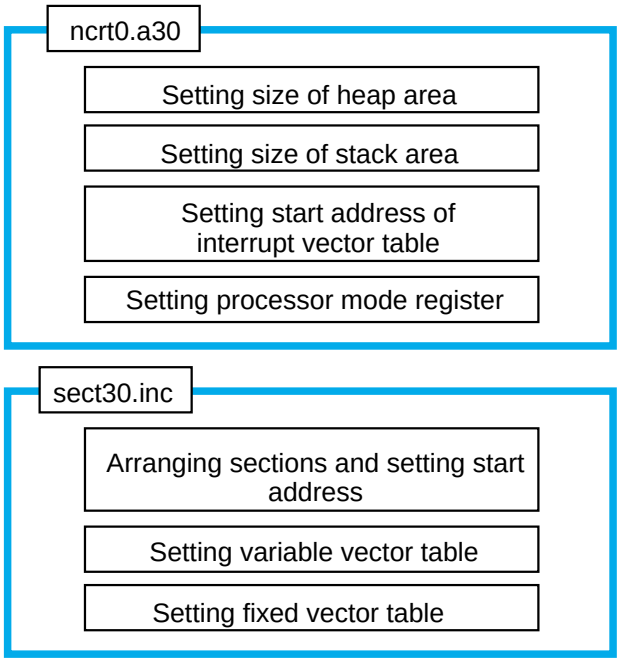


Figure 2.2.5 Points to Be Modified in Sample Startup Program

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.

```
;------------------------------------------------------------------------
;  HEAP SIZE  definition
;------------------------------------------------------------------------
HEAPSIZE                .equ        0
```

When not using memory management functions, set '0' and turn the heap area initialization section into comments.

```
;=========================================
;  heap area initialize
;------------------------------------------------------------------------
;               .glb            _mbase
;               .glb            _mnext
;               .glb            _msize
;               mov.w           #(heap_top&0FFFFH),_mbase
;               mov.w           #(heap_top>>16),_mbase+2
;               mov.w           #(heap_top&0FFFFH),_mnext
;               mov.w           #(heap_top>>16),_mnext+2
;               mov.w           #(heap_top&0FFFFH),_msize
;               mov.w           #(heap_top>>16),_msize+2
```

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 "stk30", 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.

```
;------------------------------------------------------------------------
;  STACK  SIZE  definition
;------------------------------------------------------------------------
STACKSIZE               .equ    300H
;
;------------------------------------------------------------------------
;  INTERRUPT  STACK  SIZE  definition
;------------------------------------------------------------------------
ISTACKSIZE              .equ    300H
```

When using multiple interrupts, set the total size of interrupt stacks used for them.

Figure 2.2.7 Setting the Stack Size

Setting the Start Address of Interrupt Vector Table ("ncrt0.a30")

Set the start address of the interrupt vector table. The value set here is set in the interrupt table register "INTB" within "ncrt0.a30".

```
;------------------------------------------------------------------------
;  INTERRUPT  VECTOR  ADDRESS  definition
;------------------------------------------------------------------------
VECTOR_ADR          .equ    0FFD00H
                    :
;=========================================
;  interrupt  section  start
;------------------------------------------------------------------------
          .glb              start
          .section          interrupt
start:
;------------------------------------------------------------------------
;  after  reset , this  program  will  start
;------------------------------------------------------------------------
                    :
          ldintb            #VECTOR_ADR
                    :
```

Set in interrupt table register "INTB"

Figure 2.2.8 Setting the Start Address of Interrupt Vector Table

Setting the Processor Operation Mode ("ncrt0.a30")

Set the processor operation mode. In the same way, add the instructions here that directly controls the operation of the M16C/60, M16C/20, M16C/Tiny, R8C/Tiny Series, 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.

After a reset, the program starts from this label.

```
;=========================================
;  interrupt  section  start
;------------------------------------------------------------------------
          .glb              start
          .section          interrupt
start:
;------------------------------------------------------------------------
;  after  reset , this  program  will  start
;------------------------------------------------------------------------
          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     #istack_top-1,isp
          ldc     #stack_top-1,fb
          ldc     #data_SE_top,sb

          ldintb  #VECTOR_ADR
```

Add settings matched to the system.

Figure 2.2.9 Setting the Processor Operation Mode

Arranging Each Section and Setting Start Address ("sect30.inc")

Arrange the sections generated by NC30 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            0F0000H
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 Vector Table ("sect30.inc")

Add the setup items related to the variable vector table to the section definition file "sect30.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 + 44 )
        .lword      dummy_int               ; DMA0 ( for user )
        .lword      dummy_int               ; DMA1 ( for user )
        .lword      dummy_int               ; input key ( for user )
        .lword      dummy_int               ; AD Convert ( for user )
        .org        ( VECTOR_ADR + 63 )
        .lword      dummy_int               ; UART0 trance ( for user )
        .lword      dummy_int               ; UART0 receive ( for user )
        .lword      dummy_int               ; UART1 trance ( for user )
        .lword      dummy_int               ; UART1 receive ( for user )
        .lword      dummy_int               ; TIMER A0 ( for user )
        .lword      dummy_int               ; TIMER A1 ( for user )
        .lword      dummy_int               ; TIMER A2 ( for user )
        .lword      dummy_int               ; TIMER A3 ( for user )
        .lword      dummy_int               ; TIMER A4 ( for user ) (vector 25)
        .lword      dummy_int               ; TIMER B0 ( for user ) (vector 26)
        .lword      dummy_int               ; TIMER B1 ( for user ) (vector 27)
        .lword      dummy_int               ; TIMER B2 ( for user ) (vector 28)
        .lword      dummy_int               ;  INT0 ( for user ) (vector 29)
        .lword      dummy_int               ;  INT1 ( for user ) (vector 30)
        .lword      dummy_int               ;  INT2 ( for user ) (vector 31)
        .lword      dummy_int               ;  vector 32 ( for user )
        .lword      dummy_int               ;  vector 33 ( for user )
        .lword      dummy_int               ;  vector 34 ( for user )
        .lword      dummy_int               ;  vector 35 ( for user )
        .lword      dummy_int               ;  vector 36 ( for user )
        .lword      dummy_int               ;  vector 37 ( for user )
        .lword      dummy_int               ;  vector 38 ( for user )
        .lword      dummy_int               ;  vector 39 ( for user )
        .lword      dummy_int               ;  vector 40 ( for user )
        .lword      dummy_int               ;  vector 41 ( for user )
        .lword      dummy_int               ;  vector 42 ( for user )
        .lword      dummy_int               ;  vector 43 ( for user )
        .lword      dummy_int               ;  vector 44 ( for user )
        .lword      dummy_int               ;  vector 45 ( for user )
        .lword      dummy_int               ;  vector 46 ( for user )
        .lword      dummy_int               ;  vector 47 ( for user )
```

Figure 2.2.11 Setting Variable Vector Table

Setting the Fixed Vector Table ("sect30.inc")

Set the start address of the fixed vector table and the vector address of each interrupt. Figure 2.2.12 shows an example of how to write these addresses.

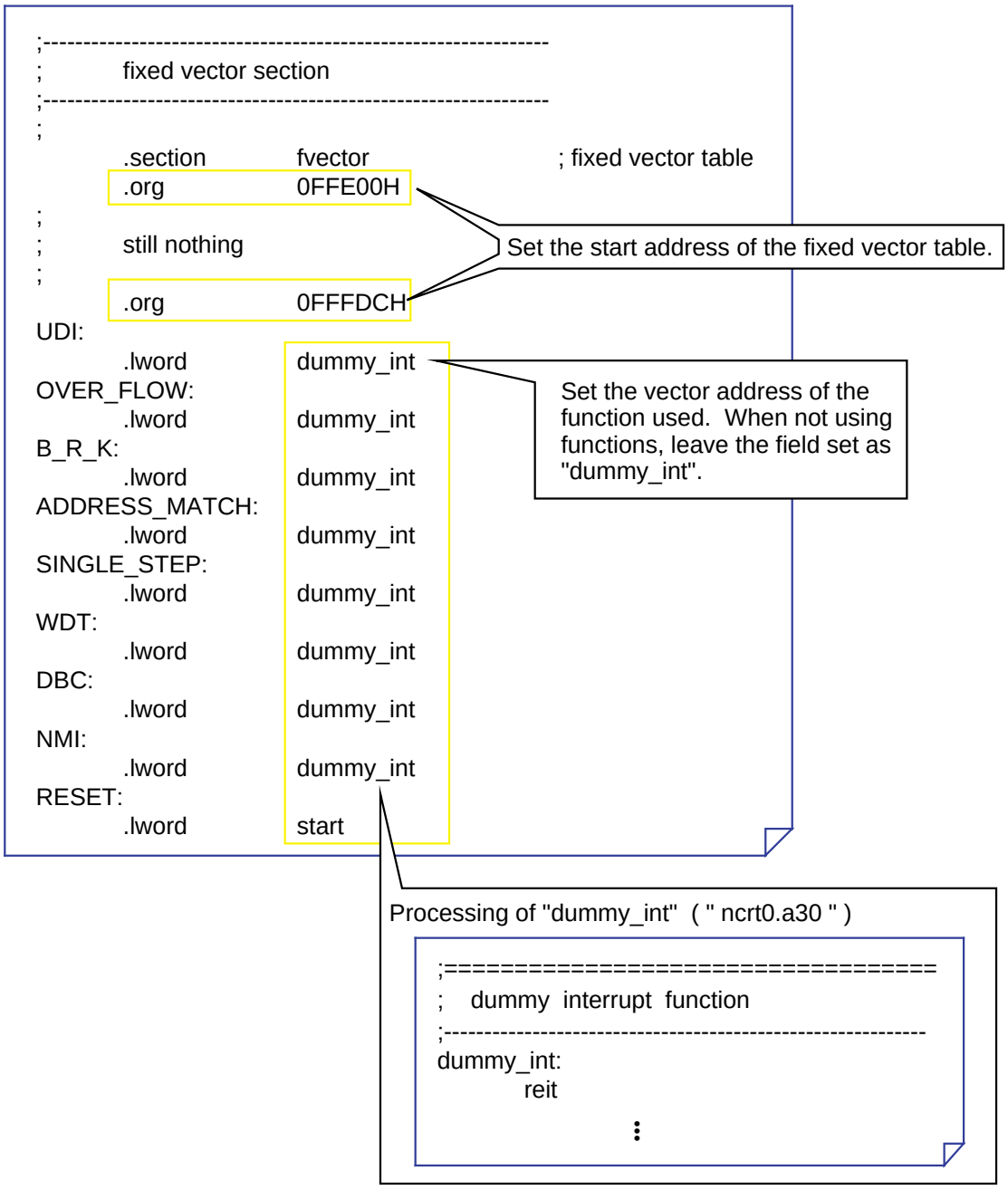


Figure 2.2.12 Setting Fixed Vector Table

Precautions for Operating in Single-Chip Mode

When operating the M16C/60, M16C/20, M16C/Tiny, R8C/Tiny Series in single-chip mode, note that the "near ROM" and the "far RAM" areas are not used. Delete the "ncrt0.a30" and the "sect30.inc" blocks shown in Figure 2.2.13 or turn them into comment statements.

ncrt0.a30: far area initialization program ("FAR area initialize")
sect30.inc: near ROM area allocation ("Near ROM data area")
far RAM area allocation ("Far RAM data area")

(" ncrt0.a30 ")

```
;=========================================
;   FAR area initialize.
;-----------------------------------------------------------------
;   bss_FE & bss_FO zero clear
;-----------------------------------------------------------------
;         BZERO ebss_Esz,ebss_E_top
;         BZERO ebss_Osz,ebss_O_top
;-----------------------------------------------------------------
;  Copy data_FE(FO) section from data_IFE(IFO) section
;-----------------------------------------------------------------
;         BCOPY edata_Esz,edata_E_top,edata_EI_top
;         BCOPY edata_Osz,edata_O_top,edata_OI_top
;         ldc       #stack_top-1,sp
;         ldc       #stack_top-1,fb
```

Leave these lines as comments.

(" sect30.inc ")

