FUJITSU SEMICONDUCTOR
CONTROLLER MANUAL

CM42-00327-1E

F^2MC-16 FAMILY

16-BIT MICROCONTROLLER

EMBEDDED C PROGRAMMING MANUAL

FOR fcc907

FUJITSU

F^2MC-16 FAMILY

16-BIT MICROCONTROLLER

EMBEDDED C PROGRAMMING MANUAL

FOR fcc907

FUJITSU LIMITED

PREFACE

■ Objectives and Intended Reader

The F^2MC-16L/16LX/16/16H/16F (hereafter collectively referred to as the F^2MC-16 Family) are 16-bit microcontrollers designed for embedded systems.

This manual provides information required for using the fcc907 F^2MC-16 family C compiler to create an embedded system. The manual explains how to create C programs that effectively use the F^2MC-16 family architecture and provides notes related to the creation of C programs.

This manual is intended for engineers who use the fcc907 to develop application programs for the F^2MC-16 family. Be sure to read this manual completely.

■ Trademarks

Softune is a registered trademark of Fujitsu Limited.

Other system names and product names in this manual are trademarks of their respective companies or organizations. The symbols ™ and ® are sometimes omitted in the text.

■ Structure of This Manual

This manual consists of the three parts listed below.

PART I "VARIABLE DEFINITIONS AND VARIABLE AREAS"

Part I describes the variable definitions and variable areas for creating C programs.

CHAPTER 1 "OBJECTS MAPPED INTO MEMORY AREAS"

This chapter briefly describes the memory mapping for a systems in which an F^2MC-16 family microcontroller was embedded.

CHAPTER 2 "VARIABLE DEFINITIONS AND VARIABLE AREAS"

This chapter describes the variable definitions and variable areas to which the results of compilation are output. It also describes the variable areas for variables that are initialized at definition and the variable area for those that are not. In addition, the chapter describes variables declared as "static."

CHAPTER 3 "READ-ONLY VARIABLES AND THEIR VARIABLE AREA"

This chapter describes how to use variables declared with the type-qualifier "const" that are only read at execution time and provides notes on their use. This chapter also discusses the reduction of the variable area and object efficiency for referencing when the "const" type modifier is used.

CHAPTER 4 "USING AUTOMATIC VARIABLES TO REDUCE THE VARIABLE AREA"

This chapter explains how to reduce the variable area by using automatic variables. Area is allocated to automatic variables at execution time.

CHAPTER 5 "ACCESSING VARIABLES THAT USE BIT FIELDS"

This chapter describes how to access variables that use bit fields.

PART II "USING STACK AREA EFFICIENTLY"

Part II describes how to use the stack area efficiently.

CHAPTER 6 "FUNCTION CALLS AND THE STACK"

This chapter briefly describes the stack area used when a function is called.

CHAPTER 7 "REDUCING FUNCTION CALLS BY EXPANDING FUNCTIONS IN LINE"

This chapter describes how to reduce the stack area by using inline expansion of functions in function calls.

CHAPTER 8 "REDUCING THE ARGUMENTS TO CONSERVE STACK AREA"

This chapter describes how to reduce the number of arguments in function calls so that less stack area is required.

CHAPTER 9 "CONSERVING STACK AREA BY IMPROVEMENTS ON THE AREA FOR FUNCTION RETURN VALUES"

This chapter explains the function return values for the register and the stack. Reducing the return values for the stack can reduce the used stack area.

PART III "USING LANGUAGE EXTENSIONS"

Part III describes the language extensions specific to the fcc907. Part III also discusses items in the extended language specifications that require special attention.

CHAPTER 10 "WHAT ARE LANGUAGE EXTENSIONS?"

This chapter describes the fcc907-specific extended language specifications, such as the qualifier for extensions, _ _asm statement, and "#pragma."

CHAPTER 11 "NOTES ON ASSEMBLER PROGRAMS IN C PROGRAMS"

This chapter provides notes on including assembler code with the _ _asm statements and #pragma asm/endasm of the extended language specifications.

CHAPTER 12 "NOTES ON DEFINING AND ACCESSING THE I/O AREA"

This chapter provides notes on specifying and mapping when using the _ _io type qualifier.

CHAPTER 13 "MAPPING VARIABLES QUALIFIED WITH THE _ _direct TYPE QUALIFIER"

This chapter provides notes on specifying and allocating variables declared with the _ _direct type qualifier.

CHAPTER 14 "CREATING AND REGISTERING INTERRUPT FUNCTIONS"

This chapter provides notes on using language extensions of the fcc907 to enable interrupt processing.

PART IV "MAPPING OBJECTS EFFECTIVELY"

This part explains how to map objects effectively.

CHAPTER 15 "MEMORY MODELS AND OBJECT EFFICIENCY"

This chapter describes the memory models of the fcc907 and explains object efficiency.

CHAPTER 16 "MAPPING VARIABLES QUALIFIED WITH THE TYPE QUALIFIER CONST"

This chapter provides notes on mapping variables declared with the type qualifier const.

CHAPTER 17 "MAPPING PROGRAMS IN WHICH THE CODE AREA EXCEEDS 64 Kbytes"

This chapter describes how to map programs when the code area exceeds 64 Kbytes.

CHAPTER 18 "MAPPING PROGRAMS IN WHICH THE DATA AREA EXCEEDS 64 Kbytes"

This chapter describes how to map programs when the data area exceeds 64 Kbytes.

In this manual, the designation <Notes> indicates items requiring special attention.

The sections entitled "Tip" provide information on functions that is useful for creating programs.

The Softune C Checker analyzes C source programs and outputs a warning for items requiring attention to ensure that the fcc907 does not output an error message.

The Softune C Analyzer analyzes function calls within the C source code of the program and displays information about such items as variables, relationship between function references, and the used amount of stack areas.

1. The contents of this document are subject to change without notice. Customers are advised to consult with FUJITSU sales representatives before ordering.
2. The information and circuit diagrams in this document are presented as examples of semiconductor device applications, and are not intended to be incorporated in devices for actual use. Also, FUJITSU is unable to assume responsibility for infringement of any patent rights or other rights of third parties arising from the use of this information or circuit diagrams.
3. The contents of this document may not be reproduced or copied without the permission of FUJITSU LIMITED.
4. FUJITSU semiconductor devices are intended for use in standard applications (computers, office automation and other office equipments, industrial, communications, and measurement equipments, personal or household devices, etc.).
   CAUTION:
   Customers considering the use of our products in special applications where failure or abnormal operation may directly affect human lives or cause physical injury or property damage, or where extremely high levels of reliability are demanded (such as aerospace systems, atomic energy controls, sea floor repeaters, vehicle operating controls, medical devices for life support, etc.) are requested to consult with FUJITSU sales representatives before such use. The company will not be responsible for damages arising from such use without prior approval.
5. Any semiconductor devices have inherently a certain rate of failure. You must protect against injury, damage or loss from such failures by incorporating safety design measures into your facility and equipment such as redundancy, fire protection, and prevention of over-current levels and other abnormal operating conditions.
6. If any products described in this document represent goods or technologies subject to certain restrictions on export under the Foreign Exchange and Foreign Trade Control Law of Japan, the prior authorization by Japanese government should be required for export of those products from Japan.

CONTENTS

PART I VARIABLE DEFINITIONS AND VARIABLE AREAS

This part describes the variable definitions and variable areas for creating C programs. This part first briefly describes memory mapping and the variables used for creating an F^2MC-16 family microcontroller embedded system. It then briefly describes the relationship between the variable definitions and variable areas. It concludes by describing how to efficiently create C programs.

CHAPTER 1 OBJECTS MAPPED INTO MEMORY AREAS

This chapter briefly describes objects that are mapped into memory areas before taking up the subject of the variable.

1.1 Program Components

This section briefly describes the program components.
Programs can be roughly divided into code and data.

■ Program Components

Programs created in C (C programs hereafter) and programs created in Assembler (assembler programs hereafter) can both be roughly divided into code and data sections.

Program
- Code (machine instruction): Read only
- Data: Read and write

❍ Code

This section in the program contains the machine instructions to be executed by the CPU.

The algorithm, which is coded as functions in a C program, is compiled and converted to machine instruction code.

The term "Code" refers to a set of execution instructions that are only read at execution.

❍ Data

The data is accessed by the program.

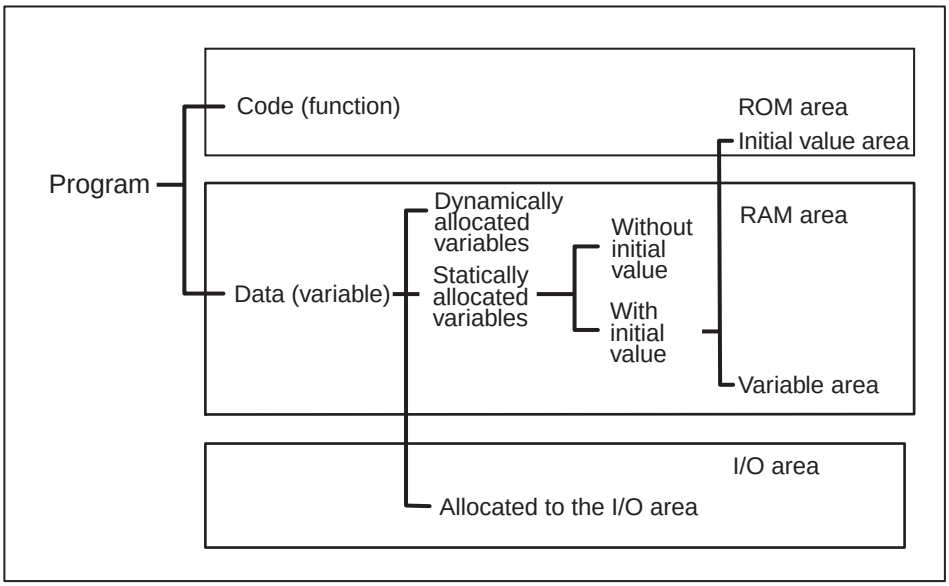
In a C program, the data includes variables, character strings, literal constants, and initial values.

Data can be read and written depending on the processing.

A C program can be classified as shown in Figure 1.1-1 "Classification of Objects in Programs for Embedded Systems and Allocation of Objects in the Memory Area". Variables, which are data items, can be classified into three types: Variables that are allocated dynamically, variables that are allocated statically, and variables that are allocated to the I/O area.

Dynamically allocated variables are allocated in a stack. Statically allocated variables can be classified into variables that are initialized and variables that are not. Initialized variables can be allocated both in the initial value area and the variable area.

Figure 1.1-1 Classification of Objects in Programs for Embedded Systems and Allocation of Objects in the Memory Area



1.2 Mapping into Memory Areas

This section briefly describes the types of memory areas and the objects that are mapped into them.

In embedded systems that use an F^{2}MC-16 family microcontroller, the memory area can be mainly classified into ROM, RAM, and I/O area.

■ Mapping into Memory Areas

An embedded system that uses an F^{2}MC-16 family microcontroller uses three types of memory areas: ROM area, RAM area, and I/O area.

❍ **Read only memory (ROM) area**

Objects mapped into the ROM area can only be read.

The code and initial value areas are allocated in the ROM area.

❍ **Random access memory (RAM) area**

Objects mapped into the RAM area can be read and written.

The data areas that are read and written to during program execution are allocated in the RAM area.

Stacks are also allocated in the RAM area.

❍ **Input/output (I/O) area**

I/O objects are mapped into the I/O area.

As shown in Figure 1.2-1 "Objects Generated by the C Compiler and Mapping into Memory Areas", code and the initial values of variables that can only be read at execution time are mapped into the ROM area. Variables that are read and written at execution time are mapped into the RAM area.

<Notes>

Since the values in the RAM area are undefined at system start, variables that are mapped into the RAM area must be initialized as described below before program execution:

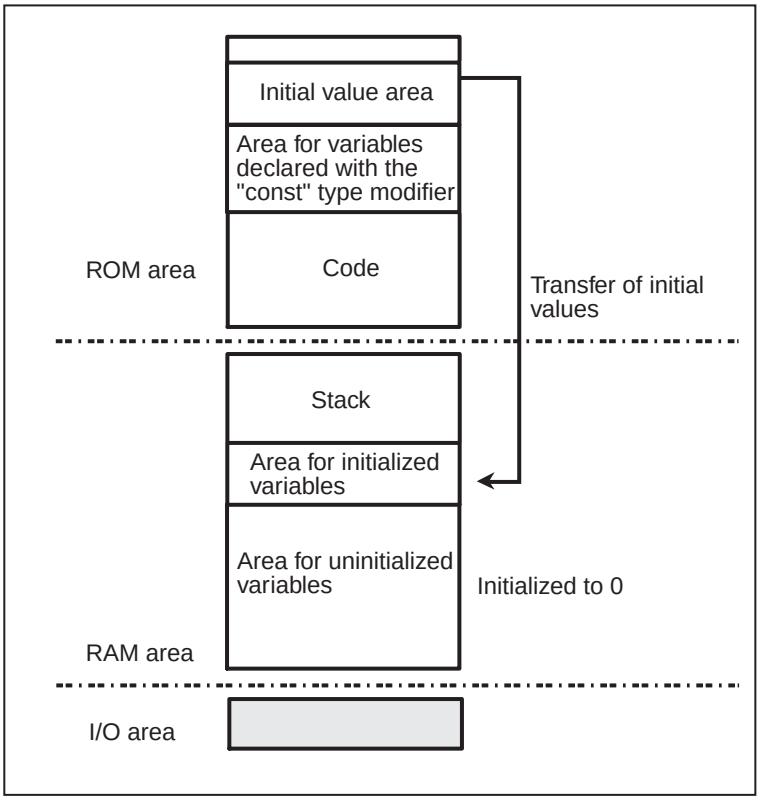
- Variable areas that are not initialized must be initialized to 0.
- The variables in the RAM area must be initialized using the initial values in the ROM area.

This initialization operation are performed using an initialization program called a startup routine[1].

The objects are mapped into their memory area during linking.

1. The startup routine is a program that performs initialization before executing the C program. An example for this is the program startup.asm supplied as a sample with the C compiler.
   Refer to the C compiler manual for information about the operations performed by the startup routine.

Figure 1.2-1 Objects Generated by the C Compiler and Mapping into Memory Areas



1.3 Dynamically Allocated Variables

This section briefly describes the dynamically allocated variables.
In a C program, the automatic variables that are defined in functions and the register variables are allocated dynamically.

■ Dynamically allocated variables

In a C program, the dynamically allocated variables are the automatic variables and the register variables defined in functions.

❍ Automatic variables

- One type of local variables
- Defined in functions
- Able to be accessed only in the function in which they were defined
- Allocated in a stack

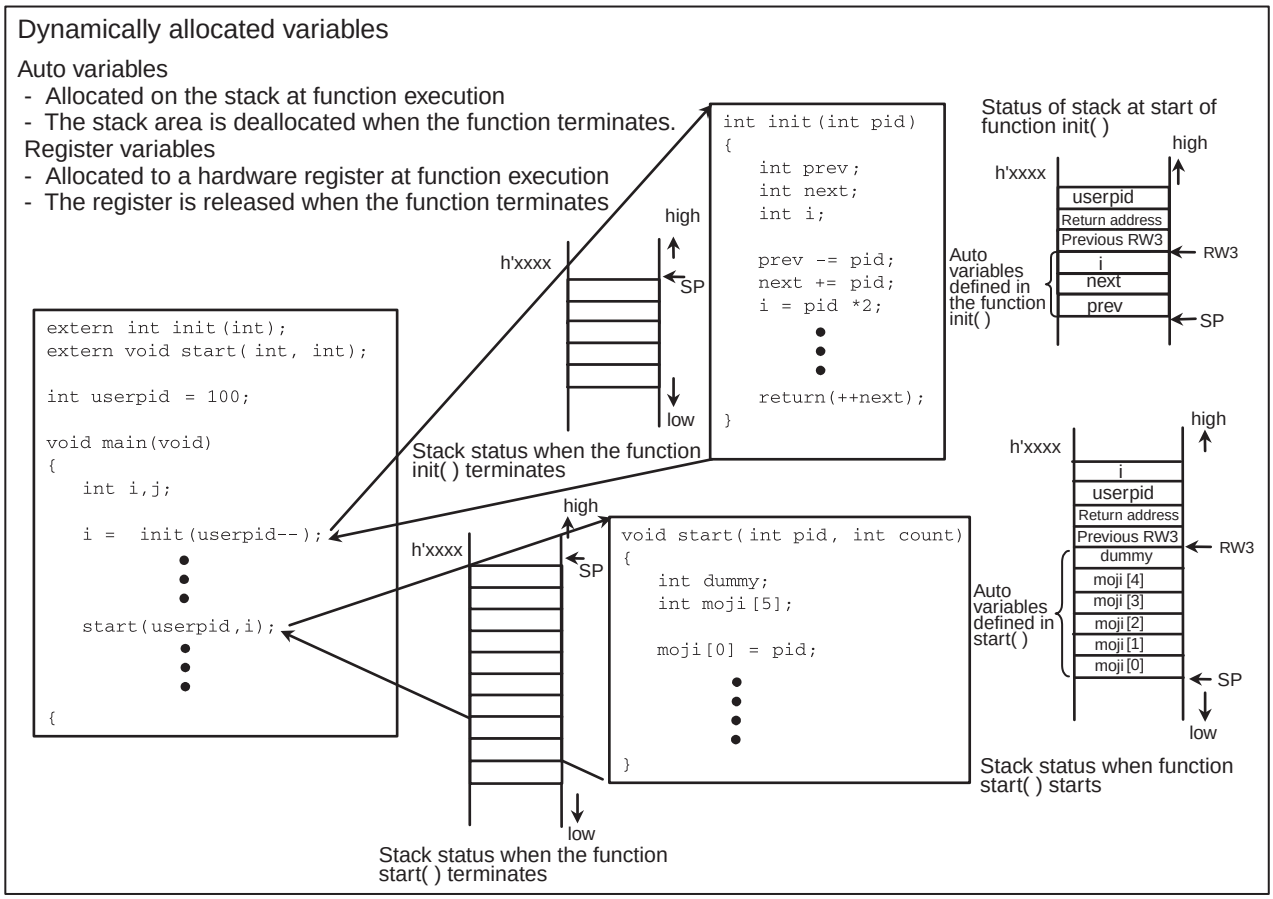
❍ Register variables

- One type of local variable
- Defined in a function
- Able to be accessed only in the function in which they were defined
- Allocated in registers

As shown in Figure 1.3-1 "Dynamically Allocated Variables", the stack area is allocated for automatic variables when a function is called. This area is deallocated when the function terminates. Automatic variables can be accessed only in the function that defined them.

During a function call, a register variable receives priority allocation to a hardware register. The register is released when the function terminates. As with automatic variables, register variables can be accessed only in the function in which they are defined.

Figure 1.3-1 Dynamically Allocated Variables



1.4 Statically Allocated Variables

This section briefly describes the statically allocated variables.
In a C program, external variables that are defined outside a function and variables declared as "static" are both allocated statically in a fixed RAM area.

■ Statically Allocated Variables

In a C program, external variables that are defined outside a function and variables declared as "static" are both allocated statically.

❍ **External variables**

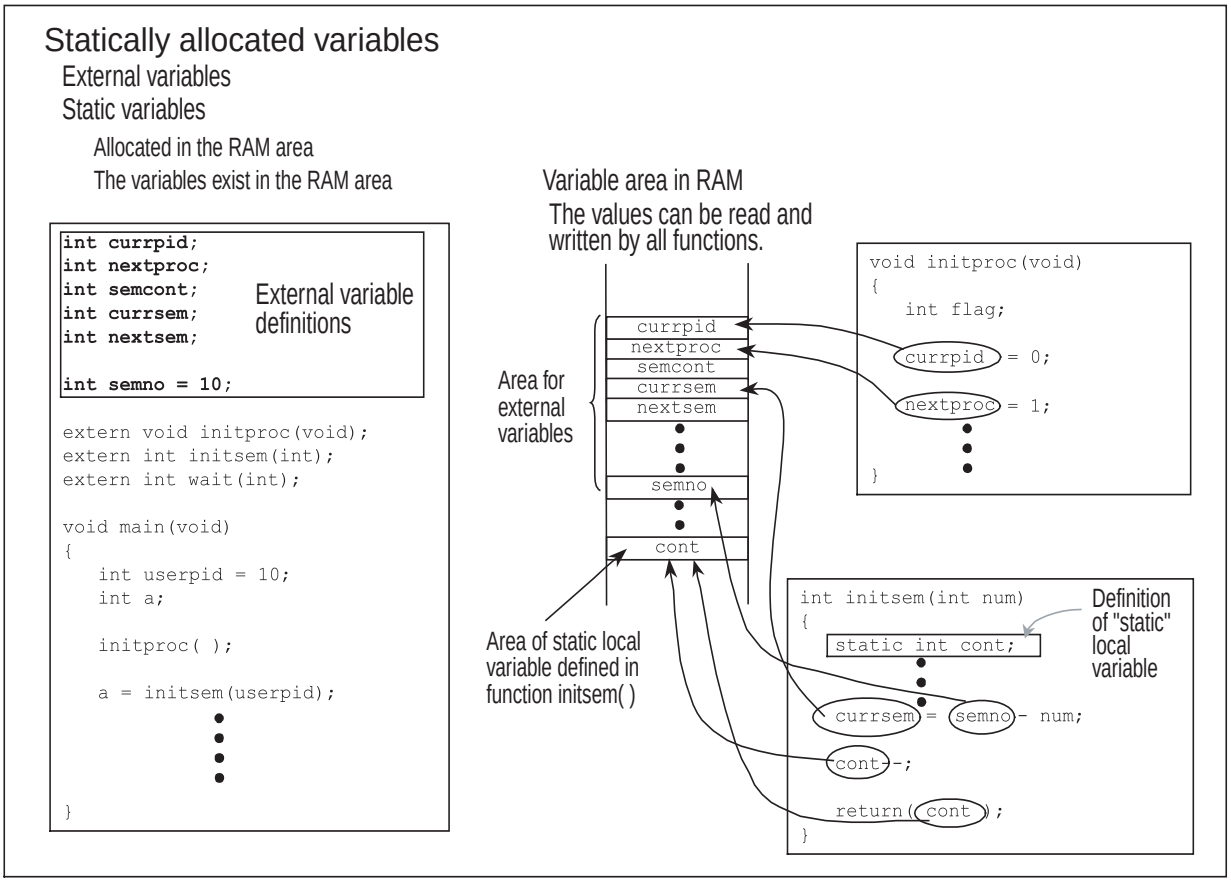
- Defined outside a function
- Able to be accessed from the entire module
- Statically allocated in memory

❍ **Static variables**

- Able to be accessed only within their defined scope
- Statically allocated in memory

As shown in Figure 1.4-1 "Statically Allocated Variables", external variables and static variables are allocated in a fixed RAM area at program execution. External variables can be accessed by all functions. Static variables are valid only within their defined scope. For details of static variables, see Section 2.4 "Variables Declared as "static" and Their Variable Area".

Figure 1.4-1 Statically Allocated Variables



CHAPTER 2 VARIABLE DEFINITIONS AND VARIABLE AREAS

This chapter briefly describes the variable definitions and variable areas to which variables are output as a result of compilation. It then describes the relationship between initial values and the variable areas used for variables. The chapter also describes variables declared as "static," which is one type of static variables that have a special format.

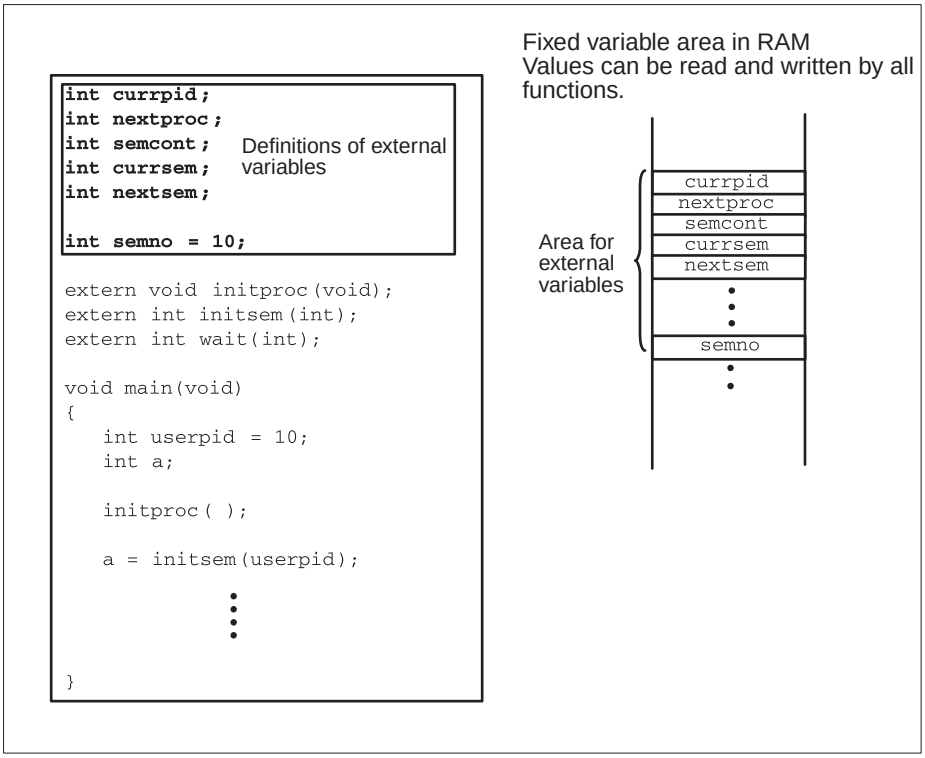
2.1 External Variables and Their Variable Area

This section briefly describes the external variables and the variable areas. The external variables are defined outside a function. The area for external variables is fixedly allocated in RAM.

■ External Variables

As shown in Figure 2.1-1 "Definitions of External Variables", the external variables, which are defined outside a function, are statically allocated. They are allocated in the memory area and can be accessed from the entire module.

Figure 2.1-1 Definitions of External Variables



The name of the section to which a variable is output as a result of compilation depends on the storage class, type qualifier, and whether an initial value is specified at definition. For details, see the fcc907 manual. Table 2.1-1 "Variables and Data Section to Which a Variable Is Output (for Small and Medium Models)" and Table 2.1-2 "Variables and Data Section to Which a Variable Is Output (for Large and Compact Models)" list the relationship between the external variable definitions and the section to which a variable is output as a result of compilation.

Table 2.1-1 Variables and Data Section to Which a Variable Is Output (for Small and Medium Models)

Type qualifier					Specification of Initial value	Variable area		Initial value area	
_ _io	_ _direct	const	_ _near	_ _for		Section type	Section name	Section type	Section name
						DATA	DATA		
					o	DATA	INIT	CONST	DCONST
		o			o	CONST	CONST	DATA	CINIT
	o					DIR	DIRDATA		
	o				o	DIR	DIRINIT	DIRCONST	DIRCONST
o						IO	IO		
			o			DATA	DATA		
			o		o	DATA	DINIT	CONST	DCONST
		o	o		o	CONST	CONST	DATA	CINIT
				o		DATA	DATA_*		
				o	o	DATA	DINIT_*	CONST	DCONST_*
				o	o	CONST	CONST_*	DATA	CINIT_*

Table 2.1-2 Variables and Data Section to Which a Variable Is Output (for Large and Compact Models)

Type qualifier					Specification of Initial value	Variable area		Initial value area	
_ _io	_ _direct	const	_ _near	_ _for		Section type	Section name	Section type	Section name
						DATA	DATA_*		
					o	DATA	INIT_*	CONST	DCONST_*
		o			o	CONST	CONST_*	DATA	CINIT_*
	o					DIR	DIRDATA		
	o				o	DIR	DIRINIT	DIRCONST	DIRCONST
o						IO	IO		
			o			DATA	DATA		
			o		o	DATA	DINIT	CONST	DCONST
		o	o		o	CONST	CONST	DATA	CINIT
				o		DATA	DATA_*		
				o	o	DATA	DINIT_*	CONST	DCONST_*
				o	o	CONST	CONST_*	DATA	CINIT_*

[Tip]

Softune C Checker:

The Softune C Checker outputs a warning for variables in an analyzed module that are not accessed at all during external access. Accordingly, define external variables only after

verifying the intended scope. Meaningless access declarations make a program look poorly written.

2.2 Initial Values and Variable Area for External Variables

This section describes the relationship between the initial values and variable areas of external variables.
In fcc907, when an initial value is specified at definition of an external variable, variable area is allocated in both the ROM and RAM areas.

■ Initial Values and the Variable Area for External Variables

Variables can be classified into the following three types according to how initialization is handled when the variables are defined. Whether an initial value is required depends on the way in which the variable is to be used.

Initial value not required

No initial value specification (The variable does not need to be initialized to 0.)

Initial value 0

No initial value specification (The variable must be initialized to 0.)

Initial value other than 0

An initial value other than 0 is specified

The fcc907, handles two types of external variables: external variables for which an initialization value is specified when they are defined (initialized variables hereafter) and external variables for which no initialization value is specified when they are defined (uninitialized variables hereafter).

The variable area and initial value area sections are output for initialized variables. For uninitialized variables, the section variable area are output.

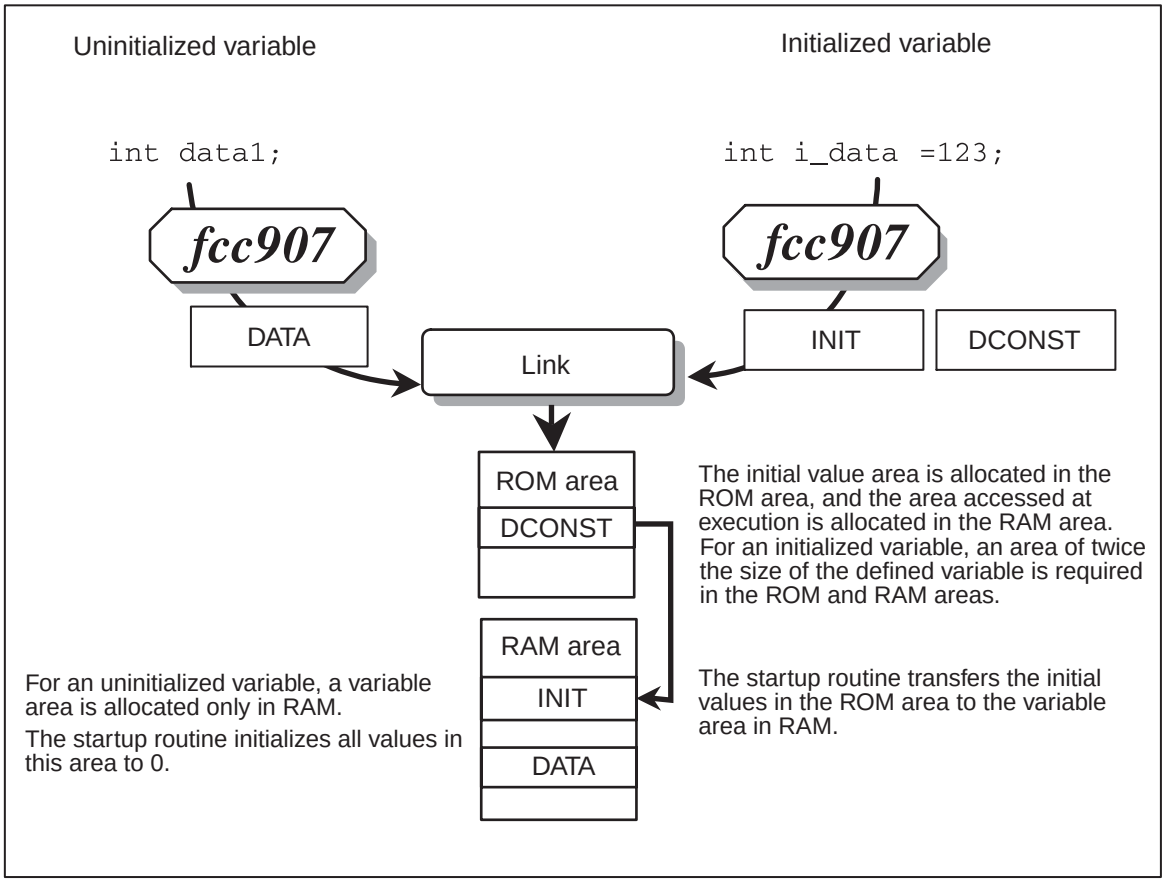
Figure 2.2-1 "Variable Areas and Memory Mapping" shows the relationship between the output sections and memory mapping for initialized and uninitialized variables. For initialized variables, a variable area is allocated in both ROM and RAM. The RAM area values are undefined at system start. After system start, the startup routine transfers the initial values from ROM to the RAM variable area. This operation completes initialization of the variable.

For uninitialized variables, a variable area is allocated only in RAM. The value of this RAM area is also undefined at system start. After system start, the startup routine initializes all values in the variable area for uninitialized variables to 0.

<Notes>

Although the startup routine provided as a sample initializes all uninitialized variables to 0, perform initialization based on the program system that is to be created.

Figure 2.2-1 Variable Areas and Memory Mapping



2.3 Initialized Variables and Initialization at Execution

This section describes initialized variables and the initialization of uninitialized variables at program execution.

■ Initialized Variables and Initialization at Execution

As shown in Figure 2.2-1 "Variable Areas and Memory Mapping", initialized variables require an initial value area and a variable area, which means that the totally required area is twice that of defined variables. For uninitialized variables, only a variable area needs to be allocated. Because the initialization value is only accessed the first time, a method is also provided that allows to initialize the variable when the respective function is executed, making it unnecessary to specify an initial value at definition time.

Figure 2.3-1 "Initialized Variables and Initial Value Assignment at Function Execution" shows an example of a function in a variable is initialized beforehand, and an example of a function in which the value is set at the beginning of the function.

See function list1( ) in (1), "Definition as an initialized variable," in Figure 2.3-1 "Initialized Variables and Initial Value Assignment at Function Execution". Function list1( ) allocates a 2-byte area in the variable area, INIT section, and initial value area DCONST section for variable i_data for which an initial value specified. The INIT section is allocated in RAM and the DCONST section is allocated in ROM. The startup routine transfers the initial value from ROM to the variable area in RAM.

See function list2( ) in (2), "Assigning a value when the variable is used," in Figure 2.3-1 "Initialized Variables and Initial Value Assignment at Function Execution". Function list2( ) allocates only a 2-byte variable area DATA for the variable i_data in RAM. However, a code for assigning a value to the variable is required. Compared with (1), the area for the value is smaller by 2 bytes, but the code area is bigger by 6 bytes.

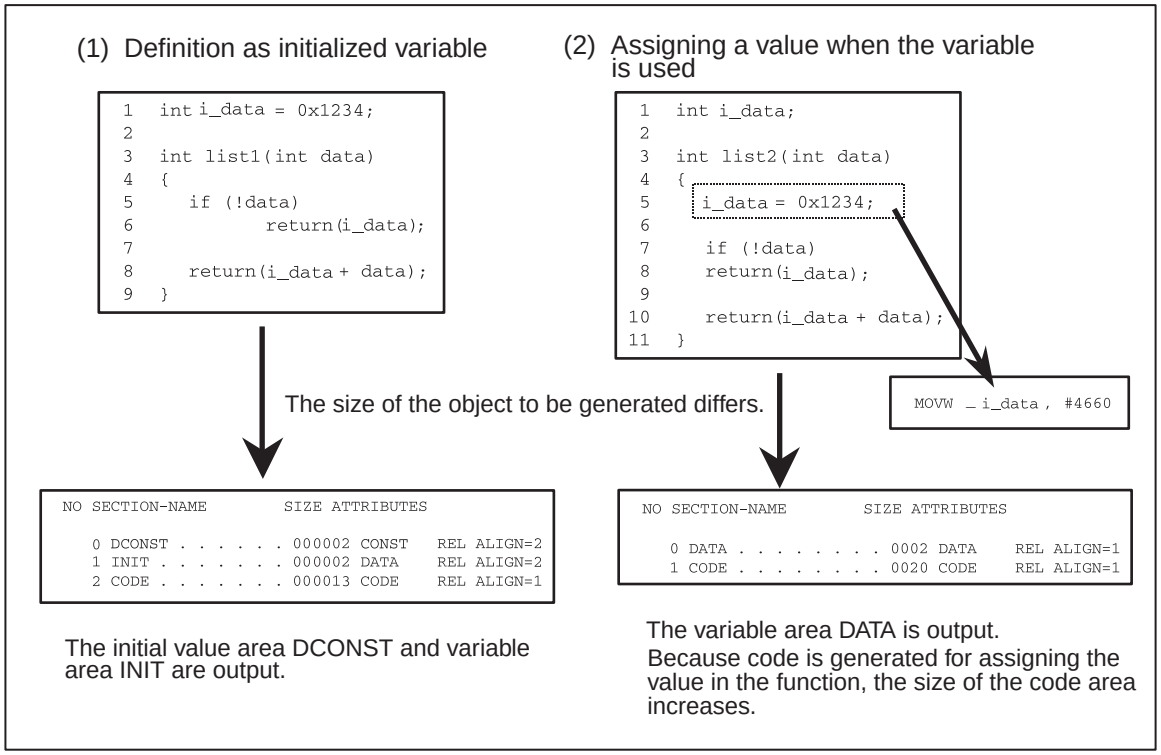
The startup routine is used to transfer the initial value of the variable to the variable area in RAM. To assign an initial value in the function, a 6-byte code is required whenever a 2-byte variable is assigned.

If we take the case of a variable that is initialized using 10 different values, code of (6 bytes x 10) = 60 bytes is required. When a variable is defined as an initialized variable, the value area in the ROM will increase by 20 bytes. Because the startup routine handles transfer, it is assumed that the size of the code will not increase. Thus, when the increase of 60 bytes in code is conmpared with the increase of 20 bytes in the variable area, it can be said that use of the ROM area is more economically when an initialized variable is defined.

<Notes>

Setting an initial value for a variable that does need not to be initialized wastes ROM area. Setting an initial value of 0 at definition time and using the startup routine to initialize uninitialized variables to 0 wastes initial value area. Set the initial value of an external variable only after carefully checking whether initialization is necessary.

Figure 2.3-1 Initialized Variables and Initial Value Assignment at Function Execution



2.4 Variables Declared as "static" and Their Variable Area

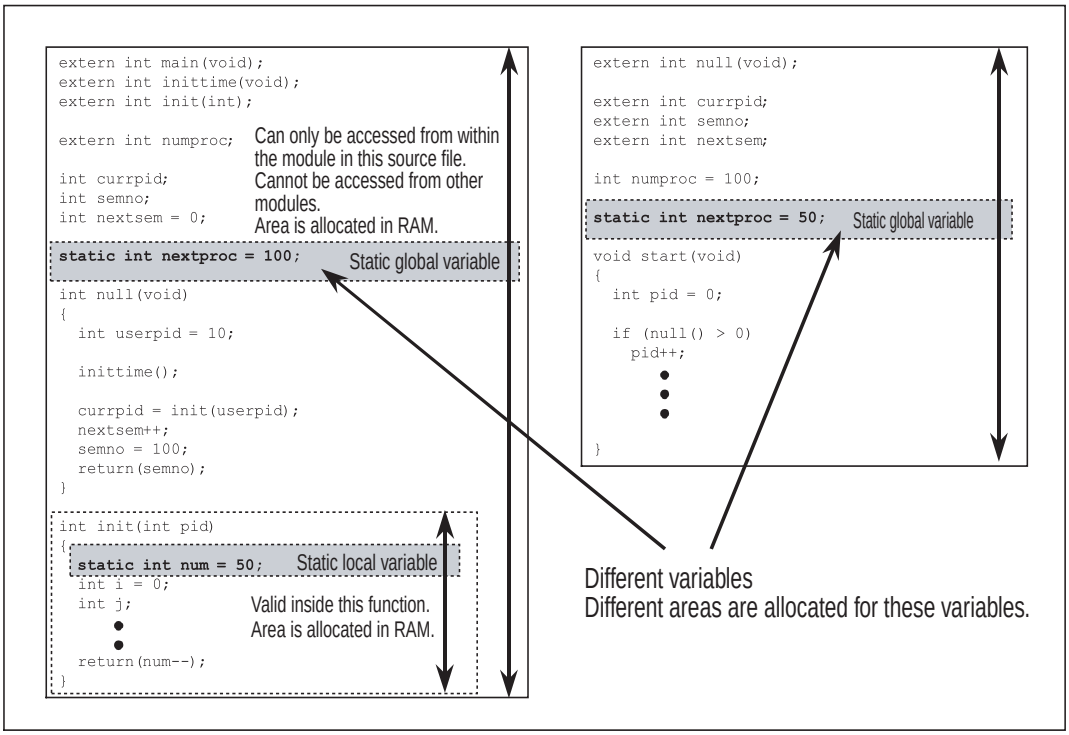
This section briefly describes variables declared as "static" and the variable area they require. Variables declared as "static" are only one type of variables that are allocated statically.
For a variable declared as "static", area in RAM is allocated for the variable statically. The scope of variables declared as "static" depends on where they are defined. A variable that is defined outside a function is referred to as a static global variable. A variable that is defined inside a function is referred to as a static local variable. Even if the module or function where the variables are defined terminates, the values are retained in the variable area within RAM.

■ Variables Declared as "static" and Their Variable Area

Whether a variable is dynamically or statically allocated depends on where it is defined. Area for external variables is allocated in RAM if the variable has been defined outside a function. Because the area is always present in RAM, the area can be accessed from the entire module.

For a variable declared as "static", area in RAM is allocated for the variable statically. However, as shown in Figure 2.4-1 "Scope of Variables Declared as "static"", the scope of the variable depends on where it is defined. A variable that is defined outside a function is referred to as a static global variable. A variable that is defined within a function is referred to as a static local variable. Static global and static local variables are output to the same section for external variables.

Figure 2.4-1 Scope of Variables Declared as "static"



Section 2.4.1 "Example of Function with Static Global Variable" provides an example of a function that uses a static global variable. Section 2.4.2 "Example of a Function with a Static Local Variable" provides an example of a function that uses a static local variable.

The scope of a variable declared as "static" depends on where the variable is defined. Even if the module or function where the variable is defined terminates, the value is retained in the variable area in RAM.

The advantage of using a variable defined as a static local variable in a function as a counter variable for the number of times the function is called is that the value will be retained. On the other hand, if a variable declared as "static" is used for a task where the value need not be retained, RAM area will be used inefficiently. Define a static variable only after carefully investigating whether this is necessary.

[Tip]

Softune C Checker:

The Softune C Checker outputs a warning for variables that have been declared as "static" in the analyzed module, but have not been accessed at all. Accordingly, carefully check the scope of variables and define variables as static variables only when necessary.

In addition, for Variables declared as "static" for which no initial value has been specified, a warning requesting that the variables be initialized will be output. If necessary, specify an initial value.

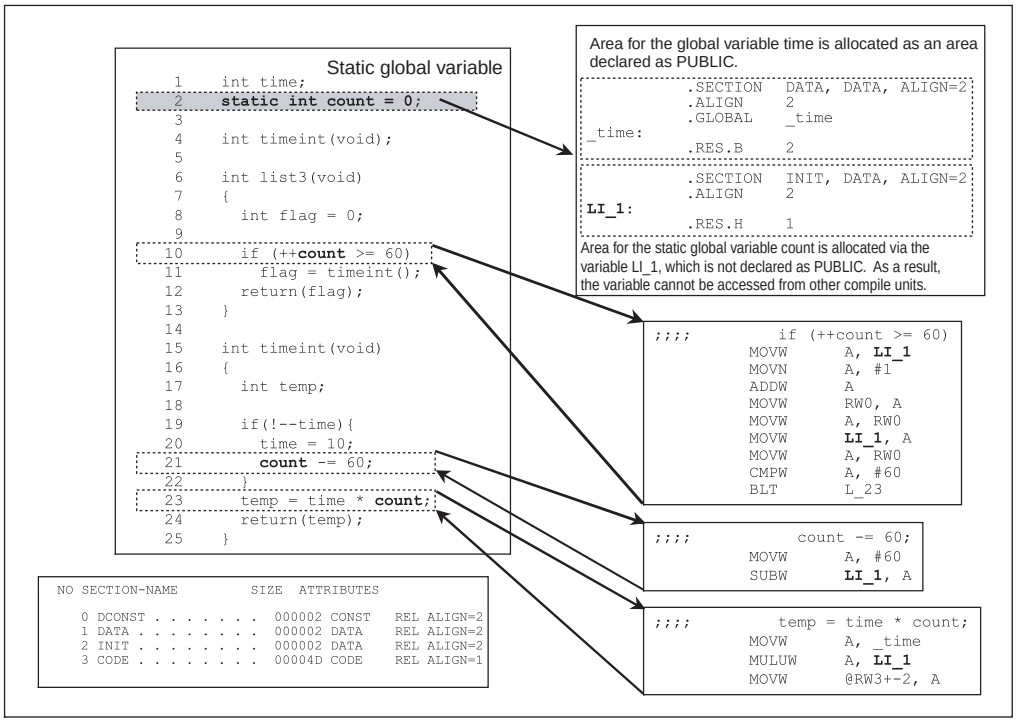
2.4.1 Example of Function with Static Global Variable

Figure 2.4-2 "Example of a Function that has a Static Global Variable" shows an example of a function that has a static global variable. The variable count, which is declared as "static" outside the function, is a static global variable.

■ Example of a Function with Static Global Variable

Area for the static global variable count is allocated via the variable LI_1, which is not declared as PUBLIC. RAM area is therefore allocated for the variable count and the value retained. Note, however, that this variable cannot be accessed from other compile units.

Figure 2.4-2 Example of a Function with Static Global Variable



2.4.2 Example of aunction with Static Local Variable

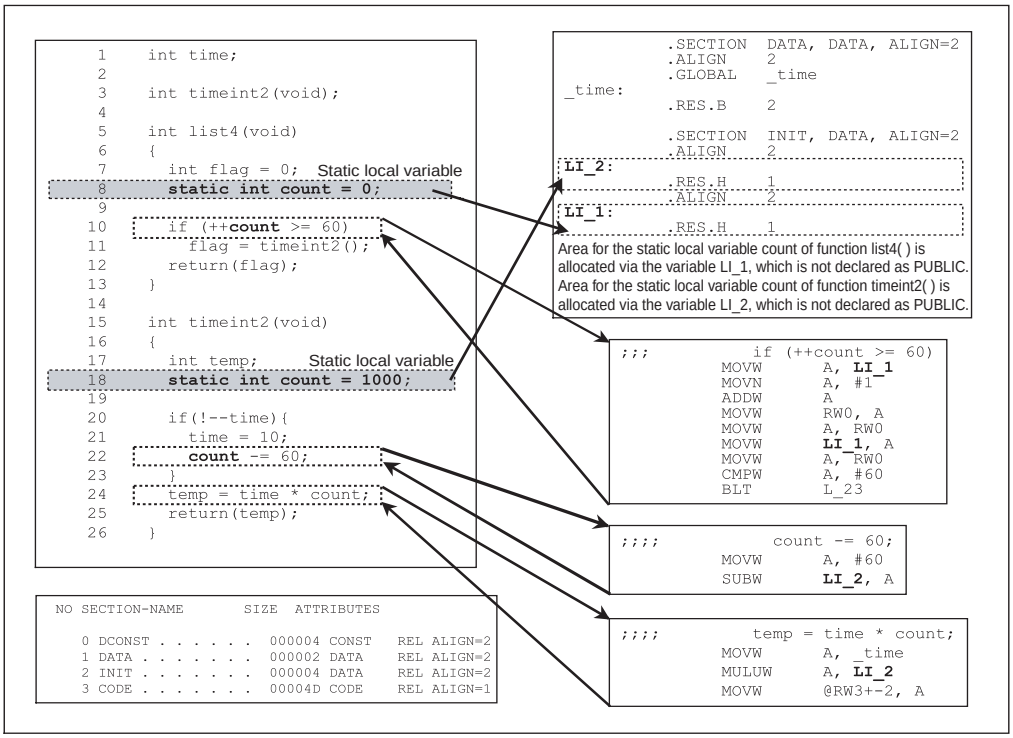
Figure 2.4-3 "Example of a Function with Static Local Variable" shows an example of a function that has a static local variable. The variable count, which is declared as "static" in the function, is a static local variable.