```
;-----------------------------------------------------
;   Near  ROM  data  area
;-----------------------------------------------------
;         .section rom_NE,ROMDATA
; 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:
```

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/60, M16C/20, M16C/Tiny, R8C/Tiny series is 1 Mbytes. NC30 divides this area into a "near area" in addresses from 00000 to 0FFFF and a "far area" in addresses from 00000 to FFFFF for management purposes.
This section explains how to arrange and access variables and functions in these areas.

The near and the far Areas

NC30 divides a maximum 1 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.

Table2.3.1 near Area and far Area

Area name	Feature
near area	This space is where the M16C/60, M16C/20, M16C/Tiny, R8C/Tiny series can access data efficiently. It is a 64-Kbyte area in absolute addresses from 00000 to 0FFFF, in which stacks and internal RAM are located.
far area	This is the entire 1-Mbyte memory space in absolute addresses from 00000 to FFFFF that can be accessed by the M16C/60, M16C/20, M16C/Tiny, R8C/Tiny. Internal ROM, etc. are located in this area.

Default near/far Attributes

NC30 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
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 NC30:
–ffar_RAM (–fFRAM) : Changes the default attribute of RAM data to "far".
–fnear_ROM (–fNROM) : Changes the default attribute of ROM data to "near".

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.

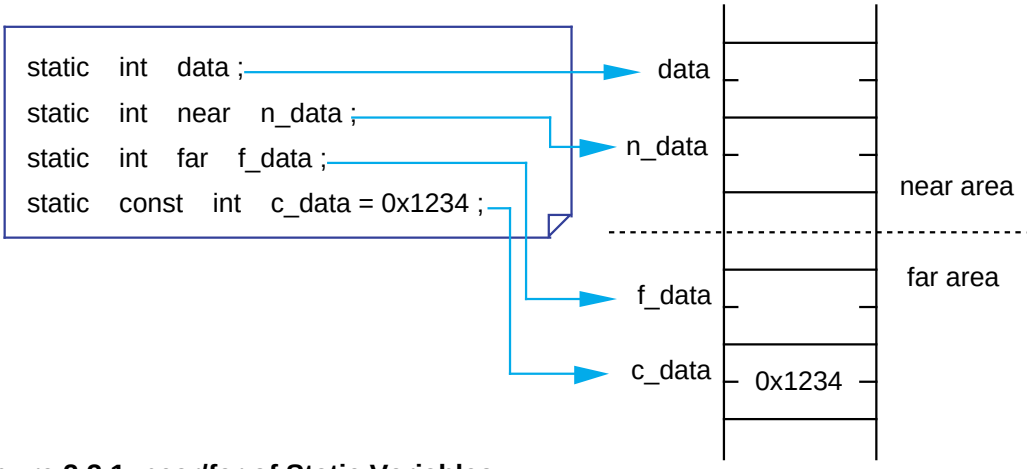


Figure 2.3.1 near/far of Static Variables

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 '&'.

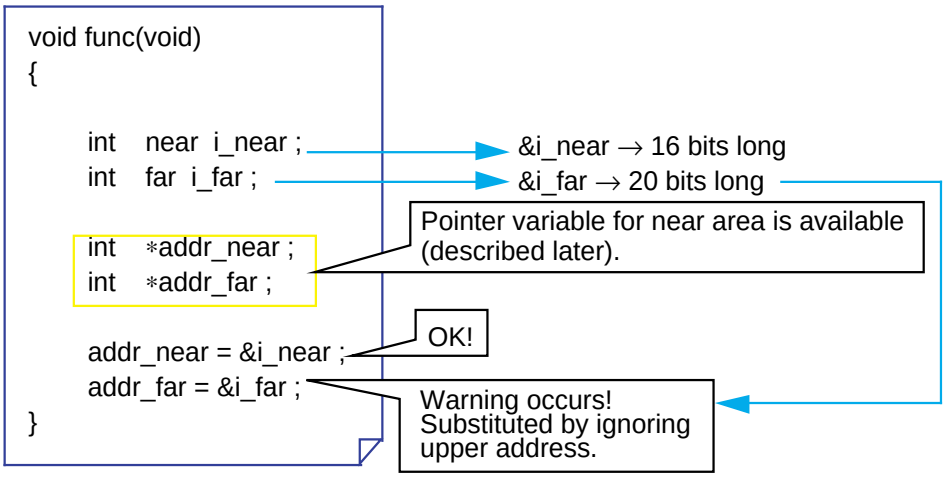


Figure 2.3.2 near/far of Automatic Variables

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.

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

near →16 bits long (16-bit absolute)
far→ 20 bits long (20-bit absolute)

```
int   near   *near_data ;
int   far    *far_data ;
```

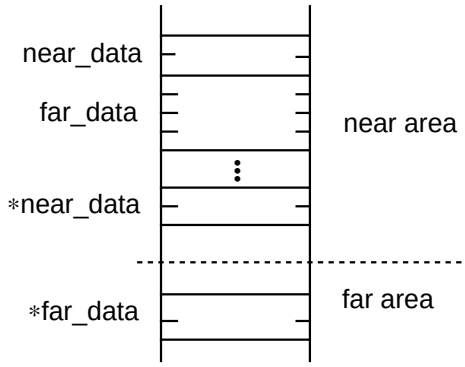


Figure 2.3.3 Specifying Address Size Stored in Pointer

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

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

near →Located in near area
far → Located in far area

```
int   *near   near_data ;
int   *far    far_data ;
```

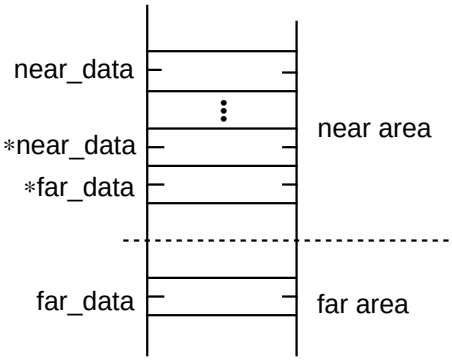


Figure 2.3.4 Specifying Area to Locate the Pointer

near/far of Functions

The attributes of NC30 functions are fixed to the far area for reasons of the M16C/60, M16C/20, M16C/Tiny, R8C/Tiny series architecture. If near is specified for an NC30 function, NC30 outputs a warning when compiling the program and forcibly locates it in the far area.

Using SB Relative Addressing (#pragma SBDATA)

```
#pragma    SBDATA    variable name
```

For the variables declared in this way, NC30 generates AS30 pseudo-instruction ".SBSYM" and uses the SB relative addressing mode when referencing them. This makes it possible to generate highly ROM-efficient code.

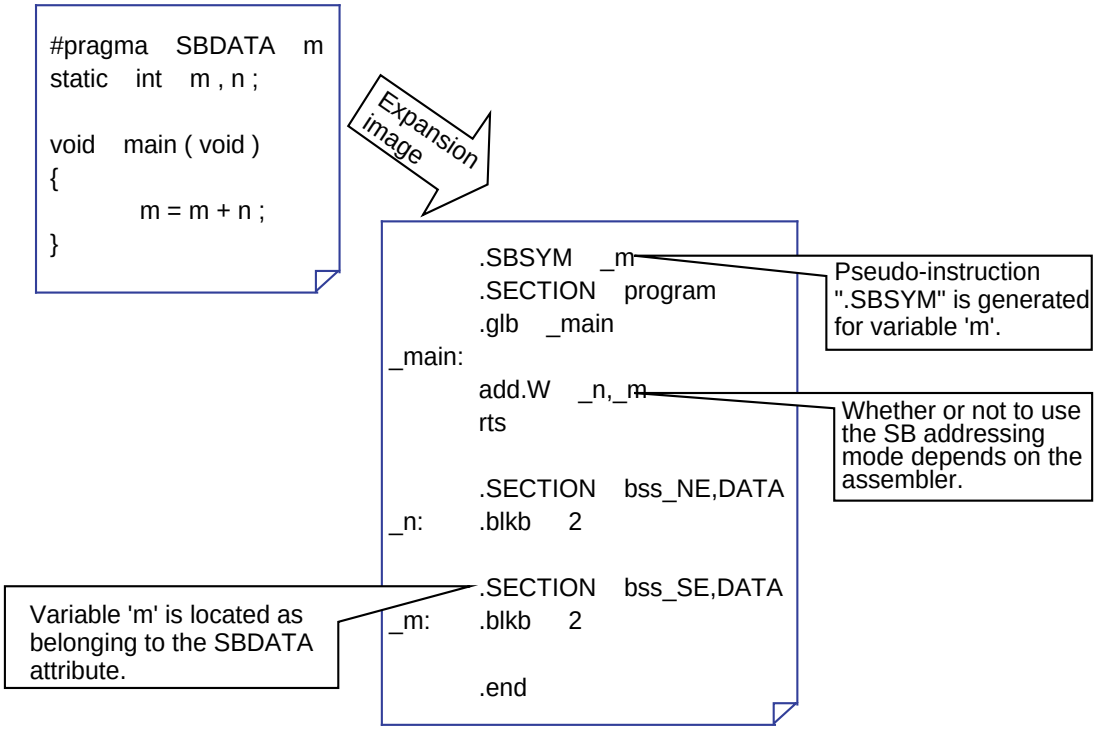


Figure 2.3.5 Image Depicting Expansion of "#pragma SBDATA"

Locating Both Pointer and Indicated Data in far Area

What declaration is necessary to locate both a pointer itself and its indicated data in a far area? The following shows the format and a description example.

```
[storage class] Δ type specifier Δ far Δ * far Δ variable name;
Example: int    far   * far    ff_data:
```

Conversely, when locating both in a near area, near/far specification is unnecessary. This is because the variables and pointers in NC30 assume the near attribute by default.

2.3.2 Handling of Bits

NC30 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 NC30.
This section explains each method of use.

Bit Field

NC30 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. NC30 has two rules according to which memory is allocated for bit fields. Figure 2.3.6 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.)

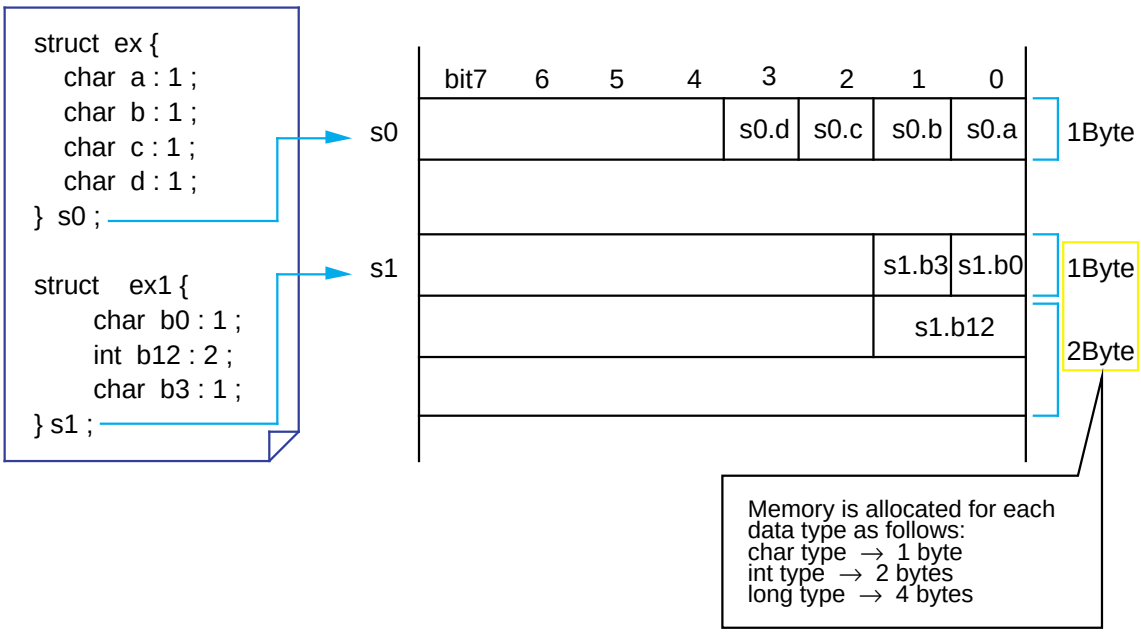


Figure 2.3.6 Example of Memory Allocation for Bit Fields

Generating Bit Instruction (#pragma BIT)

NC30's bit field is such that although bit symbols can be handled in the program, it is an arithmetic/logic instruction, and not a bit instruction, that is generated. To output a code-efficient "direct 1-bit instruction", write an extended function "#pragma BIT" along with bit field declaration.
Figure 2.3.7 shows an example of how to write such a statement and how it will be expanded.

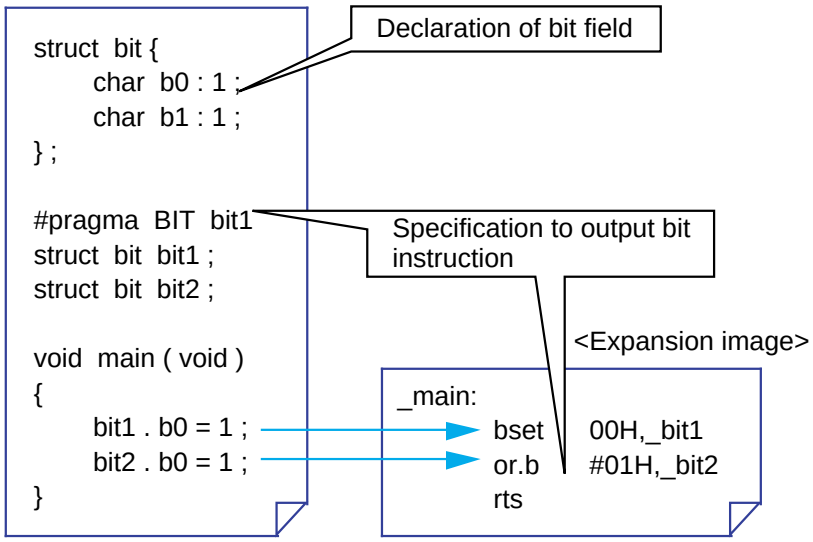


Figure 2.3.7 Typical Description of "#pragma BIT"

In addition to the data where "#pragma BIT" is declared, the direct 1-bit instruction is generated by the following:

- Variables where "#pragma SBDATA" is declared
- Variables where "#pragma ADDRESS" is declared and that area located at absolute addresses 00000 to 01FFF
- near-type variables for which the '-fbit' option is specified

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 NC30: one by using a pointer, and one by using an extended function of NC30.
This section explains each method of specification.

Specifying Absolute Addresses Using a Pointer

Use of a pointer allows you to specify absolute addresses. Figure 2.3.8 shows a description example.

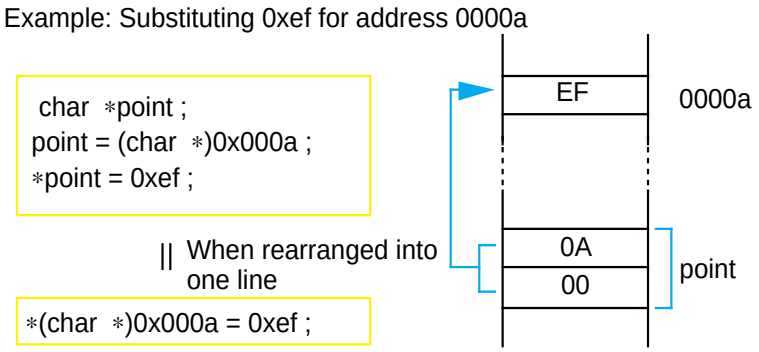
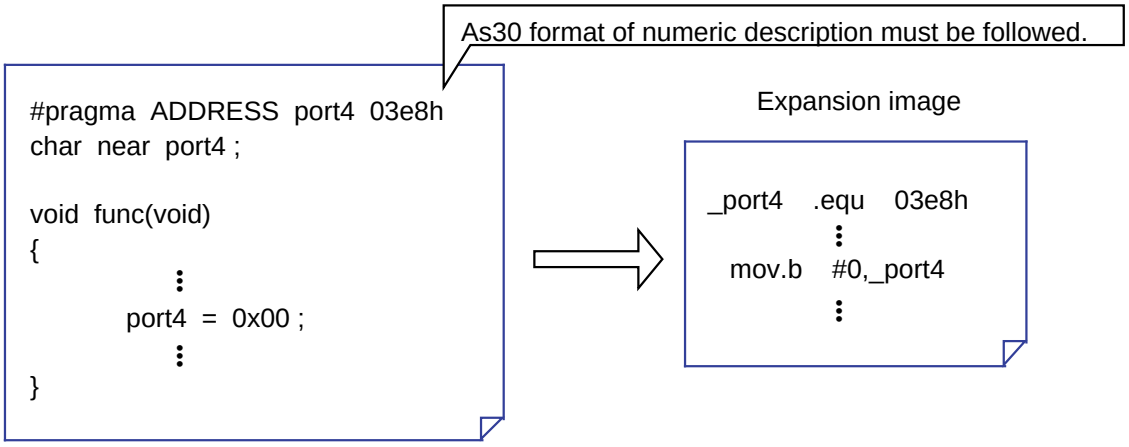


Figure 2.3.8 Specifying Absolute Addresses Using a Pointer

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.

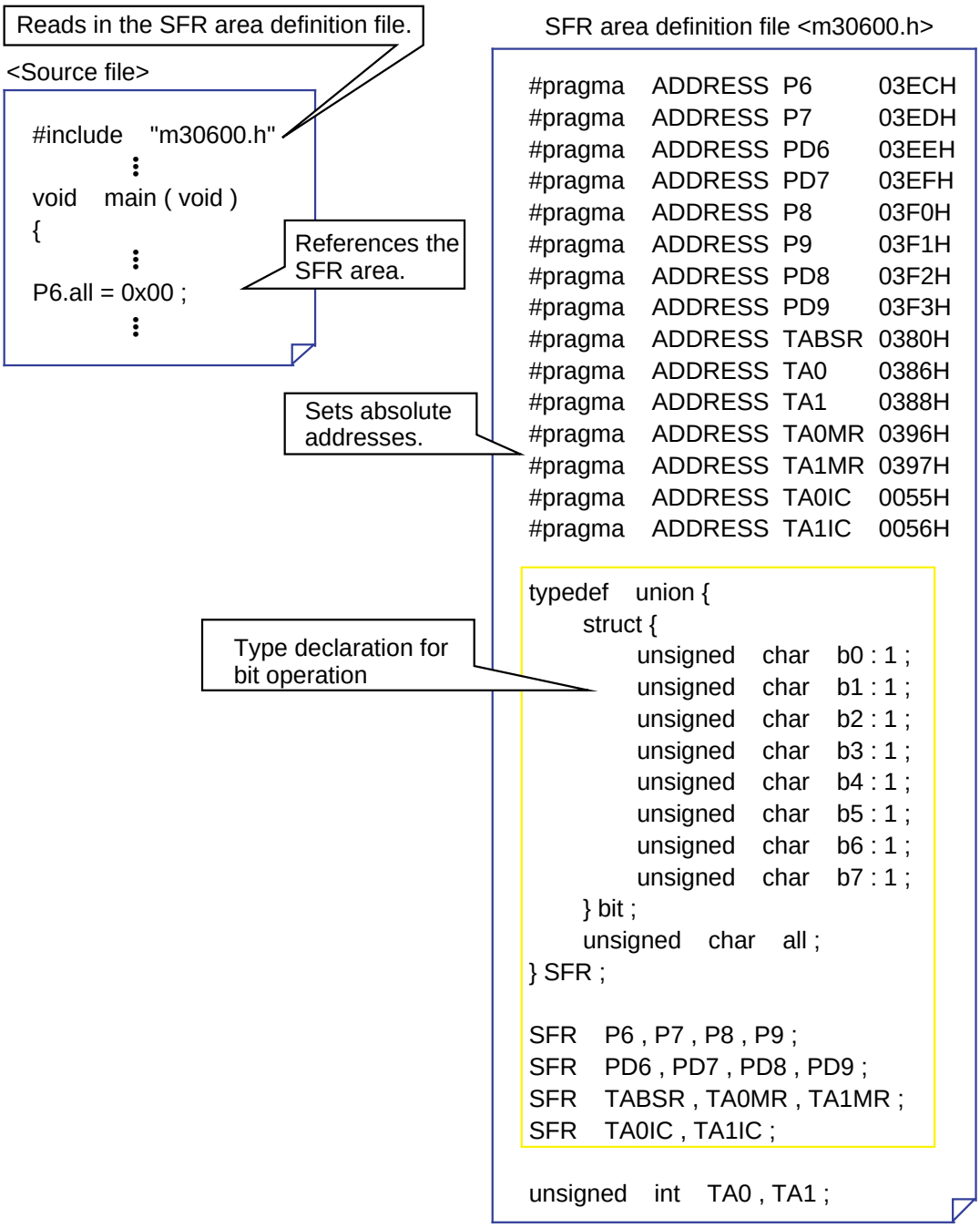


"#pragma ADDRESS" is effective for only variables defined outside a function and those declared in a function as being a static variable.

Figure 2.3.9 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.



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, NC30 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 2.3.10 shows a description example.

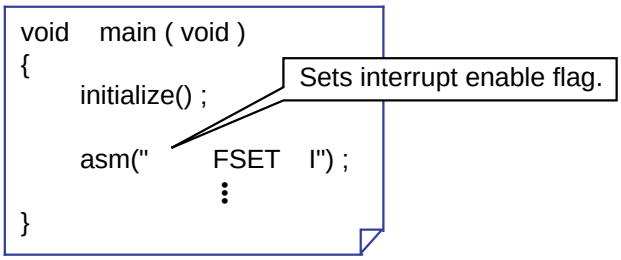


Figure 2.3.10 Typical Description of asm Function

Accessing Automatic Variables in Assembly Language (asm Function)

When it is necessary to access automatic variables inside the function, write a statement using "$$[FB]" as shown in Figure 2.3.11. Since the compiler replaces "$$" with the FB register's offset value, automatic variable names in the C language can be used in assembly language programs.

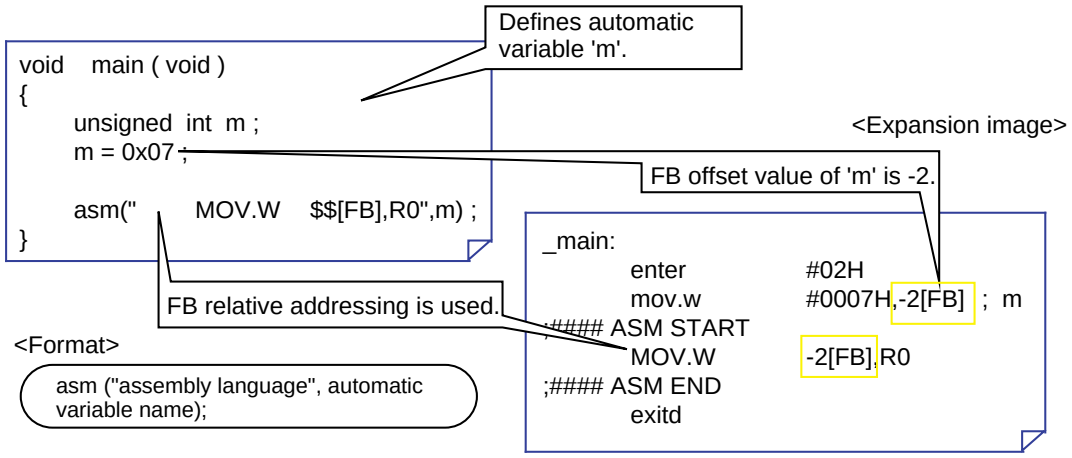


Figure 2.3.11 Using Automatic Variables in asm Function

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, NC30 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.

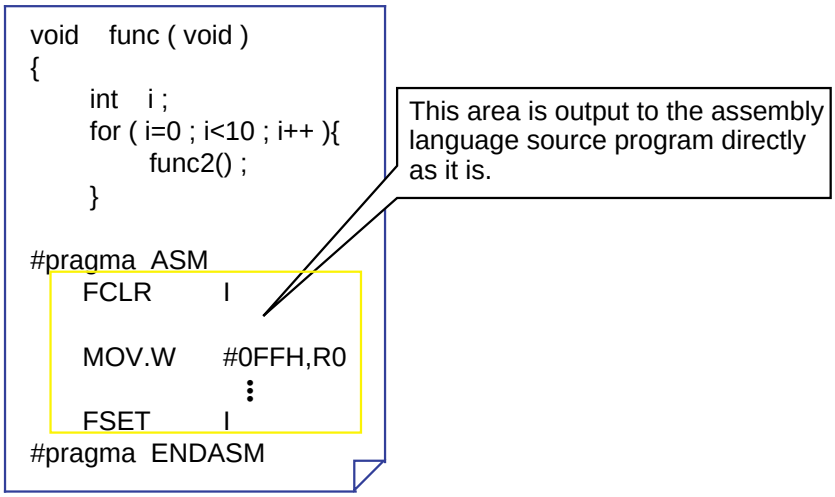


Figure 2.3.12 Example for Using "#pragma ASM" Function

Suppressing Optimization Partially by Using asm Function

When the startup option '-O' is added, NC30 optimizes generated code when compiling the program. However, if this optimization causes inconveniences such as when an interrupt occurs, NC30 allows you to suppress optimization partially by using the asm function. Figure 2.3.13 shows an example for using the asm function for this purpose.

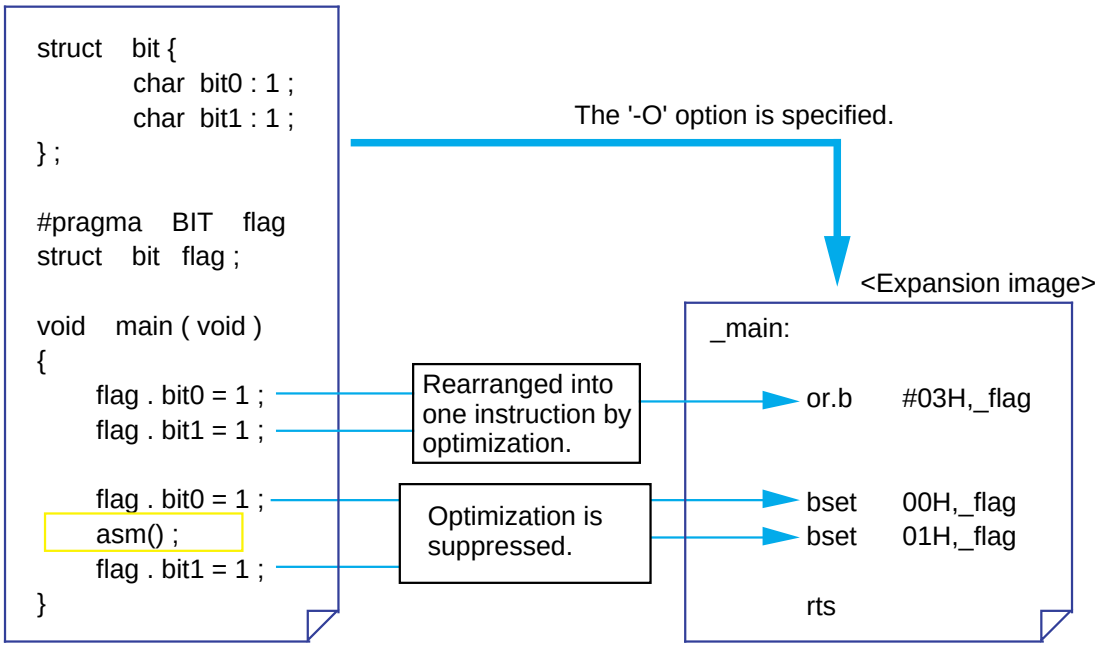


Figure 2.3.13 Suppressing Optimization Partially by Using asm Function

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, NC30 allows you to call an assembly language subroutine from the C language program or vice versa.
This section explains interfacing between functions in NC30.

Entry and Exit Processing of Functions

The following lists the three primary processings performed in NC30 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.