■ Example of a Function with Static Local Variable

Area for the static local variable count defined in function list4( ) is allocated via the variable LI_1, which is not declared as PUBLIC.

Similarly, area for the static local variable count defined in function timeint( ) is allocated via the variable LI_2, which also is not declared as PUBLIC. A separate area in RAM is allocated for each of the static local variables "count" and their values are retained. The scope of these variables is within the defined function. The variables cannot be accessed from other functions even within the same compilation unit.

Figure 2.4-3 Example of a Function with Static Local Variable



CHAPTER 3 READ-ONLY VARIABLES AND THEIR VARIABLE AREA

This chapter describes how to use read-only variables.
A value is read or written for a variable at execution. Therefore, the variable areas are mapped into RAM areas, which can be read and written. However, there are variables that are at execution only read and do not need to be changed. Examples for this type of variable are messages, such as opening or error messages. Mapping variables that are read-only in RAM areas in the same way as normal external variables has the result that these RAM areas are only read at execution. As a result, valuable RAM space will be wasted. This chapter describes two methods for reducing the required areas within RAM.

3.1 Numeric Constants and #define Definition

This section describes how to use the #define definition to define read-only variables as numeric constants.
Because this method does not allocate variable areas, RAM area usage can be reduced.

■ Numeric Constants and #define Definition

Figure 3.1-1 "Defining External Variables and Defining Variables Using the #define Statement" shows an example of defining read-only variables as initialized external variables and using the #define statement to define the read-only variables as numeric constants in a macro definition.

See function list5( ) of (1), "External variable definitions," in Figure 3.1-1 "Defining External Variables and Defining Variables Using the #define Statement". Because initialized variables have been defined for function list5( ), the variable area INIT section and initial value area DCONST section are generated. At linkage, the initial value area DCONST section is mapped into the ROM area. The variable area INIT section is mapped into the RAM area. The startup routine transfers the initial value in the ROM area to the RAM area. The following variables are defined for function list5( ):

- char-type variable (1 byte) c_max
- int-type variable (2 bytes) maxaddr
- float-type variable (4 bytes) pai
- double-type variable (8 bytes) d_data

The variable area INIT is allocated in the RAM area for these variables. Read-only variables are not written to at execution. From the viewpoint of economical use of the RAM area, this 15-byte variable area will be wasted.

The value of an external variable is referenced on the basis of the address of the external variable.

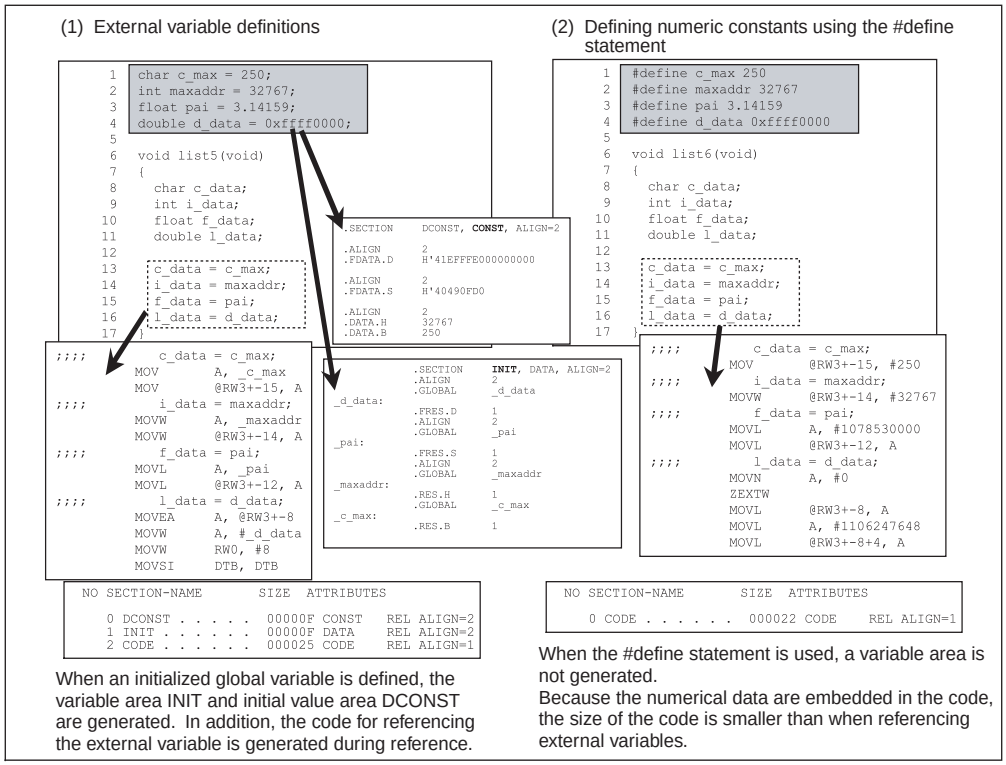
As shown below, the size of the code generated at reference depends on the variable type.

- To reference a char-type (1 byte) variable: 6 bytes
- To reference an int-type (2 bytes) variable: 5 bytes
- To reference a float-type (4 bytes) variable: 7 bytes
- To reference a double-type (8 bytes) variable: 11 bytes

See the function list6( ) of (2), "Defining numeric constants using the #define statement," in Figure 3.1-1 "Defining External Variables and Defining Variables Using the #define Statement". Function list6( ) defines c_max, maxaddr, pai, and d_data using the macro definition of the #define statement. The value of the macro-defined numeric constant is embedded in the code, and a variable area is not generated. Because the code for referencing the external variable is not generated, the total code length will be relatively short. The execution speed will also be increased. The code to be generated depends on the numeric constant.

Macro-defined variables have no type. Therefore, type conversion may be performed at assignment depending on the type of the variable to be assigned. This can lead to unexpected results.

Figure 3.1-1 Defining External Variables and Defining Variables Using the #define Statement



The above results for read-only variables can be summarized as follows:

Defining a variable as an initialized external variable

Variable area is allocated in RAM even though no writing is performed.

Defining a variable as a numeric constant

The variable area is not allocated in RAM.

Since the value is directly embedded in the code, the execution speed is higher than for using external variables.

Because the type of these values is not clearly defined, unexpected operation results can occur due to type conversion.

From the viewpoint of economical use of RAM area, it is more efficient to define read-only variables as numeric constants. As the values of numeric constants are directly accessed, processing speed will increase. However, if the number of accesses to numeric constants increases, the size of the generated code generated will increase proportionally to the number of accesses to numeric constants.

Whether to define read-only variables as normal external variables or as numeric constants must be decided based on the nature of the program system to be created. For a program system where the processing speed is more important than the size of the ROM area, it will be more efficient to use constant values defined using the #define statement.

3.2 Defining Variables Using the const Type Qualifier

This section describes how to define read-only variables using the "const" type qualifier.
Because this method directly accesses the initial value areas allocated in ROM, the size of the RAM area can be reduced.

■ Defining Variables Using the "const" Type Qualifier

Figure 3.2-1 "Output Section of a Variable Declared with the const Type Qualifier and Mapping into Memory" shows the relationship between the section to which a variable is output as a result of compilation and mapping into memory.

A const type-qualified variable is normally output to the variable area CONST section only. This CONST section is mapped into the ROM area. When a variable is accessed, the variable area in the ROM area is accessed directly.

Handling of a const-type qualified variable depends on the hardware, compiler, and memory model to be used. See Chapter 2 "MAPPING VARIABLES QUALIFIER WITH THE TYPE QUALIFIER const" for details on mapping a const-type qualified variable.

Figure 3.2-1 Output Section of a Variable Declared with the const Type Qualifier and Mapping into Memory

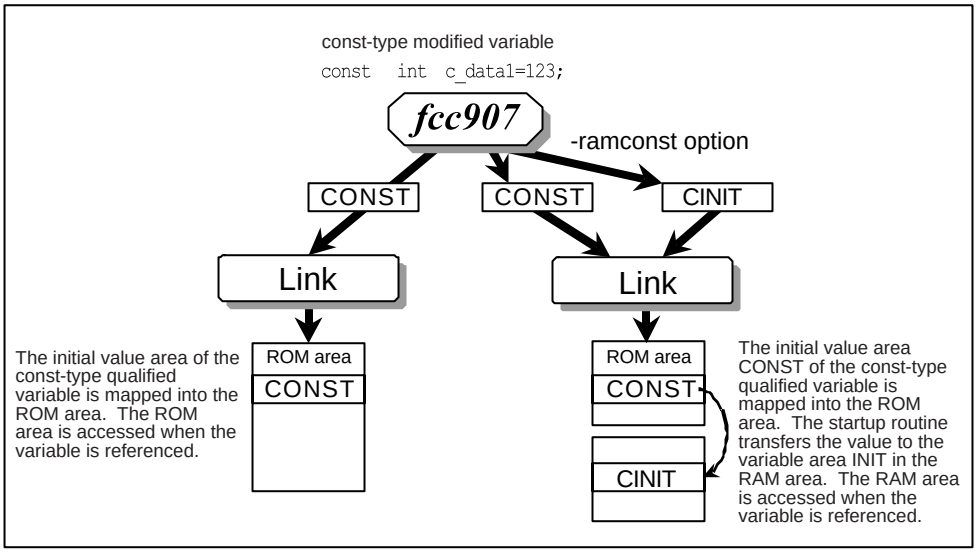


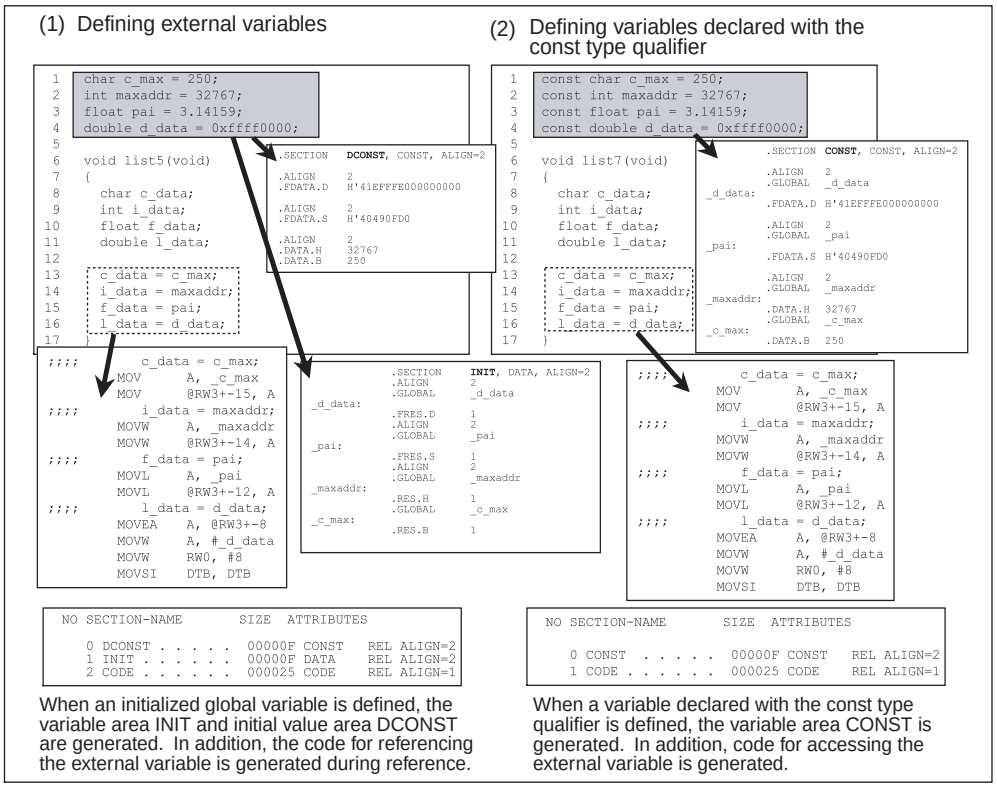
Figure 3.2-2 "Defining External Variables and Defining Variables Using the const Type Qualifier" shows a function that defines a read-only value as an initialized external variable and a function that defines the value as variable declared with the const type qualifier.

See function list5( ) of (1), "External variable definitions," in Figure 3.2-2 "Defining External Variables and Defining Variables Using the const Type Qualifier". Because initialized variables have been defined for function list5( ), the variable area INIT section and initial value area DCONST section are generated. At linkage, the DCONST section is mapped into the ROM area. The INIT section is mapped into the RAM area. The startup routine transfers the initial value in the ROM area to the RAM area. Function list5( ) outputs char-type variable c_max, int-type variable maxaddr, float-type variable pai, and double-type variable d_data to the variable area INIT. Read-only variables are not written to at execution. As a result, this 15-byte variable

area and the RAM area will not be used economically.

See function list7( ) of (2), "Defining variables declared with the const type qualifier," in Figure 3.2-2 "Defining External Variables and Defining Variables Using the const Type Modifier". Function list7( ) outputs a variable to the 15-byte variable area CONST section. At linkage, the CONST section is mapped into the ROM area. Because the ROM area is directly accessed at accessing, the RAM area can be used economically.

Figure 3.2-2 Defining External Variables and Defining Variables Using the const Type Qualifier



[Tip]

Softune C Checker:

The Softune C Checker outputs a warning in the following cases:

- A variable has been declared with multiple const type qualifiers.
- A variable declared with the const type qualifier has been defined, but no initial value has been set.
- An attempt was made to change the value of a variable declared with the const type qualifier.

Use this for reference when defining variable declared with the const type qualifier.

Softune C Analyzer:

Among the external variables of an analyzed program, the Softune C Analyzer displays variables whose values are not changed by the program as candidates for declaration as "const." This is helpful for determining which variables to declare with the "const" type qualifier.

CHAPTER 4 USING AUTOMATIC VARIABLES TO REDUCE THE VARIABLE AREA

This chapter describes how to reduce variable areas using "automatic" variables. For automatic variables, the variable areas are allocated on the stack when the function is called. The variable areas are deallocated at the termination of the function. Variables that are referenced only from within the function are defined as automatic variables to reduce the variable areas.

4.1 Automatic Variables and Statically Allocated Variables

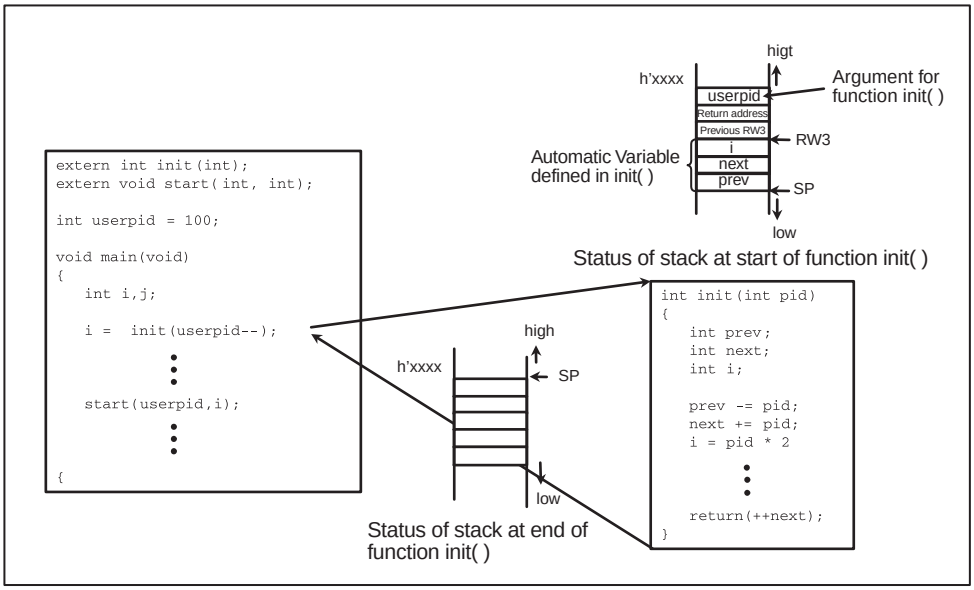
This section explains which variables are allocated as automatic variables and which are statically allocated.
As shown in Figure 4.1-1 "Automatic Variables and Status of Variable Areas on the Stack", an automatic variable is a variable that has been defined in a function. When the function is called, variable area is allocated in the stack for the automatic variable. The allocated variable area is released when the function terminates.

■ Variable Areas of Automatic Variables

Because variable area is allocated for automatic variables dynamically, automatic variables are also referred to as a dynamically allocated variables. Automatic variables can be referenced only from within a function.

The position on the stack where the Automatic Variable area is allocated depends on the status of the variable at function call. The Automatic Variables are not initialized at allocation. Therefore, if a variable defined as an automatic variable is used without being initialized, the value of the variable will be unpredictable.

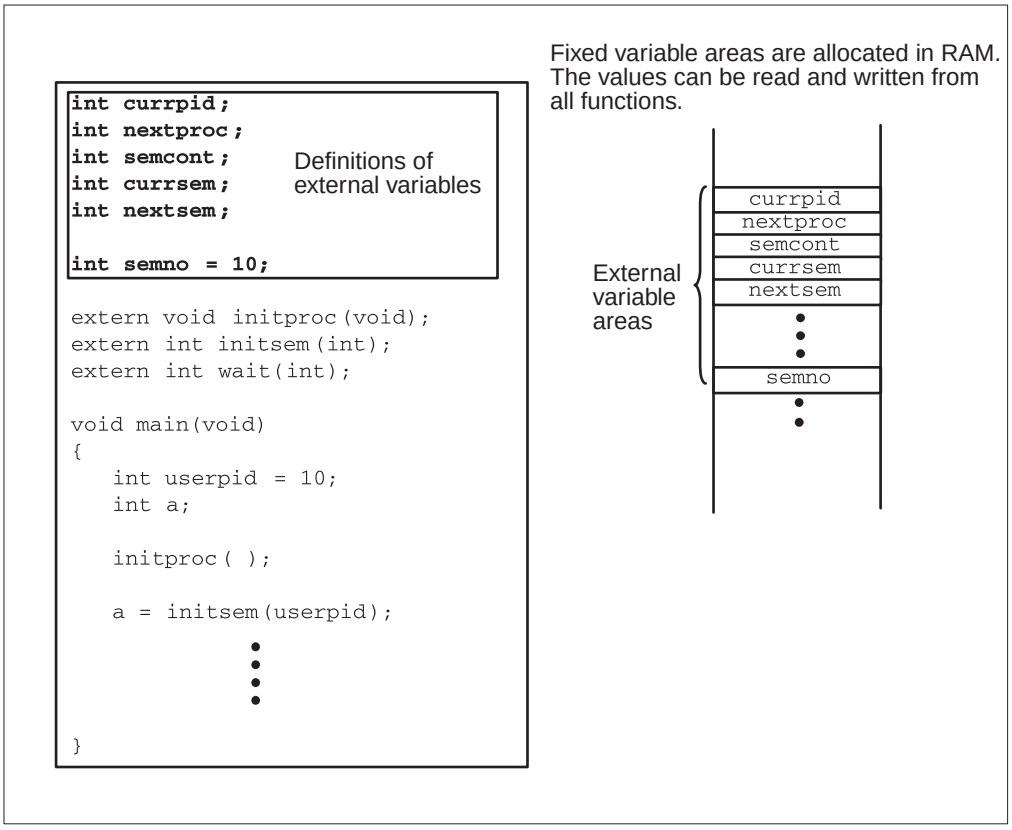
Figure 4.1-1 Automatic Variables and Status of Variable Areas on the Stack



■ Statically Allocated Variables and Variable Areas in RAM

As shown in Figure 4.1-2 "Statically Allocated Variables and Variable Areas in RAM", variable areas are allocated in the RAM area for statically allocated variables. The areas of the statically allocated variables are always located in the RAM area. External variables defined outside a function and variables declared as "static" are the statically allocated variables. External variables can be accessed from everywhere in the program. Variables declared as "static" can be classified into static local variables and static global variables depending on the location of their definition. The scope of the two types of variable differs.

Figure 4.1-2 Statically Allocated Variables and Variable Areas in RAM

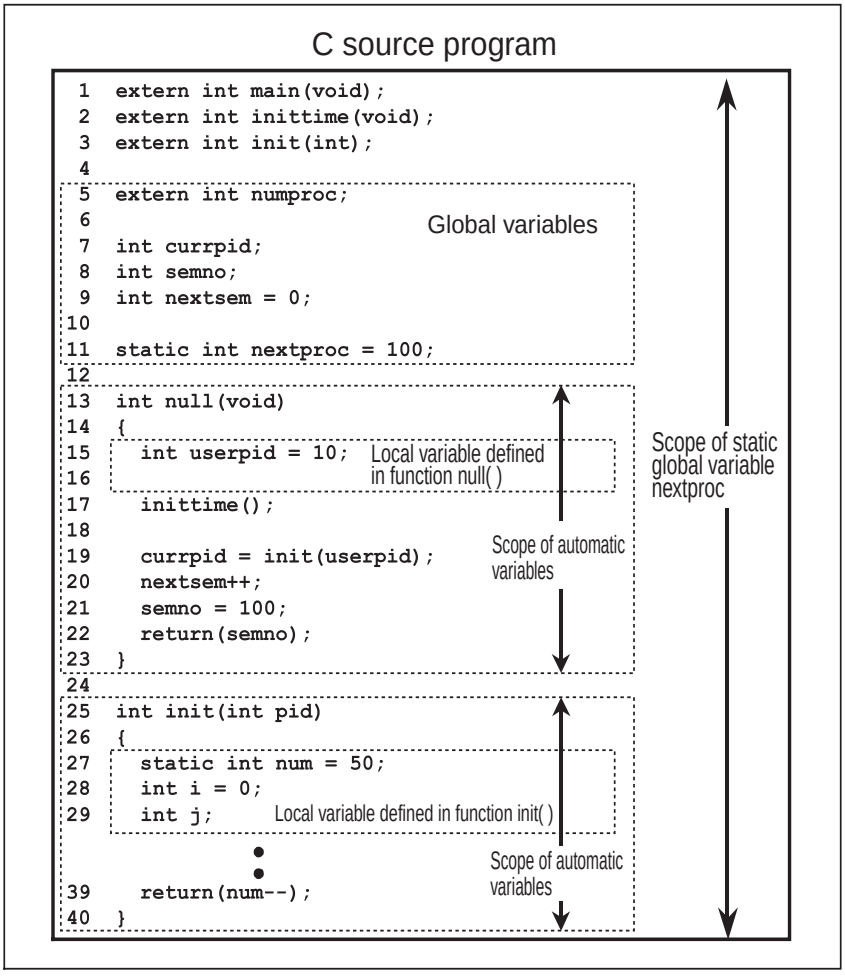


■ Definition and Scope of Automatic Variables and Statically Allocated Variables

Figure 4.1-3 "Definitions and Scope of Automatic Variables and of Statically Allocated Variables" shows scope and definitions of automatic variables and statically allocated variables.

Statically allocated variables can be divided into initialized variables and uninitialized variables. As described above, initial value area is allocated in the ROM area and variable area is allocated in the RAM area for an initialized variable. For an uninitialized variable, variable area is allocated in the RAM area. These statically allocated variables are initialized to their initial values or to 0 before control is passed to the C program.

Figure 4.1-3 Definitions and Scope of Automatic Variables and of Statically Allocated Variables



[Tip]

Softune C Checker:

The Softune C Checker outputs the following warnings for automatic variables:

- An automatic variable is not used
- An automatic variable is accessed without specifying a value

Softune C Analyzer:

The Softune C Analyzer lists the analysis results and the access status of external variables. This list can be used to check from which function a defined external variable is accessed. Variables that are only accessed by a defined module can also be identified from these results.

4.2 Using Automatic Variables

This section describes the merits of using automatic variables.
Reducing the number of external variables and using automatic variables that can only be locally accessed within a function can result in more economical use of the variable area.

■ External Variables and Automatic Variables

External variables can be divided into external variables declared as "const" and those that are not. Area for external variables that are not declared with the const type qualifier is allocated in RAM. However, careful review of the created program will often find that variables that are accessed only within a specific function have nevertheless been defined as external variables. Defining a variable whose usage range is restricted as external variable will increase the size of the variable area. Reducing the number of external variables and using automatic variables, which can be accessed only from within a function, can result in more economical use of the variable area.

As shown in Figure 1.3-1 "Dynamically Allocated Variables", and Figure 4.1-1 "Automatic Variables and Status of Variable Areas on the Stack", area for an automatic variable is allocated on the stack when a function is executed. The area is released when the function terminates. Compared with defining an external variable for each module, this enables more economic use of the variable area. However, if function calls are deeply nested, the amount of variable area allocated on the stack will increase. Figure 4.2-1 "Nesting of Function Calls and Stack States" shows nesting of function calls and the respective stack states.

Figure 4.2-1 Nesting of Function Calls and Stack States

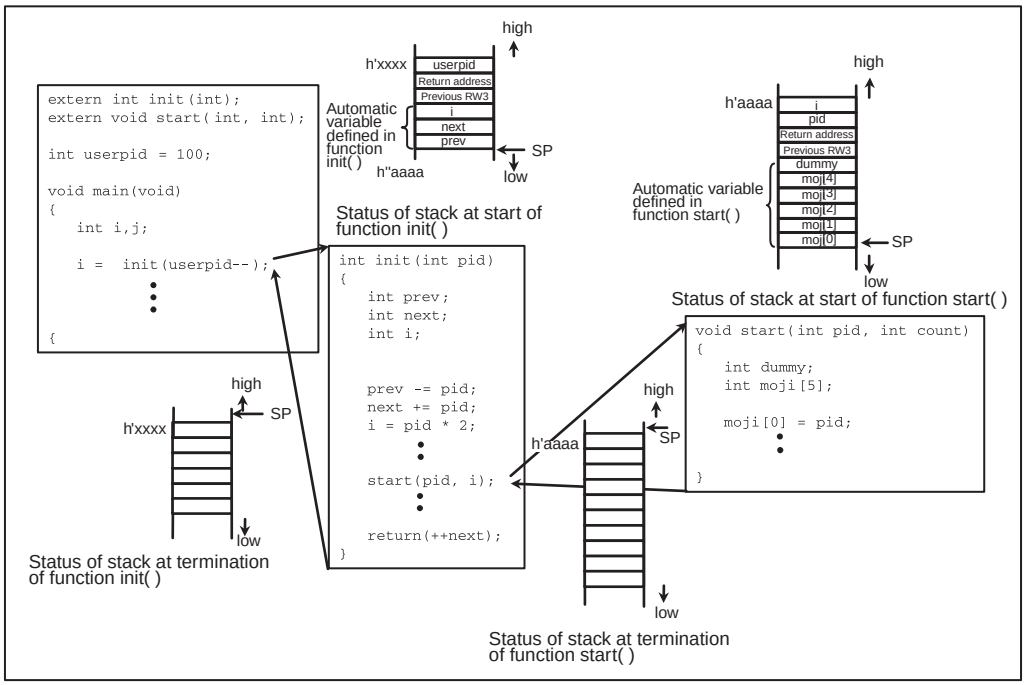
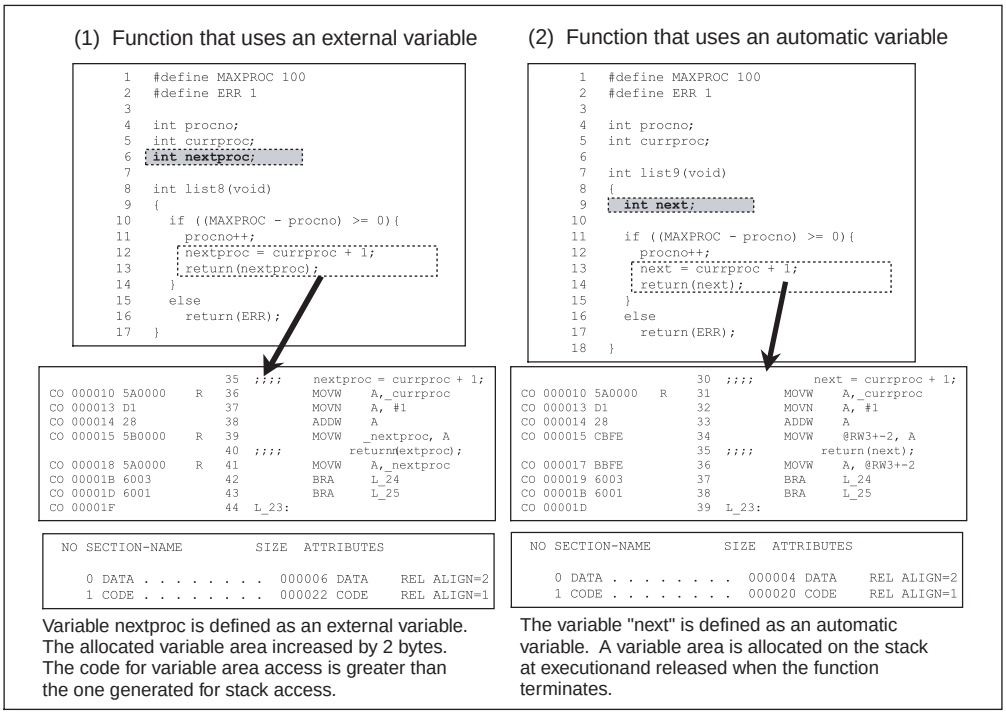


Figure 4.2-2 "Using External Variables and Automatic Variables" shows an example for defining a variable that is accessed only from within a function as an external variable and an example of

defining the variable as a automatic variable.

Figure 4.2-2 Using External Variables and Automatic Variables




```
(1) Function that uses an external variable

 1  #define MAXPROC 100
 2  #define ERR 1
 3
 4  int procno;
 5  int currproc;
 6  int nextproc;
 7
 8  int list8(void)
 9  {
10    if ((MAXPROC - procno) >= 0){
11      procno++;
12      nextproc = currproc + 1;
13      return(nextproc);
14    }
15    else
16      return(ERR);
17  }

                        35  ;;;;    nextproc = currproc + 1;
CO 000010 5A0000    R   36          MOVW    A,_currproc
CO 000013 D1            37          MOVW    A, #1
CO 000014 28            38          ADDW    A
CO 000015 5B0000    R   39          MOVW    _nextproc, A
                        40  ;;;;          return(nextproc);
CO 000018 5A0000    R   41          MOVW    A,_nextproc
CO 00001B 6003          42          BRA     L_24
CO 00001D 6001          43          BRA     L_25
CO 00001F               44  L_23:

 NO SECTION-NAME          SIZE  ATTRIBUTES

    0 DATA . . . . . .  000006 DATA    REL ALIGN=2
    1 CODE . . . . . .  000022 CODE    REL ALIGN=1

(2) Function that uses an automatic variable

 1  #define MAXPROC 100
 2  #define ERR 1
 3
 4  int procno;
 5  int currproc;
 6
 7  int list9(void)
 8  {
 9    int next;
10
11    if ((MAXPROC - procno) >= 0){
12      procno++;
13      next = currproc + 1;
14      return(next);
15    }
16    else
17      return(ERR);
18  }

                        30  ;;;;        next = currproc + 1;
CO 000010 5A0000    R   31          MOVW    A,_currproc
CO 000013 D1            32          MOVW    A, #1
CO 000014 28            33          ADDW    A
CO 000015 CBFE          34          MOVW    @RW3+-2, A
                        35  ;;;;        return(next);
CO 000017 BBFE          36          MOVW    A, @RW3+-2
CO 000019 6003          37          BRA     L_24
CO 00001B 6001          38          BRA     L_25
CO 00001D               39  L_23:

 NO SECTION-NAME          SIZE  ATTRIBUTES

    0 DATA . . . . . .  000004 DATA    REL ALIGN=2
    1 CODE . . . . . .  000020 CODE    REL ALIGN=1
```

See function list8( ) of (1), "Function using an external variable" in Figure 4.2-2 "Using External Variables and Automatic Variables". Because the variable nextproc is defined as an external variable for the function list8( ), the variable area allocated in RAM increased by 2 bytes. An external variable is accessed based on the variable address. Therefore, the resulting code is larger than the code for stack access.

See function list9( ) of (2), " Function that uses an automatic variable," in Figure 4.2-2 "Using External Variables and Automatic Variables". Function list9( ) defines the variable "next" as an automatic variable and allocates the variable area on the stack at function execution. The automatic variable allocated on the stack is accessed through the frame pointer (RW3). Therefore, the resulting code is smaller than the code for external variable access based on the address. In addition, the RAM area can be used more economically because the area is released when the function terminates.

In the example shown in Figure 4.2-2 "Using External Variables and Automatic Variables", the difference in the sizes of the data area is only 2 bytes for the external variable nextproc. The difference in the code generated for variable access is also 2 bytes. It can be expected that the size of the generated code will increase with the number of accesses to the external variable.

The amount of variable area that can be saved by reducing the number of external variables by one will only be a few bytes. However, it can be assumed that there are several dozens or several hundreds of modules. Therefore, reducing the number of wasteful external variables in each module can economize on the variable area.

In this way, defining variables that are accessed only within specific functions as external variables will result in wasteful use of the RAM and ROM areas. Therefore, by keeping the definitions of external variables to a minimum can economize on the variable area.

Similar to external variables, it is also important to keep the definitions of static variables to the minimum number required.

When designing the system, carefully investigate the scope of the variables to be defined to avoid meaningless definitions.

[Tip]

Softune C Analyzer:

The Softune C Analyzer lists the analysis results and the access status of external variables. This list can be used to determine from which function a defined external variable is accessed. Variables that are only accessed by a defined module can also be identified from these results.

The Softune C Analyzer checks for function calls that use large amounts of the stack in the program system based on the amount of stack use calculated by the fcc907. The Softune C Analyzer then visually displays the routes and amounts of usage. This information is useful for reducing the amount of stack usage.

CHAPTER 5 ACCESSING VARIABLES THAT USE BIT FIELDS

This chapter describes how to access variables that use bit fields. Using a bit field enables accessing each bit in a byte to be accessed.

5.1 Boundary Alignment of fcc907

This section briefly describes the boundary alignment of the fcc907.
For the fcc907 processing, variables are allocated to memory in accordance with the variable allocation size and boundary alignment.

■ Boundary Alignment of fcc907

Table 5.1-1 "Boundary Alignment of fcc907" lists the relationship between fcc907 variable types, allocation size, and boundary alignment.

In the fcc907 maps variables in memory based on allocation size and boundary alignment. When an odd number of char-type variables is defined, the subsequent 2- or 4-byte variable is mapped to an odd address. Unused areas are not generated. However, accessing a 2- or 4-byte variable that was mapped to an odd address may take longer than accessing a 2- or 4-byte variable that was mapped to even address. Care must be taken when variables of the type char are defined in array elements or members of a structure.

Table 5.1-1 Boundary Alignment of fcc907

Variable type		Allocation size (bytes)	Boundary alignment (bytes)
	char	1	1
signed	char	1	1
unsigned	char	1	1
	short	2	2
unsigned	short	2	2
	int	2	2
unsigned	int	2	2
	long	4	2
unsigned	long	4	2
	float	4	2
	double	8	2
long	double	8	2
near Pointer/address		2	2
for Pointer/address		4	2

5.2 Bit Field Definitions and Boundary Alignment

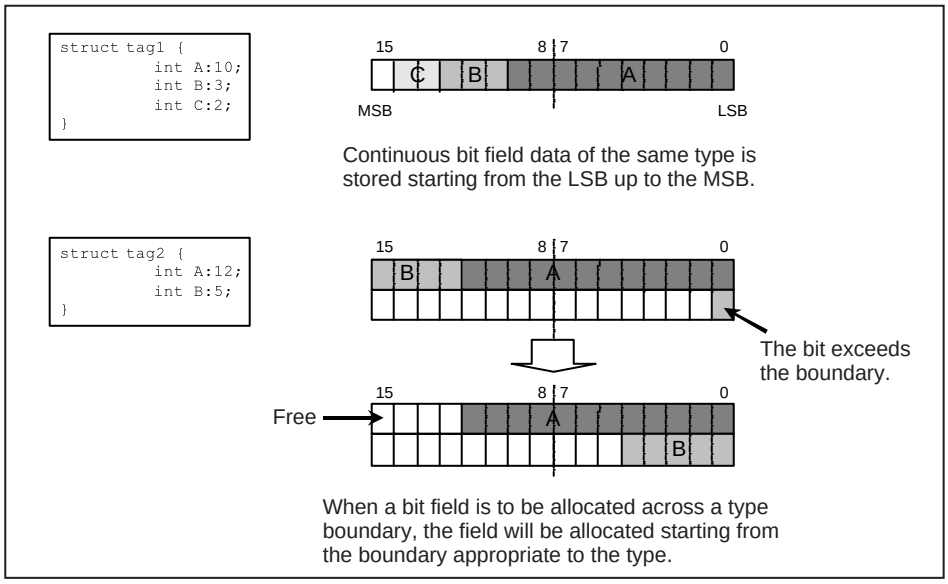
This section describes bit field definitions and boundary alignment for memory allocation.
Bit fields allow accessing each bit within a byte. However, depending on the boundary alignment conditions, it may not be possible to access some areas.

■ Bit Field Definitions and Boundary Alignment

Bit fields allow to access each bit within a byte.

Figure 5.2-1 "Bit Field Allocation 1 for the F^{2}MC-16 Family" shows the bit field assignment for the fcc907.

Figure 5.2-1 Bit Field Allocation 1 for the F^{2}MC-16 Family



As shown in Figure 5.2-1 "Bit Field Allocation 1 for the F^{2}MC-16 Family", the fcc907 allocates contiguous bit field data starting from the least significant bit (LSB) regardless of the type.

When a bit field is to be allocated over a type boundary, the field is allocated starting from a boundary that is appropriate for the type.

Figure 5.2-1 "Bit Field Allocation 1 for the F^{2}MC-16 Family" shows an example of bit field allocation with boundary alignment for structure tag2. In this example, int-type 12-bit bit field A is first allocated in memory. An attempt is then made to allocate int-type 5-bit bit field B. If one bit lies off the boundary, the boundary alignment operates so that B is mapped starting from a boundary appropriate to the type "int." In the process, an empty space of four bits is generated.

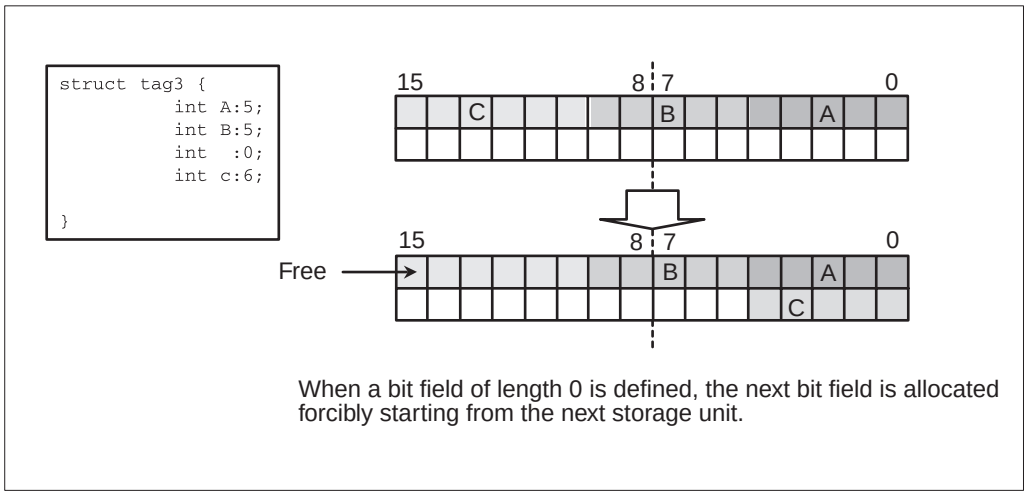
■ Bit Fields of Bit Field Length 0

When a bit field of length 0 is defined, the next field is forcibly allocated starting with the next storage unit.

Figure 5.2-2 "Bit Field Allocation 2 for the F^{2}MC-16 Family" shows an example of allocation of a

bit field of length 0. In this example, an int-type 5-bit bit field A is first allocated in memory. Next, a 5-bit int-type bit field B is allocated. Then, a 6-bit int-type bit field C is to be allocated. However, a bit field of length 0 has been defined before bit field C. As a result, the C area is allocated after empty space up to the next storage unit is forcibly allocated. Because the int-type boundary alignment is made in units of one byte, a 6-bit free area is generated.

Figure 5.2-2 Bit Field Allocation 2 for the F^{2}MC-16 Family

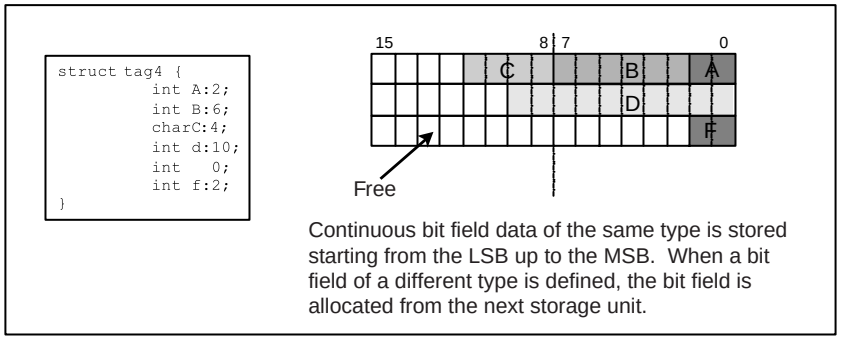


■ Definitions of Bit Fields of Different Types

Continuous bit fields of the same type are stored from the least significant bit (LSB) up to the most significant bit (MSB). When a bit field of a type that differs from that of the preceding bit field is defined, the new bit field is forcibly allocated starting with the next storage unit.

Figure 5.2-3 "Bit Field Allocation 3 for the F^{2}MC-16 Family" shows an example of allocation when different type bit fields are defined. In this example, int-type 2-bit bit field A and then an int-type 6-bit bit field are allocated in memory before a char-type 4-bit bit field is defined. Even though the types are different, no free areas are generated because the bit fields are allocated precisely on the boundaries. Int-type 10-bit bit field D is then defined. Because the type is different, the D area is allocated after free empty space up to the next storage unit is allocated. Finally, because a bit field of length 0 has been defined, int-type bit field F is allocated starting from the next storage unit.

Figure 5.2-3 Bit Field Allocation 3 for the F^{2}MC-16 Family



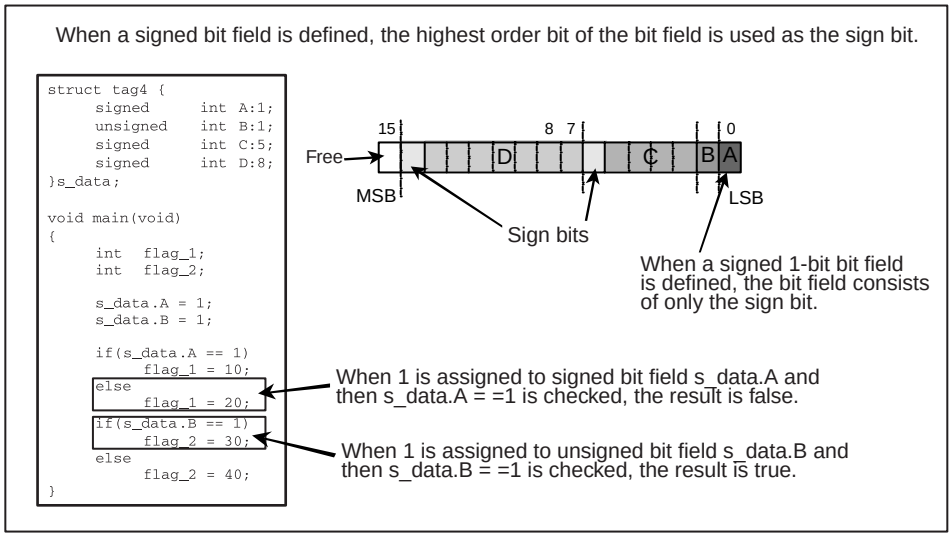
■ Signed Bit Fields

When a signed bit field is defined, the highest order bit of the bit field is used as the sign bit.

When a signed 1-bit bit field is defined, the bit field consists of only the sign bit.

Figure 5.2-4 "Definitions of Signed Bit Fields" shows an example of a definition of signed bit fields. In this example, 1-bit bit field A is defined as a signed bit field. If s_data.A=1 is assigned before checking for s_data.A = =1, the obtained result will be false.

Figure 5.2-4 Definitions of Signed Bit Fields



[Tip]

Softune C Checker:

The Softune C Checker outputs a warning message for structure variables or union variables in which a free field occurs. If a warning message is output, check the definitions of the structures and unions again.

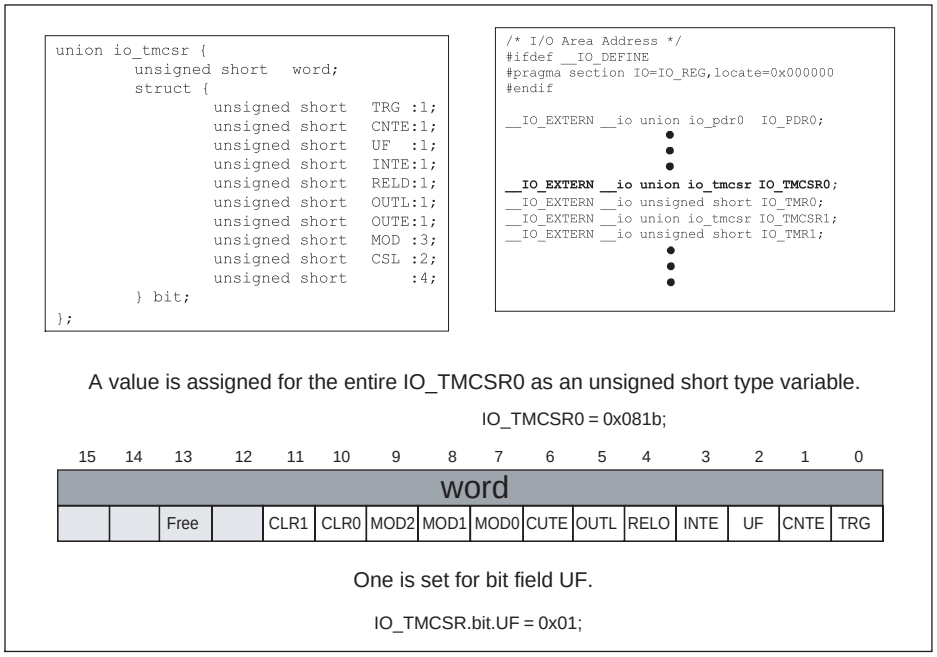
5.3 Accessing I/O Areas Using Bit Fields and Unions

This section describes how to access bit fields in bit units and entire bit fields of unions. This method is not directly related to using less RAM area, but it can facilitate access to registers mapped into the I/O area.