```
int    func ( int , int ) ;

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

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

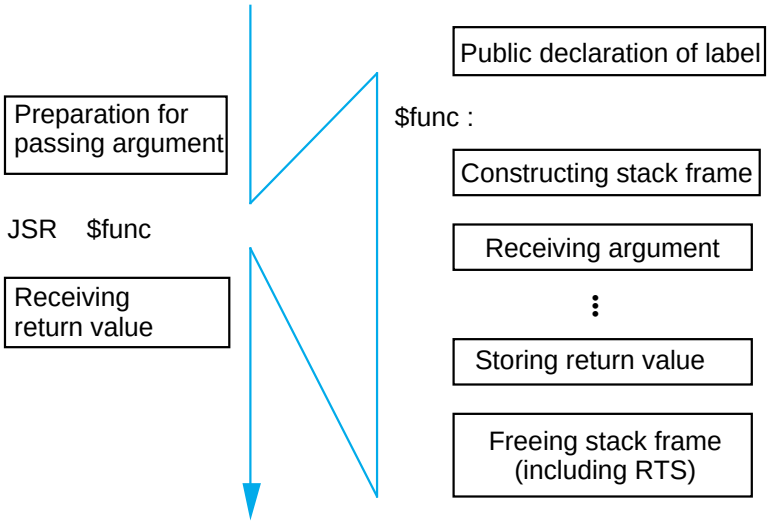


Figure 2.4.1 Operations for Calling a Function

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.

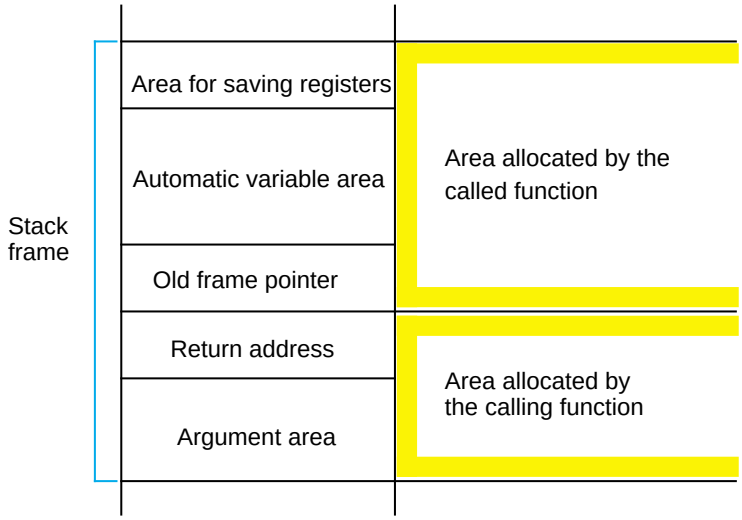


Figure 2.4.2 Structure of a Stack Frame

Constructing a Stack Frame

Figure 2.4.3 shows how a stack frame is constructed by tracing the flow of a C language program.

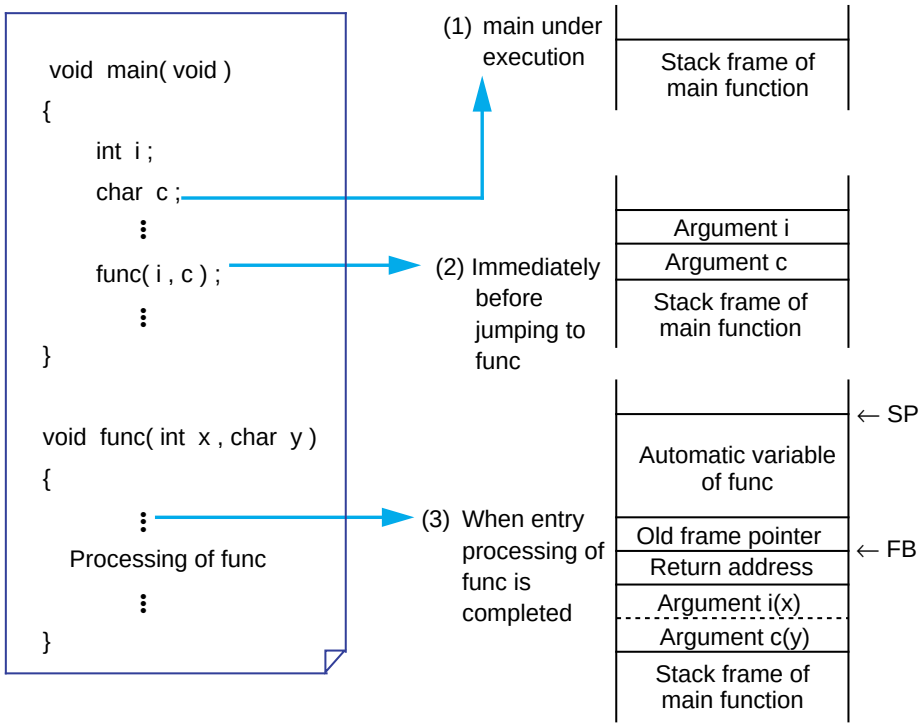


Figure 2.4.3 Constructing a Stack Frame

Rules for Passing Arguments

NC30 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) One or more arguments are the type that can be assigned to a register.
(3) No short-cut form is used in the argument part of prototype declaration.

Table 2.4.1 Rules for Passing Arguments

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

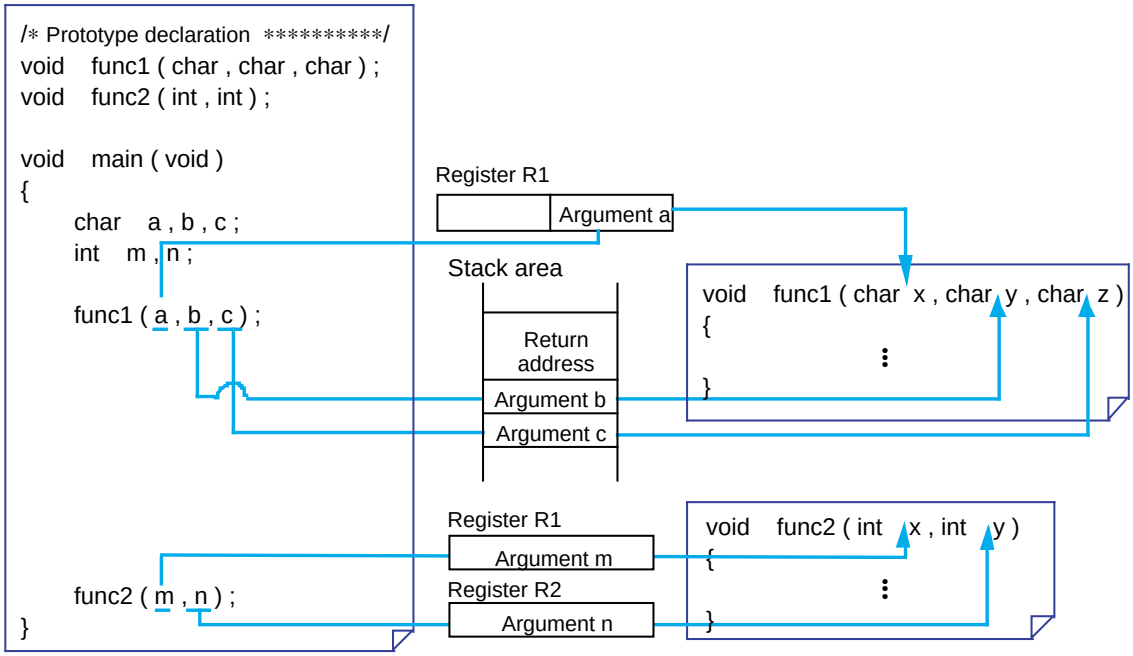


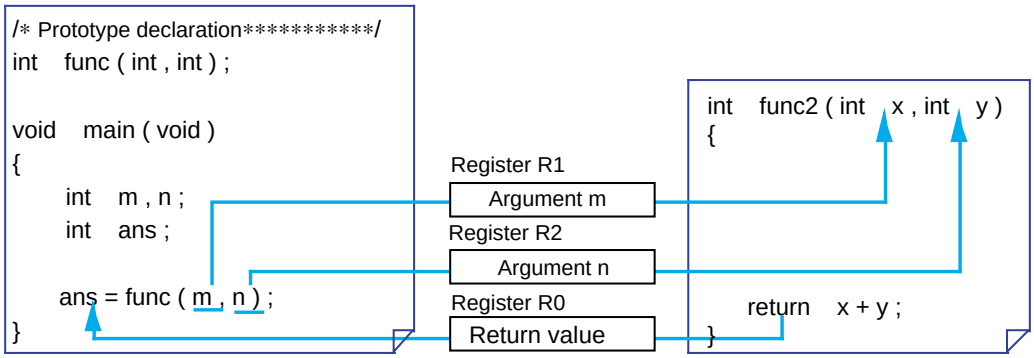
Figure 2.4.4 Example for Passing Arguments to Functions

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 struct

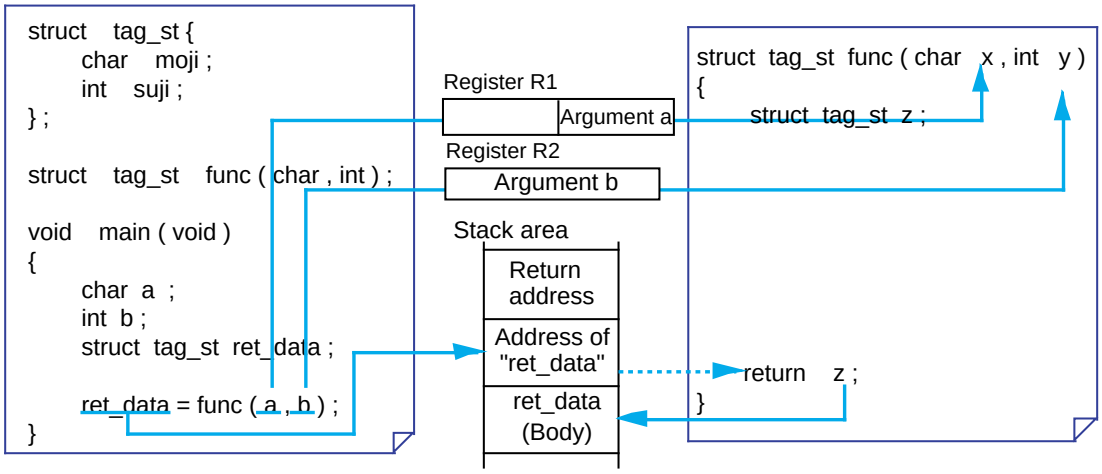


Figure 2.4.5 Example for Passing Return Value

Rules for Symbol Conversion of Functions into Assembly Language

In NC30, 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 #pragma PARAMETER	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 difference in time further, NC30 provides a facility called "inline storage class". When functions are specified to be an inline storage class, NC30 generates code for them as macro functions when compiling the program. This means that ordinary stack manipulation is nonexistent, and that processing in the called function can be executed immediately after a call.

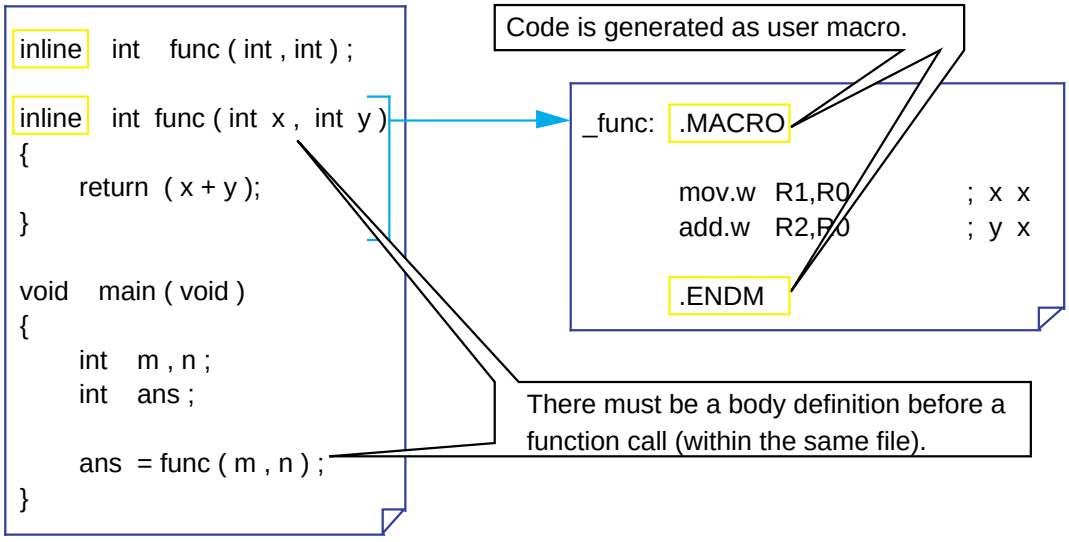


Figure 2.4.6 Example for Writing Inline Storage Class

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    PARAMETER    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:
- Make sure a function arguments are an 8-bit or 16- bit integer or a 16-bit pointer.
- 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.

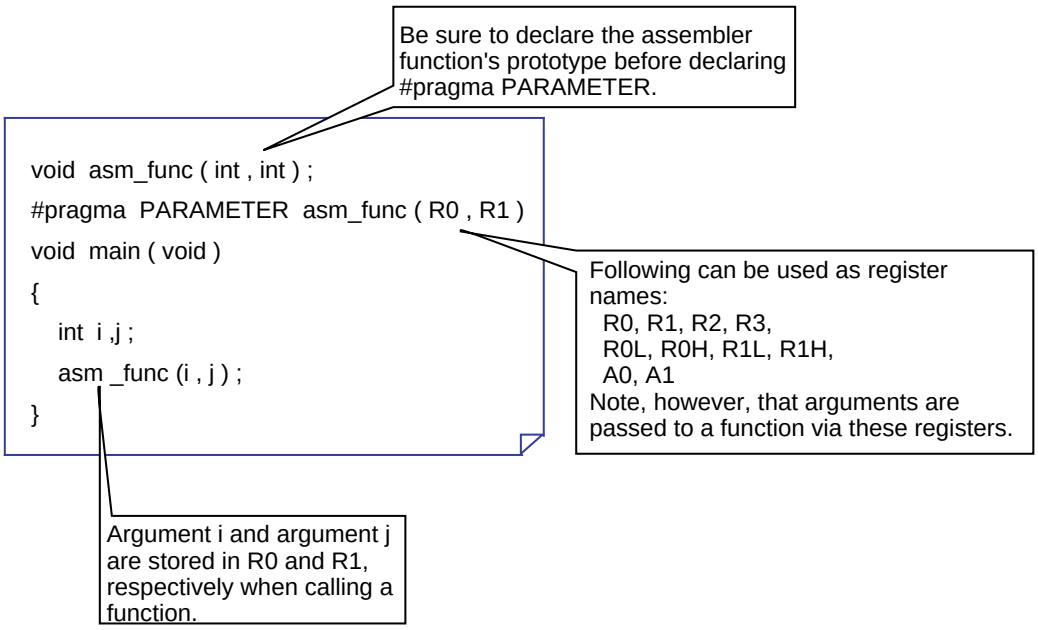


Figure 2.4.7 Example for Writing #pragma PARAMETER

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) Declare the subroutine's prototype in the C language program, from which the subroutine is to be called. At this time, declare the external reference using the storage class specifer "extern".

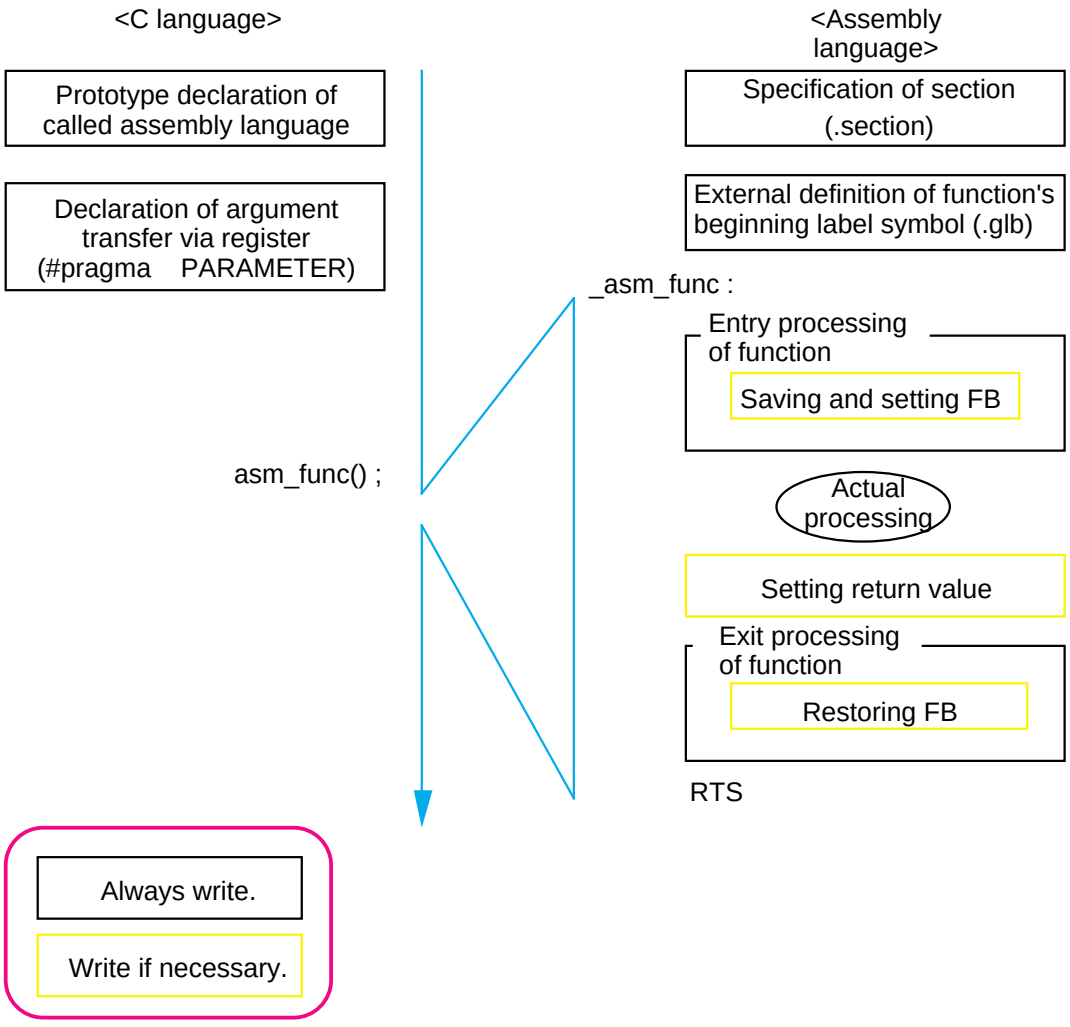


Figure 2.4.8 Calling Assembly Language Subroutine

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.

<Count-up part>

```
/* Prototype declaration */
void    led (int) ;
#pragma    PARAMETER    led (A0)

/* Specification of variables used in SB relative
addressing    */
#pragma    SBDATA    counter

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

      led ( counter ) ;
}
```

Sets the method for calling assembly language function.

There is no conversion of subroutine name because # pragma PARAMETER is specified.

Calls assembly language function "led()".

<LED display part>

```
P7    .equ    03edh

      .section    program
      .glb    led
_led :
      lde.b       table[a0] , P7
      rts
;-------------------------------------------------------
; LED display data table
;-------------------------------------------------------
.section    rom_FE , ROMDATA
table :
      .byte    0c0h , 0f9h , 0a4h , 0b0h , 099h
      .byte    092h , 082h , 0f8h , 080h , 090h

      .end
```

Sets the allocated area.