■ Accessing I/O Areas Using Bit Fields and Unions

If a structure is defined as a bit field, each field can be accessed or assigned individually, but the entire structure cannot be accessed as such. Moreover, data cannot be assigned to the entire structure in a batch operation. Defining the structure as a union as shown in Figure 5.3-1 "Accessing the I/O Area with Bit Fields and Unions" enables to access both the values of individual bits or the entire structure. In this example, bit field structures and variables of the type "unsigned short" are defined as unions. Therefore, data can be accessed either bit units or as variables of the type "unsigned short."

Figure 5.3-1 Accessing the I/O Area with Bit Fields and Unions



The values of the hardware registers that are allocated to the input-output areas of the F^{2}MC-16 family can be referenced in bit units or collectively. When a union is defined for such hardware registers, a value can be assigned in the manner shown below.

```
IO_TMCSRO.word = 0x081b;
```

A value can also be directly assigned to a bit field as shown below.

```
IO_TMCSRO.bit.UF = 0x01;
```

This approach facilitates access to registers mapped into the I/O area.

PART II USING STACK AREA EFFICIENTLY

Part II describes how to use stack areas efficiently in C programs.
Part II first briefly describes the states of the stack areas at a function call. It then describes how to use the stack areas efficiently.

CHAPTER 6 FUNCTION CALLS AND THE STACK

Before describing how to use the stack area effectively, this chapter describes the areas that are allocated on the stack when a function is called.
When a function is called, areas, such as the areas for arguments, are allocated on the stack as necessary.

6.1 Areas Allocated on the Stack during Function Calls

When a C program calls a function, a return address storage area and a previous frame pointer (RW3) save area are always allocated on the stack.

■ Areas Allocated on the Stack at Function Call

When a C program calls a function, the following areas are allocated on the stack as shown in Figure 6.1-1 "Areas Allocated on the Stack when a Function is Called":

❍ **Actual argument and dummy argument areas**

Used to hand over arguments during function calls.

- Actual argument: Argument specified by the calling function
- Dummy argument: Argument accessed by the called function

❍ **Return address save area**

Used to store the address for returning to the calling function.

This area is acquired or released by the calling function.

❍ **Previous frame pointer save area**

Used to save the value of the frame pointer (RW3 register) of the source calling the function.

❍ **Local variable area**

Used to store local variables or work variables.

This area is allocated at function entry, and released at function exit.

The size of this area depends on the number of the local variables to be stored. The greater the number of variables defined in the function, the larger the area allocated.

❍ **Register save area**

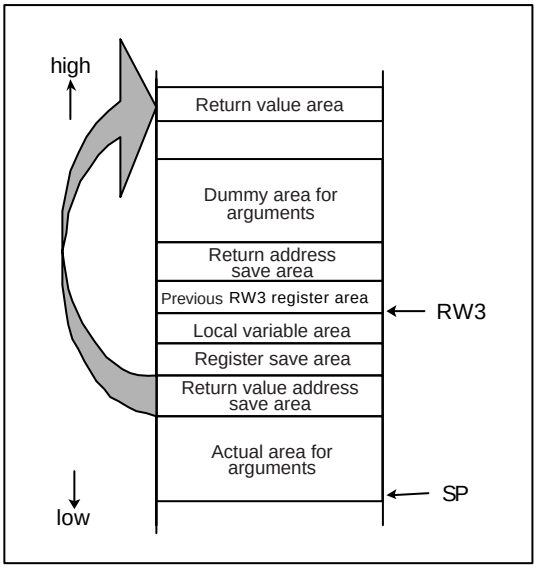
This area is used to save registers that must be preserved for the calling source.

This area is not allocated when no registers need to be saved.

❍ **Return address value save area**

This area is used to save the leading address of the area used to store the return value of functions that return double type, long double type, structure type, or union type value.

Figure 6.1-1 Areas Allocated on the Stack When a Function Is Called



Out of the areas shown in Figure 6.1-1 "Areas Allocated on the Stack When a Function Is Called", the return address storage area and old frame pointer save area are always allocated at function call. Other areas are allocated depending on the defined function. The greater the number of arguments to be passed to the function and number of local variables to be defined in the function, the larger the areas allocated on the stack.

6.2 Stack States When Function Calls Are Nested

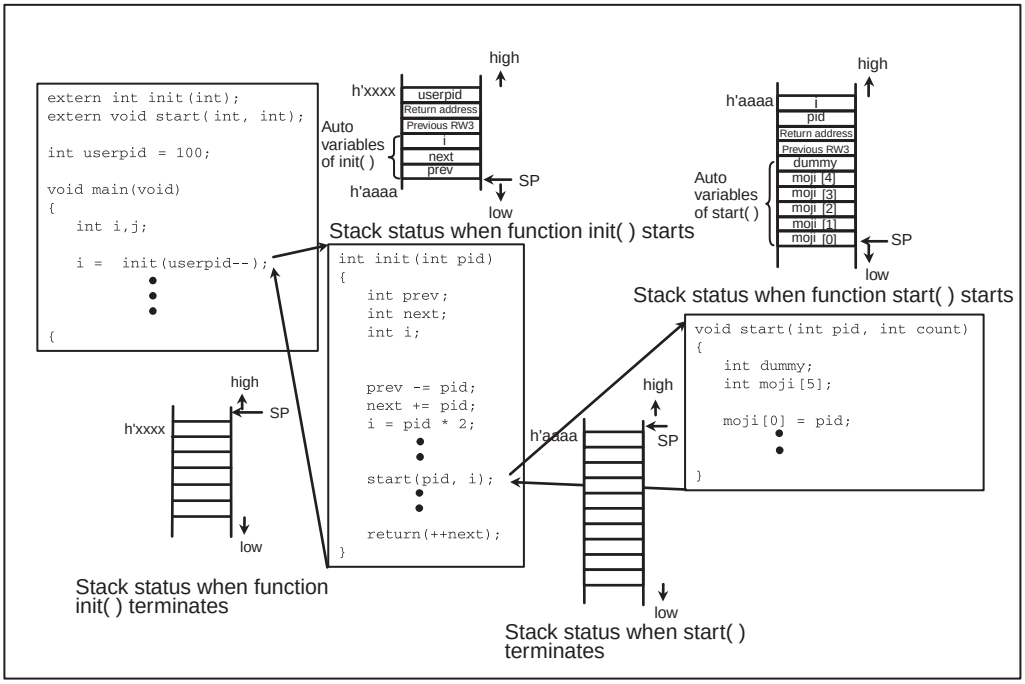
The areas allocated on the stack for a function are released when the function terminates. The deeper the nesting of function calls nesting, the greater is the amount of stack used.

■ Stack States When Function Calls Are Nested

Figure 6.2-1 "Nesting of Function Calls" shows the stack states for nested function calls. The areas allocated on the stack are released when the function terminates. However, releasing stack areas is not sufficient to guarantee that the stack is used efficiently. If function calls are deeply nested, new areas will be allocated above the previously allocated areas. As a result, the used stack areas will increase by that amount.

The best method for reducing used stack space is to avoid function calls. However, this is impractical because this would mean that one program system would have to consist of a single function only. Of the areas described above, the return address and old frame pointer areas are always allocated when a function is called. The other areas depend on the called function. Therefore, stack use can be minimized if both the number of function calls and the areas allocated on the stack when a function is called are reduced.

Figure 6.2-1 Nesting of Function Calls



CHAPTER 7 REDUCING FUNCTION CALLS BY EXPANDING FUNCTIONS IN LINE

This chapter describes how to use inline expansion of functions to reduce function calls. Expanding functions in line reduces the amount of stack area required.

7.1 Inline Expansion of Function

This section gives a simple description of the inline expansion of functions. When a specified function is called, the function body is directly expanded in line.

■ Inline Expansion of Function

The fcc907 uses the following format to specify the inline expansion of functions:

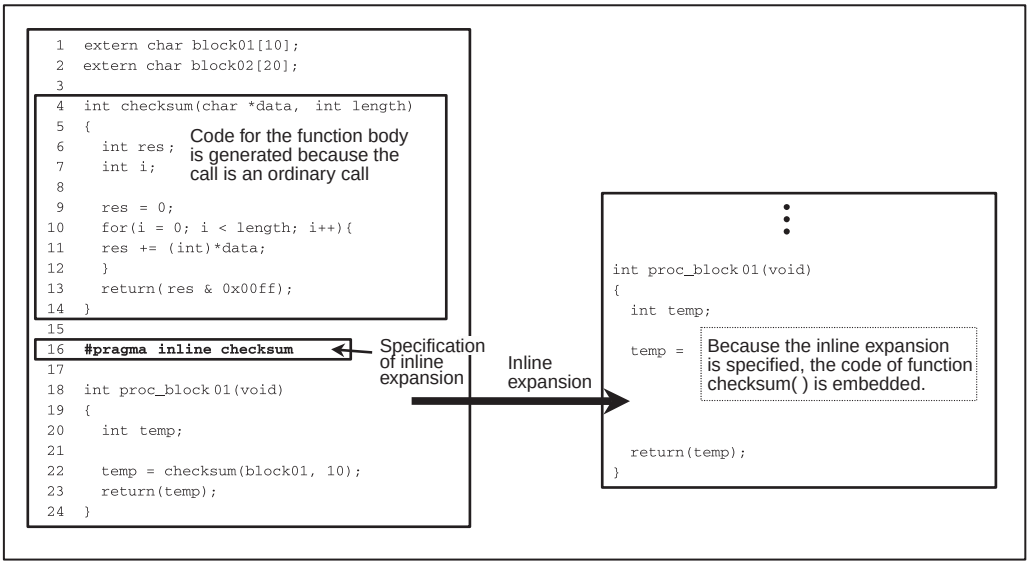
```
#pragma  inline   name-of-function-to-be-inline-expanded
```

The function to be inline-expanded can also be specified using the -x option when starting the compiler as follows:

```
-X   name-of-function-to-be-inline-expanded
```

Figure 7.1-1 "Inline Expansion of a Function" shows an example of inline expansion of a function. The inline expansion is specified with "#pragma inline function-name." When the specified function is called, it is expanded inline.

Figure 7.1-1 Inline Expansion of a Function

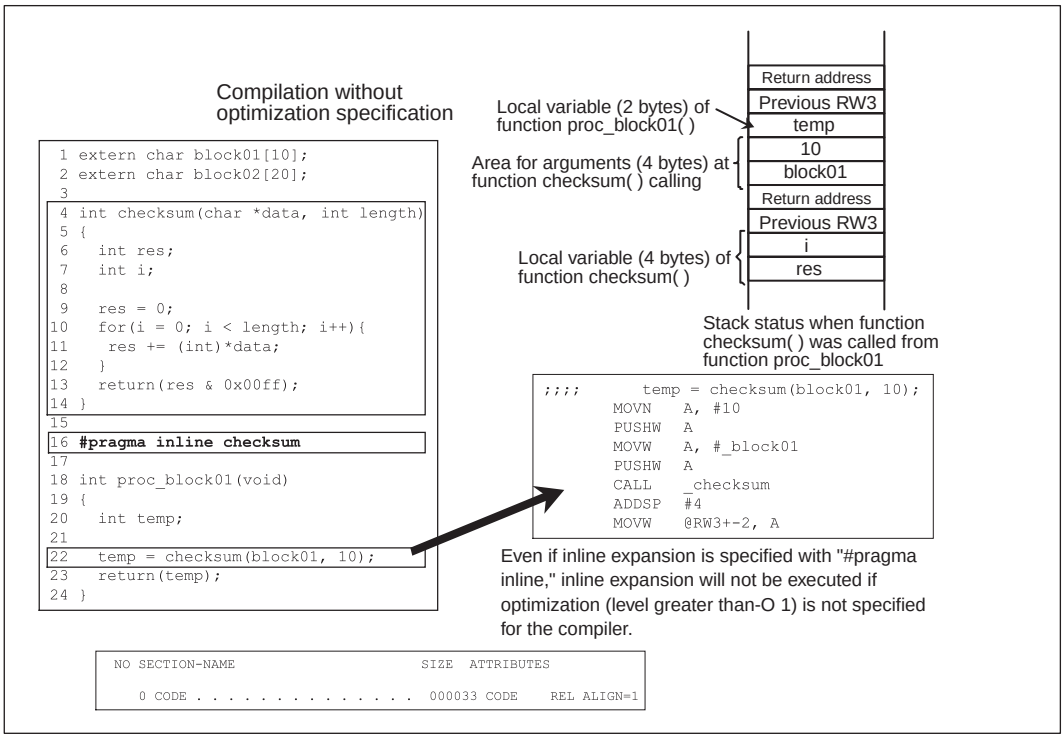


■ When Inline Expansion Is Not Executed Even Though #pragma Inline Is Specified

Figure 7.1-2 "Example in Which Inline Expansion Is Not Executed" shows an example of when inline expansion is not executed even though #pragma inline is specified.

In this example, inline expansion of function checksum( ) is specified on line 16. However, because optimization using the -O option (level greater than-O 1) has not been specified for the compiler, the usual function checksum( ) on line 22 is called.

Figure 7.1-2 Example In Which Inline Expansion Is Not Executed



<Notes>

To have the fcc907 execute inline expansion of a function, always specify optimization using the -O option in addition to specifying inline expansion.

Even though inline expansion is specified using #pragma inline, inline expansion will not be executed if optimization (level greater than-O 1) is not specified for compilation.

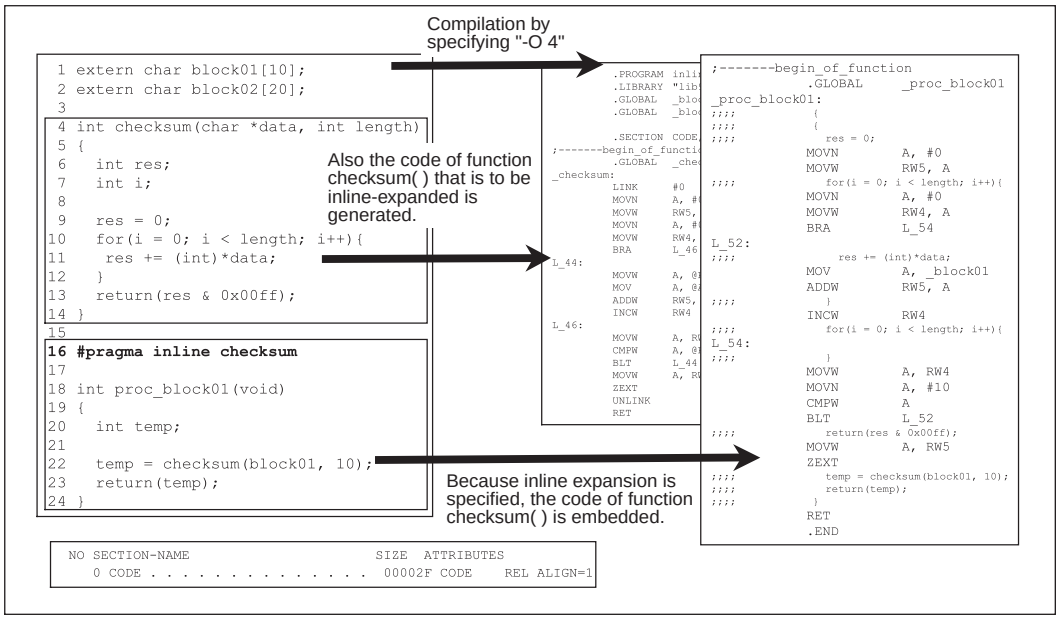
Specifying only the -O option will default to optimization level 2 (-O 2)

■ Executing Inline Expansion Using the #pragma inline Specification

Figure 7.1-3 "Inline Expansion" shows an example in which #pragma inline expansion is specified and optimization using the -O option is specified for compilation.

In this example, the inline expansion of function checksum( ) is specified on line 16. Because optimization using the (-O 4) option is specified for compilation, the function checksum( ) on line 22 is inline-expanded. Because there may be a normal function call to the function checksum( ), the code of the entire function is also generated. Specifying the inline expansion of a function reduces the size of stack used compared with using a function call. Because the code of function checksum( ) is embedded in the function proc_block01( ), faster processing can be expected. Because the code of function checksum( ) is inserted into line 22, code larger than that for the ordinary function call is generated.

Figure 7.1-3 Inline Expansion



7.2 Conditions for Inline Expansion of Function

This section explains the conditions for inline expansion of a function.
Only the functions that were defined in the same file can be inline-expanded.

■ Conditions for Inline Expansion of Function

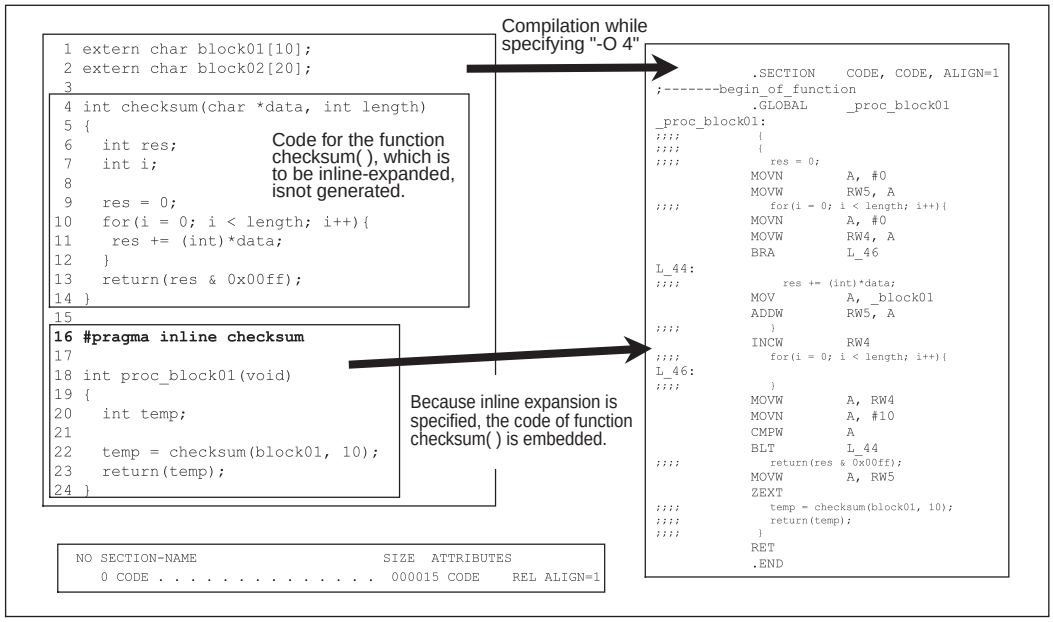
When a function is inline-expanded, the code of the function is directly inserted into the line of the function call. Therefore, inline expansion can be executed only for functions defined in the same file.

The fcc907 does not generate code if a function declared as "static" is specified for #pragma inline and optimization (level greater than-O 1) is specified.

Figure 7.2-1 "Inline Expansion of Function Declared as "static"" shows an example in which a function declared as "static" is specified for #pragma inline and optimization using the (-O 4) option is specified.

In this example, inline expansion is specified on line 16. Because the function checksum( ) is declared as "static", the function is not referenced from other modules. Therefore, because code for function checksum( ) will not be generated, the size of the code will be smaller. However, if inline expansion is frequently executed, code larger than that for function checksum( ) can be generated.

Figure 7.2-1 Inline Expansion of Function Declared as "static"



<Notes>

In the following cases, inline expansion is not executed even if specified:

- Optimization with the "-O option was not specified for compilation.
- Inline expansion was specified for a recursively called function.
- Inline expansion was specified for a function for which a structure or union was specified as argument.
- Inline expansion was specified for a file in which the setjmp function is called.
- Inline expansion was specified in a file containing the _ _asm statement.
- Arguments between functions do not match.

[Tip]

For the fcc907:

The number of lines of a function to be inline-expanded can be specified with the following size option for compilation.

-xauto size-option

When this option is specified, the functions that are specified with the size option are inline-expanded in compilation units. When the size option is not specified, functions consisting of thirty lines or less are inline-expanded. Also in this case, the optimization (-O 1 or more) must be specified with the "-O" option.

-K ADDSP-option

Specifying the ADDSP option can reduce the overhead for function call processing and generate high-speed objects that are smaller than usual. However, this option will collectively release the actual argument areas accumulated on the stack for function calls. If this option is not specified, the amount of stack used will increase.

Softune C Analyzer:

The upper limit of the number of lines of a function to be inline-expanded can be specified. When analysis is executed with this option specified, the Softune C Analyzer will list the functions that are candidates for inline expansion after the analysis is completed. This function is helpful in determining the functions that will be expanded in line.

CHAPTER 8 REDUCING ARGUMENTS TO CONSERVE STACK AREA

This chapter describes how to use fewer arguments in function calls as means of reducing the amount of stack area used.
The best way to conserve the stack is to avoid all function calls, but this is not practical. CHAPTER 7 "REDUCING FUNCTION CALLS BY EXPANDING FUNCTIONS IN LINE" already explained described how to use inline expansion to conserve stack area. However, depending on the function size and processing conditions, it may not be possible to conserve stack area by inline expansion. This chapter describes a second method for stack conservation: Conserving stack area by reducing the argument count.

8.1 Passing Arguments During Function Calls

**This section describes how to pass arguments during function calls.
When a function is called, the fcc907 stacks these arguments and passes them to the called function. Reducing the number of arguments for function calls conserves stack area. The following section describes how argumetns are passed at the example of a variable that is defined as a structure.**

■ Argument Passing and Stack Usage Size

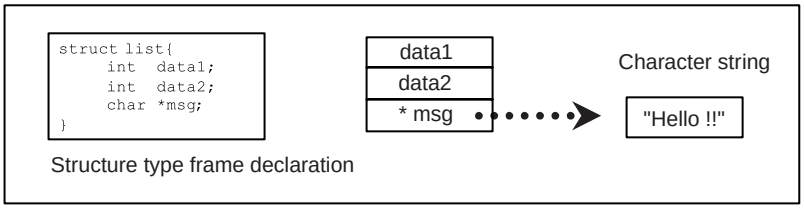
When a function is called, the fcc907 stacks these arguments and passes them to the called function. The greater the number of arguments, the larger the stack area used. Reducing the number of arguments for function calls conserves stack area.

The following three methods for passing arguments are explained for variables defined as a structure:

- Normal Argument Passing
- Argument Structure Passing
- Address Passing of Structures

Figure 8.1-1 "Variable that is Defined as a Structure" shows an example for a variable that is defined as a structure.

Figure 8.1-1 Variable That Is Defined as a Structure



8.1.1 Normal Argument Passing

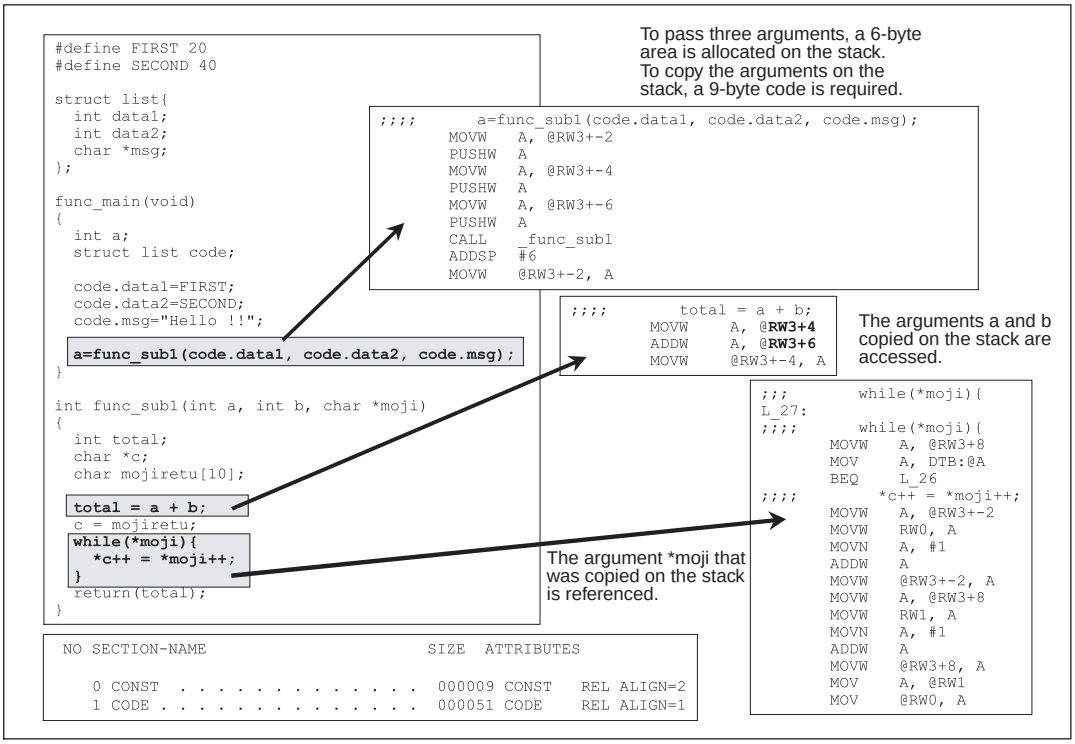
During normal argument passing, arguments are stored on the stack sequentially before calling the function. Therefore, the greater the number of arguments, the larger the stack area used.

■ Normal Argument Passing

Figure 8.1-2 "Normal Argument Passing" shows an example for normal argument passing. In this example, a 6-byte area for saving three arguments is allocated on the stack. To copy these arguments on the stack, a 9-byte code is required.

A 28-byte stack area is required for processing from calling function func_sub1( ) to its execution. (See (1), "Normal argument passing," in Figure 8.1-5 "Stack Usage Size Depending on Argument Type during Function Calls".)

Figure 8.1-2 Normal Argument Passing



8.1.2 Argument Structure Passing

Argument structure passing can be performed with very simple C code. However, in this method of argument passing, all structure elements are copied on the stack and then passed to the function. Therefore, the larger the number of elements of the structure to be passed, the larger the stack area used.

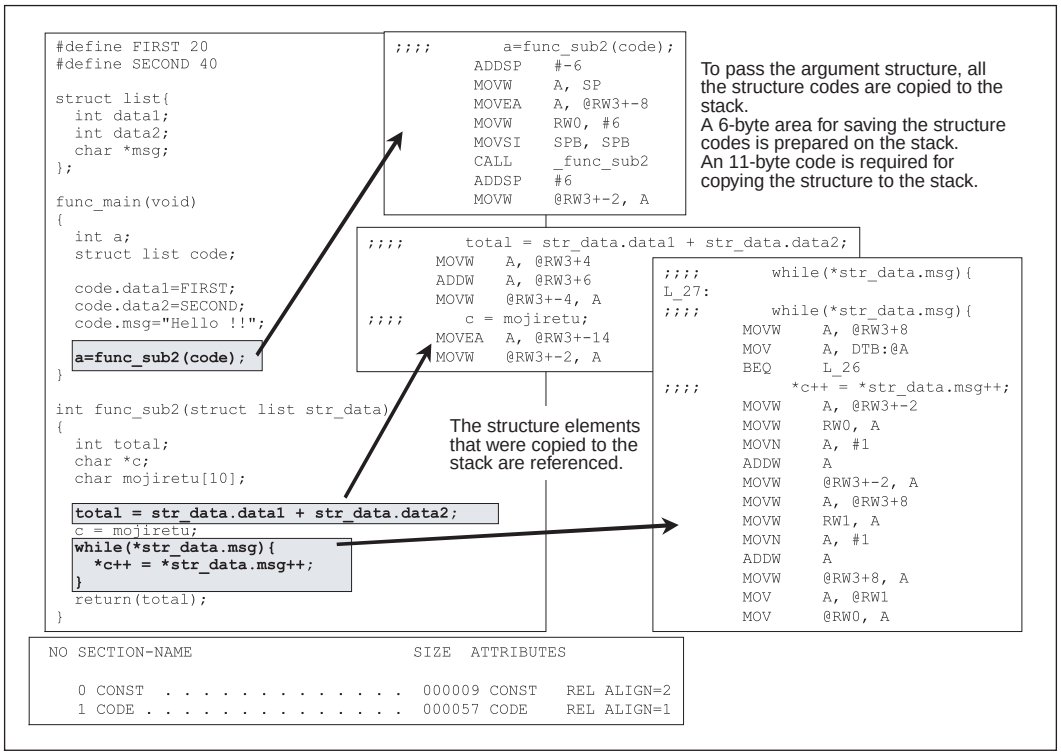
■ Argument Structure Passing

Figure 8.1-3 "Argument Structure Passing" shows an example of argument structure passing. In this example, a 6-byte area for arguments is allocated on the stack in the same way as explained in Section 8.1.1 "Normal Argument Passing". In addition, an 11-byte code is required for copying the structure to the stack.

A 28-byte stack area is required for processing from calling function func_sub2( ) to its execution. (See (2), "Argument structure passing," in Figure 8.1-5 "Stack Usage Size Depending on Argument Type during Function Calls".)

It is very easy when coding in C to specify a structure as an argument, but this method is not very efficient in terms of the generated code and the required stack size.

Figure 8.1-3 Argument Structure Passing



8.1.3 Structure Address Passing

In structure address passing, only the structure address is stored on the stack before calling the function.

■ Structure Address Passing

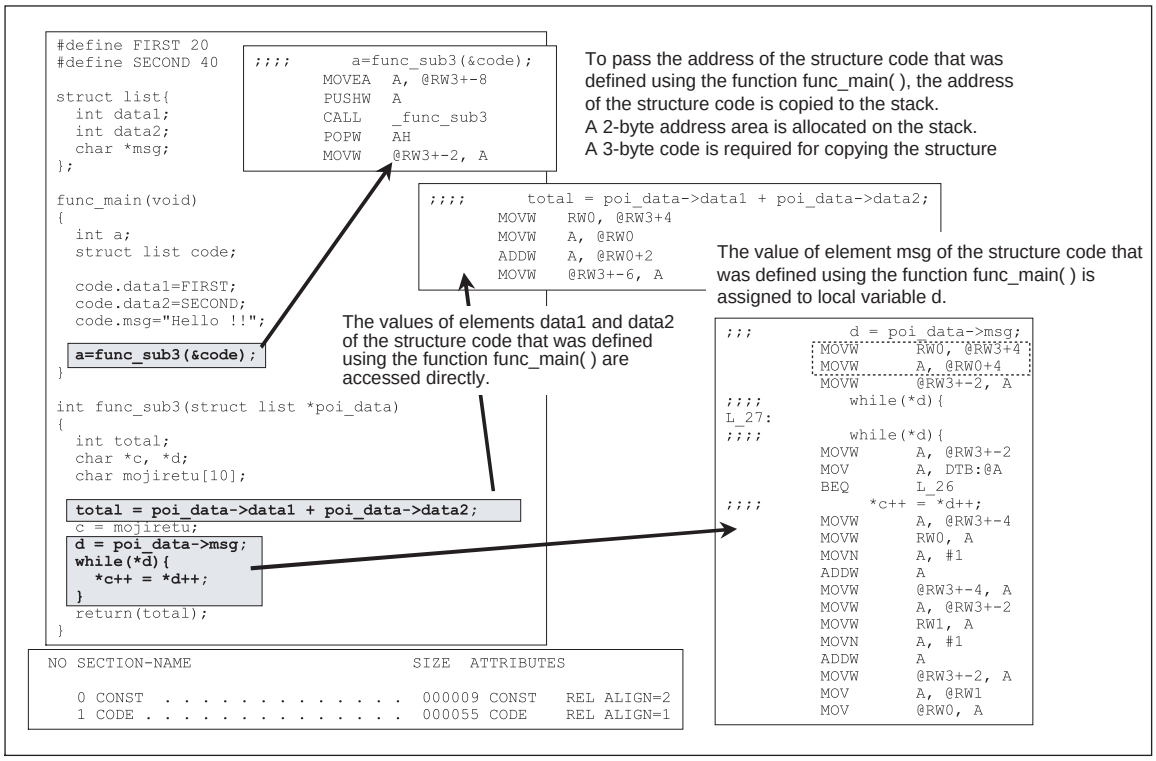
Figure 8.1-4 "Structure Address Passing" shows an example of structure address passing.

In this example, a 2-byte area for arguments is allocated on the stack. The code for copying the arguments consists of four bytes, which is much smaller than the normal argument passing and argument structure passing.

A 26-byte stack area is required for processing from calling function func_sub3( ) to its execution. (See (3), "Structure address passing," in Figure 8.1-5 "Stack Usage Size Depending on Argument Type during Function Calls".)

Specifying a structure address as an argument is the most efficient method in terms of conserving the area for arguments to be used.

Figure 8.1-4 Structure Address Passing



8.1.4 Stack Status During Function Calls

This section describes the status of the stack for function calls as explained in Sections 8.1.1 "Normal Argument Transfer", 8.1.2 "Argument Structure Passing", and 8.1.3 "Structure Address Passing".

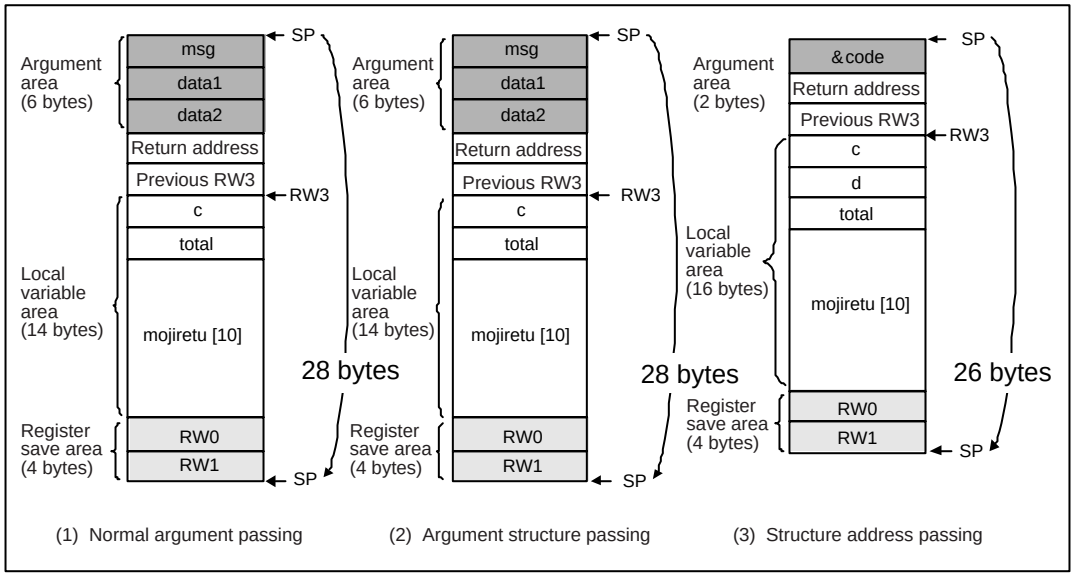
■ Stack Status at Function Call

Figure 8.1-5 "Stack Usage Size Depending on Argument Type during Function Calls" shows the status of the stack used for function calls explained in Sections 8.1.1 "Normal Argument Transfer", 8.1.2 "Argument Structure Passing", and 8.1.3 "Structure Address Passing". This figure shows the relationship between reducing the arguments and conserving the stack area when calling a function.

In these examples, the stack size used does not differ much because only three arguments were passed. However, when, for example, ten 4-byte arguments are to be passed, the stack sizes may differ considerably.

Therefore, when many arguments are to be passed during a function call, the most efficient method is to use a structure argument and to pass only its address.

Figure 8.1-5 Stack Usage Size Depending on Argument Type during Function Calls



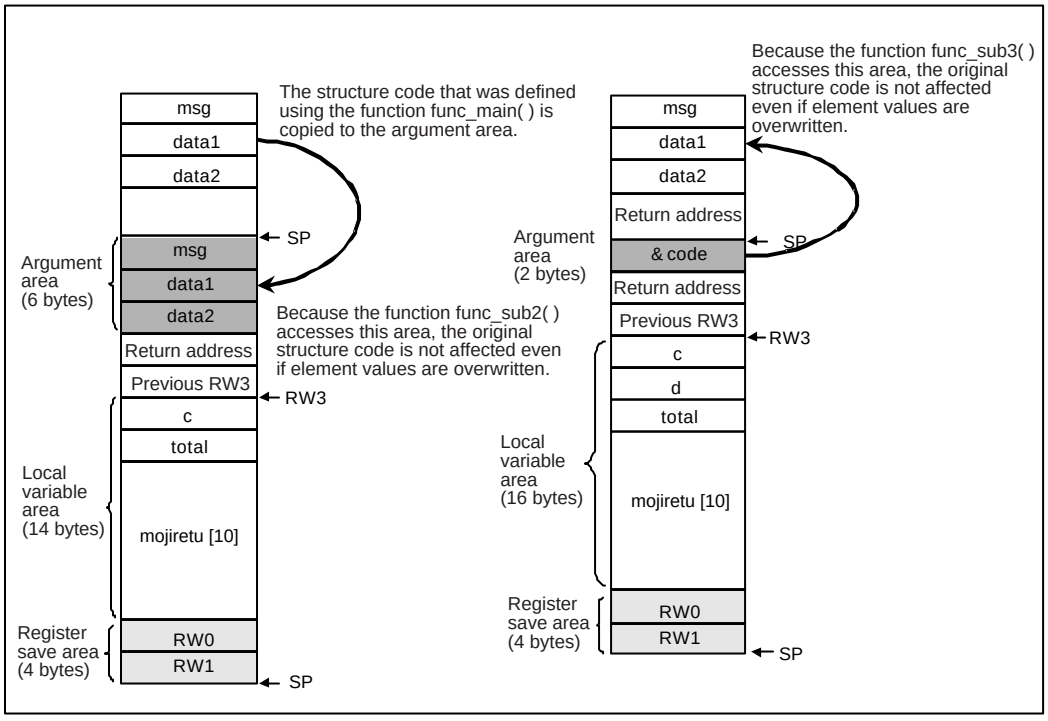
8.2 Conditions for Structure Address Transfer

This section describes the conditions that must be satisfied to pass a structure address as a function argument.

■ Conditions for Passing Structure Addresses

As explained in Section 8.1 "Passing Arguments During Function Calls", when a large number of arguments is to be passed, it is most efficient in terms of stack use to define the arguments in a structure and to pass only the address of that structure.. However, the following conditions must be satisfied to pass the address of such a structure.

Figure 8.2-1 Structure Passing and Structure Address Passing



In argument structure passing (see Section 8.1.2 "Argument Structure Passing"), each element of the structure is copied to the stack and then passed to the respective function. Therefore, even if the value in an element of the structure is changed, the value of the structure in the calling source does not change. However, in the structure address transfer (see Section 8.1.3 "Structure Address Passing"), the structure is directly accessed for processing. Therefore, if the value of an element of the structure is changed, the structure value that was held before the function call will be lost. In the example of structure address passing in Section 8.1.3 "Structure Address Passing", loss of the information for structure code element msg was avoided by adding the local variable d was added to the function func_sub3( ) so that the value could be assigned to the variable d before being used.

When the value in the calling source must be kept unchanged during structure address passing, the receiving function must operate in the way described above.. In the example of structure address passing explained in Section 8.1.3 "Structure Address Passing", the stacking efficiency is highest even though this type of processing is performed.

[Tip]

Softune C Checker:

The Softune C Checker will output a warning if some arguments were not referenced at all by the called function. Also, if a structure or union was specified in an argument, a warning message is output to the effect that performance may be reduced. Examine the method of argument passing considering the contents of these warning messages.

CHAPTER 9 CONSERVING STACK AREA BY IMPROVEMENTS ON THE AREA FOR FUNCTION RETURN VALUES

This chapter describes how to conserve stack area by improvements on the function return value area.
As already described, the number of function calls and the number of arguments required for function calls can be reduced by using inline expansion. The size of stack used can also be reduced by improvements with respect to the return values of a function. This chapter describes this third method of stack conservation, reducing the size of the function return value area.

9.1 Return Value of Functions

This section describes the return values of functions.
The type of a function is the type of the value returned when the function terminates.
The type of this return value determines whether the return value is to be returned to the register or stack.

■ Return Value of Functions

The return value of functions have the same type as ordinary variables. When defining a function, the type of the return value for the function must be specified. Table 9.1-1 "Function Return Values and Return Value Interface" lists the relationship between function return values and the interface for return values.

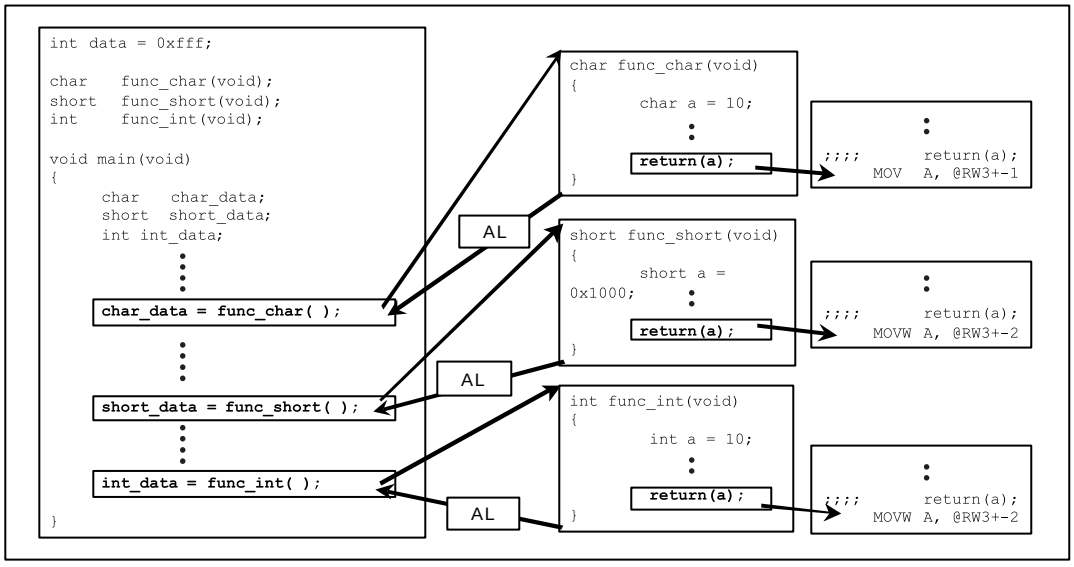
Table 9.1-1 Function Return Values and Return Value Interface

Type of return value		Allocated size (bytes)	Return value interface
	void	---	---
	char	1	AL
signed	char	1	AL
unsigned	char	1	AL
	short	2	AL
unsigned	short	2	AL
	int	2	AL
unsigned	int	2	AL
	long	4	A
unsigned	long	4	A
	float	4	A
	double	8	On stack
long	double	8	On stack
near Pointer/address		2	AL
for Pointer/address		2	A

■ Function Return Values Returned via the AL Register

The fcc907 places return values of up to two bytes into the AL register and then returns these values to the calling function. When the value to be returned by a function is of the type "char" (1 byte), "short" (2 bytes), or "int" (2 bytes), the value is stored in the AL register of the F^{2}MC-16 family as shown in Figure 9.1-1 "Returning Function Return Values Using the AL Register" and then returned to the function caller. Therefore, when such a function is to be called, the return address save area or return value area shown in Figure 6.1-1 "Areas Allocated on the Stack When a Function Is Called" is not required.

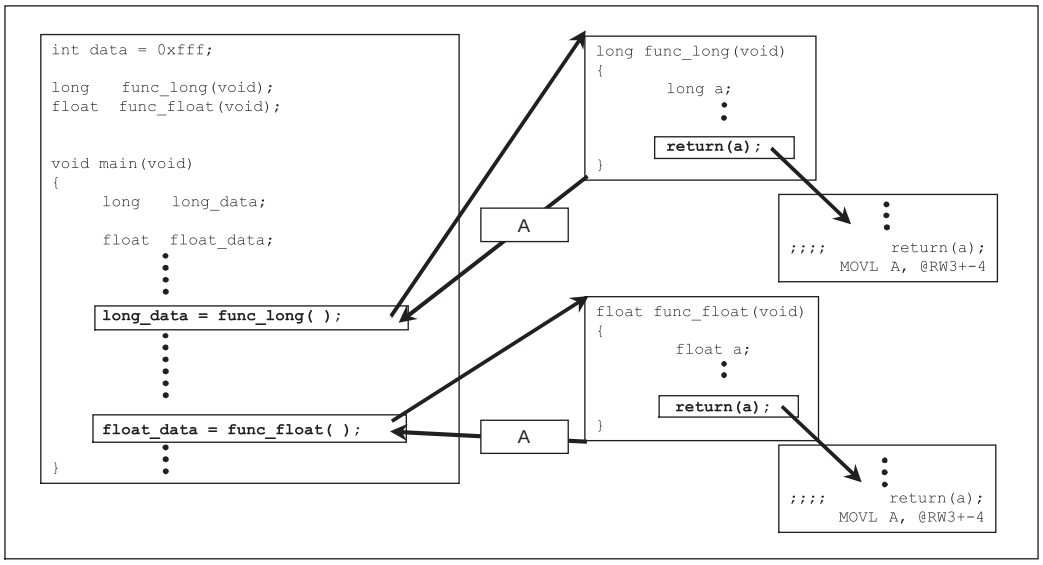
Figure 9.1-1 Returning Function Return Values Using the AL Register



■ Function Return Values Returned via the A Register

The fcc907 places 4-byte return values into the A register and then returns these values to the calling function. When the value to be returned by a function is a "long" (4 bytes) or "float" (4 bytes) type, the value is stored in the A register of the F^{2}MC-16 Family as shown in Figure 9.1-2 "Returning Function Return Values Using the A Register" and then returned to the function caller. Therefore, when such a function is to be called, the return value address save area or return value area shown in Figure 6.1-1 "Areas Allocated on the Stack When a Function Is Called" is not required.

Figure 9.1-2 Returning Function Return Values Using the A Register

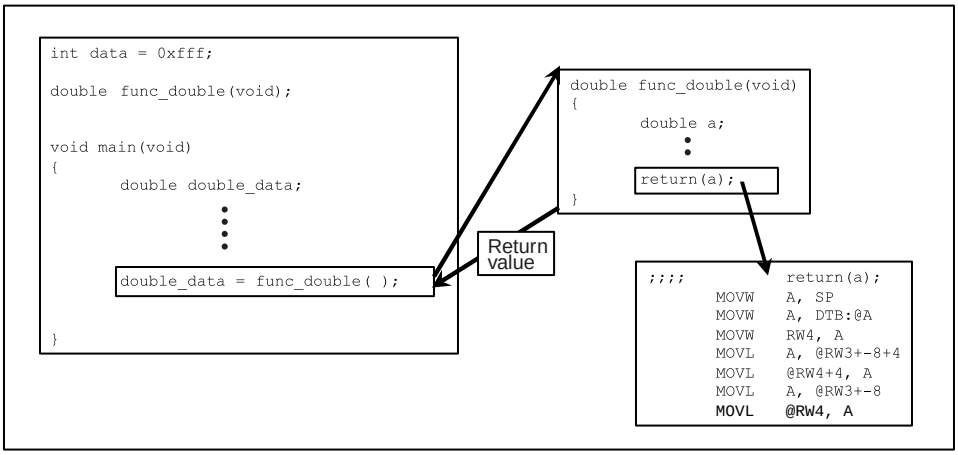


■ Returning Function Return Value via the Stack

When a function does not place a return value into the AL register (2 bytes) or A register (4 bytes), the return value is returned via the stack. In this case, the return address save area and return value area shown in Figure 6.1-1 "Areas Allocated on the Stack when a Function is Called" are allocated.

Some functions return values of other types such as double (8 bytes). Such functions return values via stack areas as shown in Figure 9.1-3 "Returning Function Return Values via Stack Areas".