Externally defines subroutine name.

Example 2.4.1 Calling Subroutine

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 2.4.9 shows a description example.

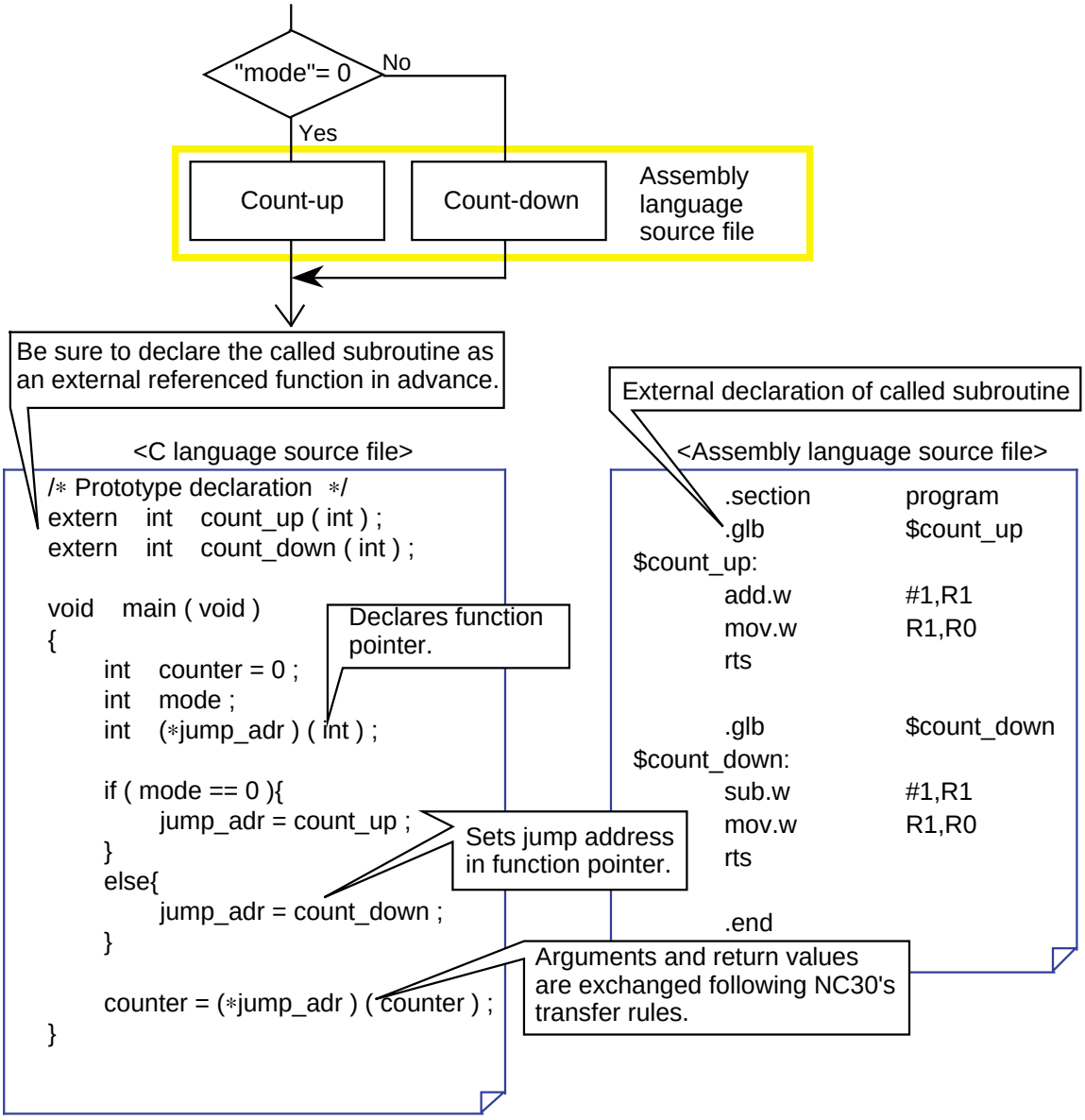
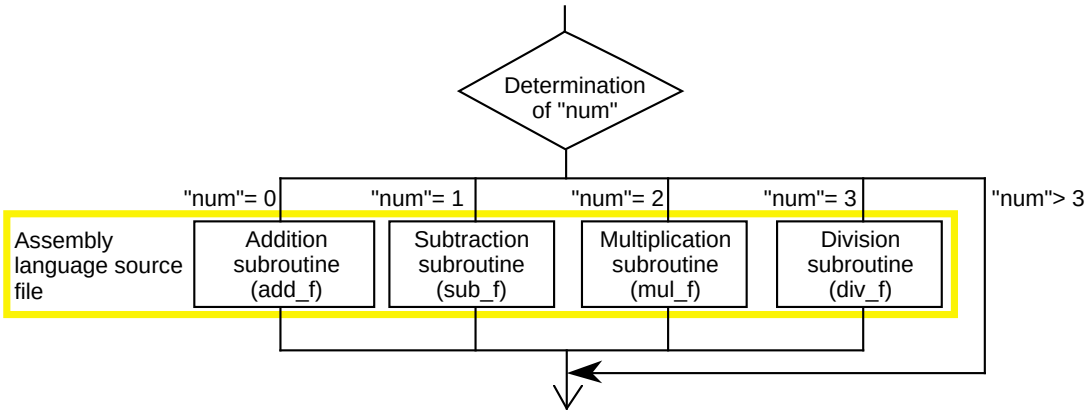


Figure 2.4.9 Calling Subroutine by Indirect Addressing

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".



<C language source file>

```
/* Prototype declaration */
int    cal_f ( int , int , int ) ;

extern    int    ( *jmptbl[] )( int , int ) ;

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

        num = 2 ;
        if ( num < 4 ) {
                val = cal_f( num , x , y ) ;
        }
}
int    cal_f( m , x , y )
{
        int    z ;
        int    (*p )( int , int ) ;

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

Externally references relevant table name as function pointer.

Gets jump address.

Uses function pointer to call subroutine.

<Assembly language source file>

```
        .section        program
add_f:
        mov.w   R1,R0
        add.w   R2,R0
        rts

sub_f:
        mov.w   R1,R0
        sub.w   R2,R0
        rts

mul_f:
        mov.w   R1,R0
        mul.w   R2,R0
        rts

div_f:
        mov.w   R2,R3
        mov.w   R1,R0
        exts.w  R0
        div.w   R3
        mov.w   R2,R0
        rts