Figure 9.1-3 Returning Function Return Values via Stack Areas



■ Functions Returning Pointer-Type Values

Some functions have "pointer" type return values. The size of the pointer handled by the fcc907 depends on the memory model specified at compilation and the _ _near-type or _ _far-type qualifier specification.

Figure 9.1-4 Functions Returning a Return Value of the Type "pointer"

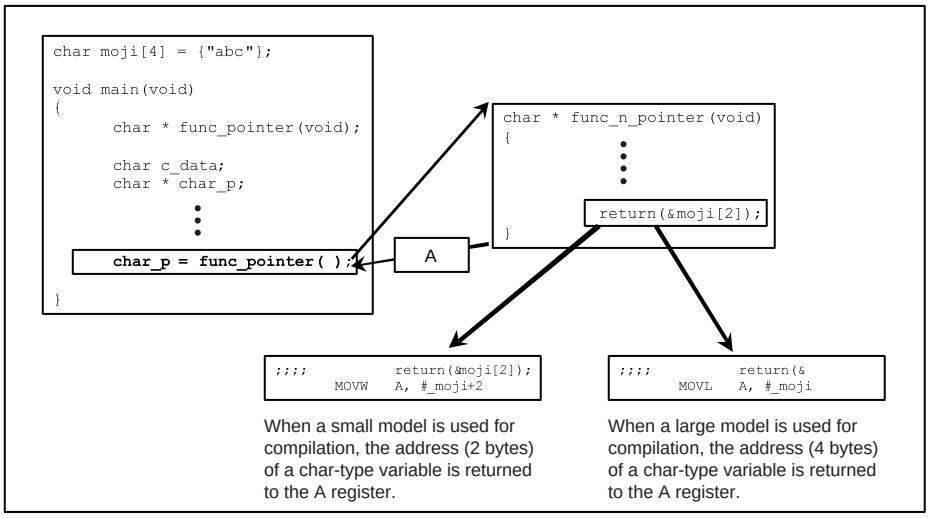


Table 9.1-2 Memory Models and Addressing at Access

	Small model	Medium model	Compact model	Large model
Function access	16-bit addressing	24-bit addressing	16-bit addressing	24-bit addressing
Variable access		16-bit addressing	24-bit addressing	

Table 9.1-2 "Memory Models and Addressing at Access" shows the relationship between memory models and addressing.

When a small model is used for compilation or when the pointer has been clearly qualified using the _ _near type, the size of the pointer will be two bytes. As a result, the pointer will be returned to the AL register. When a large model is used for compilation or when the pointer has been clearly qualified using the _ _far type, the pointer will be four bytes. As a result, the pointer will be returned to the A register. For a medium model, the pointer will be returned to the A register (4 bytes) when a function address is returned or to the AL register (2 bytes) when a variable address is returned. For a compact model, the pointer will be returned to the AL register (2 bytes) when a function address is returned or to the A register (4 bytes) when a variable address is returned. These functions can be summarized as follows:

❍ **Pointer returned to the AL register (2 bytes)**

_ _near-type qualified pointer

Variable/function access when a small model is used for compilation

Variable access when a medium model is used for compilation

Function access when a compact model is used for compilation

❍ **Pointer returned to the A register (4 bytes)**

_ _far-type qualified pointer

Variable/function access when a large model is used for compilation

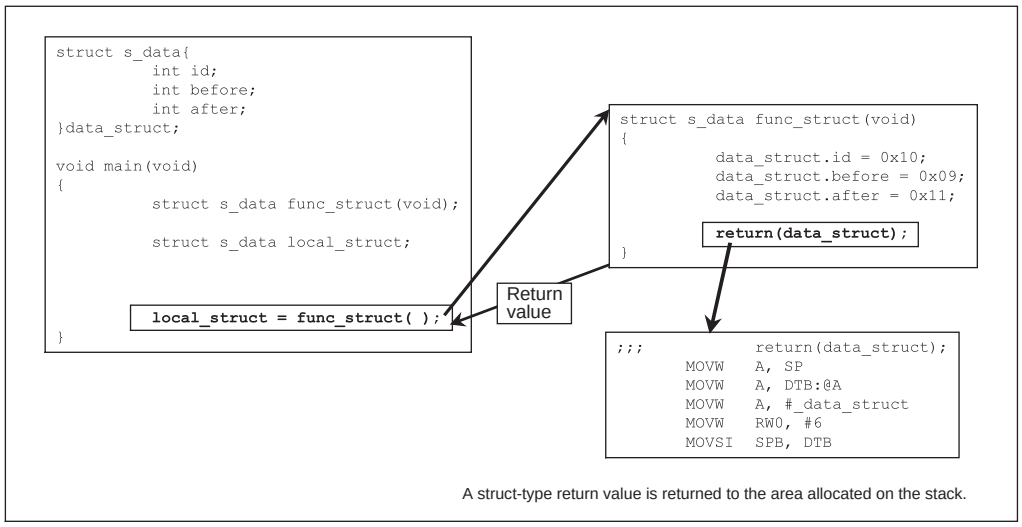
Function access when a medium model is used for compilation

Variable access when a compact model is used for compilation

■ Functions Returning Structure-Type Values

Some functions have return values of the type "structure." The size of the structure to be returned depends on the members defined in the structure. When a function is called that returns a structure, the function places a structure-type return value on the stack as shown in Figure 9.1-5 "Functions Returning a Return Value of the Type "structure"". For details of calling functions that return a structure, see Section 9.2 "Functions Returning Structure-type Values and Stack Conservation."

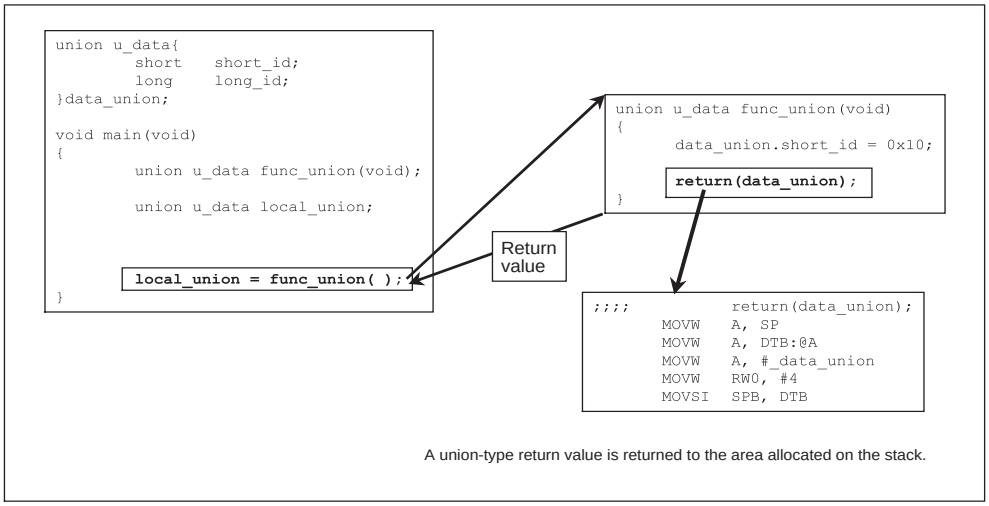
Figure 9.1-5 Functions Returning a Return Value of the Type "structure"



■ Functions Returning Union-Type Values

Some functions return a union. The size of union to be returned depends on the members defined in the union, in the same way as for the above discussed functions with a structure-type return value. When a function that returns a union is called, the function places a union-type return value on the stack as shown in Figure 9.1-6 "Functions Returning a Return Value of the Type "union"". For details of function calls to functions that return a union, see Section 9.3 "Functions Returning Union-type Values and Stack Conservation."

Figure 9.1-6 Functions Returning a Return Value of the Type "union"



9.2 Functions Returning Structure-type Values and Stack Conservation

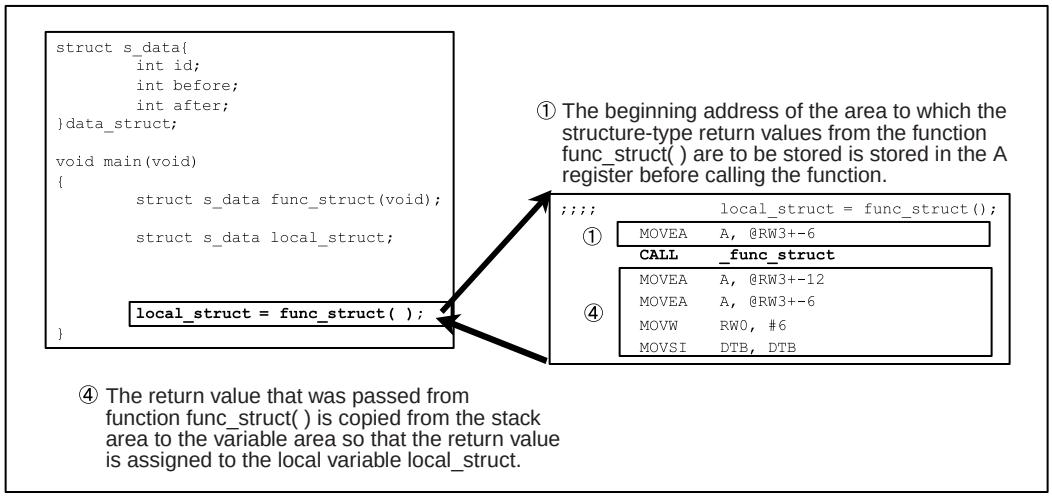
This section describes improvements with respect to the return values for a function that returns a value of the type "structure."
When a function is called that returns a value of the type "structure", the return value is not placed into an register but is stored on the stack. The larger the structure-type return value, the larger the stack area used.

■ Calling a Function Returning a Structure-type Value

Figure 9.2-1 "Calling a Function That Returns a Structure" and Figure 9.2-2 "Stack Status for Calling Functions That Return a Structure" show an example of a function that has a return value of the type "structure." In this example, the function main( ) calls a function of the type s_data. To call the function func_struct( ) that returns a value of the type "structure", the following operations are necessary:

1. The calling function main( ) loads the start address of the area to which the func_struct( ) will output its return value into the A register before calling the function func_struct( ). (See Figure 9.2-1 "Calling a Function that Returns a Structure".)
2. The called function func_struct( ) saves the value of the A register to the stack before starting with function processing. (See Figure 9.2-2 "Stack Status for Calling Functions That Return a Structure".)
3. When function processing terminates, the return value of the type "structure" is passed to the calling function main( ) based on the beginning address of the area for saving the return values from the function. (See Figure 9.2-2 "Stack Status for Calling Functions That Return a Structure".)
4. The calling function main( ) copies the return value from the stack to a local variable. (See Figure 9.2-1 "Calling a Function That Returns a Structure".)

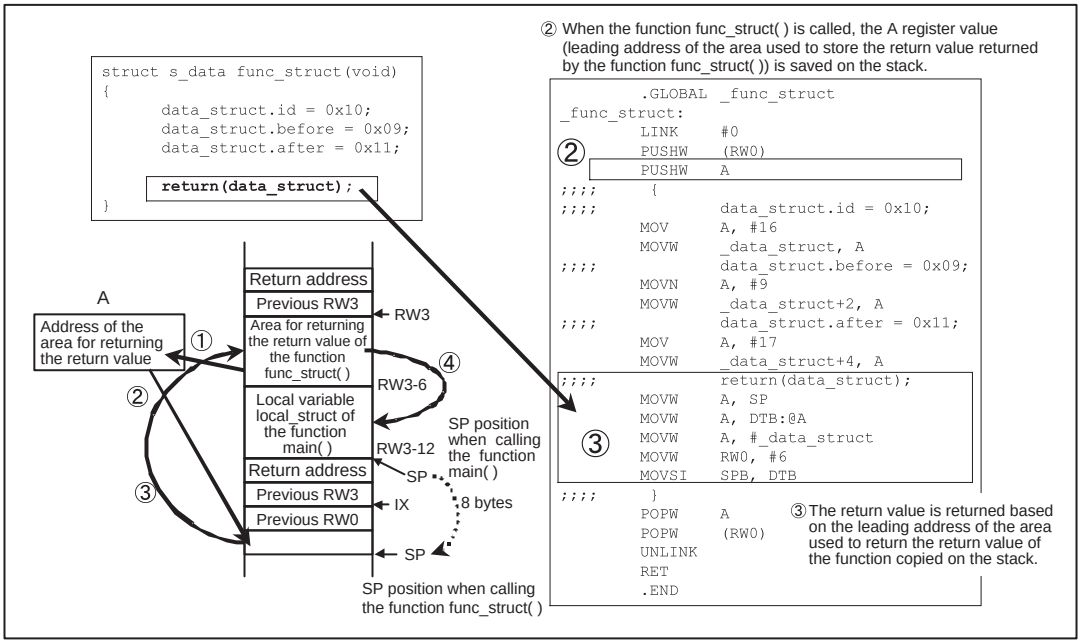
Figure 9.2-1 Calling a Function That Returns a Structure



To call a function that returns a structure, the area for saving the structure-type return value must be prepared as well as the argument to be passed to the function and the local variables

of the called function. The larger the returned structure, the larger the stack size used. In this example, the return value that is saved on the stack is copied to structure local_struct because the structure that was returned from the function func_struct( ) is assigned to the structure local_struct of the local variable.

Figure 9.2-2 Stack Status for Calling Functions That Return a Structure



■ Calling a Function Passing the Address of the Structure Variable to which the Return Value is to be Passed

In the function processing shown in Figure 9.2-1 "Calling a Function That Returns a Structure" and Figure 9.2-2 "Stack Status for Calling Functions That Return a Structure", the structure that was returned from function func_struct( ) is assigned to the local structure variable local_struct. Therefore, the return value is copied from the stack to the structure local_struct.

In this case, the function should be defined in such a way that the function passes the address of the structure variable to which the return value is to be passed. This reduces the size of the stack used.

Figure 9.2-3 "Passing the Structure Address to the Function" and Figure 9.2-4 "Stack Status When Calling a Function That Passes a Return Value to a Specified Structure" show how the call to the function returning a structure was improved by changing the function call shown in Figure 9.2-1 "Calling a Function That Returns a Structure" and Figure 9.2-2 "Stack Status for Calling Functions That Return a Structure". The function func_struct_addr( ) is called as follows:

1. The address of the structure local_struct is stored as argument on the stack before calling the function func_struct_addr( ). (See Figure 9.2-3 "Passing the Structure Address to the Function".)
2. The called function func_struct_addr( ) directly writes a value to the local variable local_struct of the function main( ) in accordance with the address stored on the stack. (See Figure 9.2-4 "Stack Status When Calling a Function that Passes a Return Value to a Specified Structure".)

Figure 9.2-3 Passing the Structure Address to the Function

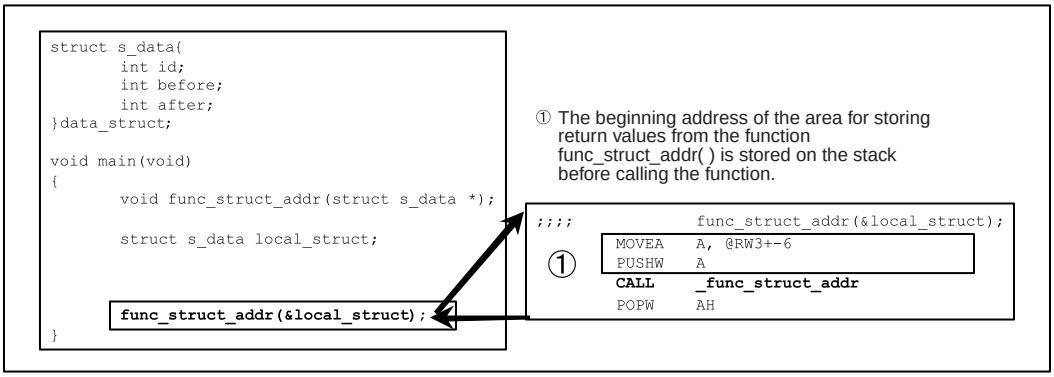
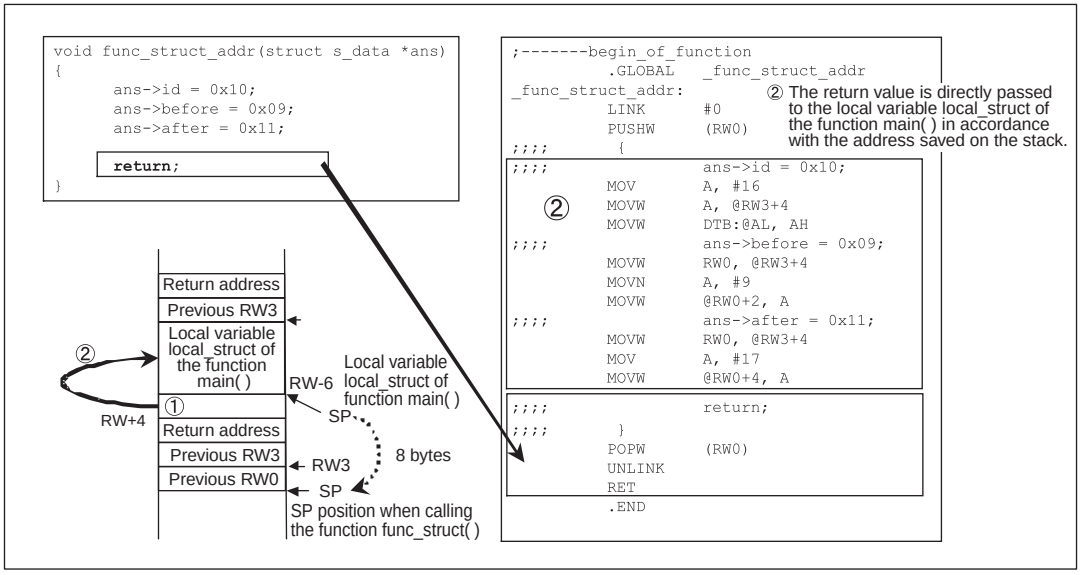


Figure 9.2-4 Stack Status When Calling a Function That Passes a Return Value to a Specified Structure



9.3 Functions Returning Union-type Values and Stack Conservation

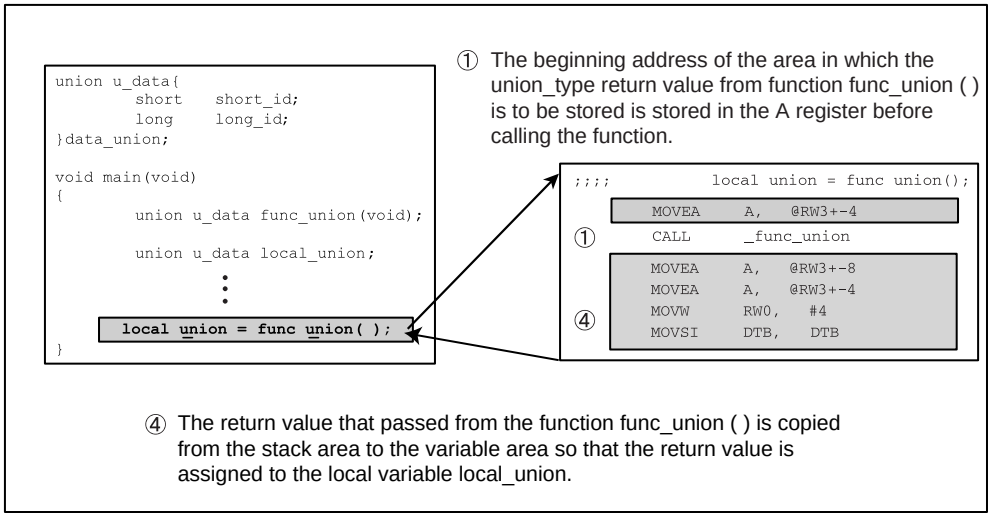
This section describes improvements with respect to the return values for a function that returns a value of the type "union."
When a function that returns a value of the type "union" is called, the return value is not placed into registers but is stored on the stack. The larger the value of the returned union, the larger the stack area used.

■ Calling a Function Returning a Union-Type Value

Figure 9.3-1 "Calling a Function That Returns a Union" and Figure 9.3-2 "Stack Status When Calling a Function That Returns a Union" show an example for a function that returns a value of the type "union." In this example, the function main( ) calls a function of the type "u_data." Calling the function func_union( ) that returns a value of the type "union" requires the following operations:

1. The calling function main( ) loads the start address of the area to which the function func_union( ) will return a value into the A register before calling the function func_union( ). (See Figure 9.3-1 "Calling a Function That Returns a Union".)
2. The called function func_union( ) saves the value of the A register to the stack before starting with function processing. (See Figure 9.3-2 "Stack Status When Calling a Function That Returns a Union".)
3. When function processing terminates, the return value of the type "union" is passed to the calling function main( ) based on the start address of the area for storing the return values from the function. (See Figure 9.3-2 "Stack Status When Calling a Function That Returns a Union".)
4. The calling function main( ) copies the return value from the stack to the local variable. (See Figure 9.3-2 "Stack Status When Calling a Function That Returns a Union".)

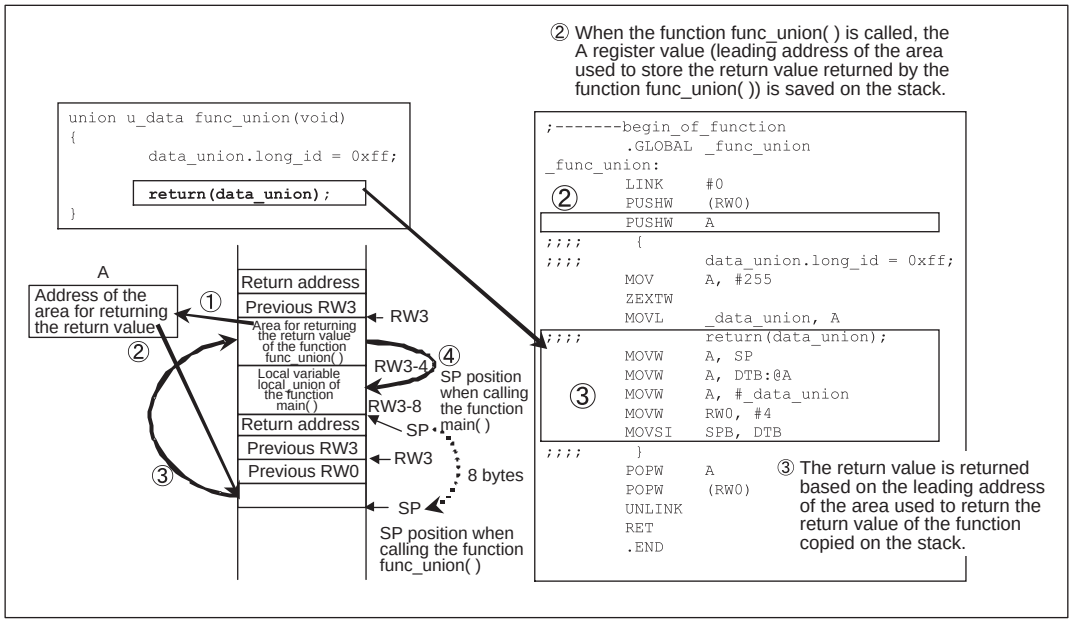
Figure 9.3-1 Calling a Function That Returns a Union



For calling a function that returns a union, the area for saving the union-type return value must

be prepared as well as the argument to be passed to the function and the local variables of the called function. The larger the returned union, the larger the stack size used. In this example, because the union that was returned from function func_union( ) is assigned to the union local_union of the local variable, the return value that is saved on the stack is copied to the union local_union.

Figure 9.3-2 Stack Status When Calling a Function That Returns a Union



■ Calling a Function Passing the Address of a Union Variable to Which the Return Values Are to Be Passed

In the function processing shown in Figure 9.3-1 "Calling a Function That Returns a Union" and Figure 9.3-2 "Stack Status When Calling a Function That Returns a Union", the union that was returned from function func_union( ) is assigned to a local variable union local_union. Therefore, the return value is copied from the stack to the union local_union.

In this case, the function should be defined in such a way that the function passes the address of the union variable to which the return value is to be passed. This reduces the size of stack used.

Figure 9.3-3 "Passing the Union Address to a Function" and Figure 9.3-4 "Stack Status When Calling a Function That Passes a Return Value to the Specified Union" show how the call to the function returning the union was improved by changing the function call shown in Figure 9.3-1 "Calling a Function That Returns a Union" and Figure 9.3-2 "Stack Status When Calling a Function That Returns a Union". The function func_union_addr( ) is called as follows:

1. The address of the union local_union is stored as argument on the stack before calling the function func_union_addr( ). (See Figure 9.3-3 "Passing the Union Address to a Function".)
2. The called function func_union_addr( ) directly writes a value to the local variable local_union of function main( ) in accordance with the address stored on the stack. (See Figure 9.3-4 "Stack Status When Calling a Function That Passes a Return Value to the Specified Union".)

Figure 9.3-3 Passing the Union Address to a Function

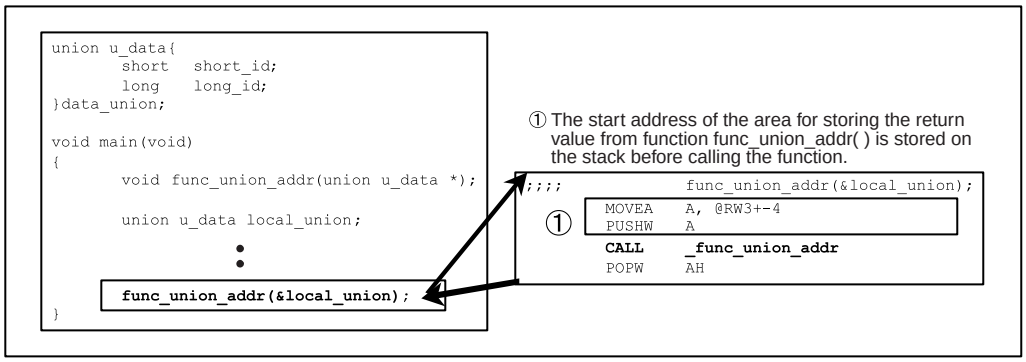
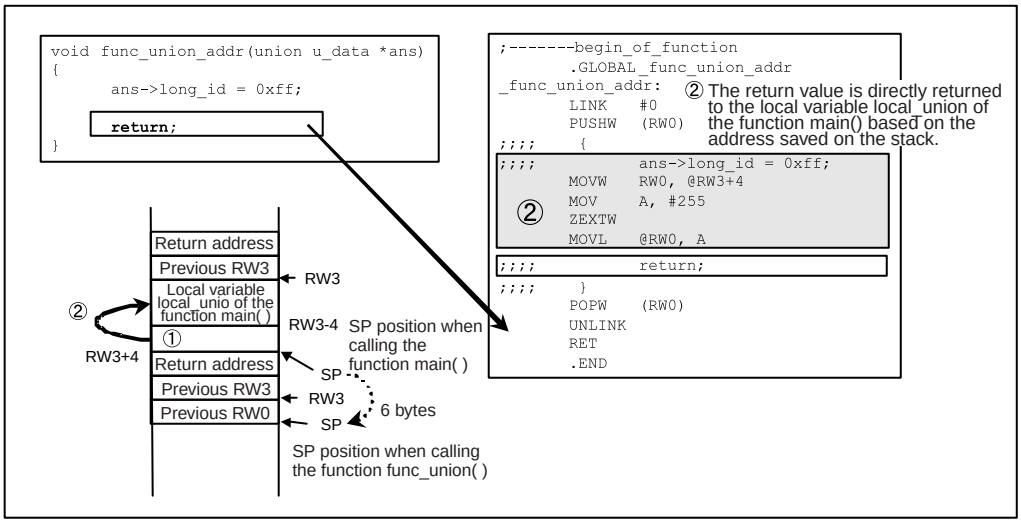


Figure 9.3-4 Stack Status When Calling a Function That Passes a Return Value to the Specified Union



PART III USING LANGUAGE EXTENSIONS

Part III describes the fcc907 language extensions.

The fcc907 supports specifications for using the F^{2}MC-16 family architecture. These specifications are referred to as the language extensions. Part 3 begins with an overview of the language extensions. It then provides notes on including assembler code in a C program and on the specification and placement of the _ _io area and _ _direct type qualifier. This part also provides notes on creating and registering interrupt functions.

CHAPTER 10 WHAT ARE LANGUAGE EXTENSIONS?

The fcc907 provides the following functionality through language extensions:

- **Coding of Assembler instructions using an _ _asm statement**
- **Extended type qualifiers**
- **Extended functions using #pragma**
- **Interrupt-related built-in functions**
- **Other built-in functions**

This chapter describes these functions.

10.1 Coding Assembler Instructions Using an _ _asm Statement

This section briefly describes how to include Assembler instruction into a C program using an _ _asm statement.
The _ _asm statement is used to include an Assembler instruction into a C program.

■ Coding Assembler Instructions Using an _ _asm Statement

The _ _asm statement is used to include an Assembler instruction into a C program. Write the _ _asm statement as follows:

```
_ _asm ("Assembler instruction");
```

C programs cannot directly set the values of CPU registers. Moreover, some operations of C programs cannot be executed fast enough. To execute such operations, you can use an _ _asm statement to include instead an Assembler instruction into the C program.

The fcc907 uses the _ _asm statement for coding Assembler instructions both inside a function or outside functions.

Figure 10.1-1 Function in Which _ _asm Statement Is Used

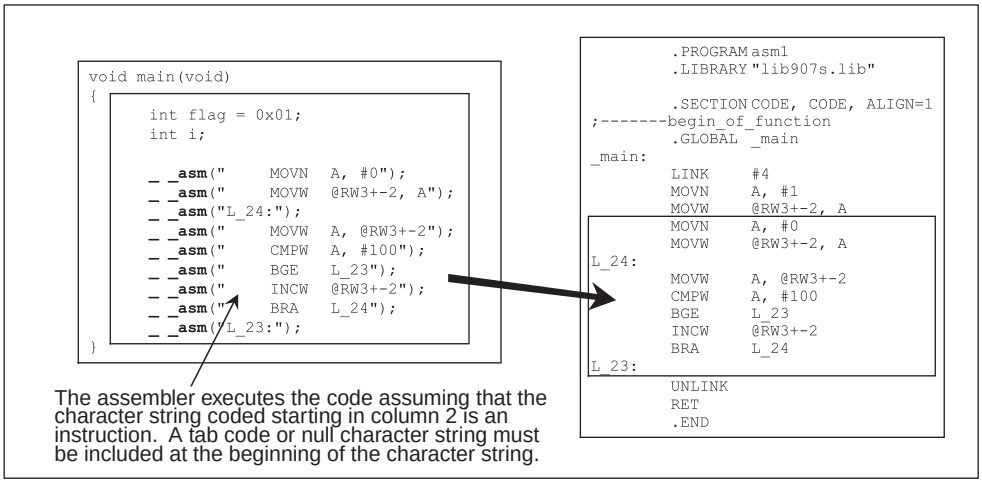


Figure 10.1-1 "Function in Which _ _asm Statement Is Used" shows an example for the coding of an _ _asm statement. When an _ _asm statement is included, an Assembler instruction is expanded at the location of the statement is included in the text.

See CHAPTER 11 "NOTES ON ASSEMBLER PROGRAMS IN C PROGRAMS" for information about including assembler code using the _ _asm statement.

10.2 Extended Type Qualifiers

This section describes the extended type qualifier, which is one of the language extensions.

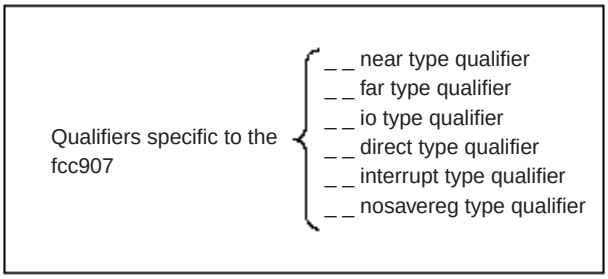
The fcc907 provides the following six extended type qualifiers in addition to the ordinary type qualifiers (const and volatile):

- **_ _near type qualifier**
- **_ _far type qualifier**
- **_ _io type qualifier**
- **_ _direct type qualifier**
- **_ _interrupt type qualifier**
- **_ _nosavereg type qualifier**

These six type qualifiers are dependent on the F^2MC-16 family architecture.

■ Extended Type Qualifiers

The fcc907 provides the following extended type qualifiers:



Sections 10.2.1 "_ _ near Type Qualifier and _ _far type Qualifier" to 10.2.5 "_ _nosavereg Type Qualifier" briefly describe the functions of the above type qualifiers and provide notes on their use.

10.2.1 _ _near Type Qualifier and _ _far Type Qualifier

This section describes the _ _near type qualifier and _ _far type qualifier of the fcc907 type qualifiers. These type qualifiers can be specified for variables and functions. The _ _near-type qualified variables and functions are accessed using 16-bit addressing. The _ _far-type qualified variables and functions are accessed using 24-bit addressing.

■ Specifications of the _ _near type qualifier and _ _far type qualifier

The _ _near type qualifier can be specified for variables and functions. The _ _near-type qualified variables and functions are accessed using 16-bit addressing regardless of the memory model specified at compilation. In addition, the _ _far-type qualified variables and functions are accessed using 24-bit addressing.

Figure 10.2-1 Specification of the _ _near-type Qualifier (for a Large Model)

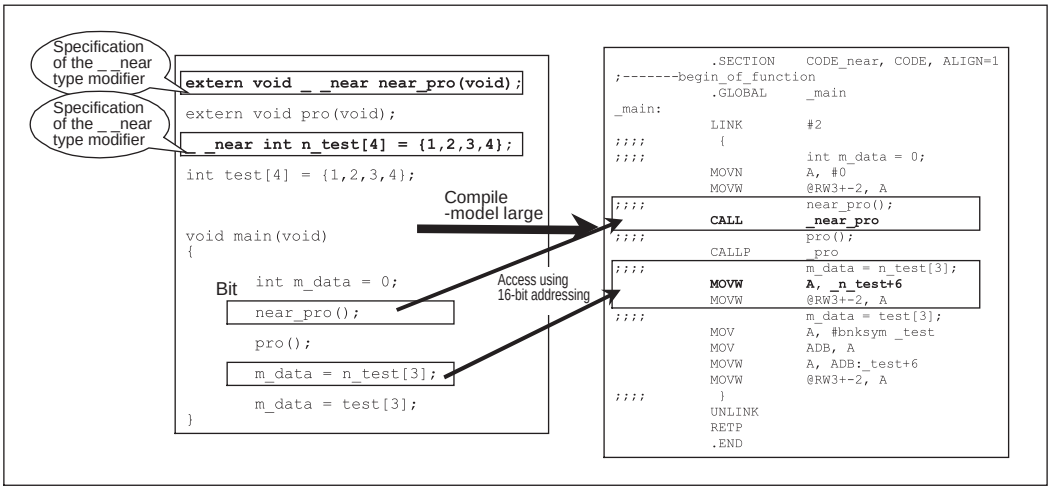


Figure 10.2-1 "Specification of the _ _near-type Qualifier (for a Large Model)" shows an example of compiling a program that includes _ _near-type qualified variables and functions for a large model. In this example, _ _near type qualification has been performed for the function near_pro( ) and int-type array n_test[ ]. For compilation using a large model, the variables and functions are accessed using 24-bit addressing. The _ _near-type qualified function near_pro( ), however, is called using 16-bit addressing. In addition, the element n_test[3] of the _ _near-type qualified array is accessed using 16-bit addressing.

Figure 10.2-2 Specification of the _ _far-type Qualifier (for a Small Model)

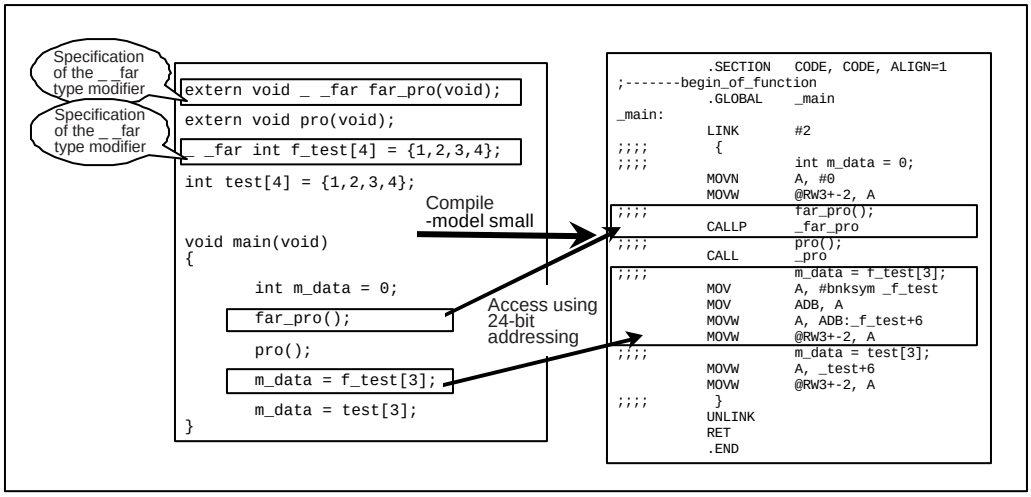


Figure 10.2-2 "Specification of the _ _far-type Qualifier (for a Small Model)" shows an example of compiling a program that includes _ _far-type qualified variables and functions for a small model. In this example, _ _far type qualification has been performed for the function far_pro( ) and int-type array f_test[ ]. For compilation using a small model, the variables and functions are accessed using 16-bit addressing. The _ _far-type qualified function far_pro( ), however, is called using 24-bit addressing. In addition, the element f_test[4] of the _ _far-type qualified array is accessed using 24-bit addressing.

As described above, the _ _near-type qualified variables and functions are accessed using 16-bit addressing regardless of the memory model specified at compilation. In the same way, the _ _far-type qualified variables and functions are accessed using 24-bit addressing regardless of the memory model specified at compilation.

<Notes>

The _ _near type qualifier and _ _far-type qualifier cannot be specified for local variables.

10.2.2 _ _io Type Qualifier

This section describes the _ _io type qualifier, which is an fcc907 extended type qualifier. The _ _io type qualifier is specified for a variable mapped into the I/O area.

■ Variables with _ _io Type Qualifier

The _ _io type qualifier is one of the type qualifiers specific to the fcc907.In the fcc907, the _ _io type qualifier is specified for a variable mapped into the I/O area (addresses h'0000' to h'00ff'). A variable qualified by the _ _io type qualifier is accessed via I/O addressing. In I/O addressing, the addresses h'0000' to h'00ff' can be accessed. In I/O addressing, the user specifies only the lower 8 bits of the address to be accessed because the high-order byte of the address is automatically assumed to be h'00'. This format allows to express a memory address in one byte. Because machine instructions using I/O addressing are generated when a variable qualified by the _ _io type qualifier is accessed, the generated code is smaller than the code generated for accessing a variable via normal addressing.

See CHAPTER 12 "NOTES ON DEFINING AND ACCESSING THE I/O AREA" for information about mapping variables into the I/O area.

Figure 10.2-3 _ _io Type Qualifier Specification and Access

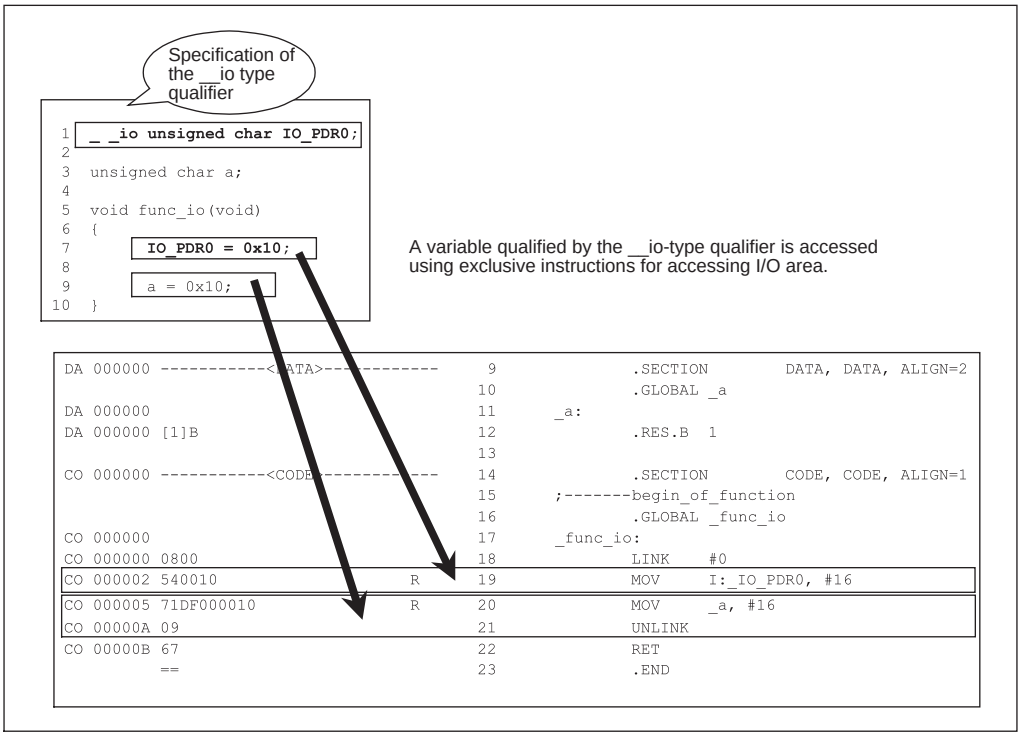


Figure 10.2-3 "_ _io Type Qualifier Specification and Access" shows an example of _ _io type qualifier specification and access.

In this example, the _ _io type qualifier is specified when variable IO_PDR0 is defined. This variable is accessed using I/O addressing.

When external variable a is accessed, code using 16-bit addressing is generated. When a machine instructions are generated using I/O addressing when a variable qualified by the _ _io

type qualifier is accessed, the generated code uses I/O addressing and is therefore smaller than the code generated for variable access using normal addressing.

<Notes>

When defining variables with the _ _io type qualifier specified, variable areas are allocated in the order defined. A variable such as a dummy must be defined for those locations where a variable is not defined.

[Tip]

Softune C Checker:

The Softune C Checker outputs a warning if the _ _io type qualifier, a language extension, is used in a definition and declaration. This check function is useful for creating programs for which portability is important.

10.2.3 _ _direct Type Qualifier

This section describes the _ _direct type qualifier, which is an fcc907 extended type qualifier. The _ _direct type qualifier is specified for variables mapped into the direct area.

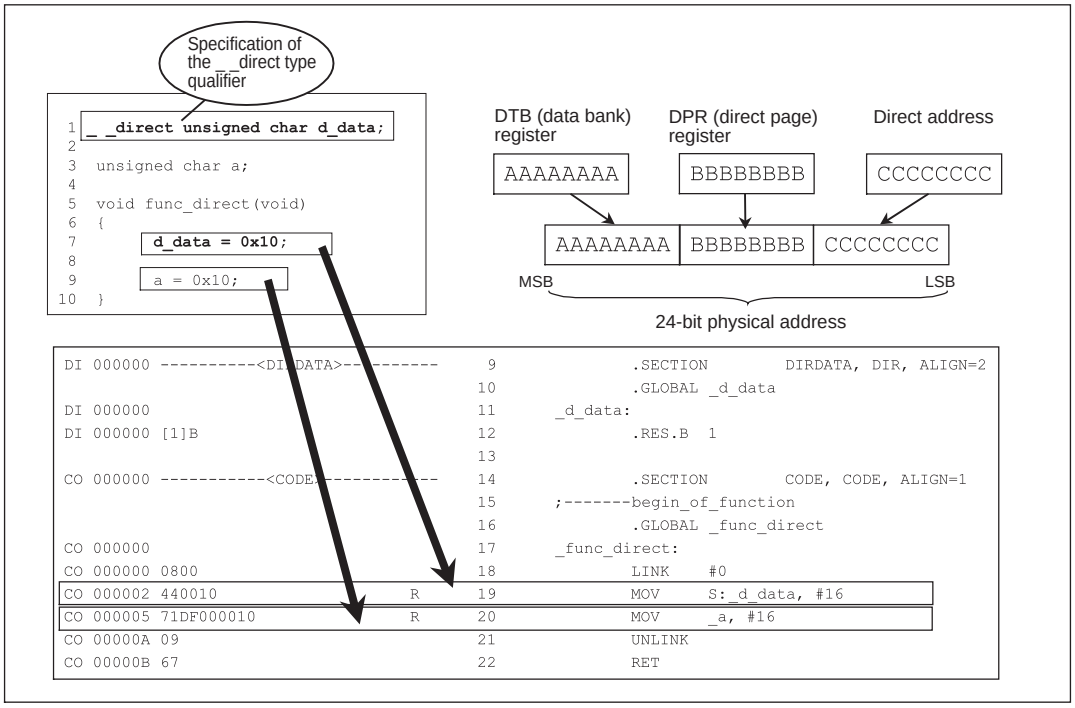
■ Variables with _ _direct Type Qualifier

The _ _direct type qualifier is one of the type qualifiers specific to the fcc907.

For the fcc907, _ _direct-type qualified variables are accessed using direct addressing. In direct addressing, the address of a variable mapped in the direct area (page pointed to by the dtb register) is accessed in eight bit units. As a result, a smaller code than that used when accessing using normal addressing can be generated.

Figure 10.2-4 "Defining and Accessing a Variable Qualified Using the _ _direct Type Qualifier" shows an example of defining and accessing a variable qualified using the _ _direct type qualifier. In this example, the _ _direct type qualifier is specified when the variable d_data is defined. See CHAPTER 13 "MAPPING VARIABLES QUALIFIED WITH THE TYPE QUALIFIER _ _direct" for details on variables qualified using the _ _direct type qualifier.

Figure 10.2-4 Defining and Accessing a Variable Qualified Using the _ _direct Type Qualifier



[Tip]

Softune C Checker:

The Softune C Checker outputs a warning if the _ _direct type qualifier, a language extension, is used in a definition or declaration. The fcc907 and fcc896 support the same function for defining and accessing variables qualified by the _ _direct type qualifier. This check function is useful for porting programs between the fcc907 and fcc896.

10.2.4 _ _interrupt Type Qualifier

This section describes the _ _interrupt type qualifier, which an fcc907 extended type qualifier. The _ _interrupt type qualifier is specified for an interrupt function.

■ Functions with _ _interrupt Type Qualifier

The _ _interrupt type qualifier is one of the fcc907-specific type qualifiers.

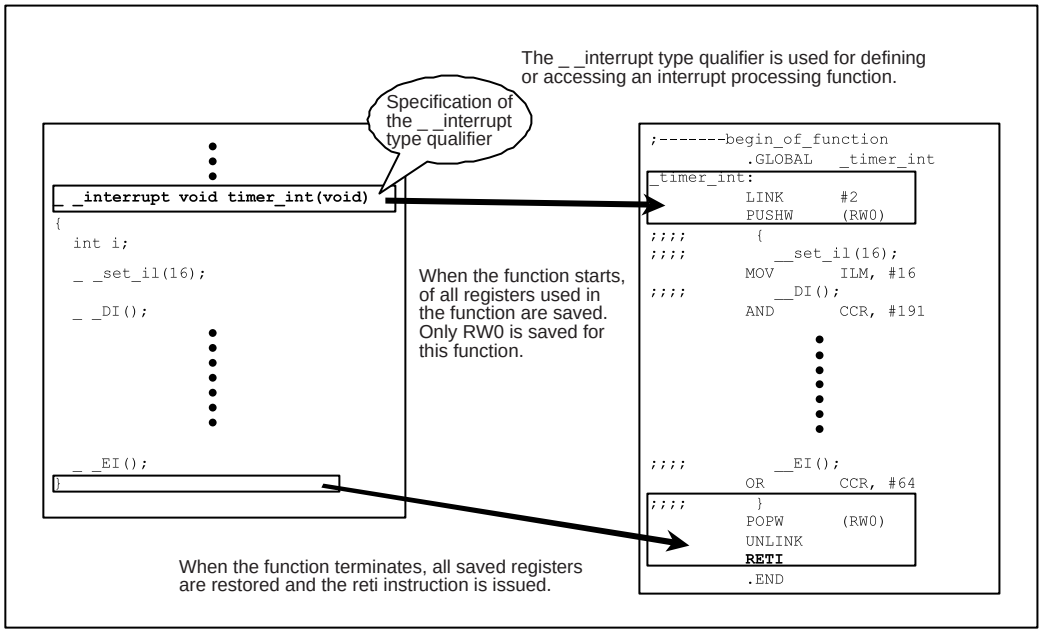
The fcc907 uses the _ _interrupt type qualifier for the specification of interrupt functions.

When an interrupt function qualified by the _ _interrupt type qualifier is called, it saves the contents of work registers before performing any processing. When the function ends, it restores all saved registers, returns control to the location where the interrupt occurred, and resumes processing. Use of this type qualifier facilitates coding of interrupt functions in C.

Figure 10.2-5 "_ _interrupt Type Qualifier Specification" shows an example of coding an interrupt function qualified by the _ _interrupt type qualifier. In this example, when an interrupt occurs and the interrupt function int_func( ) is executed, register RW0 is saved on the stack. Next, registers R0 and R1 are saved on the stack.

When the interrupt terminates, the function restores the saved registers and issues the reti instruction. The reti instruction restores the values of the PC and PS saved to the stack and returns control to the location where the interrupt occurred.

Figure 10.2-5 _ _interrupt Type Qualifier Specification



See CHAPTER 14 "CREATING AND REGISTERING INTERRUPT FUNCTIONS" for information about functions qualified by the _ _interrupt type qualifier.

[Tip]

Softune C Checker:

The Softune C Checker outputs a warning if the _ _interrupt type qualifier, a language extension, is used in a definition or declaration. The fcc907 supports the same function for

coding interrupt functions qualified by the _ _interrupt-type as the fcc896 and fcc911. This check function is useful for porting programs between the fcc896 or fcc911 and fcc896.

10.2.5 _ _nosavereg Type Qualifier

This section describes the _ _nosavereg type qualifier, which an fcc907 extended type qualifier. The _ _nosavereg type qualifier is specified for an interrupt function together with the _ _interrupt type qualifier.

■ Functions with _ _nosavereg Type Qualifier

The _ _nosavereg type qualifier is one of the type qualifiers specific to the fcc907.

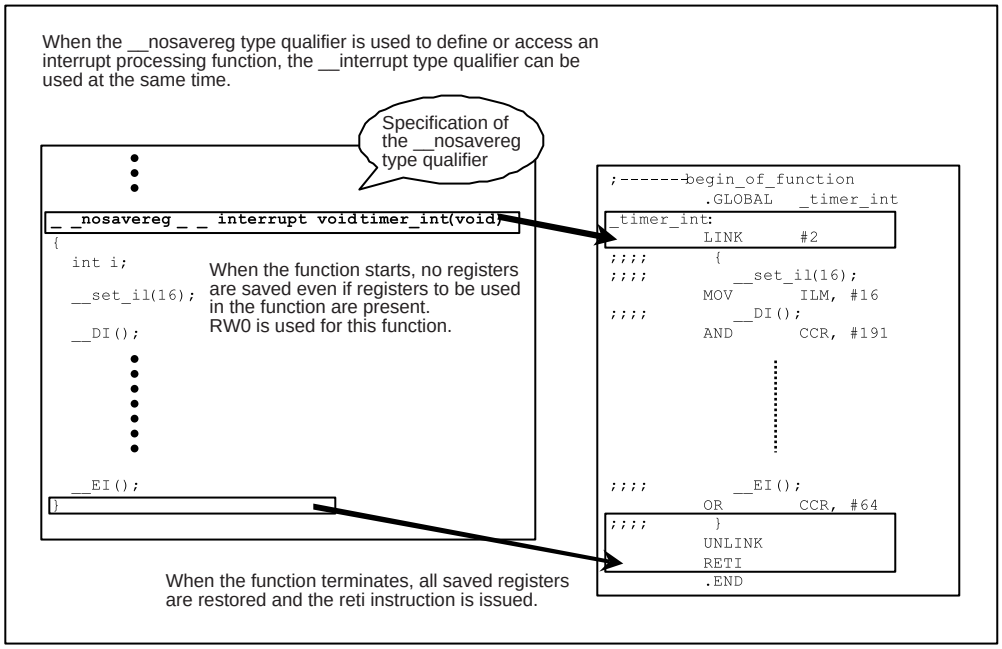
In the fcc907, the _ _nosavereg type qualifier is specified for an interrupt function together with the _ _interrupt type qualifier.

When an interrupt function qualified using the _ _nosaverreg type qualifier is called, the interrupt function executes processing without saving registers. This applies even if registers to be used in the function are present. When the function terminates, it issues the reti instruction and processing resumes at the location where the interrupt occurred. Because the #pragma register/noregister for switching the register banks can also be used at the same time, high-speed interrupt processing is enabled.

Figure 10.2-6 "_ _nosavereg Type Qualifier Specification" shows an example of coding an interrupt function with the _ _nosavereg type qualifier specified. In this example, the function is executed without registers being saved when the interrupt function timer_int( ) is executed. The RW0 register is used in this function.

When the interrupt terminates, the function restores the saved registers and issues the reti instruction. The reti instruction restores the values of the PC and PS saved on the stack and returns control to the location where the interrupt occurred.

Figure 10.2-6 _ _nosavereg Type Qualifier Specification



See CHAPTER 14 "CREATING AND REGISTERING INTERRUPT FUNCTIONS" for information about functions qualified by the _ _nosavereg type qualifier.

[Tip]

Softune C Checker:

The Softune C Checker outputs a warning if the _ _nosavereg type qualifier, a language extension, is used in a definition or declaration. The fcc907 supports the same function for coding interrupt functions qualified by the _ _nosavereg type qualifier as the fcc896. This check function is useful for porting programs between the fcc907 and fcc896.

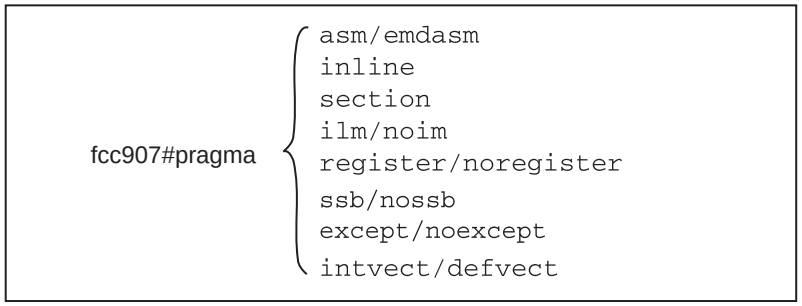
10.3 Extended Functions Using #pragma

This section describes #pragma as used in the fcc907.
The fcc907 provides the following eight #pragma types as extended functions:

- **asm/endasm**
- **inline**
- **section**
- **ilm/noilm**
- **register/noregister**
- **ssb/nossb**
- **except/noexcept**
- **intvect/defvect**

■ Extended Functions Using #pragma

The fcc907 provides the following #pragma functions:



A control line that begins with #pragma specifies operations specific to the fcc907. Sections 10.3.1 "Inserting Assembler Programs Using #pragma asm/endasm" to 10.3.8 "Generating an Interrupt Vector Table Using #pragma intvect/defvect" briefly describe the #pragma functions and provide notes on their use.

10.3.1 Inserting Assembler Programs Using #pragma asm/ endasm

This section describes #pragma asm/endasm.
The #pragma asm/endasm can be used to code assembly instructions in C programs.

■ Inserting Assembler Programs Using #pragma asm/endasm

The #pragma asm directive specifies the start of insertion of an assembler program.

```
#pragma asm
```

The #pragma endasm directive specifies the end of insertion of an assembler program.

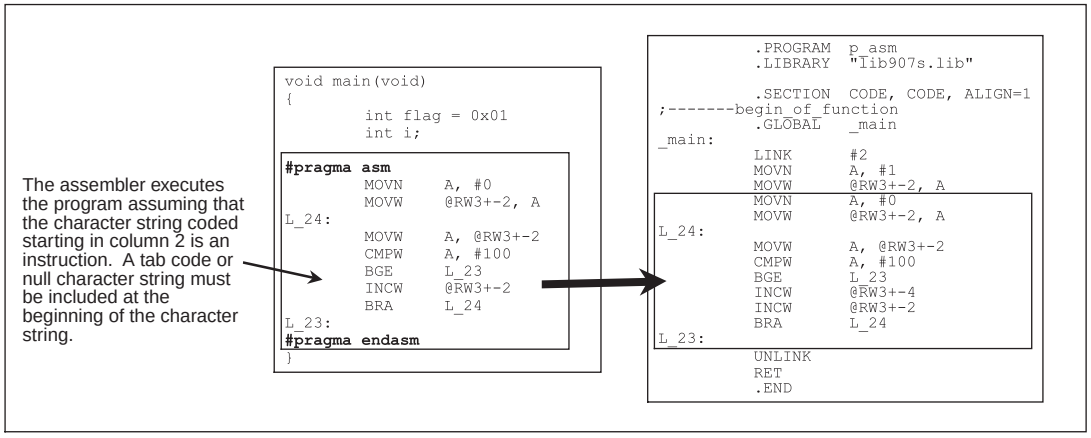
```
#pragma endasm
```

C programs cannot directly set the contents of CPU registers. Moreover, some operations in C programs cannot be executed fast enough. To execute such operations, you can use #pragma asm/endasm to include Assembler programs into the C program.

Figure 10.3-1 "Coding #pragma asm/endasm" shows an example of coding #pragma asm/ endasm. At the location where #pragma asm/endasm are used, Assembler instructions are expanded.

See CHAPTER 11 "NOTES ON ASSEMBLER PROGRAM IN C PROGRAMS" for information about including Assembler modules using #pragma asm/endasm.

Figure 10.3-1 Coding #pragma asm/endasm



10.3.2 Specifying Inline Expansion Using #pragma inline

This section describes inline expansion using #pragma inline. The #pragma inline directive is used to specify a function that is to be expanded.

■ Inline Expansion Using #pragma inline

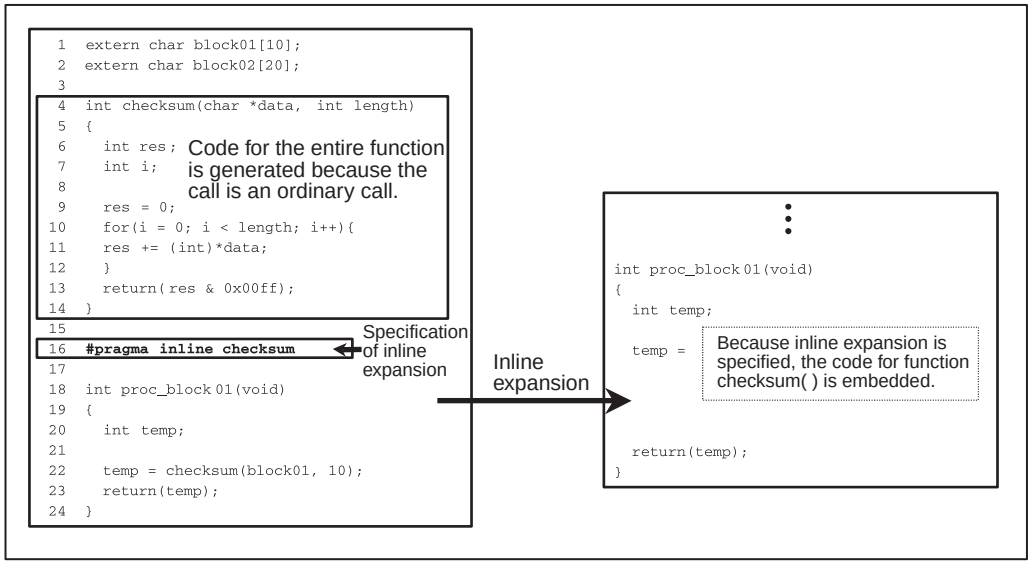
The #pragma inline directive is used to specify a function that is to be expanded. The specified function is expanded in line during compilation. After this specification, the specified function is expanded in line whenever it is called.

```
#pragma inline   name-of-function-expanded-inline
```

Figure 10.3-2 "Inline Expansion of a Function Using #pragma inline" shows an example of using #pragma inline.

In this example, inline expansion of the function checksum is specified on line 16. Therefore, when the function proc_block01( ) is called, function checksum will be expanded in line.

Figure 10.3-2 Inline Expansion of a Function Using #pragma inline



See CHAPTER 7 "REDUCING FUNCTION CALLS BY EXANDING FUNCTIONS IN LINE" for information about expanding functions in line.

<Notes>

When inline expansion is specified using #pragma inline, use the -O option to specify optimization during compilation. If optimization if not specified, inline expansion will not be executed.

[Tip]

For the fcc907:

The following option can be used to specify the function to be expanded in line during compilation.

-x function-name option

Use the following option to specify the number of lines of the function to be expanded in line during compilation.

-xauto size option

Optimization must be specified using the -O option.

10.3.3 Using #pragma section to Change Section Names and Specify Mapping Address

This section briefly describes how to use #pragma section to change section names and section attributes and to specify mapping addresses.

■ Using #pragma section to Change Section Names and Specify Mapping Addresses

The #pragma section directive can change the default section names output by the fcc907 to user-specified section names. In addition, #pragma section can change the section attributes.

```
#pragma section default-section-name [=new-section-name][, attr=attribute][, locate=mapping-address]
```

The fcc907 can specify the sections listed in Table 10.3-1 "Default Sections That Can Be Specified Using #pragma section" for the default section, and can specify the section attributes listed in Table 10.3-2 "Default Section Attributes That Can Be Specified Using #pragma section" for "attr."

For the mapping address, specify the beginning address of where the specified section is to be mapped.

Table 10.3-1 Default Sections That Can Be Specified Using #pragma section

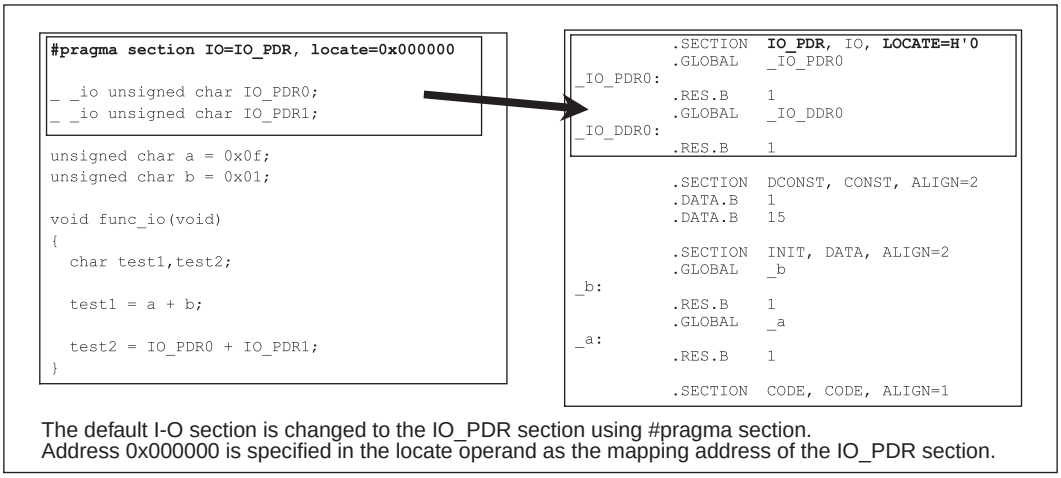
Section name	Section type
CODE	Code area
INIT	Area for variables that are initialized
DCONST	Initial value area for variables with the initial value specified
CONST	Area for variables qualified by the const type qualifier
CINIT	RAM area for const-type qualified variables when a CPU that does not have the mirror ROM function is used
DATA	Area for variables that are not initialized
DIRINIT	Area for variables qualified by the_ _direct type qualifier with initial value specified
DIRCONST	nitial value area for _ _direct-type qualified variables with initial value specified
DIRDATA	Area for variables qualified by the_ _direct type qualifier without initial value specified
IO	Area for variables qualified by the _ _io type qualifier
INTVECT	Interrupt vector table area
DTRANS DCLEAR	Data table for initializing external variables

Table 10.3-2 Default Section Attributes That Can Be Specified Using #pragma section

Section attribute name	Explanation
CODE	Program code area
DATA	Area for variables that are not initialized
CONST	Area for variables whose specified initial value does not change
COMMON	Shared variables and shared area
STACK	Stack area
IO	Input-output port area
IOCOMMON	Input-output area that can be shared with the linker
DIR	Direct access area
DIRCONST	Direct access area in which initial values that do not change are mapped
DIRCOMMON	Direct access area that can be shared with the linker

Figure 10.3-3 "Changing the Output Section Using #pragma section" shows an example of using #pragma section. In this example, the default I/O section is changed to the IO_PDR section. In addition, the IO_PDR section is mapped into the area beginning at address 0x000000. As a result, the variable qualified by the _ _io type qualifier is output to the IO_PDR section allocated to the area beginning with address 0x000000.

Figure 10.3-3 Changing the Output Section Using #pragma section



The default I-O section is changed to the IO_PDR section using #pragma section.
Address 0x000000 is specified in the locate operand as the mapping address of the IO_PDR section.

[Tip]

For the fcc907:

The following option can be used to specify the same operation as that of #pragma section during compilation.

-s default-section-name=new-section-name [, attribute][, mapping-address] option

10.3.4 Specifying the Interrupt Level Using #pragma ilm/noilm

This section describes #pragma ilm/noilm.
The #pragma ilm/noilm directive is used to set the function interrupt level.

■ Specifying the Interrupt Level Using #pragma ilm/noilm

The #pragma ilm directive specifies the function interrupt level. It is used to specify the interrupt level of each function.

```
#pragma ilm (interrupt-level-number)
```

The #pragma noilm releases the switched interrupt level.

```
#pragma noilm
```

Figure 10.3-4 Using #pragma ilm/noilm to Set Function Interrupt Levels

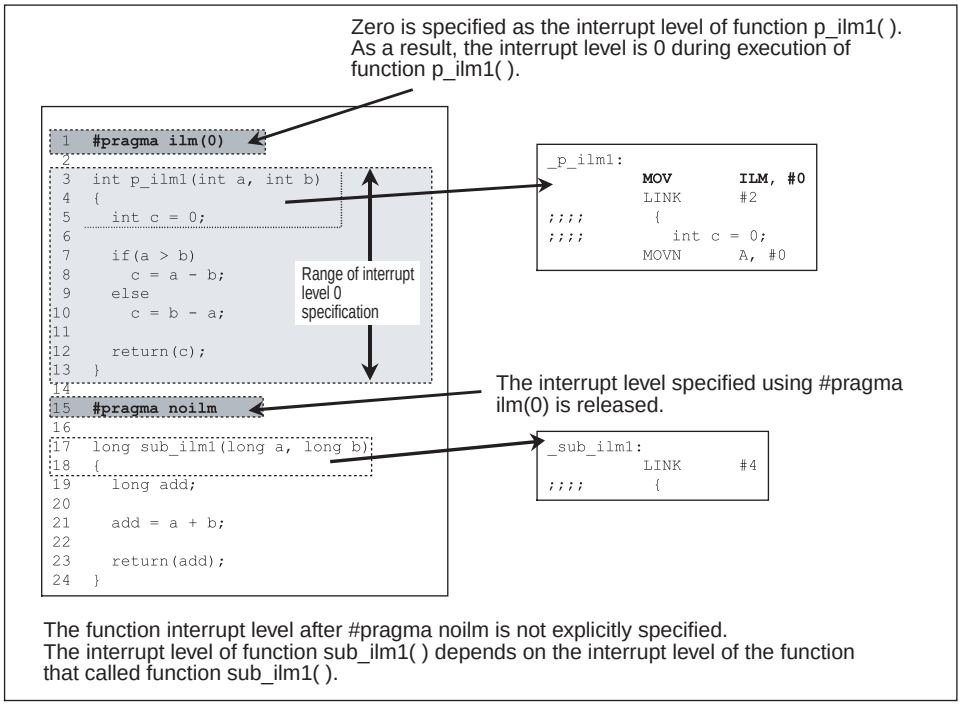


Figure 10.3-4 "Using #pragma ilm/noilm to Set Function Interrupt Levels" shows an example of a function that uses #pragma ilm.

In this example, 0 is specified as the interrupt level when function p_ilm1( ) on line 1 is executed. The specification of #pragma noilm on line 15 releases the interrupt level specified using #pragma ilm(0). As a result of the release, the interrupt level changes to 0 when function p_ilm1( ) is called, but it does not change when function sub_ilm1( ) is called.

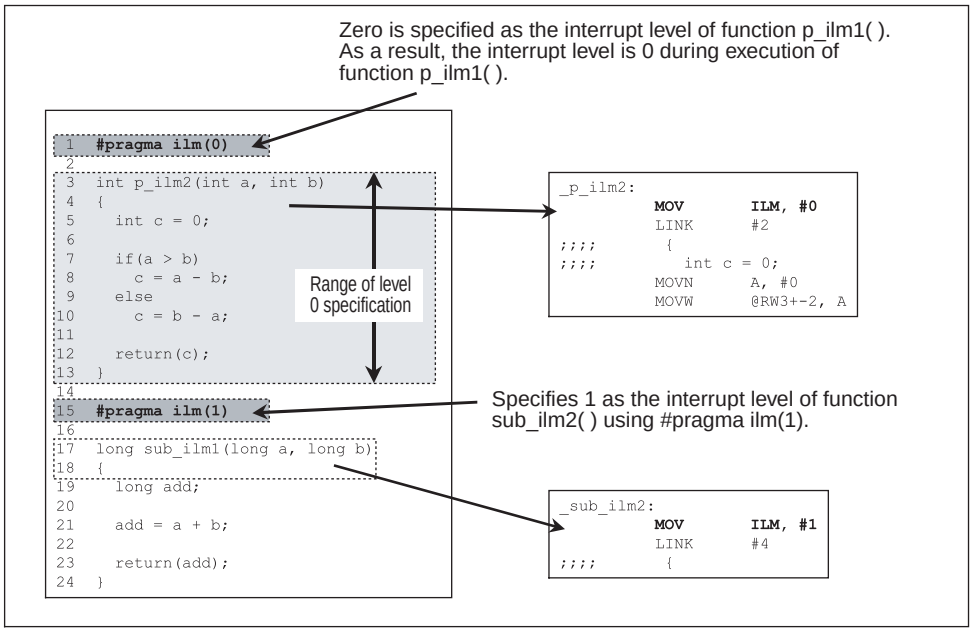
The interrupt level of function sub_ilm1( ) depends on the state when function sub_ilm1( ) is called. When function sub_ilm1( ) is executed, processing is executed using the interrupt level

of the function that called function sub_ilm1( ).

As shown in Figure 10.3-5 "Using #pragma ilm to Set the Interrupt Level for Each Function", when creating a system in which the interrupt level of a function changes, use #pragma ilm to specify the interrupt level.

The minimum unit for which #pragma ilm/noilm can specify the interrupt level is a single function. To change the interrupt level within a function, use the built-in function _ _set_il( ).

Figure 10.3-5 Using #pragma ilm to Set the Interrupt Level for Each Function



<Notes>

Code #pragma ilm/noilm outside the function. The minimum unit for which the interrupt level can be changed using #pragma ilm/noilm is a function. To temporarily change the interrupt level during execution of a function, use the built-in function _ _set_il( ).

Be aware that #pragma noilm only releases the specified #pragma ilm. It does not include a function for returning the interrupt level to what it was before #pragma ilm was specified.

10.3.5 Setting the Register Bank Using #pragma register/ noregister

This section describes #pragma register/noregister.
The #pragma register/noregister directive is used to specify the register bank used by a function.

■ Setting the Register Bank Using #pragma register/noregister

The #pragma register directive specifies the register bank used. This specification enables to change the register bank used for a function.

```
#pragma register (number-of-register-bank-used)
```

The #pragma noregister directive releases the specification of the register bank.

```
#pragma noregister
```

Figure 10.3-6 Using #pragma register/noregister for a Function

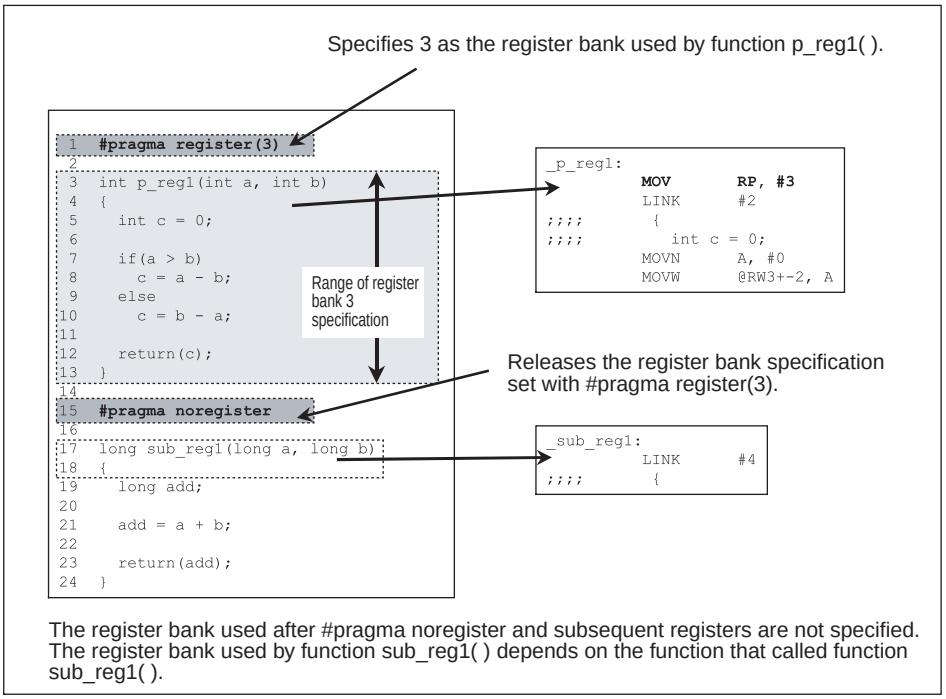


Figure 10.3-6 "Using #pragma register/noregister for a Function" shows an example of using #pragma register for a function. In this example, the register bank that will be used during execution of function p_reg1( )is set to 3 on line 1. On line 15, #pragma noregister releases the register bank specification set by #pragma register(3). As a result of the release, the register bank switches to 3 when function p_reg1( ) is called, but does not change when function sub_reg1( ) is called.

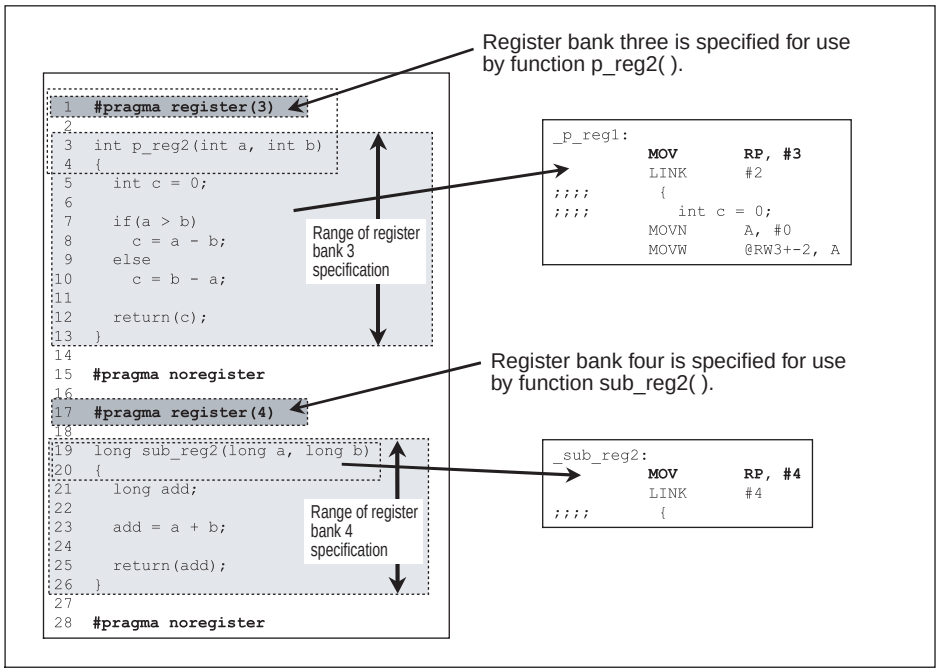
The register bank used by function sub_reg1( ) depends on the status when function sub_reg1( ) is called:

When function sub_reg1( ) is executed, the register bank used by the function that called function sub_reg1( ) is used.

Note that #pragma noregister only cancels the specified #pragma register. It does not have a function for returning to the register bank that was being used before #pragma register was specified.

As shown in Figure 10.3-7 "Using #pragma register to Specify the Register Bank for a Function", when creating a system in which the used register bank changes for a function, use #pragma register to specify the register bank used for the function.

Figure 10.3-7 Using #pragma register to Specify the Register Bank for a Function



<Notes>

Code #pragma register/noregister outside the function. The minimum unit for which the register bank can be specified using #pragma register/noregister is a function. The register bank cannot be changed using #pragma register/noregister during execution of a function.

Be aware that #pragma register only releases the specified #pragma register. It does not include a function for returning to the register bank that was being used before #pragma register was specified.

10.3.6 Setting Use of the System Bank Using #pragma ssb/nossb

This section describes #pragma ssb/nossb.
The function with #pragma ssb/nossb specified accesses the system stack when a stack is used.

■ Accessing the System Stack Using #pragma ssb/nossb

Specifying #pragma ssb sets the system stack as the stack to be accessed by a function. When a stack is accessed, this specification loads the value of the system stack bank register (SSB) and then generates a code for accessing the stack.

```
#pragma ssb
```

Specifying #pragma nossb cancels the specification that allows the system stack to be used.

```
#pragma nossb
```

Figure 10.3-8 Using #pragma ssb/nossb to Set and Allow the System Bank to Be Used

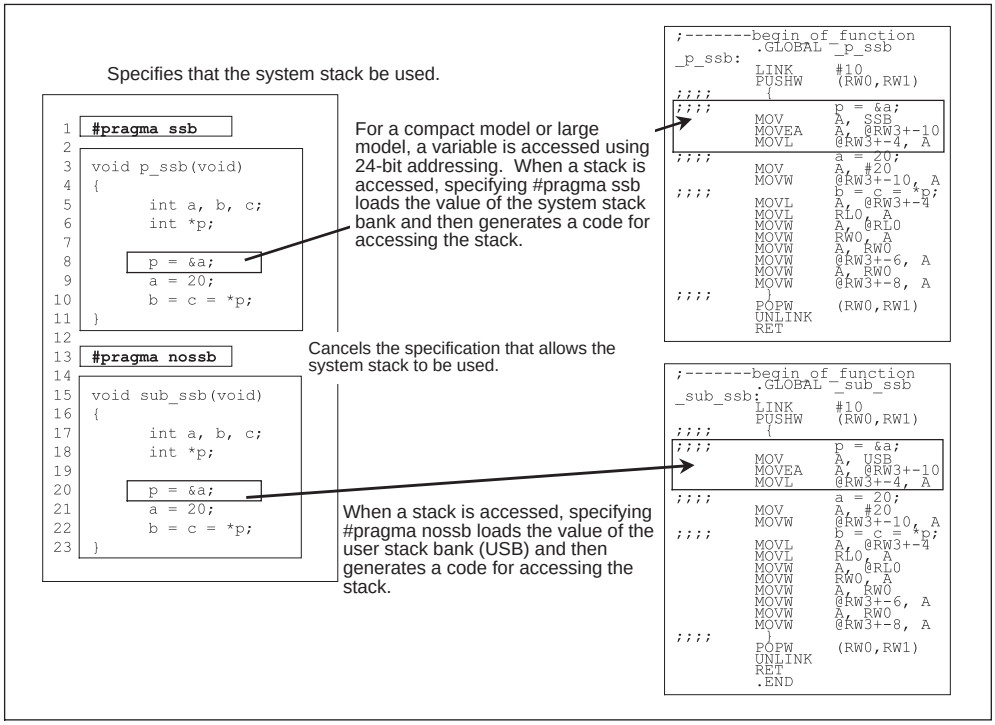


Figure 10.3-8 "Using #pragma ssb/nossb to Set and Release Use of the System Bank" shows an example of a variable using #pragma ssb/nossb. For a compact model or large model, the variable is accessed using 24-bit addressing. For normal stack access, the user stack is accessed using 24-bit addressing. However, if an interrupt function uses the system stack to

execute processing, the system stack must be accessed using 24-bit addressing. In such cases, #pragma ssb/nossb is specified to load the value of the SSB register and generate a code for accessing the stack when the stack is accessed.

In this example, #pragma ssb is specified on line 1 to specify use of the system stack when the function p_ssb( ) is executed. On line 13, #pragma nossb is specified to cancel the specification, that allows the system stack to be used, given by #pragma ssb. Then, when the function p_ssb( ) is called and the stack accessed, the SSB register value will be loaded and a code for accessing the stack is generated. If the function sub_ssb( ) is called, however, the value of the user stack bank register (USB register) will be loaded and a code for accessing the stack is generated.

<Notes>

When #pragma ssb/nossb is specified to generate a code for accessing the system stack, specify a compact or a large model at compilation. If a compact model or large model is not specified, a code for 16-bit addressing will be generated.

10.3.7 Setting the Stack Bank Automatic Identification Function Using #pragma except/noexcept

This section describes #pragma except/noexcept.
A function with #pragma except/noexcept specified loads the value of the stack being used when the stack is accessed and then accesses the stack.

■ Accessing the System Stack Using #pragma except/noexcept

Specifying #pragma except notifies the compiler that the function is operating using the system stack or user stack. This specification identifies the status of the stack being used when the stack is accessed, loads the value of the corresponding stack bank, and then generates a code for accessing the stack.

```
#pragma except
```

Specifying #pragma noexcept cancels the specification that allows the stack bank automatic identification function to be used.

```
#pragma noexcept
```

Figure 10.3-9 Using #pragma except/noexcept to Set and Release the Stack Bank Automatic Identification Function

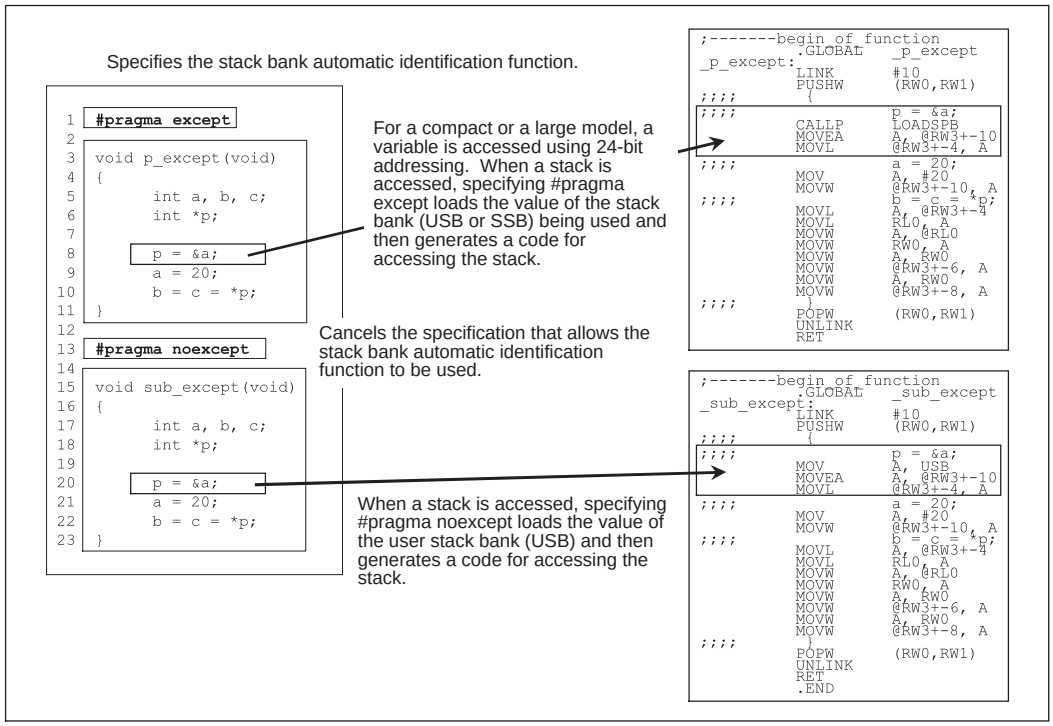


Figure 10.3-9 "Using #pragma except/noexcept to Set and Release the Stack Bank Automatic

Identification Function" shows an example of a variable using #pragma except/noexcept. For a compact or a large model, the variable is accessed using 24-bit addressing. For normal stack access, the user stack bank register (USB register) is used to access the user stack using 24-bit addressing. However, for an exception handler created using REALOS, the stack that is accessed depends on the activation status. In such cases, #pragma except/noexcept is specified to load the value of the stack bank register being used and generate a code for accessing the stack when the stack is accessed.

In this example, #pragma except is specified on line 1. This specification generates a code for automatically identifying the stack bank when the function p_except( ) is executed. On line 13, #pragma noexcept is specified to release specification of the stack bank automatic identification function specified by #pragma except. Then, when the function p_except( ) is called, the status of the stack being used is identified. As a result, the value of the stack bank being used will be loaded and a code for accessing the stack is generated. If the function sub_except( ) is called, however, the value of the user stack bank register (USB register) will be loaded and a code for accessing the stack is generated.

<Notes>

When #pragma except/noexcept is specified to use the automatic identification function for the stack being used, specify a compact model or large model at compilation to generate code for accessing the stack using 24-bit addressing. If a compact or a large model is not specified, a code for 16-bit addressing will be generated.

10.3.8 Generating an Interrupt Vector Table Using #pragma intvect/defvect

This section describes #pragma intvect/defvect.
The #pragma intvect directive is used to generate an interrupt vector table.

■ Generating Interrupt Vector Tables Using #pragma intvect/defvect

The #pragma intvect directive generates an interrupt vector table for setting an interrupt function.

```
#pragma intvect interrupt-function-name vector-number
```

The #pragma defvect directive specifies the function to be mapped to an interrupt vector that has not been specified using #pragma intvect.

```
#pragma defvect interrupt-function-name
```

Figure 10.3-10 Example of Using #pragma intvect

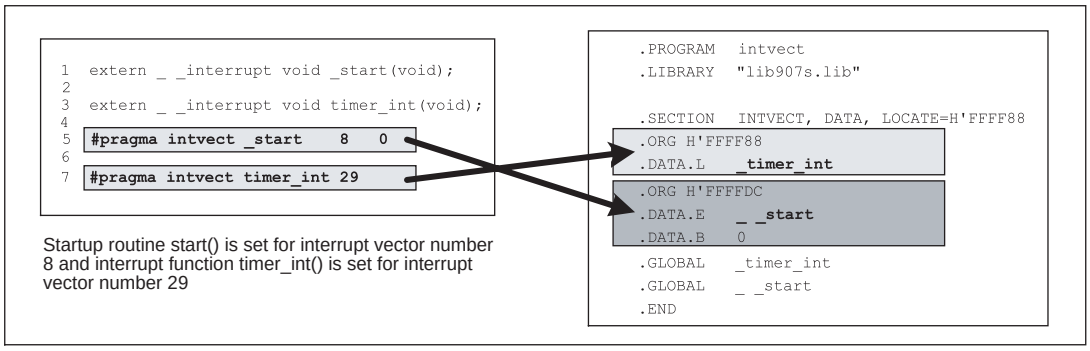


Figure 10.3-10 "Example of Using #pragma intvect" shows an example of using #pragma intvect.

In this example, startup routine start( ) is registered in interrupt vector number 8 and 16-bit reload timer interrupt processing function timer_int( ) is registered in interrupt vector number 29.

Zeros are set for vectors other than vector numbers 8 and 29 of the INTVECT section.

Figure 10.3-11 Example of Using #pragma defvect

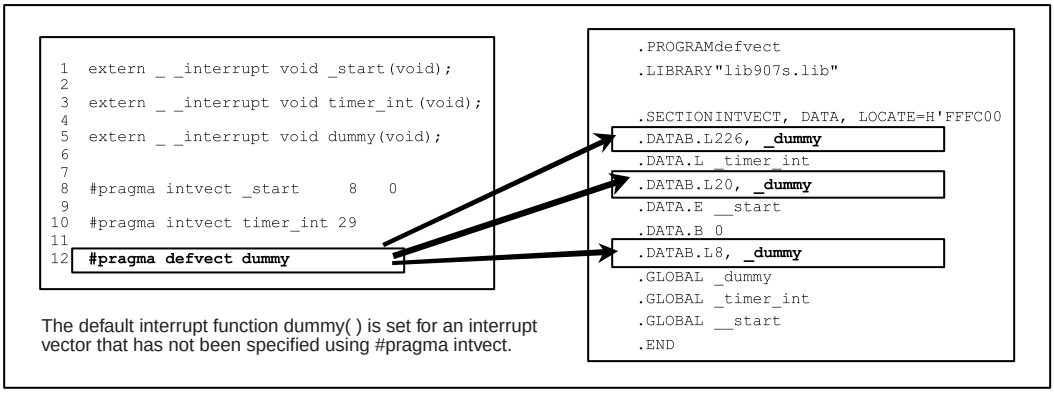


Figure 10.3-11 "Example of Using #pragma defvect" shows an example of using #pragma defvect. In this example, interrupt function dummy( ) has been registered for all vector numbers except 8 and 29, which were specified using #pragma intvect.

See CHAPTER 14 "CREATING AND REGISTERING INTERRUPT FUNCTIONS" for information about the interrupt functions.

<Notes>

Note the following points when using #pragma intvect/defvect to define interrupt vector tables.

Interrupt vector tables defined using #pragma intvect/defvect is output to an independent section named INTVECT mapped into the area beginning with address h’fffc00’. When #pragma defvect is executed, the specified interrupt function is set for all interrupt vectors that have not been specified using #pragma intvect in the INTVECT section.

When #pragma intvect/defvect is specified, define all interrupt vector tables in the same compile unit.

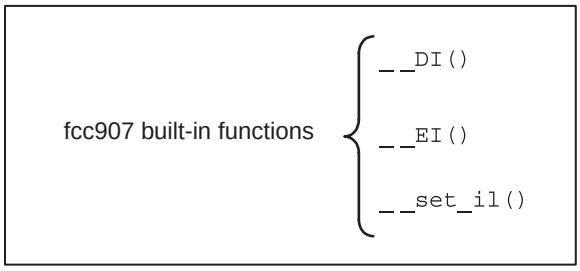
10.4 Interrupt-Related Built-in Functions

This section briefly describes the built-in functions of the fcc907.
The fcc907 provides the following three built-in functions:

- **_ _DI( )**
- **_ _EI( )**
- **_ _set_il( )**

■ Using the Interrupt-Related Built-in Functions to Add Functions

The fcc907 provides the following built-in functions related to interrupt processing:



Sections 10.4.1 "Disabling Interrupts Using _ _DI( )" to 10.4.3 "Setting the Interrupt Level Using _ _set_il( )" provides brief notes on using each of the built-in functions.

10.4.1 Disabling Interrupts Using _ _DI( )

This section describes _ _DI( ), which is used to disable interrupts.
_ _DI( ) is used to disable interrupts in the entire system.

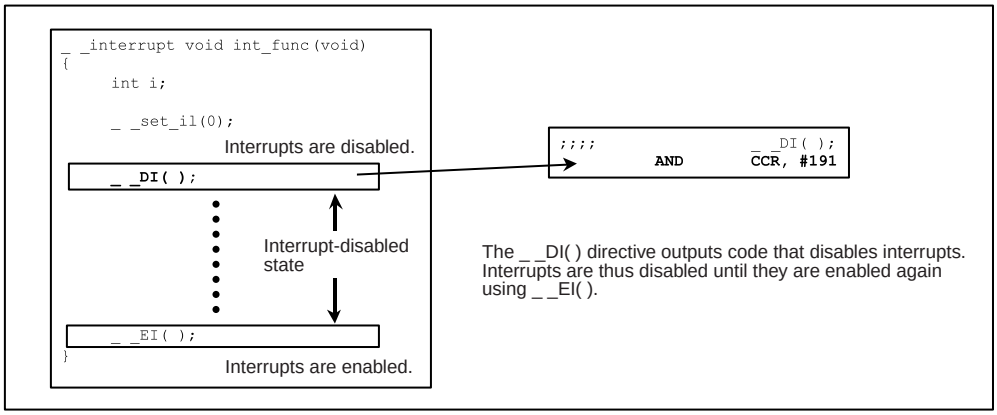
■ Disabling Interrupts Using _ _DI( )

The _ _DI( ) directive expands code that masks interrupts, thereby disabling interrupts in the entire system.

```
void _ _DI(void);
```

Figure 10.4-1 "Using _ _DI( ) to Disable System Interrupts" shows an example of using _ _DI( ) to code a function that disables system interrupts. See CHAPTER 14 "CREATING AND REGISTERING INTERRUPT FUNCTIONS".

Figure 10.4-1 Using _ _DI( ) to Disable System Interrupts



10.4.2 Enabling Interrupts Using _ _EI( )

This section describes _ _EI( ), which is used to enable interrupts.
The _ _EI( ) directive is therefore used to enable interrupts in the entire system.

■ Enabling Interrupts Using _ _EI( )

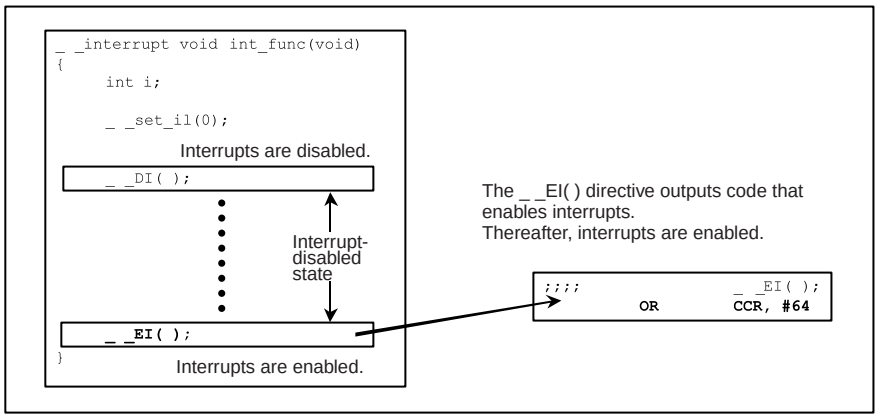
The _ _EI( ) directive expands code that releases masking of interrupts. The _ _EI( ) directive is therefore used to enable interrupts for the entire system.

```
void _ _EI(void);
```

Figure 10.4-2 "Using _ _EI( ) to Enable System Interrupts" shows an example of using _ _EI( ) to code a function that enables system interrupts.

See CHAPTER 14 "CREATING AND REGISTERING INTERRUPT FUNCTIONS" for information about interrupt processing.