        .section        rom_FE , ROMDATA
        .glb            _jmptbl
_jmptbl:
        .lword  add_f
        .lword  sub_f
        .lword  mul_f
        .lword  div_f
        .END
```

Specifies located section.

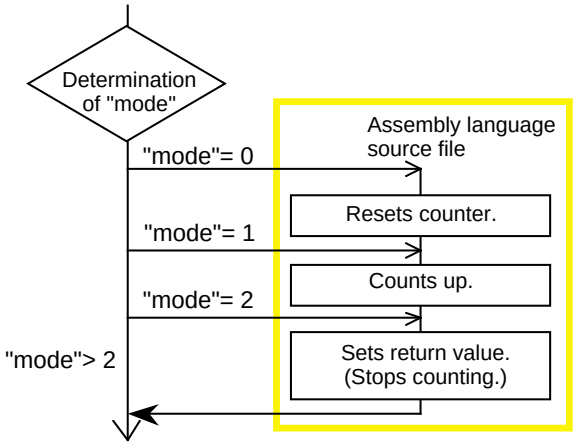
Externally declares table name.

Use pseudo-instruction ".lword" to register each subroutine's start address.

Example 2.4.2 Calling Subroutine by Table Jump

Example 2.4.3 A Slightly Different Way to Use Table Jump

Once the internal labels of a subroutine are registered in a jump table, NC30 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,R1

count:
              add.w   #1,R1

stop:
              mov.w   R1,R0
              rts

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

Registers internal labels of subroutine in jump table.

Example 2.4.3 A Slightly 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 C Language Function

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

(1) Follow NC30'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 AS30's pseudo-instruction ".glb" before calling the C language function.

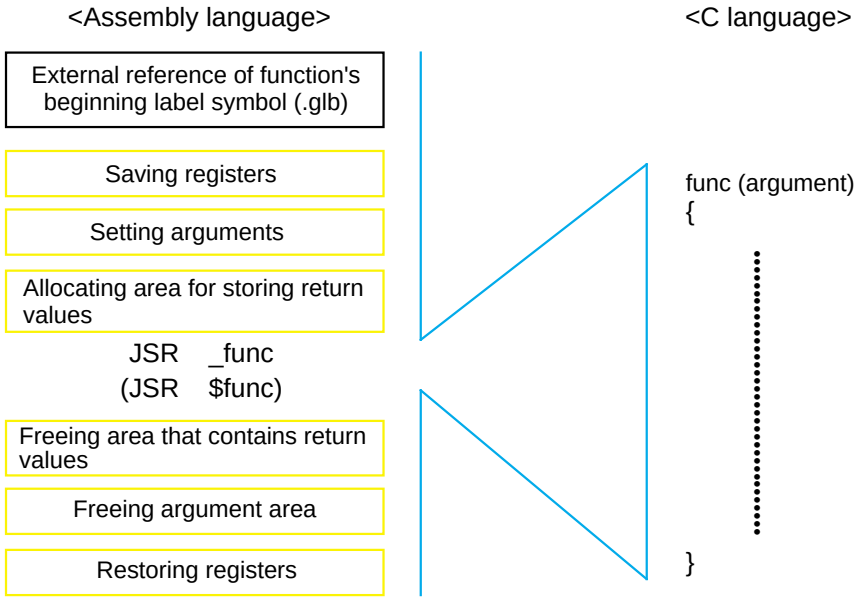


Figure 2.4.10 Calling C Language Function

2.5 Interrupt Handling

2.5.1 Writing Interrupt Handling Functions

NC30 allows you to write interrupt handling as C language functions. There are two procedures to be followed:
(1) Write interrupt processing functions.
(2) Register them in an interrupt vector table.
This section explains how to write C language functions for each type of interrupt processing.

Writing Hardware Interrupts (#pragma INTERRUPT)

#pragma Δ INTERRUPT Δ interrupt function name

When an interrupt function is declared as shown above, NC30 generates instructions to save and restore all registers of the M16C/60, M16C/20, M16C/Tiny, R8C/Tiny and the reit instruction at entry and exit of the specified function, in addition to ordinary function procedures. For both arguments and return values, void is only the valid type of interrupt processing functions. If any other type is declared, NC30 generates a warning when compiling the program.

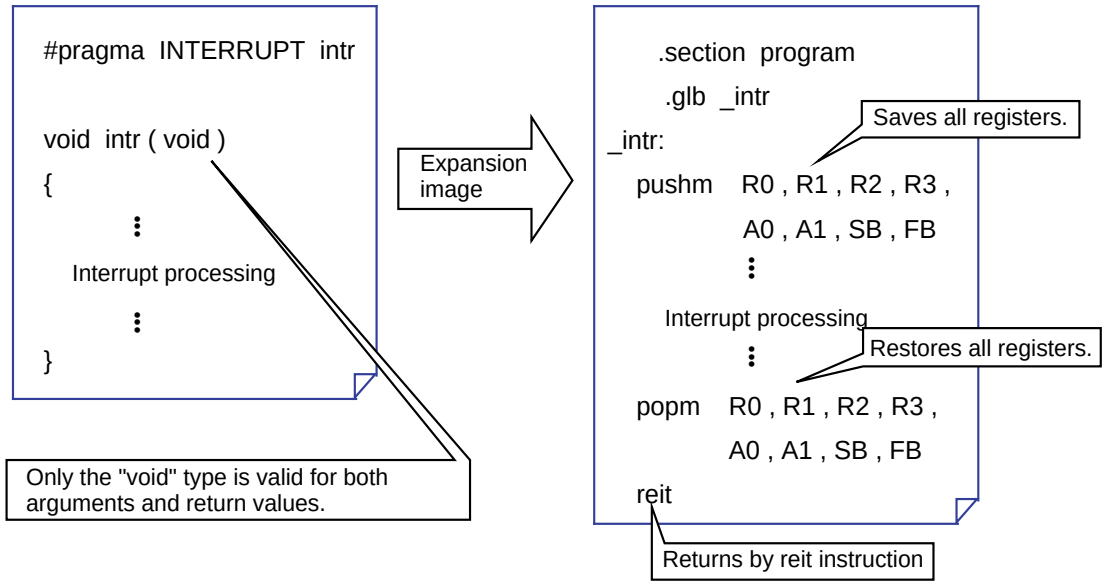


Figure 2.5.1 Image Depicting Expansion of Interrupt Handling Function

Writing Interrupt Service Routines with Improved Response Time (#pragma INTERRUPT/B)

The M16C/60, M16C/20, M16C/Tiny, R8C/Tiny has a facility to switch over the register banks while at the same time protecting register contents, etc., and making it possible to reduce the time until an interrupt handler is invoked. To utilize this facility, write a command statement as follows:

#pragma Δ INTERRUPT/B Δ interrupt function name

When an interrupt function is declared as shown above, NC30 generates instructions to switch over the register banks, in place of instructions to save and restore the registers. However, since the M16C/60, M16C/20, M16C/Tiny, R8C/Tiny register banks consist of only bank 0 and bank 1, only one interrupt can be specified at a time(Note). Therefore, use this facility for the interrupt that needs to be invoked in the shortest time possible.

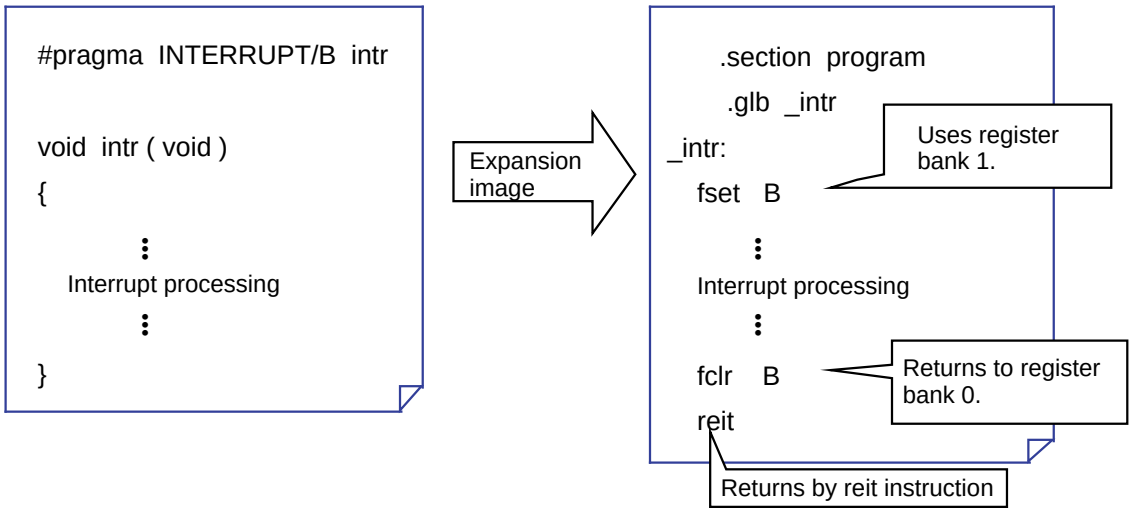


Figure 2.5.2 Image Depicting Expansion of Fast Interrupt Processing Function

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

Writing Software Interrupts (#pragma INTCALL)

To use the M16C software interrupts, write a command statement as follows:

#pragma Δ INTCALL Δ INT number Δ function name

In software interrupts, arguments can be passed to a function via registers. Furthermore, any return value except those expressed by a struct or union, can be received from the called function.

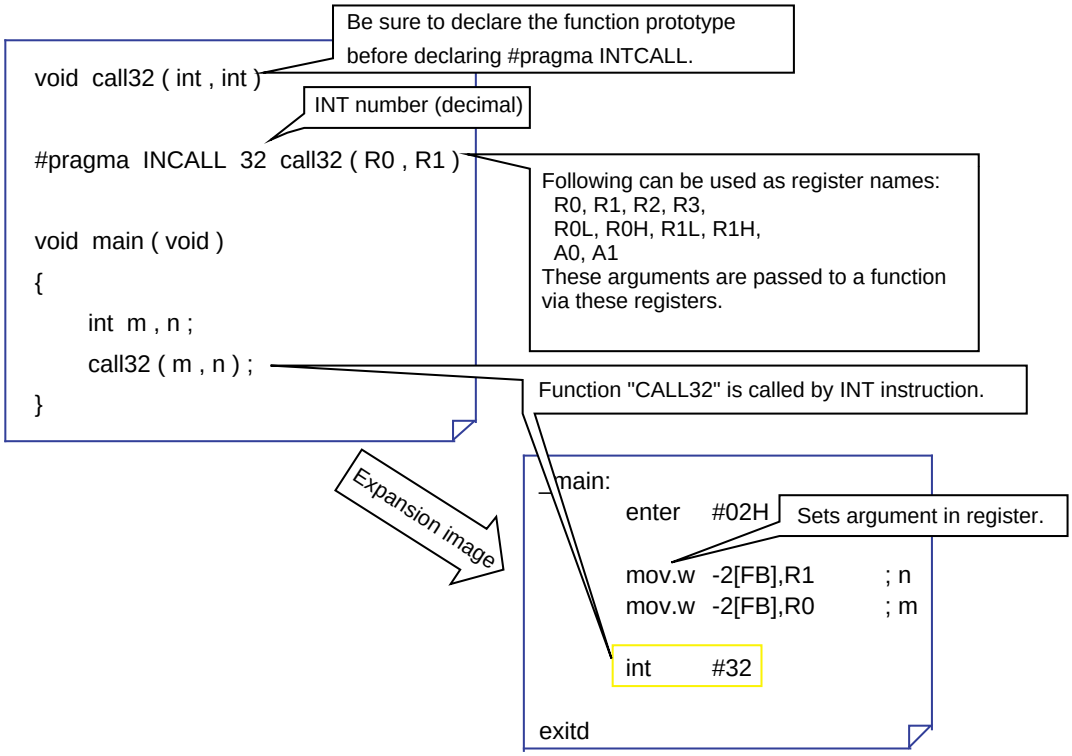


Figure 2.5.3 Example for Writing "#pragma INTCALL"

2.5.2 Registering Interrupt Processing Functions

For interrupts to be serviced correctly, in addition to writing interrupt processing functions, it is necessary to register them in an interrupt vector table.
This section explains how to register interrupt processing functions in an interrupt vector table.

Registering in Interrupt Vector Table

When interrupt processing 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 processing function names using the pseudo-instruction ".glb".
(2) Change the dummy function names "dummy_int" of the interrupts used to interrupt processing function names.

```
;-----------------------------------------------------------------------
;   variable vector  section
;-----------------------------------------------------------------------
        .section        vector                  ; variable  vector  table
        .org            VECTOR_ADR
;
        .lword          dummy_int               ; vector (BRK)
        .org            ( VECTOR_ADR + 44 )
        .lword          dummy_int               ; DMA0 (for user)
        .lword          dummy_int               ; DMA1 (for user)
        .lword          dummy_int               ; input key (for user)
        .lword          dummy_int               ; A-D Convert (for user)
        .org            ( VECTOR_ADR + 68 )
        .lword          dummy_int               ; uart0 trance (for user)
        .lword          dummy_int               ; uart0 receive (for user)
        .lword          dummy_int               ; uart1 trance (for user)
        .lword          dummy_int               ; uart1 receive (for user)
        .glb            _ta0
        .lword          _ta0                    ; TIMER A0 (for user)
        .lword          dummy_int               ; TIMER A1 (for user)
        .lword          dummy_int               ; TIMER A2 (for user)
        .lword          dummy_int               ; TIMER A3 (for user)
        .lword          dummy_int               ; TIMER A4 (for user)
                        :
```

Registers function "ta0()" in TA0 interrupt.

Figure 2.5.4 Interrupt Vector Table ("sect30.inc")

2.5.3 Example for Writing Interrupt Handling Function

The program shown in this description example counts up the content of "counter" each time an INT0 interrupt occurs.

Writing Interrupt Handling Function

Figure 2.5.5 shows an example of source file description.

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

unsigned    int    counter = 0 ;

void    int0 ( void )                  /* Interrupt function */
{
          if ( counter < 9 ) {

                    counter ++ ;
          }
          else {
                    counter = 0 ;
          }
}

void  main ( void )
{
          INT0IC = 1 ;                 /* Setting interrupt level */

          asm ( "  fset       i " ) ;    /* Enabling interrupt */
          while (1) ;                  /* Interrupt waiting loop */
}
```