Figure 10.4-2 Using _ _EI( ) to Enable System Interrupts



10.4.3 Setting the Interrupt Level Using _ _set_il( )

This section briefly describes how to set the interrupt level using _ _set_il( ). The _ _set_il( ) directive is used to change the interrupt level of the entire system during execution of a function.

■ Setting the Interrupt Level Using _ _set_il( )

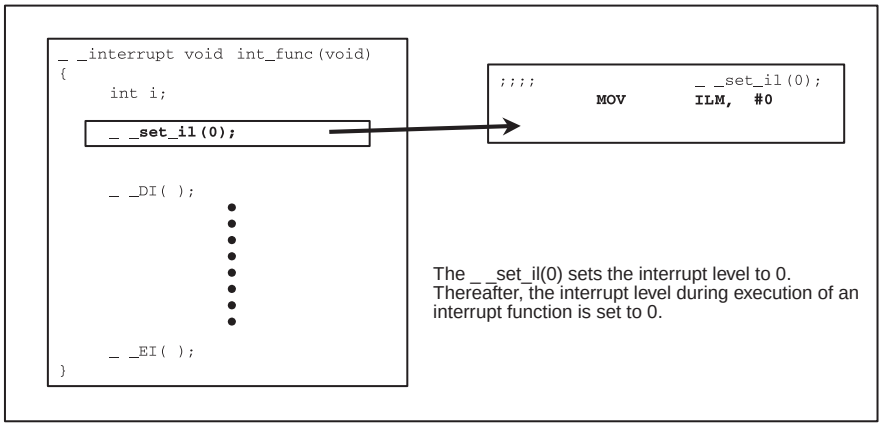
The _ _set_il( ) directive expands code that sets the interrupt level. You can therefore use this directive to determine the allowed interrupt level for the entire system.

```
void _ _set il(interrupt-level);
```

Figure 10.4-3 "Using _ _set_il( ) to Set the System Interrupt Level" shows an example of using _ _set_il( ) to code a function that sets the interrupt level for the entire system.

See CHAPTER 14 "CREATING AND REGISTERING INTERRUPT FUNCTIONS" for information about interrupt processing.

Figure 10.4-3 Using _ _set_il( ) to Set the System Interrupt Level



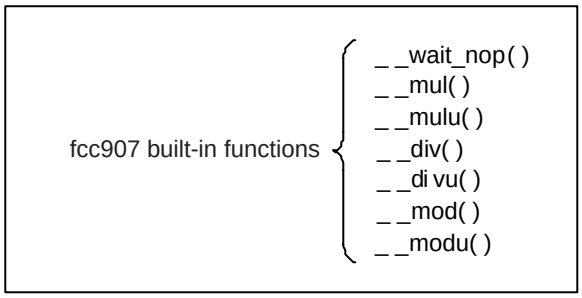
10.5 Other Built-in Functions

This section briefly describes the other built-in functions provided by the fcc907.
The fcc907 provides the following seven built-in functions:

- **_ _wait_nop( )**
- **_ _mul( )**
- **_ _mulu( )**
- **_ _div( )**
- **_ _divu( )**
- **_ _mod( )**
- **_ _modu( )**

■ Other Additional Built-in Functions

The fcc907 provides the following built-in functions not related to interrupt processing:



Sections 10.5.1 "Outputting a nop Instruction Using _ _wait_nop( )" to 10.5.7 "Unsigned 32-Bit/ Unsigned 16-Bit Remainder Calculation Using _ _modu( )" provides brief notes on using each of the built-in functions.

10.5.1 Outputting a Nop Instruction Using _ _wait_nop( )

This section briefly describes the expansion of a nop instruction using _ _wait_nop( ). The _ _wait_nop( ) is used to expand a single nop instruction at the location of the function call.

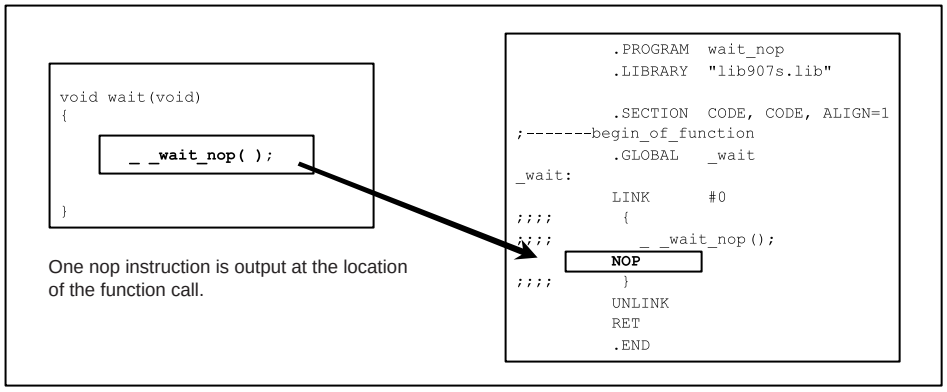
■ Outputting a nop Instruction Using _ _wait_nop( )

The _ _wait_nop( ) expands one nop instruction at the location of the function call. Code the _ _wait_nop( ) at which a nop instruction is required.

```
void _ _wait_nop(void);
```

Figure 10.5-1 "Using _ _wait_nop( ) to Output a Nop Instruction" shows an example of coding a function that uses _ _wait_nop( ).

Figure 10.5-1 Using _ _wait_nop( ) to Output a Nop Instruction



<Notes>

The fcc907 outputs one nop instruction at the location where _ _wait_nop( ) is coded. Code the _ _wait_nop( ) at which a nop instruction is required.

If the _ _asm statement is used to code a nop instruction, the various optimization operations can be suppressed.

Coding _ _wait_nop( ) can control the timing so as to minimize the side effects of optimization.

10.5.2 Signed 16-Bit Multiplication Using _ _mul( )

This section briefly describes signed 16-bit multiplication using _ _mul( ). The _ _mul( ) is used to return the result of (signed 16 bits) x (signed 16 bits) operations as signed 32 bits.

■ Signed 16-Bit Multiplication Using _ _mul( )

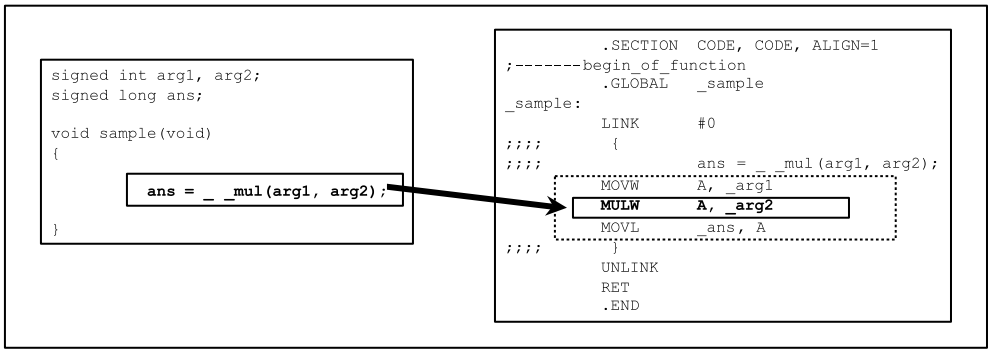
The _ _mul( ) executes multiplication operations of (signed 16 bits) x (signed 16 bits) = (signed 32 bits). The _ _mul( ) can be used to prevent an overflow of 16-bit operations.

```
signed long _ _mul(signed int, signed int);
```

This built-in function is enabled only when the MB number of the F^{2}MC-16LX/16F series has been specified using the -CPU option.

Figure 10.5-2 "Signed 16-Bit Multiplication Using _ _mul( )" shows an example of coding a function that uses _ _mul( ).

Figure 10.5-2 Signed 16-Bit Multiplication Using _ _mul( )



10.5.3 Unsigned 16-Bit Multiplication Using _ _mulu( )

This section briefly describes unsigned 16-bit multiplication using _ _mulu( ). The _ _mulu( ) is used to return the result of (unsigned 16 bits) x (signed 16 bits) operations as unsigned 32 bits.

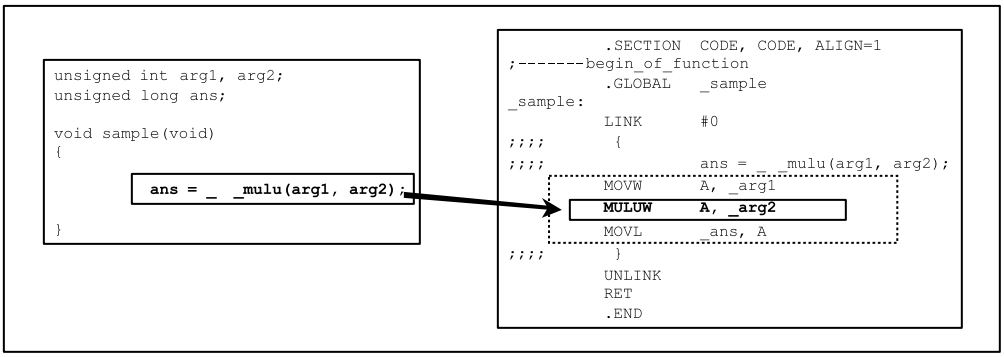
■ Unsigned 16-Bit Multiplication Using _ _mulu( )

The _ _mulu( ) executes multiplication operations of (unsigned 16 bits) x (unsigned 16 bits) = (unsigned 32 bits). The _ _mulu( ) can be used to improve the efficiency of 16-bit operations.

```
unsigned long _ _mulu(unsigned int, unsigned int);
```

Figure 10.5-3 "Unsigned 16-Bit Multiplication Using _ _mulu( )" shows an example of coding a function that uses _ _mulu( ).

Figure 10.5-3 Unsigned 16-Bit Multiplication Using _ _mulu( )



10.5.4 Signed 32-Bit/Signed 16-Bit Division Using _ _div( )

This section briefly describes signed 32-bit/signed 16-bit division using _ _div( ). The _ _div( ) is used to return the result of (signed 32 bits)/(signed 16 bits) operations as signed 16 bits.

■ Signed 32-Bit/Signed 16-Bit Division Using _ _div( )

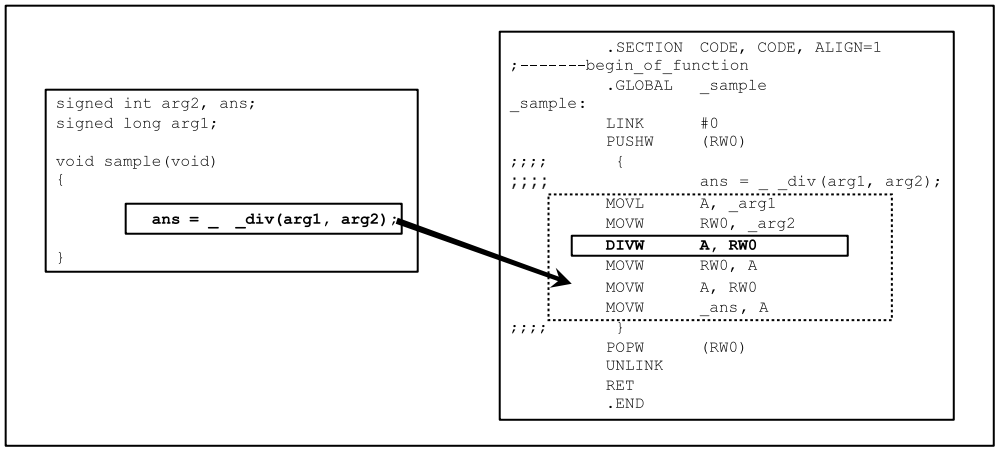
The _ _div( ) executes division operations of (signed 32 bits)/(signed 16 bits) = (signed 16 bits). The _ _div( ) can be used to improve the efficiency of 32-bit operations.

```
signed int _ _div(signed long, signed int);
```

This built-in function is enabled only when the MB number of the F^{2}MC-16LX/16F series has been specified using the -CPU option.

Figure 10.5-4 "Signed 32-Bit/Signed 16-Bit Division Using _ _div( )" shows an example of coding a function that uses _ _div( ).

Figure 10.5-4 Signed 32-Bit/Signed 16-Bit Division Using _ _div( )



10.5.5 Unsigned 32-Bit/Unsigned 16-Bit Division Using _ _divu( )

This section briefly describes unsigned 32-bit/unsigned 16-bit division using _ _divu( ). The _ _divu( ) is used to return the result of (unsigned 32 bits)/(unsigned 16 bits) operations as unsigned 16 bits.

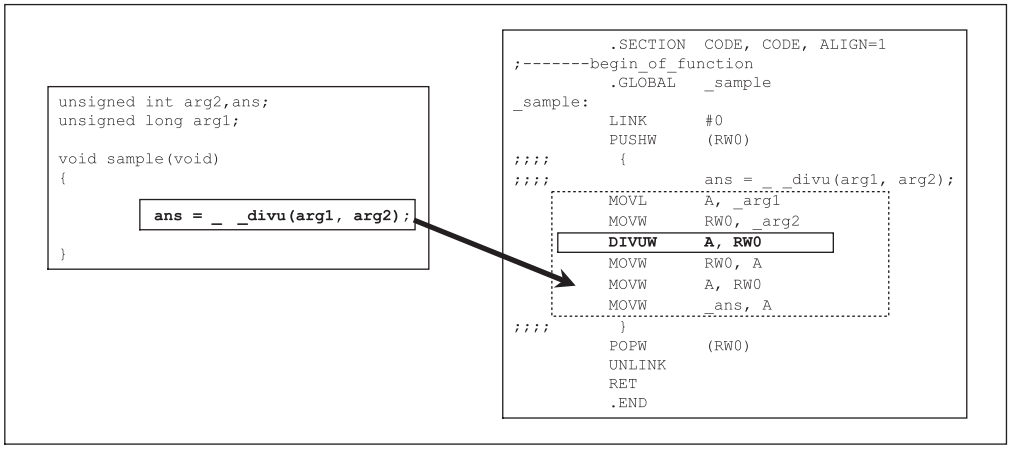
■ Unsigned 32-Bit/Unsigned 16-Bit Division Using _ _divu( )

The _ _divu( ) executes division operations of (unsigned 32 bits)/(unsigned 16 bits) = (unsigned 16 bits). The _ _divu( ) can be used to improve the efficiency of 32-bit operations.

```
unsigned int _ _divn(unsigned long, unsigned int);
```

Figure 10.5-5 "Unsigned 32-Bit/Unsigned 16-Bit Division Using _ _divu( )" shows an example of coding a function that uses _ _divu( ).

Figure 10.5-5 Unsigned 32-Bit/Unsigned 16-Bit Division Using _ _divu( )



10.5.6 Signed 32-Bit/Signed 16-Bit Remainder Calculation Using _ _mod( )

This section briefly describes the remainder of signed 32-bit/signed 16-bit division using _ _mod( ).
The _ _mod( ) is used to return the remainder of (signed 32 bits)/(signed 16 bits) operations as signed 16 bits.

■ Signed 32-Bit/Signed 16-Bit Remainder Calculation Using _ _mod( )

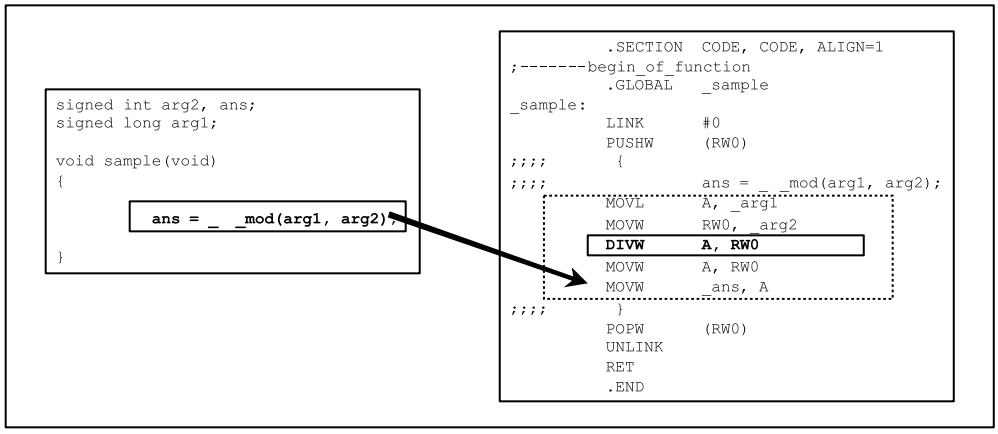
The _ _mod( ) returns the remainder of the result of (signed 32 bits)/(signed 16 bits) operations as signed 16 bits. The _ _mod( ) can be used to improve the efficiency of 32-bit operations.

```
signed int _ _mod(signed long, signed int);
```

This built-in function is enabled only when the MB number of the F^{2}MC-16LX/16F series has been specified using the -CPU option.

Figure 7.1-3 "Signed 32-Bit/Signed 16-Bit Remainder Calculation Using _ _mod( )" shows an example of coding a function that uses _ _mod( ).

Figure 10.5-6 Signed 32-Bit/Signed 16-Bit Remainder Calculation Using _ _mod( )



10.5.7 Unsigned 32-Bit/Unsigned 16-Bit Remainder Calculation Using _ _modu( )

This section briefly describes the remainder of unsigned 32-bit/unsigned 16-bit division using _ _modu( ).
The _ _modu( ) is used to return the remainder of (unsigned 32 bits)/(unsigned 16 bits) operations as unsigned 16 bits.

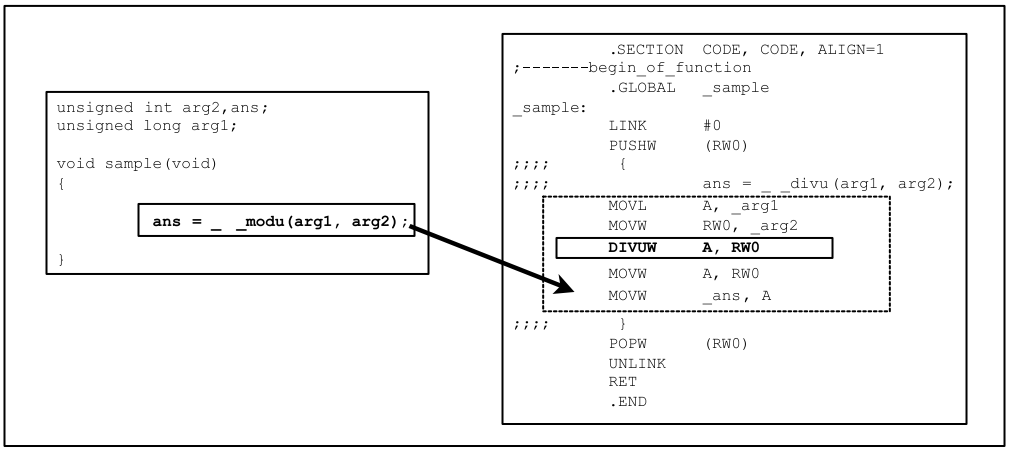
■ Unsigned 32-Bit/Unsigned 16-Bit Remainder Calculation Using _ _modu( )

The _ _modu( ) returns the remainder of the result of (unsigned 32 bits)/(unsigned 16 bits) operations as unsigned 16 bits. The _ _modu( ) can be used to improve the efficiency of 32-bit operations.

```
unsigned int _ _modu(unsigned long, unsigned int);
```

Figure 10.5-7 "Unsigned 32-Bit/Unsigned 16-Bit Remainder Calculation Using _ _modu( )" shows an example of coding a function that uses _ _modu( ).

Figure 10.5-7 Unsigned 32-Bit/Unsigned 16-Bit Remainder Calculation Using _ _modu( )



CHAPTER 11 NOTES ON ASSEMBLER PROGRAM IN C PROGRAMS

This chapter provides notes on including Assembler program in C programs.

11.1 Including Assembler Code in C Programs

This section briefly describes how to code assembler program modules. The _ _asm statement can code only one assembly language instruction. The #pragma asm/endasm can code multiple assembly language instructions.

■ Coding Assembler Programs

Assembler source programs consist of the following fields:

Symbol field	Instruction field	Operand field	Comment field	Line-feed field

The assembler executes the code assuming that the character string coded starting in column 2 is an instruction. An Assembler instruction character string coded in a C source program will be output as is to an assembly source file output by the C compiler. Therefore, a tab code or null character string is required at the beginning of the character string.

As shown in Table 11.1-1 "Coding Assembler Programs", the fcc907 can use the _ _asm statement or #pragma asm/endasm to include Assembler program in C programs.

Table 11.1-1 Coding Assembler Programs

Function	Coding method
_ _asm statement	Only one Assembler instruction can be coded per _ _asm statement.
#pragma asm/endasm	More than one Assembler instructions can be coded.

As listed in Table 11.1-2 "Location for Including Assembler Programs", coding can also be divided into coding outside or inside a function based on the coding location in the C program.

Table 11.1-2 Location for Including Assembler Programs

Coding location	Explanation
Coding inside a function	Assembler instructions are coded as part of the function.
Coding outside a function	Because the Assembler instructions are expanded as an independent section, they must be defined in the section using a section definition pseudo-instruction.

■ Accessing Variables and Functions Defined in C Programs from Assembler Programs

The names of external variables or functions defined in a C program are output as symbols with an underscore attached as the result of compilation. When variables or functions defined in a C program are referenced from an assembler program, the variables or functions are referenced with the underscore attached.

Figure 11.1-1 "Referencing Variables in a C program from an Assembler Program" shows an example of referencing variables defined in a C program from an assembler program. In this example, the external variables a and b have been defined in the C program. In function func1( ), the variable b is referenced as _b from the assembler program coded using #pragma asm/ endasm.

Figure 11.1-1 Referencing Variables in a C program from an Assembler Program

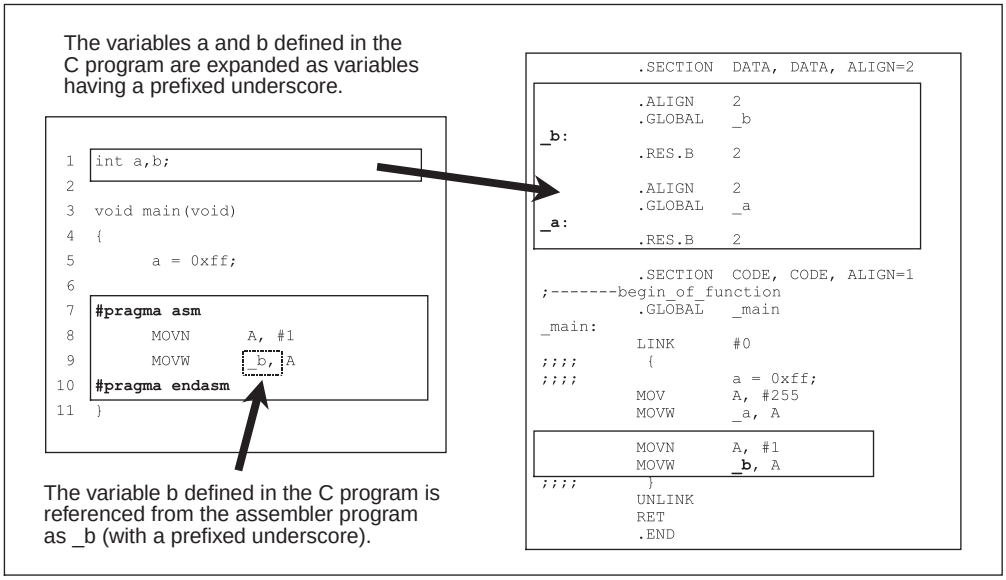
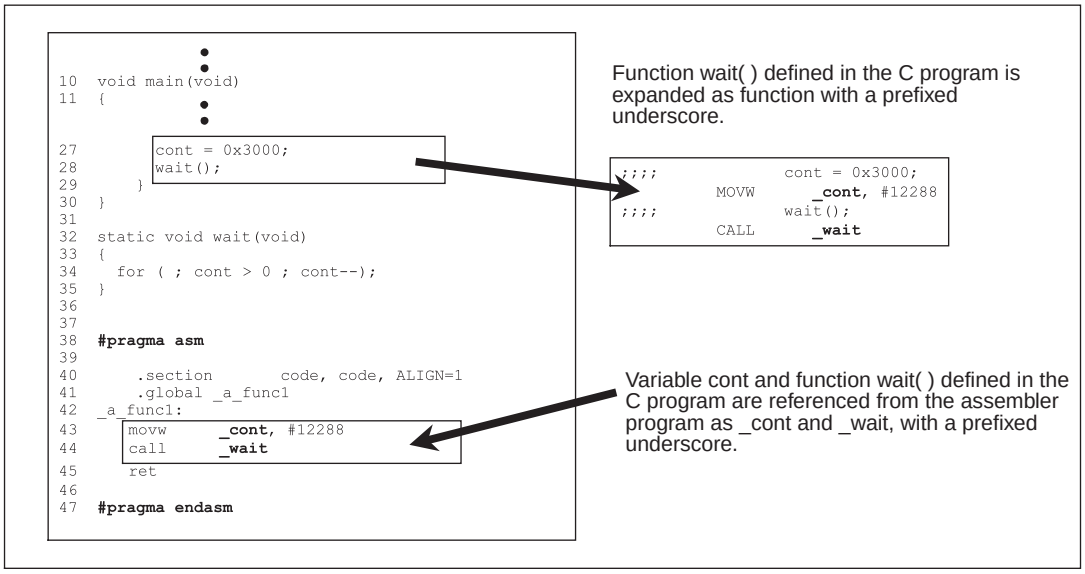


Figure 11.1-2 "Referencing a Variable and a Function in a C program from an Assembler Program" shows an example of referencing a function and a variable defined in a C program from an assembler program. In this example, function wait( ) is called after a value is assigned to variable cont outside the function in the C program. Variable cont and function wait( ) are referenced from the assembler program as _cont and _wait that have a prefixed underscore.

Figure 11.1-2 Referencing a Variable and a Function in a C Program from an Assembler Program



<Notes>

Note the following points when using the _ _asm statement or #pragma asm/endasm to include Assembler code in a C program:

- When using the _ _asm statement to code Assembler instructions, always include a tab code or null character string at the beginning of the character string.
- The accumulator (A) register can be used unconditionally. To use another register, save and restore the register (this is to be performed by the user).
- Include only one Assembler instruction per _ _asm statement.
- If several Assembler instructions are included, use either as many _ _asm statements as there are Assembler instructions, or use #pragma asm/endasm.
- If an _ _asm statement or #pragma asm/endasm is coded in a C program, optimization by specifying "-O" for compilation may be suppressed.
- The fcc907 does not check Assembler code for errors. If an Assembler instruction coded in an _ _asm statement or #pragma asm/endasm contains an error, the assembler will output an error message. Refer to the assembler manual for information about Assembler coding.

[Tip]

Softune C Checker:

The Softune C Checker will output a warning when Assembler instructions are included using the _ _asm statement or #pragma asm/endasm. The fcc896, fcc907, and fcc911 support the _ _asm statement and #pragma asm/endasm functions. However, the registers and instruction sets that can be used depend on the architecture. This check function is useful for identifying locations that can be rewritten for porting from the fcc896 or fcc911 to the fcc907.

11.2 Differences Between Using the _ _asm Statement and #pragma asm/endasm

This section briefly describes the differences between using the _ _asm statement and #pragma asm/endasm.
For including only one Assembler instruction in a function, use the _ _asm statement.

■ Including an Assembler Program Having Multiple Instructions in a Function

As listed in Table 11.1-1 "Coding Assembler Programs", an _ _asm statement can contain only one Assembler instruction. However, #pragma asm/endasm can contain several Assembler instructions at a time.

Figure 11.2-1 "Using the _ _asm Statement to Include Assembler Program in a Function" shows an example of using the _ _asm statement to include two Assembler instructions in a function.

Figure 11.2-1 Using the _ _asm Statement to Include Assembler Program in a Function

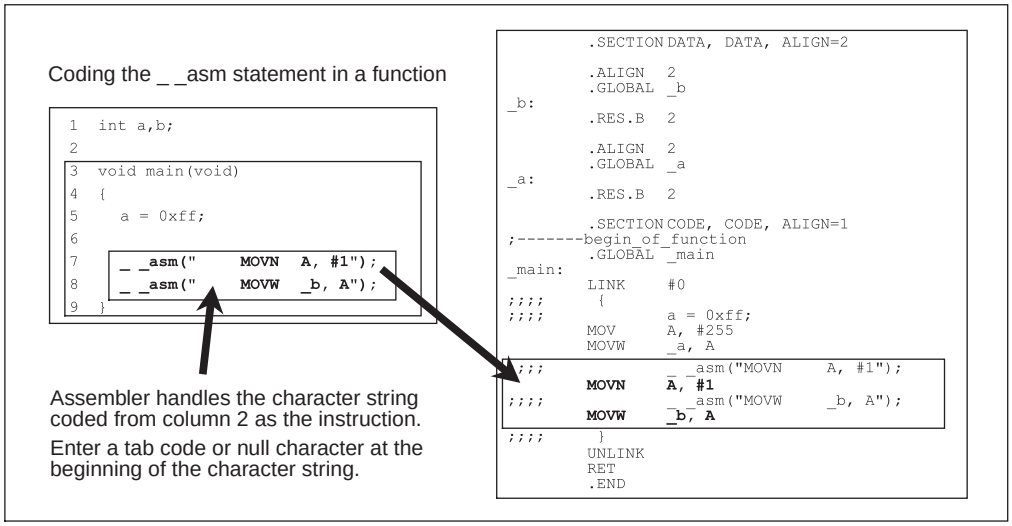
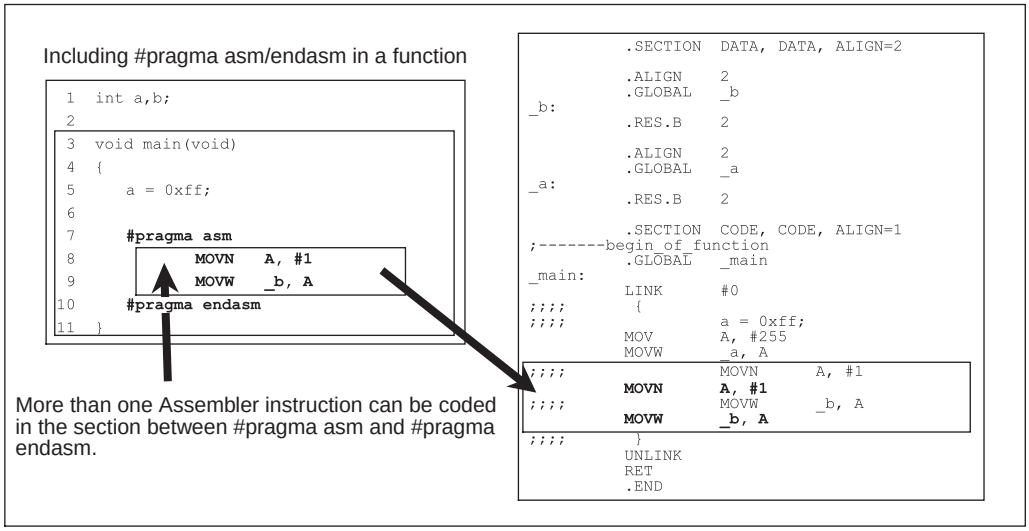


Figure 11.2-2 "Using #pragma asm/endasm to Include Assembler Programs in a Function" shows an example how the same function can be rewritten using #pragma asm/endasm.

These two examples are almost identical. However, when only one Assembler instruction is to be included in a function, we recommend to use the _ _asm statement.

Figure 11.2-2 Using #pragma asm/endasm to Include Assembler Programs in a Function



■ Coding an Assembler Program Outside a Function

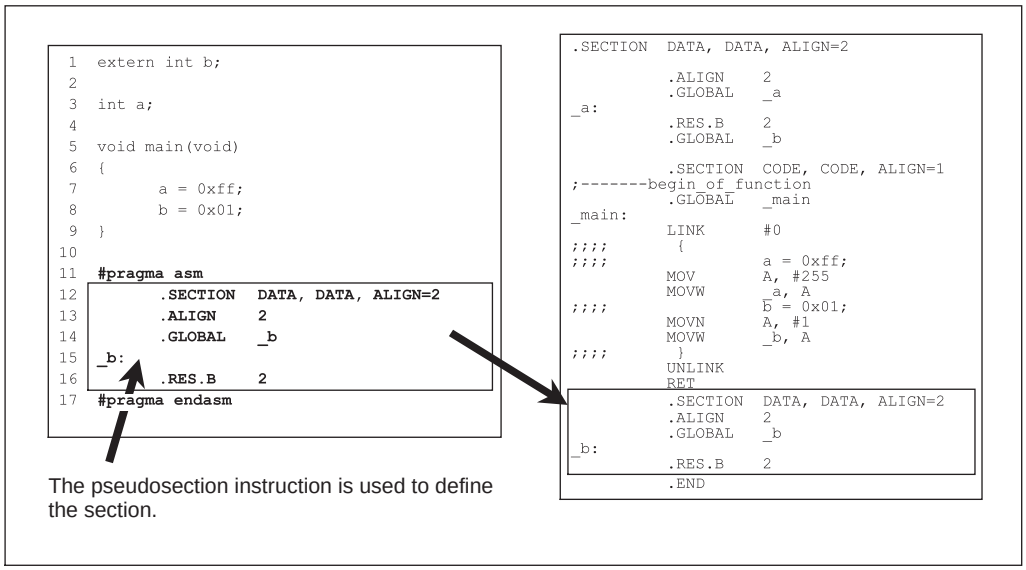
When an assembler program is coded outside a function, the coded assembler program is expanded as an independent section. To code an assembler program outside a function, use a pseudo-instruction for defining the section. If the section has not been defined, operation of the coded Assembler instructions will be unpredictable.

Figure 11.2-3 "Using #pragma asm/endasm to Code Outside a Function" shows an example of a function where #pragma asm/endasm is coded outside the function.

In this example, pseudo-instruction for defining the section is used outside the function to define the 2-byte symbol _b for the assembler. This symbol is accessed by the C function func1( ) as variable b of type int.

When coding an assembler program outside a function, use #pragma asm/endasm.

Figure 11.2-3 Using #pragma asm/endasm to Code Outside a Function



[Tip]

Softune C Checker:

The Softune C Checker will output a warning when Assembler instructions are coded using

_ _asm statement or #pragma asm/endasm. The fcc896, fcc907, and fcc911 support the

_ _asm statement and #pragma asm/endasm. However, the registers and instruction sets that can be used depend on the architecture. This check function is useful for identifying locations that can be rewritten from the fcc896 or fcc911 to the fcc907.

CHAPTER 12 NOTES ON DEFINING AND ACCESSING THE I/O AREA

This chapter describes the definition and accessing of resources mapped into the I/O area. The chapter uses as examples the I/O area of the MB90678 series of microcontrollers, which belong to the F^{2}MC-16 family of microcontrollers, to explain how resources mapped into the I/O area are defined and accessed.

12.1 M90678 Series I/O Areas

This section briefly describes the I/O areas of the F^{2}MC-16 Family.

For the F^{2}MC-16 Family, the area between addresses h'0000 and h'00bf of bank h'00 is used as the I/O area.

■ F^{2}MC-16 Family Memory Mapping

Figure 12.1-1 "F^{2}MC-16 Family Memory Mapping" shows memory mapping in the MB90678 series.

For the F^{2}MC-16 Family, the area between addresses h'0000 and h'00bf of bank h'00 is used as the I/O area. Each resource register is mapped into this area. The internal RAM area starts from address h'0100 of bank h'00. The size of the internal RAM area depends on the model. For more information, refer to the manual of the model being used.

Figure 12.1-1 F^{2}MC-16 Family Memory Mapping

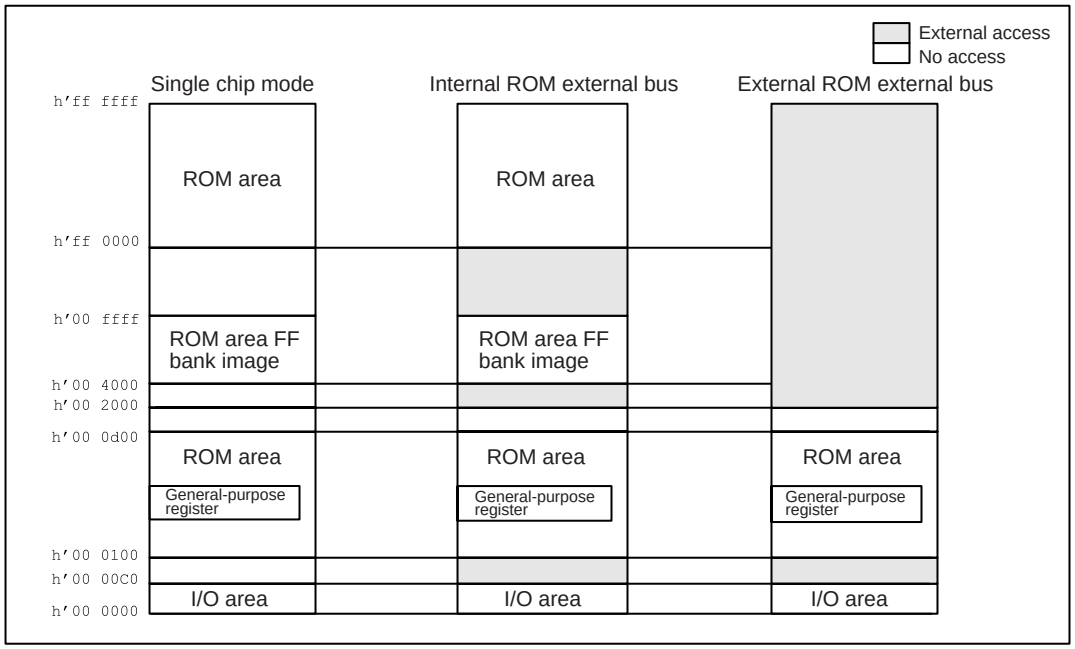


Figure 12.1-2 "MB90670/675 Series I/O Register Mapping" lists the resource registers mapped between addresses h'0000 and h'00bf of the MB90678. For details on the registers, refer to the hardware manual.

Figure 12.1-2 MB90670/675 Series I/O Register Mapping

h'005e	CCR11		OCU1
h'005c	CCR10		
h'005a	CCR01		OCU0
h'0058	CCR00		
h'0056	TCRH		24-bit free run timer
h'0054	TCRL		
h'0052	ICC		ICU
h'0050	TCCR		24-bit free run timer
h'0044	IDAR		IIC bus IF
h'0042	ICCR	IADR	
h'0040	IBSR	IBCR	
h'003e	TMR0/TMRLR0		16-bit reload timer 1
h'003c	TMCSR0		
h'003a	TMR0/TMRLR0		16-bit reload timer 0
h'0038	TMCSR0		
h'0036	PRL1		PPG1
h'0034	PRL0		PPG0
h'0032			
h'0030	PPG0	PPG1	PPG0/PPG1
h'002e	ADCR		
h'002c	ADCS		
h'002a	ELVR		DTP/external interrupt
h'0028	ENIR	EIRR	
h'0026	SIDR1/SODR1	SSR1	UART1
h'0024	SMR1	SCR1	
h'0022	UIDR0/UODR0	URD0	UART0
h'0020	UMC0	USR0	
h'001e		EICR	Wake-up interrupt
h'001a	PDDA	PDDB	Port direction register
h'0018	PDD8	PDD9	
h'0016	PDD6	PDD7	
h'0014	PDD4	ADER	Port four direction register /analog input enable register
h'0012	PDD2	PDD3	Port direction register
h'0010	PDD0	PDD1	
h'000e		EIFR	Wake-up interrupt
h'000a	PDRA	PDRB	Port data register
h'0008	PDR8	PDR9	
h'0006	PDR6	PDR7	
h'0004	PDR4	PDR5	
h'0002	PDR2	PDR3	
h'0000	PDR0	PDR1	

h'00be	ICR14	ICR15	Interrupt controller
h'00bc	ICR12	ICR13	
h'00ba	ICR10	ICR11	
h'00b8	ICR08	ICR09	
h'00b6	ICR06	ICR07	
h'00b4	ICR04	ICR05	
h'00b2	ICR02	ICR03	
h'00b0	ICR00	ICR01	
h'00a8	WDTC	TBTC	Watchdog timer /time-base timer
h'00a6	HACR	EPCR	External pin
h'00a4		ARSR	
h'00a2			
h'00a0	IBSR	IBCR	Low-power consumption
h'009e		DIRR	Delayed interrupt occurrence module
	System reserved area		
h'008e	CPR07H		ICU1
h'008c	CPR07L		
h'008a	CPR06H		
h'0088	CPR06L		
h'0086	CPR05H		
h'0084	CPR05L		
h'0082	CPR04H		
h'0080	CPR04L		
h'007e	CPR03H		ICU0
h'007c	CPR03L		
h'007a	CPR02H		
h'0078	CPR02L		
h'0076	CPR01H		
h'0074	CPR01L		
h'0072	CPR00H		
h'0070	CPR00L		
h'006e	ICR3H		ICU
h'006c	ICR3L		
h'006a	ICR2H		
h'0068	ICR2L		
h'0066	ICR1H		
h'0064	ICR1L		
h'0062	ICR0H		
h'0060	ICR0L		

12.2 Defining and Accessing Variables Mapped into the I/O Area

This section describes how to define and access the I/O area of the MB90678.

■ Operations for Accessing I/O Area Registers as Variables from C Programs

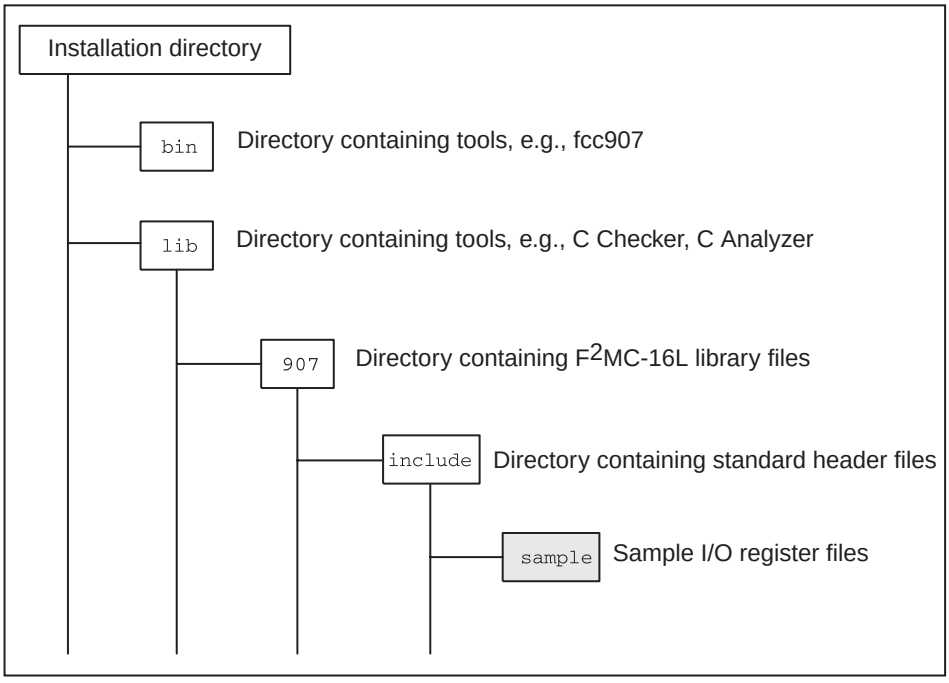
Basically, the following operations are required to access the registers in the I/O area as variables from a C program:

1. Use #pragma section to specify the mapping address of the I/O area.
2. Specify the _ _io type qualifier to define a variable to be mapped into the area.
3. Specify the _ _io type qualifier to declare access to the variable mapped into the I/O area.

■ Sample I/O Register Files Provided by the fcc907

When the fcc907 is installed, files required for defining and accessing an I/O register are created in the directories shown in Figure 12.2-1 "Directories Containing the Sample I/O Files". This section uses an example of the MB90678 series to describe the method used for defining and accessing the I/O area.

Figure 12.2-1 Directories Containing the Sample I/O Files



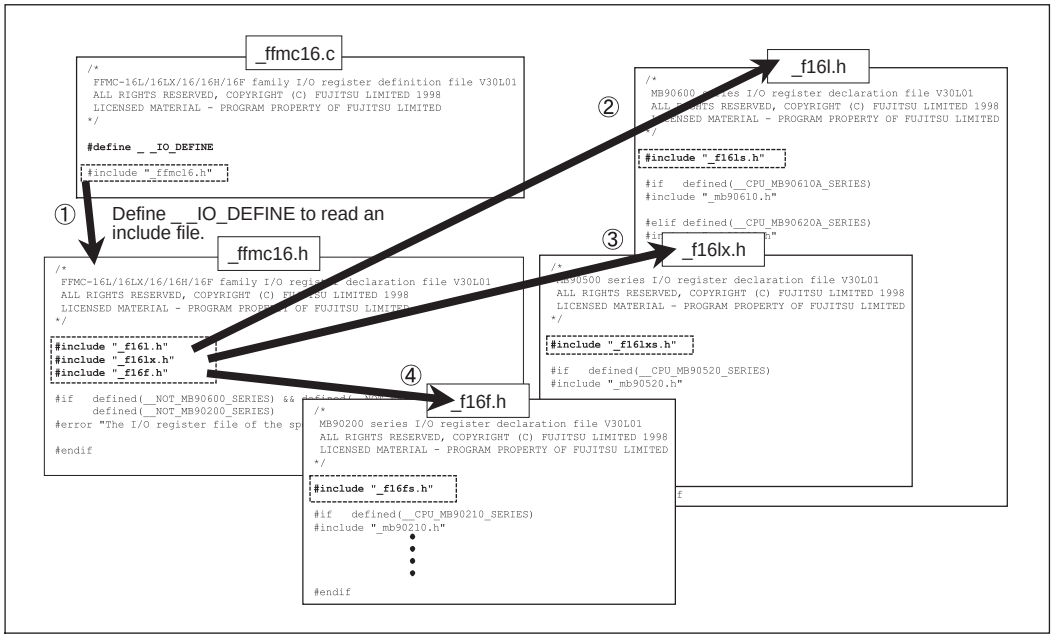
■ Defining the MB90678 I/O Registers

All I/O registers of the MB90678 hardware can be defined by specifying the following option for compilation of the files in the directories containing the sample I/O files:

```
fcc907s -cpu mb90678 -c *.c
```

The MB number specified by the -CPU option for compilation has already been defined in the predefined macro _ _CPU_MB number_ _. In the examples given below, _ _CPU_MB90675_ _ is defined. The number is used to select the required files and define the I/O area.

Figure 12.2-2 Defining Variables Mapped into the I/O Area (1)



In definition file _ffmc16.c, proceed as follows:

① Use #define to define _ _IO_DEFINE and include _ffmc16.h.

In _ffmc16.h, proceed as follows:

② Include _f16l.h.

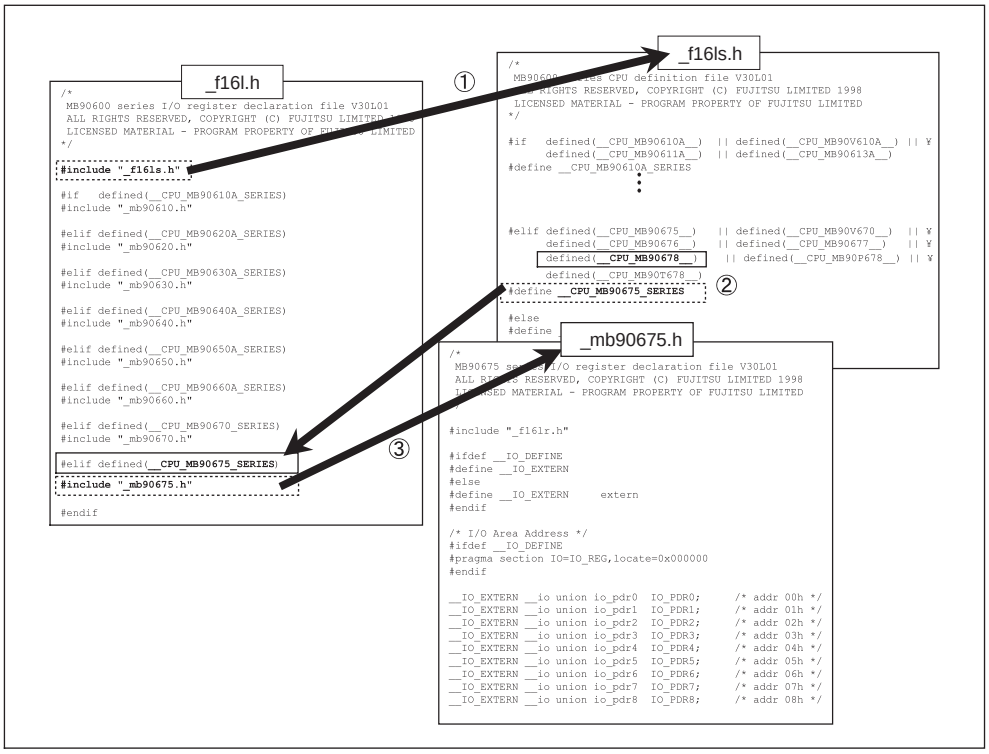
③ Include _f16lx.h.

④ Include _f16f.h.

In _f16l.h, use the predefined macro _ _CPU_MB90678_ _ to define the predefined macros of the series.

Because the _f16lx.h is a definition file for the 16lx series, and the _f16f.h is a definition file for the 16f series, these files are read only.

Figure 12.2-3 Defining the Variables Mapped into the I/O Area (2)



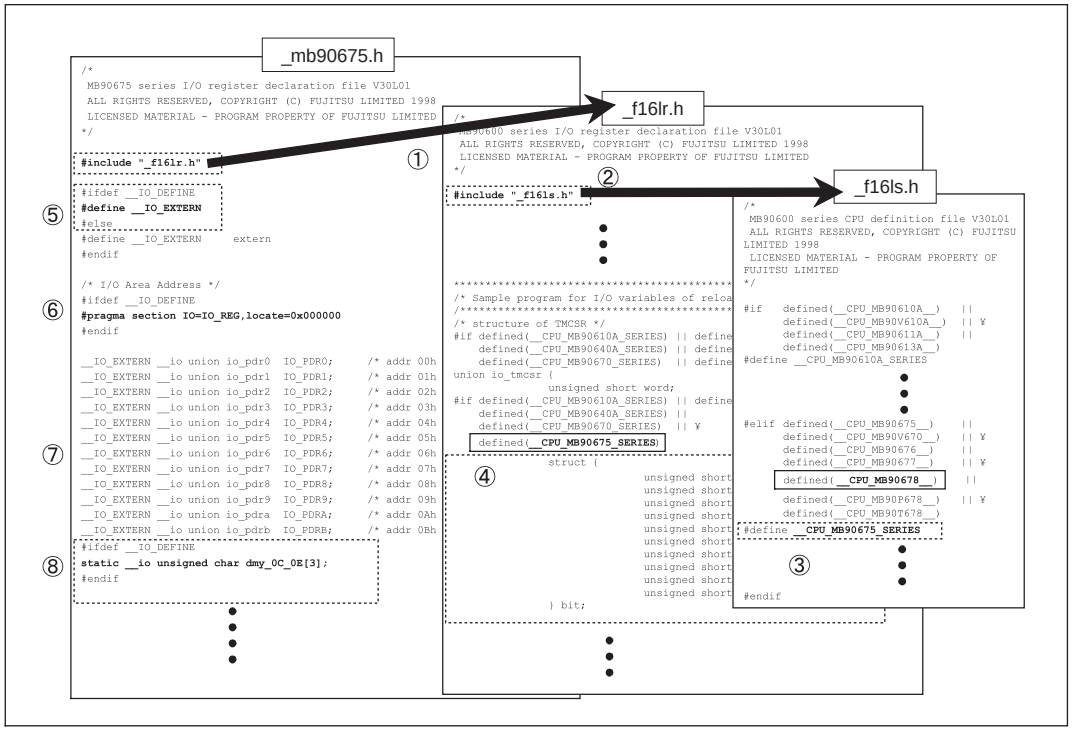
In _f16l.h, proceed as follows:

① Include _f16ls.h.

② In _f16ls.h, use the predefined macro _ _CPU_MB90678_ _ to define the predefined macro _ _CPU_MB90675_SERIES.

③ Use the predefined macro _ _CPU_MB90675_SERIES defined in _f16ls.h to include _mb90675.h.

Figure 12.2-4 Defining Variables Mapped into the I/O Area (3)



In _mb90675.h, proceed as follows:

① Include _f16lr.h.
② In _f16lr.h, include _f16ls.h.
③ In _f16ls.h, use the predefined macro _ _CPU_MB90678_ _ to define the predefined macro _ _CPU_MB90675_SERIES.
④ In _f16lr.h, use the predefined macro _ _CPU_MB90675_SERIES defined in _f16ls.h to define the required type specific to the MB90675 series.
⑤ Because _ _IO_DEFINE has been defined in _ffmc16.c, use #define to define a macro that replaces _ _IO_EXTERN with blanks.
⑥ Because _ _IO_DEFINE has been defined, use #pragma section to map the IO_REG section starting from address 0x0000.
⑦ Specify _ _IO_EXTERN and the _ _io type qualifier to define the I/O register variables specific to the MB90675 series mapped between addresses 0x0000 and 0x00bf.
⑧ Specify the static declaration and the _ _io type qualifier in an area having no I/O registers to allocate a dummy area that cannot be accessed by other functions.

■ Accessing the MB90678 I/O Registers

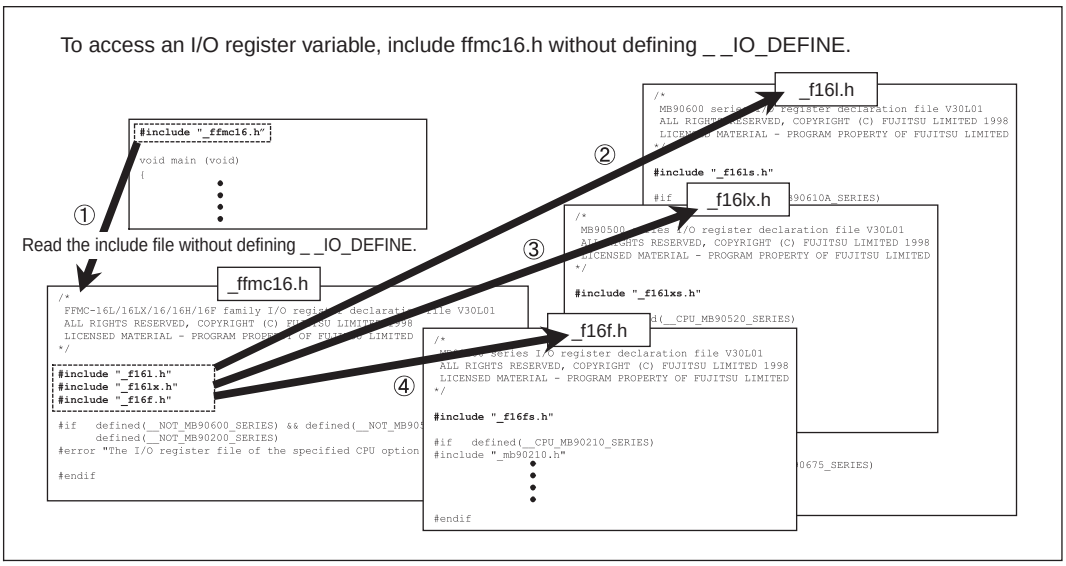
To access the registers mapped into the I/O area, include _ffmc16.h. Do not define

_ _IO_DEFINE using #define (see (1) in Figure 12.2-5 "Accessing Variables Mapped into the I/O Area (1)"). The following describes the access declaration when the MB90678 is used.

The MB number to specified in the -CPU option for compilation is defined in defined macro

_ _CPU_MB number_ _. In the examples shown below, _ _CPU_MB90675 is defined. With this definition, the required files are selected and access to the I/O area is declared.

Figure 12.2-5 Accessing Variables Mapped into the I/O Area (1)

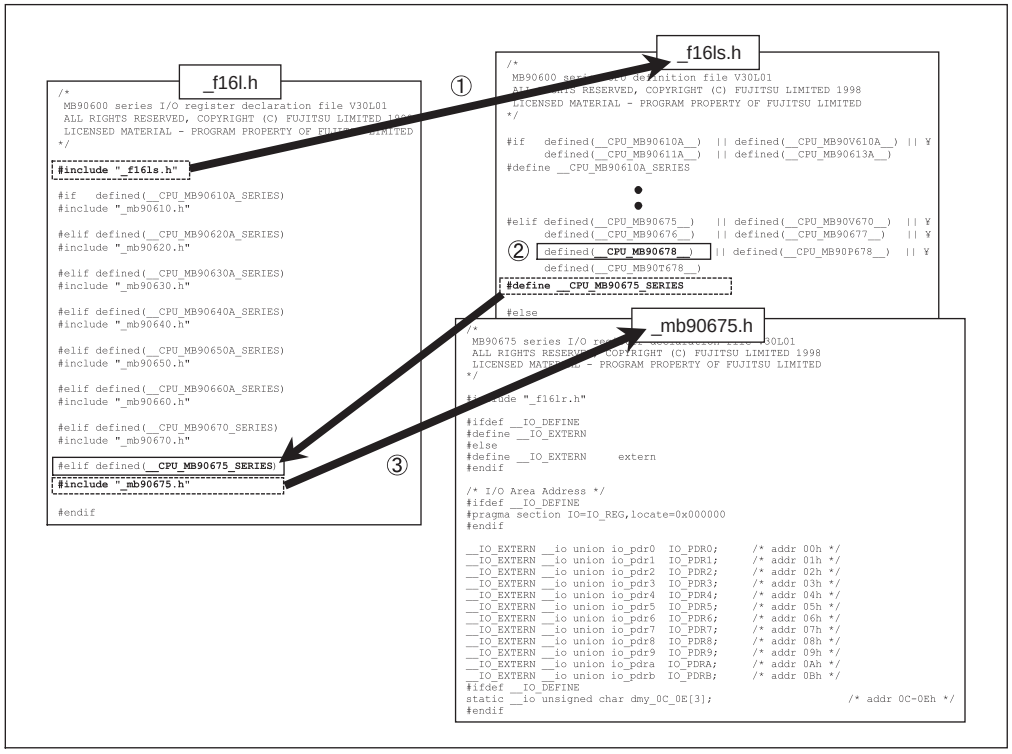


In _ffmc16.h, proceed as follows:

② Include _f16l.h.
③ Include _f16lx.h.
④ Include _f16f.h.

In _f16l.h, use the predefined macro _ _CPU_MB90678_ _ to define the predefined macros of the series.
Because the _f16lx.h is a definition file for the 16lx series, and the _f16f.h is a definition file for the 16f series, these files are read only.

Figure 12.2-6 Accessing Variables Defined in the I/O Area (2)



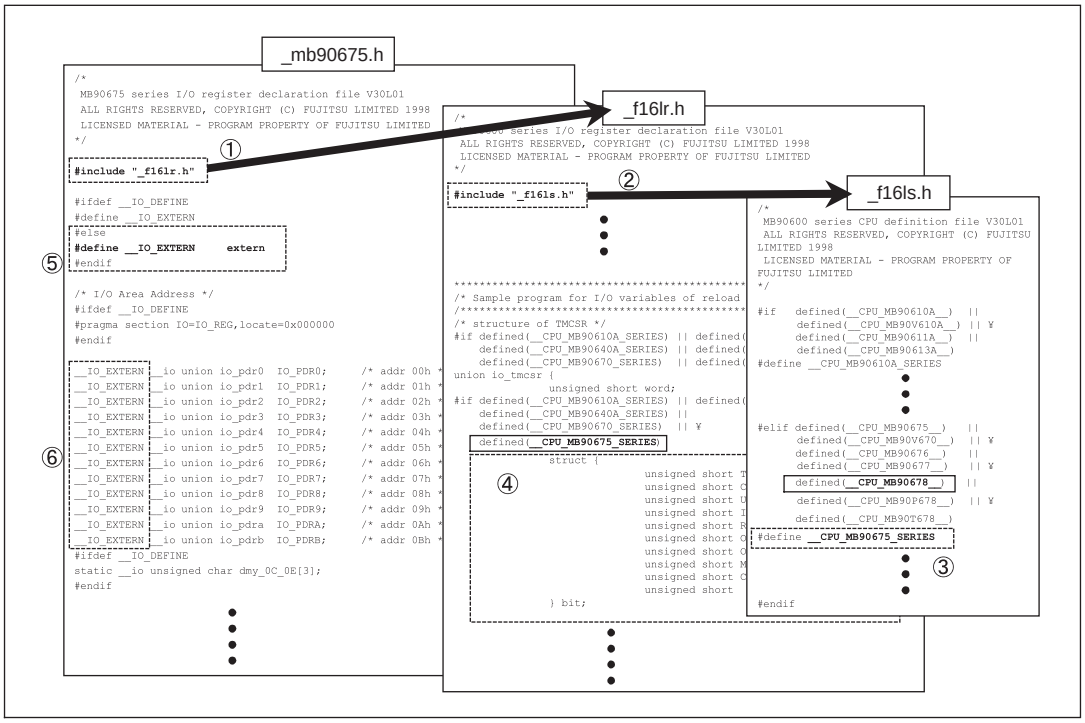
In _f16l.h, proceed as follows:

① Include _f16ls.h.

② In _f16ls.h, use the predefined macro _ _CPU_MB90678__ to define the predefined macro _ _CPU_MB90675_SERIES.

③ Use the predefined macro _ _CPU_MB90675_SERIES defined in _f16ls.h to include _mb90675.h.

Figure 12.2-7 Accessing Variables Defined in the I/O Area (3)



In _mb90675.h, proceed as follows:

① Include _f16lr.h.

② In _f16lr.h, include _f16ls.h.

③ In _f16ls.h, use the predefined macro _ _CPU_MB90675_ _ to define the predefined macro _ _CPU_MB90675_SERIES.

④ In _f16lr.h, use the predefined macro _ _CPU_MB90675_SERIES defined in _f16ls.h to define the required type specific to the MB90675 series.

⑤ Because _ _IO_DEFINE has not been defined, use #define to define a macro that replaces _ _IO_EXTERN with extern.

⑥ Specify _ _IO_EXTERN and the _ _io type qualifier to declare access to the I/O register variables specific to the MB90675 series.

<Notes>

Note the following points when defining I/O variables:

- Map variables qualified by the _ _io type qualifier to the I/O area defined from address 0x0000 to address 0x00bf. The I/O area can be accessed using highly efficient dedicated instructions.
- To define I/O variables after address 0x00bf, specify the volatile type qualifier.
- Initial values cannot be set for variables qualified by the _ _io type qualifier.
- Variables qualified by the _ _io type qualifier are handled as variables qualified by the volatile type qualifier. If the -K NOVOLATILE option is specified, the variables qualified by the _ _io type qualifier will not be handled as variables qualified by the volatile type qualifier.

CHAPTER 13 MAPPING VARIABLES QUALIFIED WITH THE _ _direct TYPE QUALIFIER

This chapter describes the variables qualified by the _ _direct type qualifier and the conditions for mapping them.
A variable qualified by the _ _direct type qualifier can be mapped in the page pointed to by the DPR register and accessed using direct addressing.

13.1 Output Sections of and Access to Variables Qualified by the _ _direct Type Qualifier

A variable qualified by the _ _direct type qualifier can be accessed by the direct addressing method specific to the F^{2}MC-16 family.
A variable qualified by the _ _direct type qualifier to which an initial value has been assigned is output to the DIRINIT section of the variable area and to the DIRCONST section of the initial value area. A variable to which an initial value has not been assigned is output to the DIRDATA section.

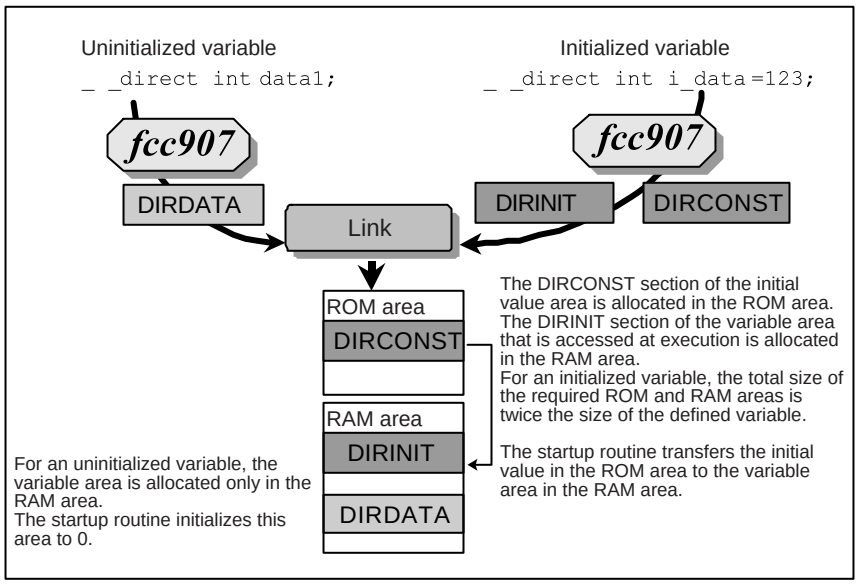
■ Output Sections of Variables Qualified by the _ _direct Type Qualifier

Like other variables, the variables qualified by the _ _direct type qualifier have different output section names depending on whether they are initialized.

An uninitialized variable qualified by the _ _direct type qualifier is output only to the DIRDATA section. This area is allocated in RAM, and is usually initialized to 0 by the startup routine.

An initialized variable is output to the DIRCONST section of the initial value area and to the DIRINIT section of the variable area. The DIRCONST section of the initial value area is allocated in the ROM area. The DIRINIT section of the variable area that is accessed at execution is allocated in the RAM area. The startup routine transfers the initial value in the ROM area to the RAM area. As a result, the total size of the required ROM and RAM areas is twice the size of the defined variable.

Figure 13.1-1 Variables Qualified by the _ _direct Type Qualifier and Their Output Sections

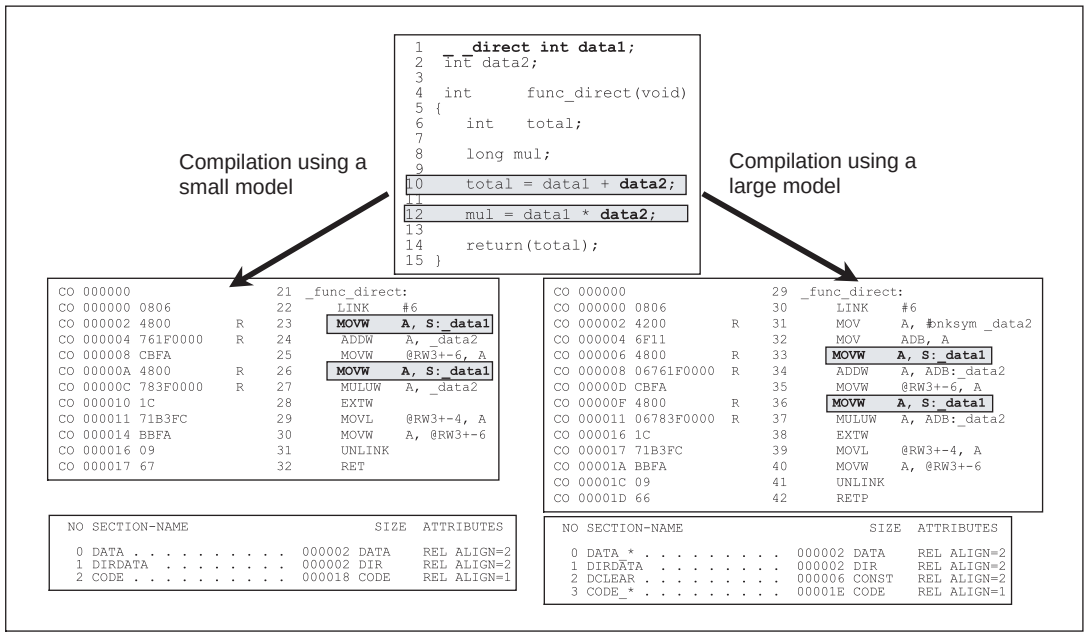


■ Accessing a Variable Qualified by the _ _direct Type Qualifier

For the fcc907, the addressing mode when a variable is accessed depends on the memory model specified at compilation. For a small or medium model, variables are accessed using 16-bit addressing. For a compact or large model, variables are accessed using 24-bit addressing and the ADB register. For a variable qualified by the _ _direct type qualifier, the variable is accessed using direct addressing where addresses are accessed in eight bit units regardless of the memory model.

Figure 13.1-2 "Accessing a Variable Qualified by the _ _direct Type Qualifier" shows the difference between normal variable access and access of a variable qualified by the _ _direct type qualifier. In this example, the address of variable data1 qualified by the _ _direct type qualifier is accessed in eight bit units. The address of variable data2, however, depends on the memory model specified at compilation. Variables accessed frequently should be qualified by the _ _direct type qualifier.

Figure 13.1-2 Accessing a Variable Qualified by the _ _direct Type Qualifier



13.2 Mapping Variables Qualified by the _ _direct Type Qualifier

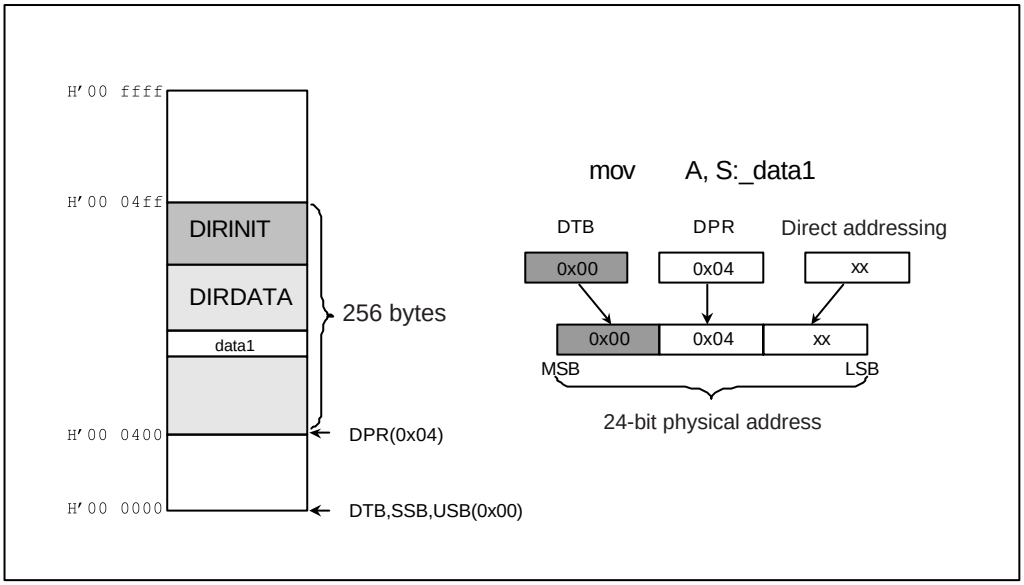
All variables qualified by the _ _direct type qualifier must be mapped in the page pointed to by the DPR register. Therefore, the total size of the variables qualified by the _ _direct type qualifier must not exceed 256 bytes.

■ Accessing Variables Using Direct Addressing

In direct addressing, only the eight low-order bits of an address of a variable accessed using 16 or 24 bits are accessed. The eight-bit values that can be accessed are 0 to 255. In direct addressing, the DTB and DPR registers are used to determine the address to be accessed as shown in Figure 13.2-1 "Areas into Which Variables Qualified by the _ _direct Type Qualifier Can Be Mapped". The following settings are required to access a variable using direct addressing:

1. Set the DPR register in the data bank which is indicated by the DTB register.
2. Allocate the areas (DIRVAR and DIRINIT) of the variables qualified by the _ _direct type qualifier into the page (256 bytes) indicated by the DPR register.

Figure 13.2-1 Areas into Which Variables Qualified by the _ _direct Type Qualifier Can Be Mapped



■ _ _direct Type Qualifier and Initialization of the DTB Register

The DTB register accessed in direct addressing is initialized to 0x00 at reset.

For a small or medium model in which the variable areas are restricted to 1 bank, there is no problem if the initial values are used as is. For a compact or large model in which the variable areas can be specified for multiple banks, the numbers of banks used to map the DIRINIT and DIRDATA sections must be set in the DTB register.

Use the startup routine to set the DTB register. For the startup routine provided by the fcc907, the DTB register has been set up based on allocation of the DATA section. Refer to these to code the startup routine based on the system to be created.

■ _ _direct Type Qualifier and Initialization of the DPR Register

The DPR register accessed in direct addressing is initialized to 0x01 at reset.

When a variable is accessed by direct addressing using the initial values of the DTB and DPR registers as is, the 256-byte area starting from address h'0100 of the 0x00 bank will be enabled for direct addressing. However, an extended intelligent I/O service descriptor is already present between addresses h'0100 and h'015f. In addition, an area for a general-purpose register is present between addresses h'180 and h'0380. Therefore, when mapping variables qualified by the _ _direct type qualifier for a small or medium model, initialize the DPR register based on use of the extended intelligent I/O service and register bank.

For the startup routine provided by the fcc907, the DPR register has been set up based on allocation of the DIRDATA section. Refer to these to code the startup routine based on the system to be created.

■ Mapping Variables Qualified by the _ _direct Type Qualifier

Figure 13.2-2 "Mapping Variables Qualified by the _ _direct Type Qualifier (Small Model)" shows the link specification and an image of the actual mapping of the variables qualified by the _ _direct type qualifier when a small model is specified.

In this example, the DIRDATA section is allocated starting from the page boundary after the DATA section. The DIRINIT section is then allocated. The DIRCONST section for initial values is allocated at the end of the section allocated in the ROM area. At execution, the initial value DIRCONST in the ROM area is transferred to the variable area in the RAM area.

Figure 13.2-2 Mapping Variables Qualified by the _ _direct Type Qualifier (Small Model)

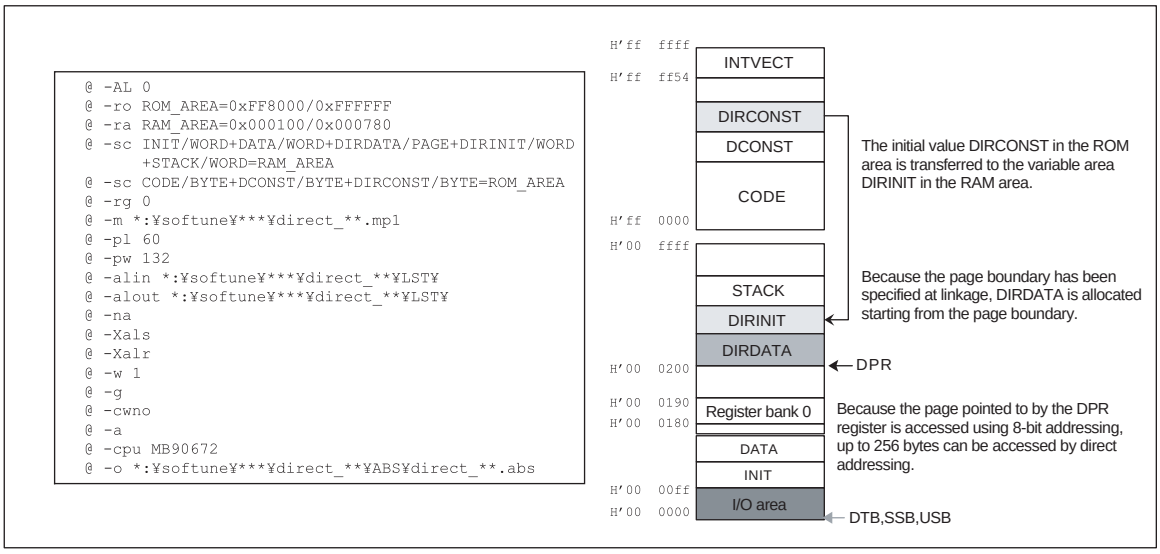
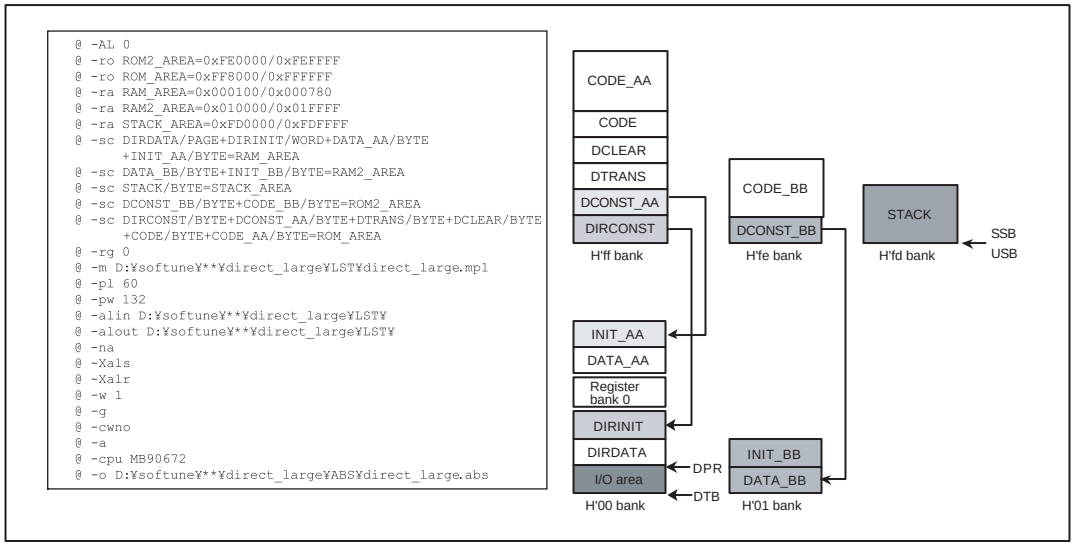


Figure 13.2-3 "Mapping Variables Qualified by the _ _direct Type Qualifier (Large Model)" shows the link specification and an image of the actual mapping of the variables qualified by the _ _direct type qualifier when a large model is specified.

In this example, the DIRDATA section is allocated starting from address h'0100 of the 0x00 bank. The DIRINIT section is then allocated. The DIRCONST section for initial values is allocated starting from the beginning of the 0xff bank. At execution, the initial value DIRCONST in the ROM area is transferred to the variable area DIRINIT in the RAM area.

Figure 13.2-3 Mapping Variables Qualified by the _ _direct Type Qualifier (Large Model)



<Notes>

The fcc907 does not provide a function for calculating the total size of variables qualified by the _ _direct type qualifier and outputting error messages. If the total variable size exceeds 256 for the whole system, an error message is output during linkage.

[Tip]

Softune C Analyzer:

The Softune C Analyzer checks the reference relationships of variables in a specified module, and displays the candidates for _ _direct type qualifier declaration in descending order of number of references. The number of generated candidates can be reduced by specifying an upper limit for the number of _ _direct type qualifier declarations. This check function is helpful in determining the variables for qualification by the _ _direct type qualifier.

CHAPTER 14 CREATING AND REGISTERING INTERRUPT FUNCTIONS

This chapter provides notes for creation and registration of interrupt functions.

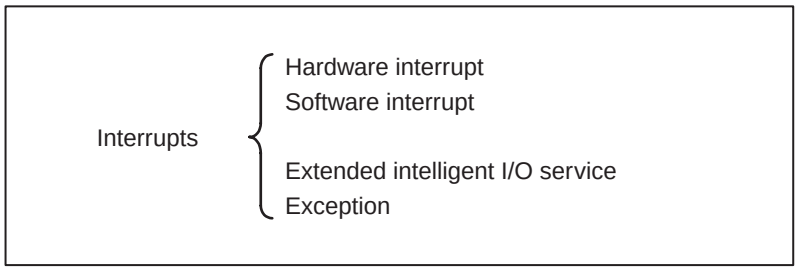
The F^{2}MC-16 family of microcontrollers has various resources for generating interrupts. The generation and processing of interrupts requires to set initial values for hardware and software.

14.1 F²MC-16 Family Interrupts

This section describes interrupt handling in the F²MC-16 family of microcontrollers. When an interrupt occurs, the processing being executed is temporarily halted and interrupt processing is executed. When interrupt processing terminates, processing resumes from where the interrupt occurred.

■ F²MC-16 Family Interrupts

The F²MC-16 Family has the following four types of interrupts. When an interrupt occurs, the processing currently being executed is temporarily halted and control is passed to the interrupt handler. When interrupt processing terminates, processing resumes from where the interrupt occurred.



■ Interrupt handling in the F²MC-16 Family

This section mainly describes the handling of internal resource interrupts in the F²MC-16 family, but also covers other types of interrupt handling.

In the F²MC-16 family, when an internal resource interrupt request or external interrupt request that is allowed occurs during program execution, control passes to the interrupt handler. The necessary interrupt handling is executed, the reti instruction is issued, control returns to the location where the interrupt was detected, and the interrupted processing is resumed.

Figure 14.1-1 "F²MC-16 Family Interrupt Handling" shows interrupt handling in the F²MC-16 family.

The following preparations are required before F²MC-16 family internal resource interrupts and external interrupts can be handled:

❍ **Hardware settings**

- Setting of system stack area
- Initialization of internal resources that can generate interrupt requests
- Setting of the resource interrupt level
- Starting of resource operation
- Enabling of internal interrupts in the CPU

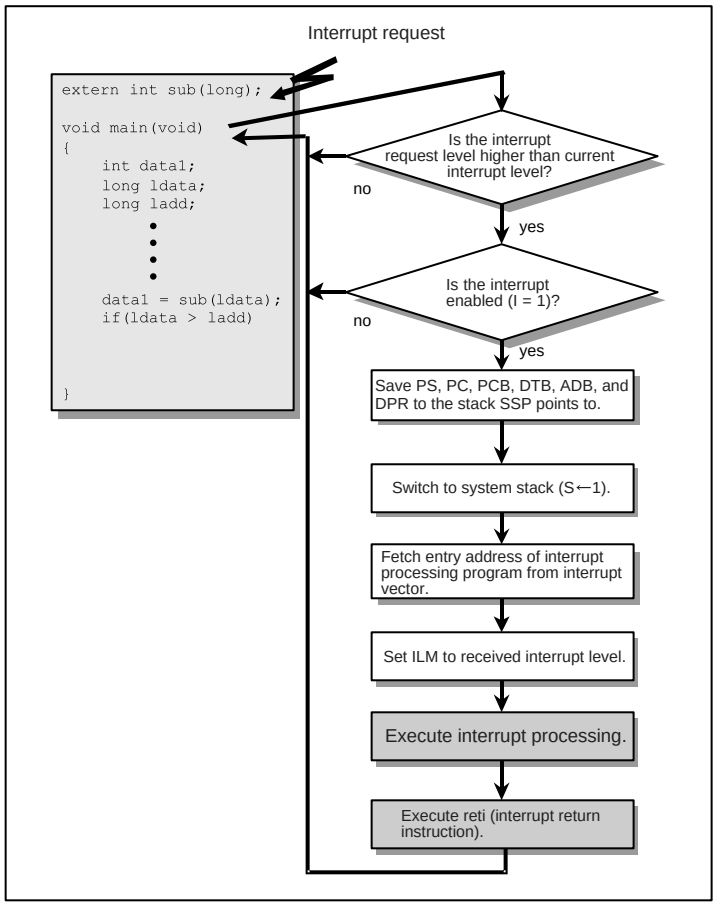
❍ **Creation of interrupt functions**

❍ **Registration of the interrupt functions in interrupt vectors**

Provided the above preparations have been made, a hardware interrupt request will be issued when an interrupt occurs. If the interrupt is allowed, the CPU saves the contents of registers and passes control to the corresponding interrupt processing handler.

Sections 14.2 "Required Hardware Settings for Interrupts" to 14.4 "Setting of Interrupt Vectors" describe the preparations for interrupt processing.

Figure 14.1-1 F^{2}MC-16 Family Interrupt Handling



14.2 Required Hardware Settings for Interrupts

This section describes the required hardware settings for interrupt handling.
The following steps must be performed to enable interrupt handling.

- **Setting the stack area**
- **Initial value of resources that can generate interrupt requests**
- **Setting the resource interrupt level**
- **Starting resource operation**
- **Enabling CPU interrupts**

■ Required Hardware Settings for Interrupts

The following steps must be performed to enable interrupt processing for F^2MC-16 family microcontrollers:

- Setting the system stack area
- Initial value of resources that can generate interrupt requests
- Setting the resource interrupt level
- Starting resource operation
- Enabling CPU interrupts

Sections 14.2.1 "Setting the System Stack Area" to 14.2.5 "Enabling CPU Interrupts" describe the required initializations.

14.2.1 Setting the System Stack Area

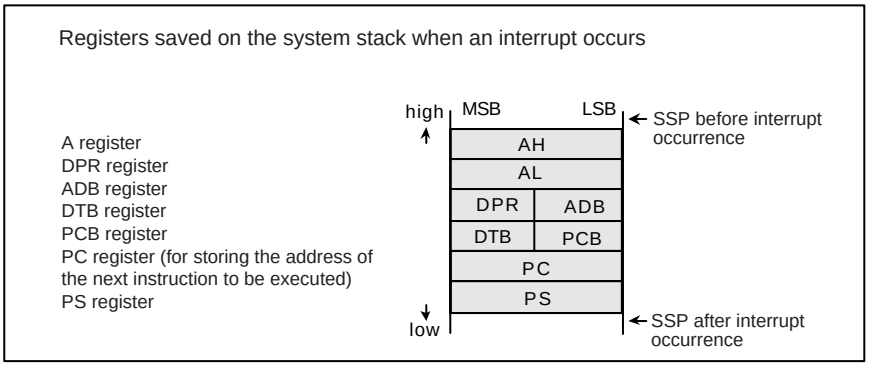
This section describes how to set the system stack areas used for interrupt handling. When an interrupt occurs, the CPU automatically saves the contents of the registers on the system stack.

■ Setting the System Stack

When an allowed F^{2}MC-16 family interrupt occurs, the CPU saves the contents of the registers shown below on the stack, and then executes interrupt processing.

- A register
- DPR register
- ADB register
- DTB register
- PCB register
- PC register
- PS register

Figure 14.2-1 Registers Saved to the System Stack when an Interrupt Occurs



The system stack must be initialized as follows to create a system in which interrupt processing can be executed:

- Allocation of system stack area
- Setting the system stack pointer (SSP)
- Specifying the address of stack allocation for the linker

Register values cannot be set directly in a C program. An assembler must be used to set the system stack pointer. Use a startup routine to allocate the system stack area and initialize the system stack pointer (SSP).

In addition, specify the mapping addresses of the system stack at linkage.

Figure 14.2-2 "Setting the System Stack Area" shows an example of using a startup routine to allocate the system stack area and setting the system stack pointer.

Figure 14.2-2 Setting the System Stack Area

```
;=====================================================================
; Sample program for initialization (small model)
;---------------------------------------------------------------------

                .PROGRAM        start
                .TITLE          start

                                •
                                •
                                •
                                •
;---------------------------------------------------------------------
; definition to stack area
;---------------------------------------------------------------------
                .SECTION        STACK,    STACK,    ALIGN=1
                .RES.B          254
SSTACK_TOP:
                .RES.B          2
                .RES.B          254
USTACK_TOP:
                .RES.B          2

                                •
                                •
                                •
                                •
;---------------------------------------------------------------------
; code area
;---------------------------------------------------------------------
                .SECTION        CODE,     CODE,     ALIGN=1
__start:
                                •
                                •
                                •
                                •
;---------------------------------------------------------------------
; set system stack
;---------------------------------------------------------------------
                AND     CCR, #0x20
                MOV     A, #BNKSYM SSTACK_TOP
                MOV     SSB, A
                MOVW    A, #SSTACK_TOP
                MOVW    SP, A
                AND     CCR, #0x00DF

                                •
                                •
                                •
                                •
; end:          bra     end

                .END    __start
```

- Areas of 256 bytes have been allocated for the system stack and user stack defining the STACK section.
- The bank containing the symbol SSTACK_TOP in the STACK section has been set in the SSB.
- The address of the symbol SSTACK_TOP has been set in the SP.

14.2.2 Initializing Resources

This section describes the initial settings for resources that generate interrupt requests. This initialization values must be defined dependent on the used resources.

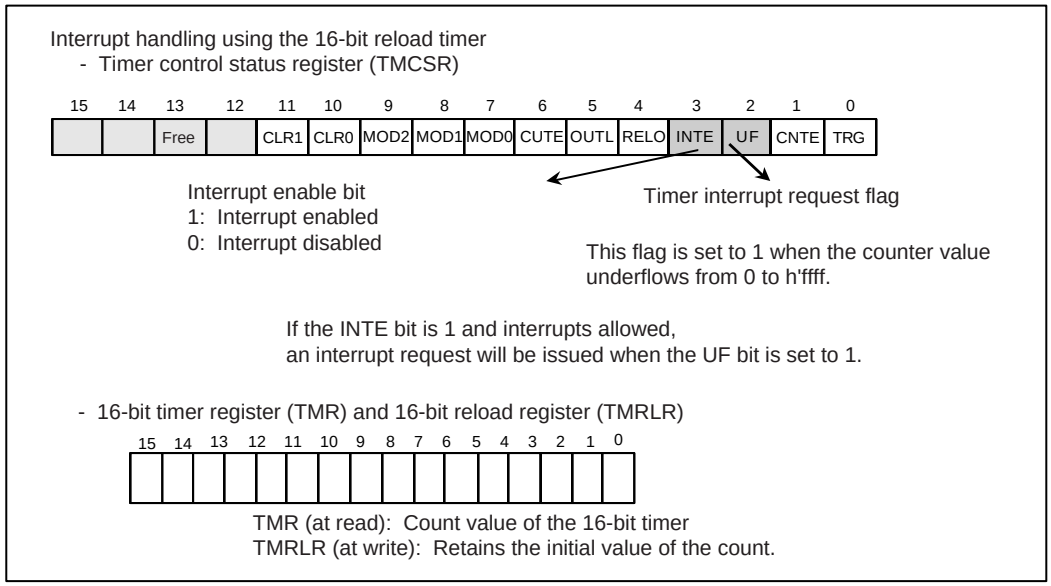
■ Initializing Resources

Before an interrupt can be generated, the resources that generate interrupt requests must be initialized.

The internal resources that can request hardware interrupts for an F^2MC-16 family microcontroller have an interrupt enable bit and interrupt request flag in a register. First, the resources that can execute interrupt processing must be initialized. The settings of the interrupt enable flag and interrupt level depend on the system to be created. Initialize each resource as required.

Figure 14.2-3 "Initializing Internal Resources (for interrupts using 16-bit timer)" shows the registers for the 16-bit reload timer, which is an internal resource. These registers must be initialized for interrupt operations that uses the 16-bit reload timer. See Figure 14.2-10 "Example of Initializing Interrupt Processing" for an example of an initialization program for interrupt processing that uses the 16-bit reload timer.

Figure 14.2-3 Initializing Internal Resources (for interrupts using 16-bit reload timer)



For information about the registers for each of its internal resources, refer to the hardware manual for the specific product.

14.2.3 Setting Interrupt Control Registers

Set the values of the interrupt control register after the resources that generate interrupt requests have been initialized.

■ Setting Interrupt Control Registers

The values of the interrupt control registers must be set after the resources that generate interrupt requests have been initialized.

An interrupt level setting register is allocated to each internal resource. The interrupt level set in the interrupt level setting register determines the priority of the interrupts that are enabled.

Figure 14.2-4 "Bit Configuration of an F^{2}MC-16 Family Interrupt Level Setting Register" shows the bit configuration of the F^{2}MC-16 Family interrupt control registers.

At a reset, the interrupt control registers are initialized to interrupt prohibited level 7. When an interrupt request is issued in a resource, the interrupt controller informs the CPU of the value corresponding to the interrupt. Set a value based on the system to be created.

For the F^{2}MC-16 Family, interrupt control registers are mapped between addresses 0x0000b0 and 0x0000bf in the I/O area. (See Figure 12.1-2 "I/O Register Mapping in the MB90670/675 Series")

Table 14.2-1 "Relationship between Interrupt Sources, Interrupt Level Setting Registers, and Interrupt Vectors for MB90675" shows the relationship between interrupt sources and interrupt control register bits. For information about the interrupt control registers, refer to the hardware manual of the specific product.