Figure 2.5.5 Example for Writing Interrupt Handling function

Registering in Interrupt Vector Table

Figure 2.5.6 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 + 68 )
            .lword          dummy_int               ; UART0 trance (for user)
            .lword          dummy_int               ; UART0 receive (for user)
            .lword          dummy_int               ; UART1 trance (for user)
            .lword          dummy_int               ; UART1 receive (for user)
            .lword          dummy_int               ; TIMER A0 (for user)
            .lword          dummy_int               ; TIMER A1 (for user)
            .lword          dummy_int               ; TIMER A2 (for user)
            .lword          dummy_int               ; TIMER A3 (for user)
            .lword          dummy_int               ; TIMER A4 (for user) (vector 25)
            .lword          dummy_int               ; TIMER B0 (for user) (vector 26)
            .lword          dummy_int               ; TIMER B1 (for user) (vector 27)
            .lword          dummy_int               ; TIMER B2 (for user) (vector 28)
            .glb            _int0
            .lword          _int0                   ; INT0 (for user) (vector 29)
            .lword          dummy_int               ; INT1 (for user) (vector 30)
            .lword          dummy_int               ; INT2 (for user) (vector 28)
                                ⋮
```

Figure 2.5.6 Example for Registering in Interrupt Vector Table

Appendices

Appendix A. Functional Comparison between NC30 and NC77
Appendix B. NC30 Command Reference
Appendix C. Questions & Answers

Appendix A. Functional Comparison between NC30 and NC77

Regarding sections

One noteworthy feature of the M16C/60, M16C/20, M16C/Tiny, R8C/Tiny series is that they support 1 Mbytes of linear memory space without "boundaries every 64 Kbytes", and that those banks that are found in the 7700 family are nonexistent. Furthermore, although the interrupt programs in the 7700 family were subjected to restrictions on allocatable addresses, they in the M16C/60, M16C/20, M16C/Tiny, R8C/Tiny series can be mapped into any desired location over the entire memory space just like ordinary other programs. Therefore, NC77's interrupt section is nonexistent in NC30, and the interrupt programs in NC30 are stored (and located) in the program section.
Moreover, the M16C/60, M16C/20, M16C/Tiny, R8C/Tiny series has two types of interrupt vector tables: a "variable" vector table that can be located at any desired address in the entire memory space and a "fixed" vector table which has its location address predetermined for each type of microcomputer. In NC30, the former is located as the vector section, and latter as the fvector section.
Table A.1 lists the differences between NC30 and NC77 regarding sections.

Table A.1 Functional Comparison Regarding Sections

Item	NC30	NC77
stack	An area used as stack. It is located at addresses from 00400H to 0FFFFH.	An area used as stack. It is located in bank 0 of the 7700 family.
vector	Stores the contents of the M16C/60's, M16C/20's, M16C/Tiny's, R8C/Tiny's interrupt vector table. The interrupt vector table can be located at any desired address in the M16C/60's, M16C/20's, M16C/Tiny's, R8C/Tiny's entire memory space by INTB register relative.	Stores the contents of the 7700 family's interrupt vector table. The address at which this interrupt vector table is located varies with each type of microcomputer.
fvector	Stores the contents of the M16C/60's, M16C/20's, M16C/Tiny's, R8C/Tiny's sfixed vector.	
interrupt	deleted Since the interrupt program is located at any desired address in the M16C/60's, M16C/20's, M16C/Tiny's, R8C/Tiny's entire memory space, it is located at "program" section.	Stores interrupt programs (functions specified by "#pragma INTERRUPT" and "#pragma HANDLER"). This section is located in bank 0 of the 7700 family.

Modified Extended Functions

With the M16C/60, M16C/20, M16C/Tiny, R8C/Tiny series, "banks" and the "m and x flags" are nonexistent. Therefore, the definitions of the near/far modifiers and part of functionality of the asm function have been modified.

Table A.2 Modified Extended Functions

Item	NC30	NC77
near/far modifier	1. Specify the addressing mode to access data. near: Access 00000H through 0FFFFH. far: Access 00000H through FFFFFH. (2. All functions assume the far attribute.)	1. Specify the addressing mode to access data. near: Access addresses within the same bank. far: Access addresses outside the bank. 2. Specify whether the "JSR" or "JSRL" instruction is used to call a function. near: JSR instruction is used. far: JSRL instruction is used.
asm function	1. Write assembly language in C language. 2. Specify auto variable by variable name. 3. Partially suppress optimization. 4. Specify register argument by variable name.	1. Write assembly language in C language. 2. Specify auto variable by variable name. 3. Partially suppress optimization. 4. Control 'm' and 'x' flags.

Added Extended Functions

For NC30, the compiler's extended functions have been added to support the features of the M16C/60, M16C/20, M16C/Tiny, R8C/Tiny series, such as bit manipulating instructions and SB relative addressing. Furthermore, to accommodate the M16C/60, M16C/20, M16C/Tiny, R8C/Tiny series 's versatile interrupt processing, new extended functions are provided for writing interrupt programs that use software interrupts or register banks. Moreover, the extended functions now include an "inline" storage class and an inline assemble function "#pragma ASM", making it possible to take full advantage of the M16C/60, M16C/20, M16C/Tiny, R8C/Tiny series capabilities.

Table A.3 Added Extended Functions

Item	NC30
#pragma ASM to #pragma ENDASM	Specifies area where statements are written in assembly language.
#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 SBDATA	Declares that SB relative addressing can be used for the data.
#pragma INTERRUPTt/B	When calling interrupt function, it switches over register banks, instead of saving registers to stack.
#pragma INTCALL	Declares function that calls software interrupts (int instruction).

Deleted Extended Functions

The extended functions of NC77 listed in Table A.4 are not supported by NC30, because they are used to operate on the registers or flags that do not exist in the M16C/60, M16C/20, M16C/Tiny, R8C/Tiny series.

Table A.4 Extended Functions Not Supported by NC30

Item	NC77
#pragma LOADDT	When calling a function, it returns data bank register (DT) to the value when compiled.
#pragma M1FUNCTION	Set the 'm' flag to 1 before calling a function.

The extended functions listed in Table A.5 are supported by NC30 also, for reasons of compatibility with NC77. However, when creating a new program, please follow the recommended uses below, without using these extended functions.

Table A.5 Extended Functions Retained for Compatibility Reason and Recommended Uses in NC30

Item	Function	Recommended use in NC30
#pragma ROM	Locates in rom section.	Use const modifier.
#pragma INTF	Specifies interrupt processing function.	Use #pragma INTERRUPT.
#pragma EQU	Specifies absolute address of variable.	Use #pragma ADDRESS.

Appendix B. NC30 Command Reference

NC30 Command Input Format

%nc30 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 Regarding Compile Driver Control

Table B.1 Options Regarding Compile Driver Control

Option	Function
–c	Creates relocatable file (attribute .r30) before terminating processing.
–D identifier	Defines identifier. It functions in the same way as "#define".
–I directory name	Specifies directory name where file specified by "#include" exists. Up to 8 directories can be specified.
–E	Invokes only preprocess command and outputs result to standard output device.
–P	Invokes only preprocess command and creates file (attribute .i).
–S	Creates assembly language source file (attribute .a30) before terminating processing.
–U predefined macro name	Undefines specified predefined macro.
–silent	Inhibits copyright message from being output at startup.

If startup options -c, -E, -P, and -S are not specified, NC30 controls the compile driver up to ln30 until it creates the absolute module file (attribute .x30).

Output File Specifying Options

Table B.2 Output File Specifying Options

Option	Function
–o file name	Specifies the name of file generated by nc30 (e.g., absolute module file, map file). Do not write file extension.

Version Information Display Options

Table B.3 Version Information Display Options

Option	Function
–v	Displays command program name and command line under execution.
–V	Displays message when compiler's each program starts up before terminating processing (does not perform compile processing).

Debug Options

Table B.4 Debug Options

Option	Function
–g	Outputs debug information to assembly language source file (attribute. a30).
–genter	When calling function, it always outputs enter instruction. Be sure to specify this option when using debugger's stack trace function.
–greg	Outputs debug information about register variables.

Alarm Options

Table B.5 Alarm Options

Option	Abbreviation	Function
–Wnon_prototype	–WNP	Outputs alarm when attempt is made to use or define the function whose prototype is not declared.
–Wunknown_pragma	–WUP	Outputs alarm when attempt is made to use unsupported "#pragma".
–Wno_stop	–WNS	Does not stop compile operation even when error occurs.
–Wstdout	None	Outputs error message to host computer's standard output device (stdout).

Optimization Options

Table B.6 Optimization Options

Option	Abbreviation	Function
–O	None	Optimizes to minimize both speed and ROM size.
–OR	None	Optimizes by emphasizing ROM size than speed.
–OS	None	Optimizes by emphasizing speed than ROM size.
–Ono_bit	–ONB	Suppresses optimization to put bits in order.
–Ono_break_source_debug	–ONBSD	Suppresses optimization that affects source line information.
–Osp_adjust	–OSA	Optimizes to remove stack correction code. This helps to reduce ROM size. However, it could result in increased stack amount.
–Ono_stdlib	–ONS	Suppresses inline embedding of standard library functions or modification of library functions.
–Ono_cse	–ONC	Suppresses optimization that deletes common instructions.

Library Specifying Options

Table B.7 Library Specifying Options

Option	Function
–lΔ<library name>	Specifies library that is used when linking.

Assemble and Link Options

Table B.8 Assemble and Link Options

Option	Function
–as30Δ<Option>	Specifies options of assemble command "as30". When passing two or more options, be sure to enclose them with double quotations (").
–ln30Δ<Option>	Specifies options of link command "ln30". When passing two or more options, be sure to enclose them with double quotations (").

Generated Code Modifying Options

Table B.9 Generated Code Modifying Options

Option	Abbreviation	Function
–fansi	None	Enables –fnot_reserve_asm, –fnot_reserve_far_and_near, –fnot_reserve_inline, and –fextend_to_int
–fnot_reserve_asm	–fNRA	Frees "asm" from reserved word. (Only _asm is valid.)
–fnot_reserve_far_and_near	–fNRFAN	Frees "far" and "near" from reserved words. (Only _far and _near are valid.)
–fnot_reserve_inline	–fNRI	Frees "inline" from reserved word. (Only _inline is valid.)
–fextend_to_int	–fETI	Expands char-type data to int type before operating on it.
–fchar_enumerator	–fCE	Handles enumerator type as being unsigned char type, and not as int type.
–fno_even	–fNE	Locates all data in odd attribute section without separating them between odd and even when outputting data.
–fshow_stack_usage	–fSSU	Outputs stack usage conditions to file (extension. stk).
–ffar_RAM	–fFRAM	Changes default attribute of RAM data to far.
–fnear_ROM	–fNROM	Changes default attribute of ROM data to near.
–fconst_not_ROM	–fCNR	Does not handle types specified by const as ROM data.
–fnot_address_volatile	–fNAV	Does not recognize variables specified by #pragma ADDRESS (#pragma EQU) as those specified by volatile.
–fsmall_array	–fSA	When referencing far-type array, if its total size is within 64 Kbytes, this option calculates subscripts in 16 bits.
–fbit	–fB	Outputs 1-bit manipulating instruction in 16-bit absolute addressing mode for variables located in near area.

Other Options

Table B.10 Other Options

Option	Abbreviation	Function
–dsource	–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).

```
%nc30 -otest ncrt0.a30 c_src.c
→Specifies the output file name.
```

2 Generate an assembler list file and a map file.

```
%nc30 -as30 "-l" -ln30 "-M" c_src.c
→Specifies the options of "as30" and "ln30".
```

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

```
%nc30 -g -S ncrt0.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 Structs

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 bit data...]
(1) Is there any bit data using a bit field for which "#pragma BIT" is not declared? Always be sure to declare "#pragma BIT".
(For direct 1-bit instructions to be generated in NC30, it is necessary to declare "#pragma BIT" as well as a bit field.)

[When compiling...]
(1) Is the optimization option "-OR" specified? If not, specify this option.
(When the optimization option "-OR" is specified in NC30, it optimizes code generation by placing emphasis on ROM efficiency.)

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".

<Source file 1>
Defines "mode".

```
void func1(void) ;

char mode ;
#pragma SBDATA mode

void main(void)
{
        mode = 1;
        func1();
}
```

<Source file 2>
References "mode".

```
extern    void    func(void) ;

extern char mode ;
#pragma SBDATA mode

void func1(viod)
{
        mode = mode + 1 ;
}
```

For "mode" to be accessed by SB relative, declare "#pragma SBDATA" in the referencing program.

Figure C.2 Example for Writing "#pragma SBDATA"

[For programs whose generated code is 64 Kbytes or less...]
(1) By using the asm function or "#pragma ASM", set ".OPTJ JMPW, JSRW" at the beginning of each file, which is the branch instruction optimizing control directive command.

<Using asm function>

```
asm("    .OPTJ    JMPW,JSRW") ;

void    func1(void) ;
char    mode ;

void    main(void)
{
        ⋮
}
```

<Using "#pragma ASM">

```
#pragma    ASM
        .OPTJ    JMPW,JSRW
#pragma    ENDASM

void    func1(void) ;
        ⋮
```

Figure C.3 Example for Setting ".OPTJ JMPW, JSRW"

REVISION HISTORY	Programming Guidelines <C Language> Application Note

Rev.	Date	Description	
		Page	Summary
1.00	Oct 20, 2003	-	First edition issued
1.10	Nov 4, 2004	-	**M16C/Tiny, R8C/Tiny series** added
		1	**Preface** Description modified **Guide to Using This Application Note** Description modified
			Chapter 1
		8	• **1.1.1 Assembly Language and C Language** Description modified
			• **Table 1.1.1 Comparison between C and Assembly Languages** "Format" row modified
		9	• **1.1.2 Program Development Procedure** Description modified
			• **Figure 1.1.1 NC30 Product List** Figure modified
		10	• **Creating Machine Language File from C Language Source File** Description modified
		11	• **1.1.3 Program Rules and Practices** Title and description modified
		12	• **Programming Style** Description modified
		13	• **Method for Writing Comments** Title and description modified
		15	• **1.2.1 "Constants" in C Language** Title modified
			• **Table 1.2.1 Method for Writing Integer Constants** "Hexadecimal" rows modified
		16	• **List of Control Codes** Description modified
		21	• **Table 1.3.1 NC30 Operators** Title and Table modified
		32	• **1.4.1 Structuring of Program** Description modified
		33	• **1.4.2 Branch Processing** Title modified
		37	• **1.4.3 Repeat Processing** Title modified
		51	• **What is an Array?** Example name modified
			• **Figure 1.7.1 Concept of Array** Figure modified
		54	• **1.7.3 Pointers** Description modified
		55	• **Figure 1.7.5 Relationship Between Pointer Variables and Variables** Figure modified
		59	• **Figure 1.7.10 Difference between Two-dimensional Array and Pointer Array** Example names modified
			Chapter 2
		73	• **Extended Function of NC30** Title modified
		90	• **Figure 2.2.11 Setting Variable Vector Table** Figure modified
		101	• **2.3.4 Using Inline Assembly** Title modified
		113	• **Example 2.3.4 A Slightly Different Way to Use Table Jump** Title modified
			Chapter 3 Deleted
			Appendices
		133	• **Reducing Generated Code (2)** 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 nonflammable 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.