Figure 14.2-4 Bit Configuration of an F^{2}MC-16 Family Interrupt Level Setting Register

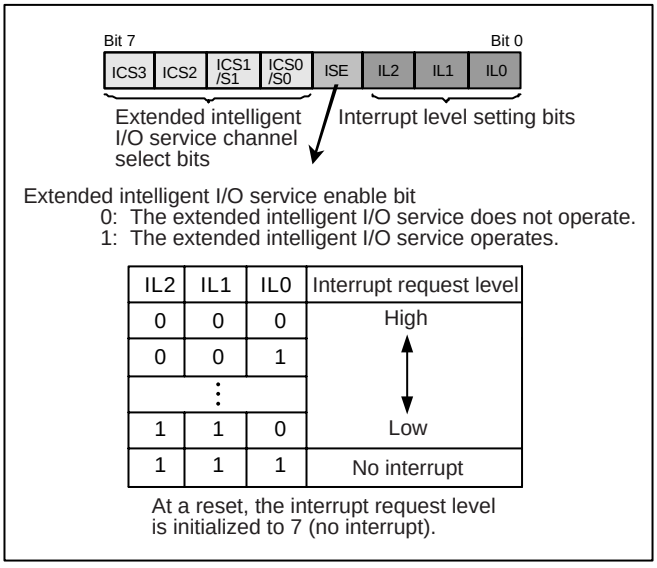


Table 14.2-1 Relationship between Interrupt Sources, Interrupt Level Setting Registers, and Interrupt Vectors for MB90675

Interrupt source	Interrupt vector		Interrupt level setting register	
	Number	Address	ICR	Address
Reset	#08	h'FFFFDC	-	-
INT9 instruction	#09	h'FFFFD8	-	-
Exception	#10	h'FFFFD4	-	-
External interrupt #0	#11	h'FFFFD0	ICR0	h'0000B0
External interrupt #1	#12	h'FFFFCC		
External interrupt #2	#13	h'FFFFC8	ICR1	h'0000B1
External interrupt #3	#14	h'FFEFC4		
OCU#0	#15	h'FFEFC0	ICR2	h'0000B2
OCU#1	#16	h'FFEFBC		
OCU#2	#17	h'FFEFB8	ICR3	h'0000B3
OCU#3	#18	h'FFEFB4		
OCU#4	#19	h'FFEFB0	ICR4	h'0000B4
OCU#5	#20	h'FFFFAC		
OCU#6	#21	h'FFFFA8	ICR5	h'0000B5
OCU#7	#22	h'FFFFA4		
24-bit free run timer overflow	#23	h'FFEFA0	ICR6	h'0000B6
24-bit free run timer intermediate bit	#24	h'FFEF9C		
ICU#0	#25	h'FFEF98	ICR7	h'0000B7
ICU#1	#26	h'FFEF94		
ICU#2	#27	h'FFEF90	ICR8	h'0000B8
ICU#3	#28	h'FFEF8C		
16-bit reload timer #0/PPG0	#29	h'FFFF88	ICR9	h'0000B9
16-bit reload timer #1/PPG1	#30	h'FFFF84		
A/D converter measurement	#31	h'FFFF80	ICR10	h'0000BA
Wake-up interrupt	#33	h'FFEF78	ICR11	h'0000BB
Time-base timer interval interrupt	#34	h'FFEF74		
UART1 send completion	#35	h'FFEF70	ICR12	h'0000BC
UART0 send completion	#36	h'FFEF6C		
UART1 receive completion	#37	h'FFEF68	ICR13	h'0000BD
I^2C interface	#38	h'FFEF64		
UART1 receive completion	#39	h'FFEF60	ICR14	h'0000BF
Delayed interrupt occurrence module	#42	h'FFEF54	ICR15	h'0000BF

14.2.4 Starting Resource Operation

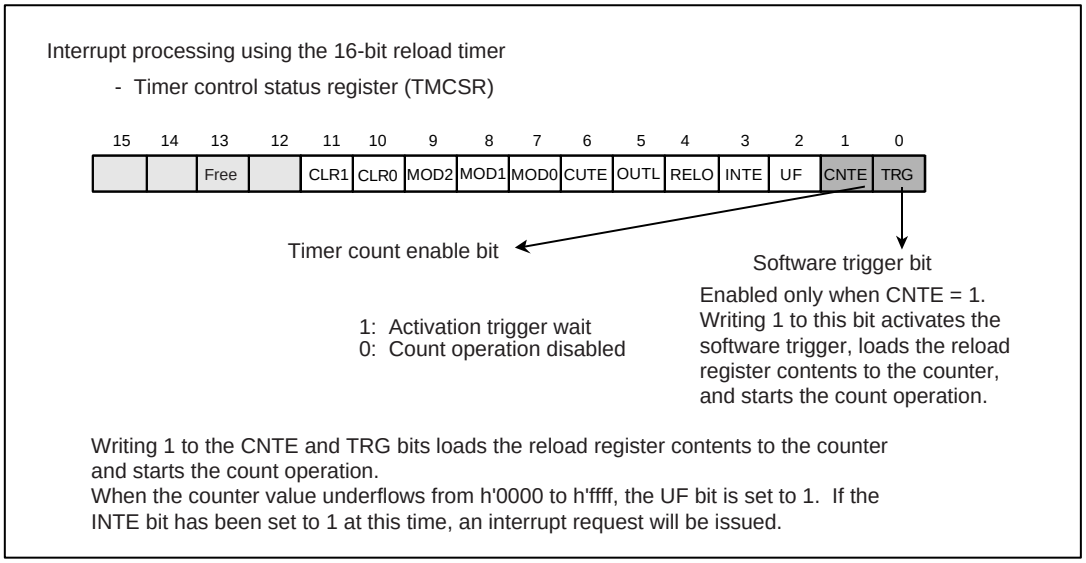
After the resources that process interrupts have been initialized and the corresponding interrupt control registers have been set, the resources start operation.

■ Starting Resource Operation

Each resource register has a bit for enabling or disabling interrupt processing and a bit for starting operation of the resource. Setting these bits enables interrupts for the corresponding resource and starts operation of the resource.

Figure 14.2-5 "Starting Internal Resource Operation (for interrupt processing using the 16-bit reload timer)" shows how to start the operation of the 16-bit reload timer, which is an internal resource. See Figure 14.2-10 "Example of Initializing Interrupt Processing" for an example of an initialization program for interrupt processing that uses the 16-bit reload timer.

Figure 14.2-5 Starting Internal Resource Operation (for interrupt processing using the 16-bit reload timer)



Some resources generate interrupt requests as soon as the resource start. As a result, an interrupt can occur before processing for interrupts has been completely initialized, with unpredictable results. Therefore, initialize resources and start their operation in a manner appropriate for the system.

For information about the registers of respective resources, refer to the hardware manual of the specific product.

14.2.5 Enabling CPU Interrupts

This section describes how to set CPU interrupts to be enabled.
The I flag and ILM value in the CPU determine the interrupt level allowed for the system.

■ Enabling CPU Interrupts

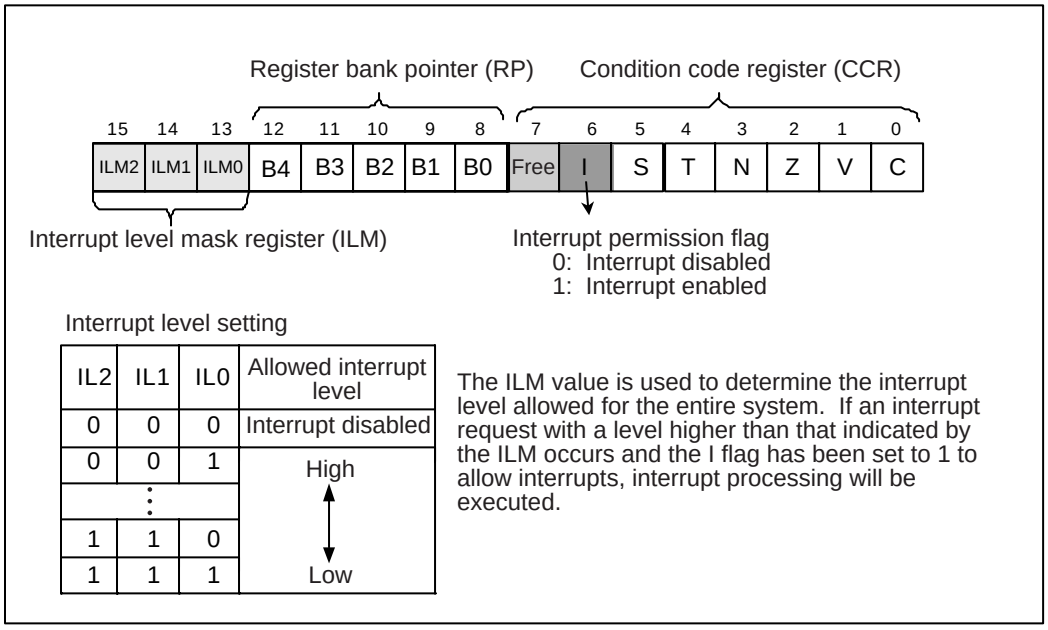
Once the resources for interrupt handling have been set up, the settings for the receiving CPU must be made.

For the F^{2}MC-16 Family, the interrupt permission flag in the program status register (PS) and the value of the interrupt level mask register (ILM) determine the hardware interrupt level allowed for the entire system.

Figure 14.2-6 "Bit Configuration of PS Register" shows the bit configuration of the PS register.

ILM indicates the interrupt level that is currently allowed. If an interrupt request of a higher level than that indicated by the ILM register occurs, interrupt processing will be executed. Level 0 is the highest level, and level 7 is the lowest level. When the system is reset, the lowest level (7) is set.

Figure 14.2-6 Bit Configuration of PS Register



For the fcc907, the _ _set_il( ) function and #pragma ilm/noilm can be used to set the interrupt level.

■ Using _ _set_il( ) to Set the Interrupt Level in a Function

Because _ _set_il( ) converts the ILM register values into an argument, an interrupt level can be set anywhere in a function.

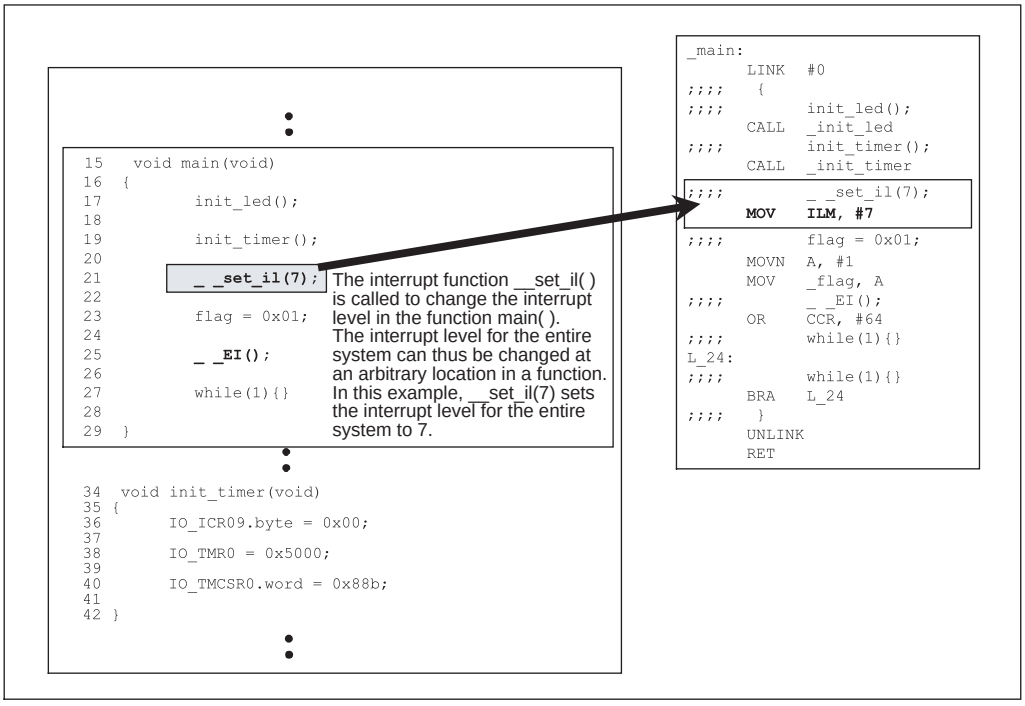
```
void  _ _set_il (interrupt-level) ;
```

Figure 14.2-7 "Using _ _set_il( ) to Set the Interrupt Level in a Function" shows a function for which an interrupt level has been set using _ _set_il( ).

Function main( ) calls the built-in function _ _set_il( ) at line 21. Because 7 is specified as an argument, code that sets 7 in the ILM register is generated.

The _ _set_il( ) can be set at an arbitrary location in a function to generate a code that changes the interrupt level.

Figure 14.2-7 Using _ _set_il( ) to Set the Interrupt Level in a Function



■ Using #pragma ilm/noilm to Set the Interrupt Level in a Function

The #pragma ilm directive can set the interrupt level for each function. When an interrupt level is set using #pragma ilm, code that sets the interrupt level is generated before processing of the function is started.

When changing the interrupt level of a function with #pragma ilm, place #pragma ilm before the function whose interrupt level is to be changed.

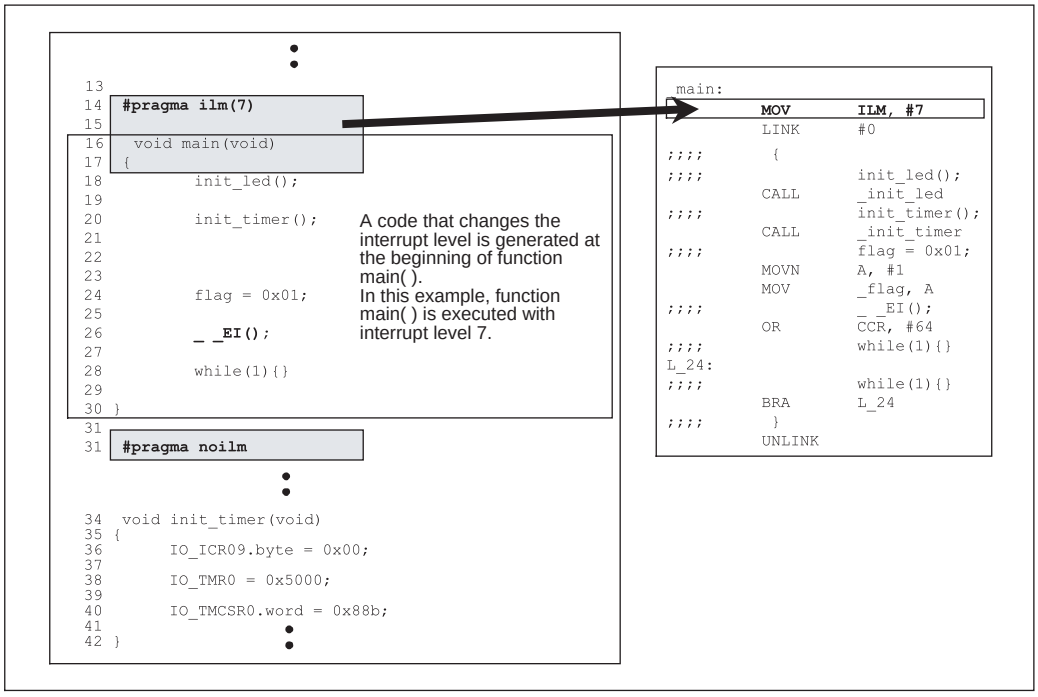
```
#pragma ilm (interrupt-level) ;
```

Use #pragma noilm to terminate the specification for changing the interrupt level of a function.

```
#pragma noilm
```

Figure 14.2-8 "Using #pragma ilm/noilm to Set the Interrupt Level in a Function" shows a function whose interrupt level is changed using #pragma ilm/noilm. In this example, interrupt level 7 is set. That is, when the processing of function main( ) starts, code that sets the ILM to 7 is generated. Because #pragma noilm has been specified after function main( ), code that sets an interrupt level will not be generated when the processing for function init_timer( ) defined from line 34 starts.

Figure 14.2-8 Using #pragma ilm/noilm to Set the Interrupt Level in a Function



[Tips]

Softune C Checker:

The Softune C Checker will output the message "The interrupt level setting function has been used" at the location where the _ _set_il( ) function or #pragma ilm/noilm has been specified. The fcc896 and fcc911 also support _ _set_il( ) and #pragma ilm/noilm. When porting, check this message to see whether the function should be used in the new program.

■ Using the I Flag to Enable Interrupts for the Entire System

Finally, after all of the initializations for interrupts have been set, the I flag is set.

When the I flag is 1, interrupts are enabled for the entire system. Resetting clears the I flag to 0. Although interrupts that are higher than the level set by the ILM register are enabled, whether the interrupts are actually process depends on the status of the I flag.

In the fcc907, interrupts can be disabled by clearing the I flag to 0 with _ _DI( ), as follows.

```
void  _ _DI(void) ;
```

Interrupts can be enabled by setting the I flag to 1 with _ _EI( ), as follows.

```
void  _ _EI(void) ;
```

Figure 14.2-9 "Example of Using _ _EI( ) in a Function to Enable Interrupts" shows an example of a function that uses _ _EI( ) to enable system interrupts.

Figure 14.2-9 Example of Using _ _EI( ) in a Function to Enable Interrupts

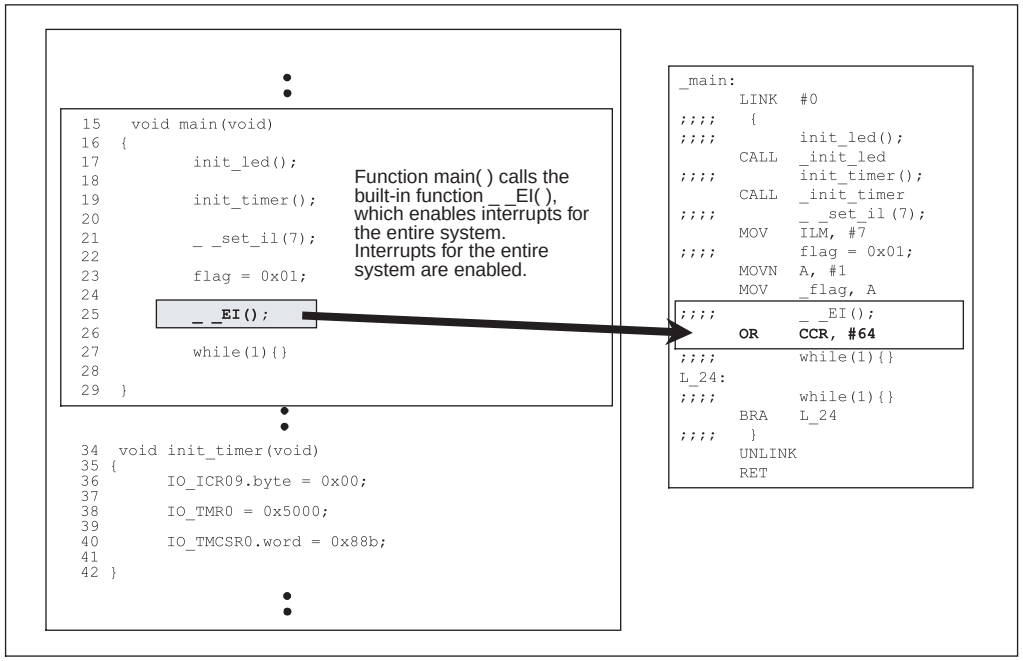
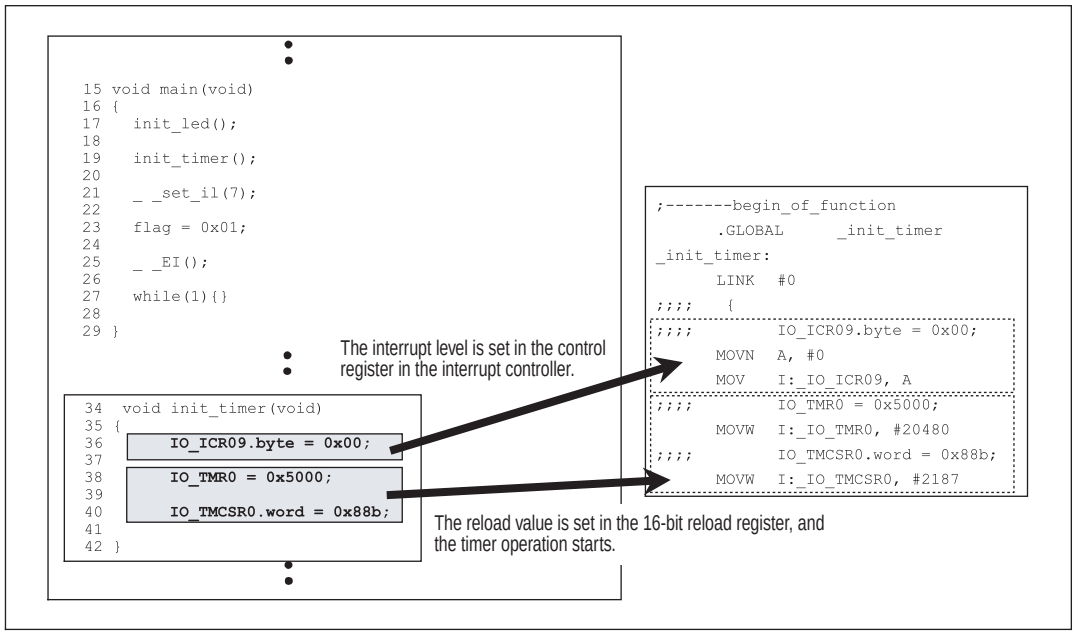


Figure 14.2-10 "Example of Initializing Interrupt Processing" shows an example of an initialization program for interrupt processing that uses the 16-bit reload timer.

In this example, function main( ) calls function init_timer( ), which initializes the 16-bit reload timer. On line 36, function init_timer( ) sets the highest interrupt level (0) in the 16-bit reload timer interrupt control register. Then, on line 38, reload value 0x5000 is set in the IO_TMR0 register. Finally, 0x088b is set in the IO_TMCSR0 register and operation of the 16-bit reload timer starts when initialization starts.

When initialization of the 16-bit reload timer terminates, control is returned to function main( ). System interrupts are then enabled after _ _set_il(7) sets the interrupt level of the entire system. Interrupts using the 16-bit reload timer are thus enabled.

Figure 14.2-10 Example of Initializing Interrupt Processing



<Notes>

Because a reset clears the I flag to 0, execute _ _EI( ) to enable interrupts of the entire system after the hardware of the system to be created has been initialized.

[Tip]

Softune C Checker:

The Softune C Checker will output messages indicating that the interrupt mask setting and interrupt mask release functions have been used at the locations where _ _EI( ) and _ _DI( ) are used. The fcc896 and fcc911 also support the _ _EI( ) and _ _DI( ) functions. When porting, check this message to see whether these functions should also be used in the new program system.

14.3 Using the _ _interrupt Type Qualifier to Define Interrupt Functions

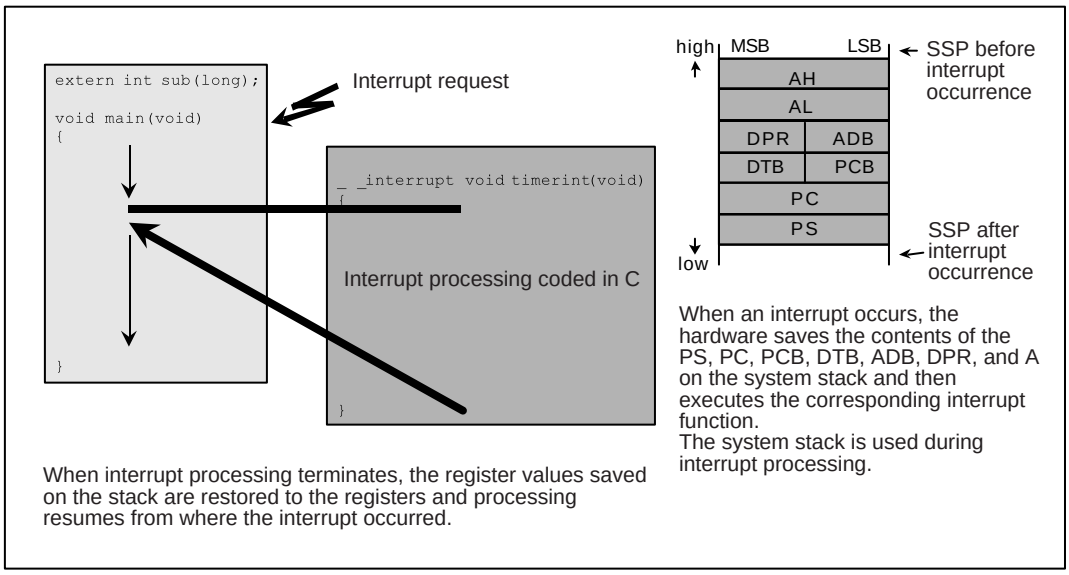
Sections 14.2.1 to 14.2.4 described the initialization required to execute interrupts. However, interrupt processing cannot be executed simply by initialization. Before interrupt processing can be executed, interrupt processing functions corresponding to the interrupts must be created.

■ Using the _ _interrupt Type Qualifier to Code Interrupt Functions

When an interrupt allowed by an F^2MC-16 family microcontroller is issued, the following procedure is used to execute interrupt processing:

1. The PS, PC, PCB, DTB, ADB, DPR, and A (12 bytes total) are saved on the stack.
2. The ILM register is updated to the level of the received interrupt.
3. The PS register S flag is set (the system stack is used).
4. Instructions starting from the address indicated by the corresponding interrupt vector are executed.

Figure 14.3-1 Executing an Interrupt Function



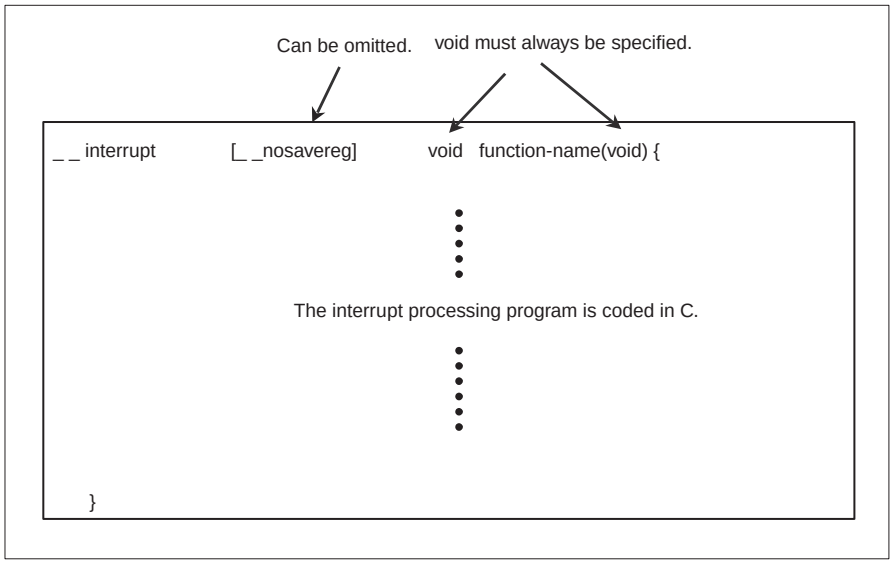
As shown in Figure 14.3-1 "Executing an Interrupt Function", the hardware automatically saves the contents of registers and passes control to an interrupt processing routine when an interrupt occurs.

When an interrupt processing routine is coded in assembly language, the reti instruction is issued at the end of the interrupt processing routine. As a result, the PS, PC, PCB, DTB, ADB, DPR, and A register values that were saved on the stack are restored and processing resumes from where the interrupt occurred.

When an interrupt processing function is coded using the fcc907, the interrupt function must be qualified with the _ _interrupt type qualifier, as shown in Figure 14.3-2 "Using the _ _interrupt Type Qualifier to Define an Interrupt Function". Based on the coding, the fcc907 compiles the

specified function as an interrupt function.

Figure 14.3-2 Using the _ _interrupt Type Qualifier to Define an Interrupt Function



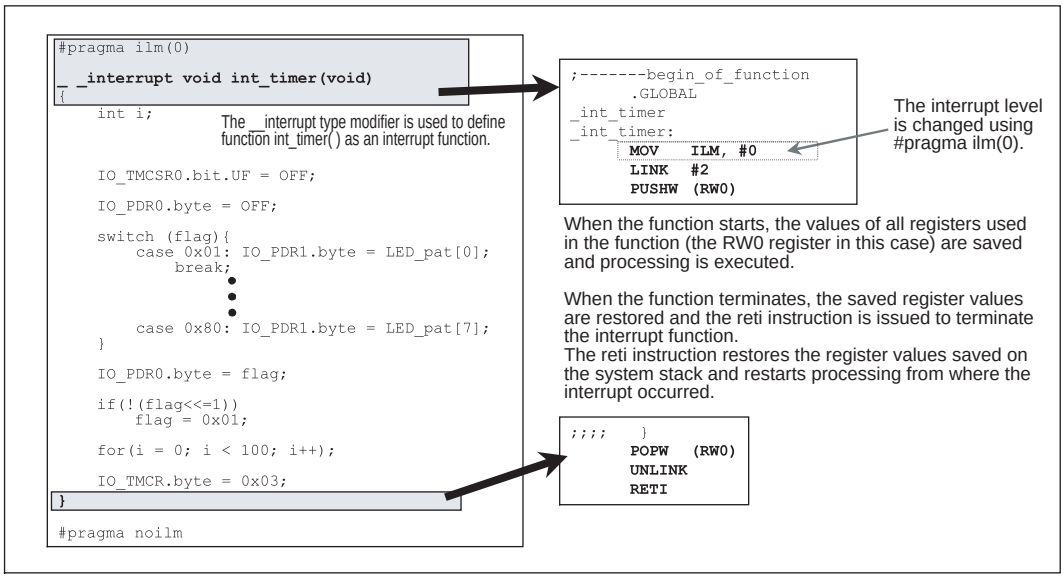
When an interrupt function qualified by the _ _interrupt type qualifier is executed, the values of all of the registers that are used in the function are saved. When the interrupt function terminates, the saved register values are restored and the reti instruction is issued. Issuing the reti instruction restores the PS, PC, PCB, DTB, ADB, and DPR register values that were saved on the stack and restarts processing from where the interrupt occurred.

Figure 14.3-3 "Example of an Interrupt Function Using the _ _interrupt Type Qualifier" shows an example of an interrupt function.

When function int_timer( ) qualified by the _ _interrupt type qualifier is called, the value of the register (the RW0 register in this case) is saved on the stack when the function starts.

When the function terminates, the saved register value is restored and the reti instruction is issued. The reti instruction restores the PS, PC, PCB, DTB, ADB, DPR, and A register values that were saved on the stack and restarts processing from where the interrupt occurred.

Figure 14.3-3 Example of an Interrupt Function Using the _ _interrupt Type Qualifier



■ Coding of Interrupt Function That Switches the Register Bank without Saving Work Registers

F^{2}MC-16 family microcontrollers can use up to 32 register banks. Because the register bank that will be used can be changed when an interrupt function starts, it becomes possible to create an interrupt function that is faster than a function that saves work registers.

When writing an interrupt function that switches to a new register bank, #pragma register/ noregister must be used to switch register banks and the interrupt function must be coded using both the _ _interrupt type qualifier and _ _nosavereg type qualifier.

When a function is qualified by the _ _nosavereg type qualifier, the values of the registers are not saved. This applies even if registers are used in the function.

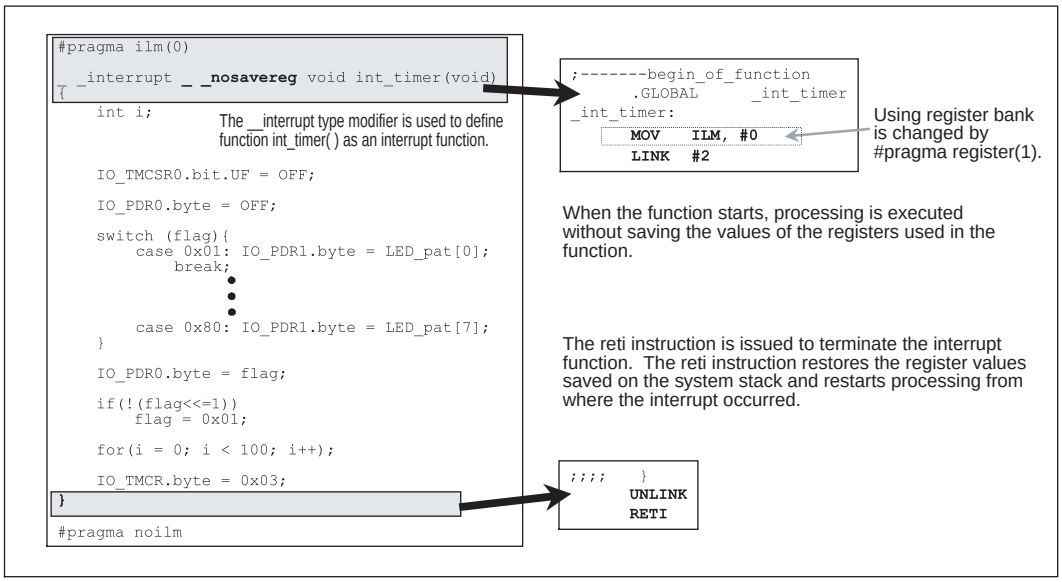
Figure 14.3-4 "Changing Register Banks When an Interrupt Function Is Executed" shows an example of an interrupt function for which the _ _nosavereg type qualifier is specified.

#pragma register(1) is specified before function int_timer( ) is defined.

When function int_timer( ) qualified by the _ _interrupt type qualifier and _ _nosavereg type qualifier is called, the code for switching the register bank to be used is output. When the register bank is switched, an area for the local variables used in the interrupt function is allocated.

When the function terminates, the saved registers are restored, and then the reti instruction is issued to restore the PS register value that was saved when the interrupt occurred. Control then returns to the register bank that was being used before the interrupt occurred.

Figure 14.3-4 Changing Register Banks When an Interrupt Function Is Executed



<Notes>

For a function qualified by the _ _interrupt type qualifier, always specify void as the function type.

When the interrupt processing terminates with the reti instruction, the registers that were saved to the system stack when the interrupt occurred are restored. Saving the register values enables the interrupted processing to be restarted. Because the registers are restored after the interrupt function returns a return value and terminates, the return value cannot be accessed. In addition, even though the return value has been placed on the stack, the location of the return value cannot be determined by the function gaining control after interrupt processing terminates because the stack returns to its pre-interrupt state by execution of the reti instruction. For this reason, the return value cannot be accessed. To

prevent such wasteful processing, type void must be specified for the interrupt function. If the processing results of an interrupt function are required, define an external variable where the processing results can be saved and accessed when necessary.

[Tip]

Softune C Checker processing:

The Softune C Checker will output a warning message for the location where the _ _interrupt type qualifier is specified indicating that a type qualifier for coding an interrupt function is used. The fcc896 and fcc911 support the _ _interrupt type qualifier. When porting, check this message to see whether the function should also be used in the new program system.

14.4 Setting of Interrupt Vectors

This section describes how to use #pragma intvect/defvect to register an interrupt function in an interrupt vector.
Using #pragma intvect enables a created interrupt function to be registered in an interrupt vector.

■ Using #pragma intvect/defvect to Register Interrupt Functions

When the hardware settings for executing interrupt processing and the definitions of the interrupt functions for the actual operation have been completed, the last step is to register the created interrupt functions.

The F^2MC-16 family provides interrupt vectors at addresses 0xFFFC00 to 0xFFFFFF. Registering the required interrupt processing functions in this area enables the required interrupt processing to be executed when an interrupt occurs.

See Table 14.2-1 "Relationship between Interrupt Sources, Interrupt Level Setting Registers, and Interrupt Vectors for MB90675" for the relationship between interrupt sources, interrupt control registers, and interrupt vectors.

The fcc907 uses #pragma intvect as follows to register interrupt functions.

pragma intvect interrupt-function-name interrupt-vector-number

Figure 14.4-1 Using #pragma intvect to Register an Interrupt Processing Function

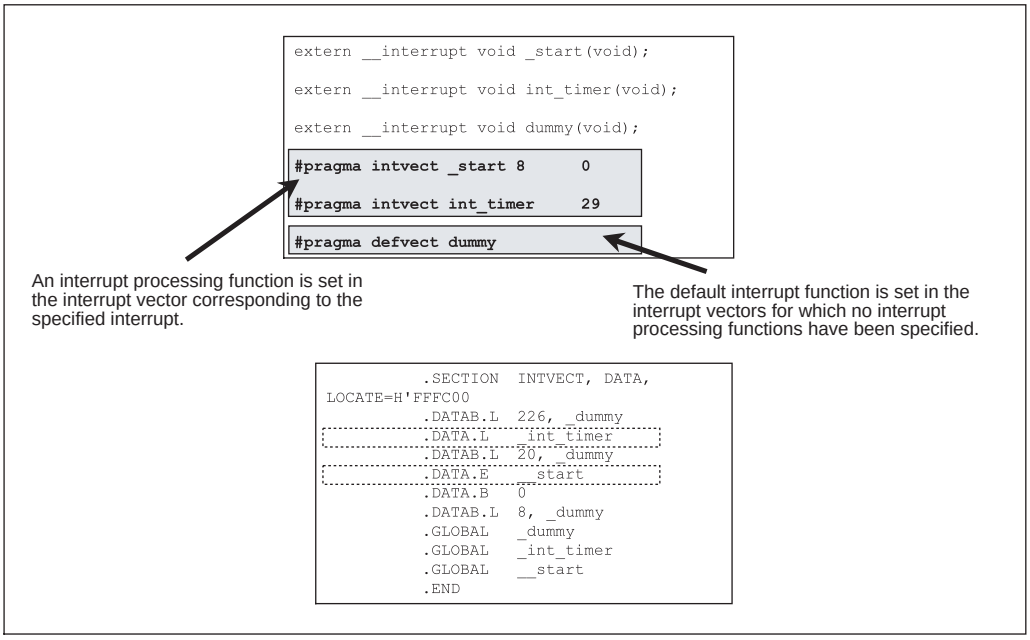


Figure 14.4-1 "Using #pragma intvect to Register an Interrupt Processing Function" shows an example of using #pragma intvect to register an interrupt processing function.

In this example, the startup routine start( ) is registered in interrupt vector 8 and the 16-bit reload timer interrupt processing function int_timer( ) is registered in interrupt vector 29.

When #pragma intvect is executed, the interrupt vector table INTVECT, which is allocated starting at address h'fffc00, is generated. The interrupt vectors that have not been assigned a vector number by #pragma intvect are filled with zeros. When #pragma defvect is executed, the specified interrupt function is set in all the vectors that that have been filled with 0. In the example shown in Figure 14.4-1 "Using #pragma intvect to Register an Interrupt Processing Function", default interrupt function dummy is specified with #pragma defvect. Function dummy is registered in all interrupt vectors except interrupt vector 8 and 29.

<Notes>

When using #pragma intvect to set an interrupt function in a vector table, always declare access for a function for which the _ _interrupt type qualifier has been specified before you code #pragma intvect. The fcc907 will output a warning message if the _ _interrupt type qualifier is omitted.

Registering of a function in an interrupt vector with #pragma intvect/defvect is only allowed in one module. If the function is registered in more than one module, an error message indicating that a section name has been specified multiple times may be output during linking.

PART IV MAPPING OBJECTS EFFECTIVELY

This part describes how to effectively map created programs into memory. When the fcc907 is used, the memory model to be selected depends on the scale of the system to be created. How objects are mapped into memory depends on the selected memory model. This part describes the following items:

- **Memory models and object efficiency**
- **Mapping variables qualified by the const type qualifier**
- **Mapping programs in which the code area exceeds 64 Kbytes**
- **Mapping programs in which the data area exceeds 64 Kbytes**

CHAPTER 15 MEMORY MODELS AND OBJECT EFFICIENCY

This chapter describes the memory models that can be used by the fcc907, including object efficiency thereof.

- **Small model**
- **Medium model**
- **Compact model**
- **Large model**

15.1 Four Memory Models

This section describes the memory models that can be used by the fcc907. The fcc907 has small-, medium-, compact-, and large-size memory models based on memory sizes capable of being handled.

■ Memory Models of the fcc907

The fcc907 has small-, medium-, compact-, and large-size memory models as shown in Figure 15.1-1 "fcc907 Memory Models" based on memory sizes capable of being handled.

Figure 15.1-1 fcc907 Memory Models

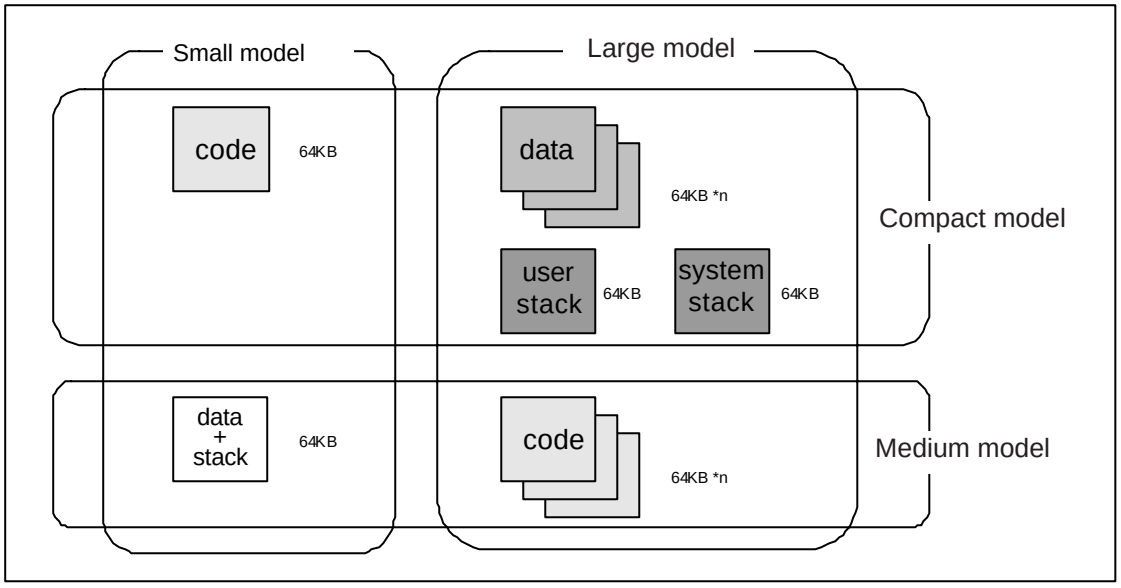


Table 15.1-1 "fcc907 Memory Models" lists the relationship between the memory models and memory areas that are handled. In addition, Table 15.1-2 "fcc907 Memory Models and Pointers" lists the relationship between the memory models and pointers at access.-

Table 15.1-1 fcc907 Memory Models

	Small model	Medium model	Compact model	Large model
Data area	One bank	One bank	Multiple banks	Multiple banks
System stack			One bank	One bank
User stack			One bank	One bank
Code area	One bank	Multiple banks	One bank	Multiple banks

Table 15.1-2 fcc907 Memory Models and Pointers

Memory model	Pointer to function	Pointer to variable
Small model	16 bits	
Medium model	24 bits	16 bits
Compact model	16 bits	24 bits
Large model	24 bits	

❍ **Small model**

For the code and data areas, 16-bit addressing objects are generated.

Specify a small model for a system that has code and data areas each within one bank (64 Kbytes). Then, when a function is accessed, the bank pointed to by the PCB is accessed using 16-bit addressing. When a variable is accessed, the bank pointed to by the DTB is accessed using 16-bit addressing.

❍ **Large model**

For the code and data areas, 24-bit addressing objects are generated.

Specify a large model for a system that uses multiple banks for the code and data areas. The data and code areas can be allocated at arbitrary locations in the memory space without being related to the PCB and DTB values. As a result, functions and variables are accessed using 24-bit addressing.

❍ **Medium model**

When data is accessed, an object of 16-bit addressing is generated. When code is accessed, a 24-bit addressing object is generated.

Specify a medium model for a system that has a data area within one bank (64 Kbytes). Then, when a variable is accessed, the bank pointed to by the DTB is accessed using 16-bit addressing.

❍ **Compact model**

When data is accessed, a 24-bit addressing object is generated. When code is accessed, a 16-bit addressing object is generated.

Specify a compact model for a system that has a code area within one bank (64 Kbytes). Then, when a function is accessed, the bank pointed to by the PCB is accessed using 16-bit addressing.

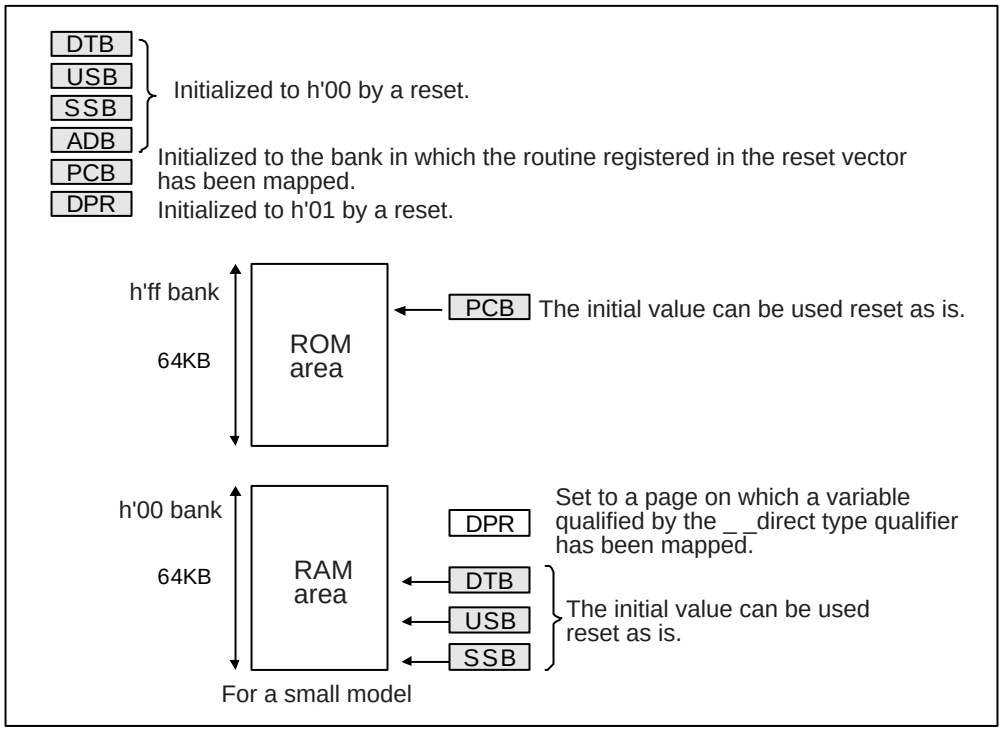
■ Memory Models and Bank Registers

The F^{2}MC-16 Family uses a reset signal to initialize the bank registers. At this time, the four registers DTB, ADB, USB, and SSB are initialized to h'00. The PCB is initialized into the bank where the routine registered in the reset vector has been mapped. In addition, at a reset, the DPR is initialized to h'01.

For a small- or medium-size model where data is accessed using 16-bit addressing, the three registers DTB, USB, and SSB must be initialized so as to point to the same bank. For a small- or medium-size model, the I/O area has been allocated in the h'00 bank. Therefore, the three registers DTB, USB, and SSB are initialized so as to point to the h'00 bank.

The three registers DTB, USB, and SSB can thus use the values initialized by a reset as is. Because a reset initializes the DPR to h'01, the DPR must be set to a page on which a variable qualified by the _ _direct type qualifier has been mapped.

Figure 15.1-2 Initializing the Bank Registers (for a Small Model)



For a compact or large model where the data area is accessed using 24-bit addressing, the restriction where the three registers are set to h'00 does not apply. However, a bank in which a variable qualified by the _ _near type qualifier and a variable qualified by the _ _direct type qualifier have been mapped must be set in the DTB register. In addition, a page on which a variable qualified by the _ _direct type qualifier has been mapped must be set in the DPR register. Use the startup routine to initialize these registers to values that match the system to be created.

For details on the DTB, DPR, USB, SSP, and PCB registers, refer to the manual of the respective hardware.

15.2 Large Models and Object Efficiency

When a large model is used, compilation generates 24-bit addressing objects. The code size for 24-bit addressing is larger and the execution speed is lower than for 16-bit addressing.
For a system in which the data area or code area exceeds 64 Kbytes, using a large, medium, or compact model can reduce program efficiency and execution speed. These problems can be avoided by selecting optimum object mapping.

■ Generated Objects of Small and Large Models

This section explains the difference between generated objects when the same source file is compiled using a small model and a large model.

Figure 15.2-1 "s_f_dif.c Source File" shows the source file (s_f_dif.c) to be compiled. This section explains the difference when this source file is compiled using a small model and when it is compiled using a large model.

Figure 15.2-1 s_f_dif.c Source File

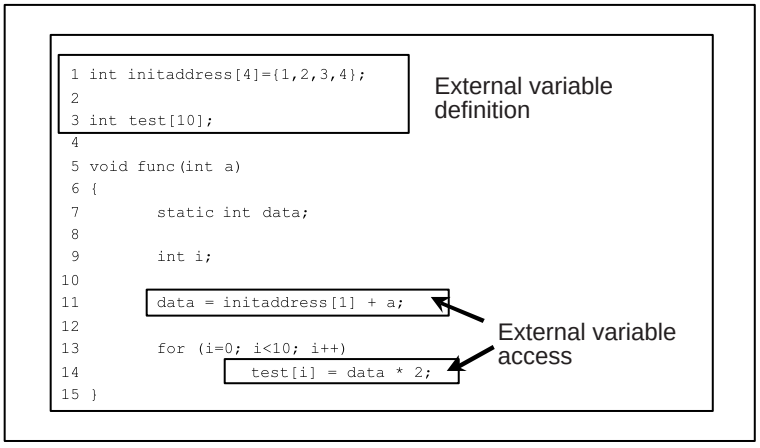
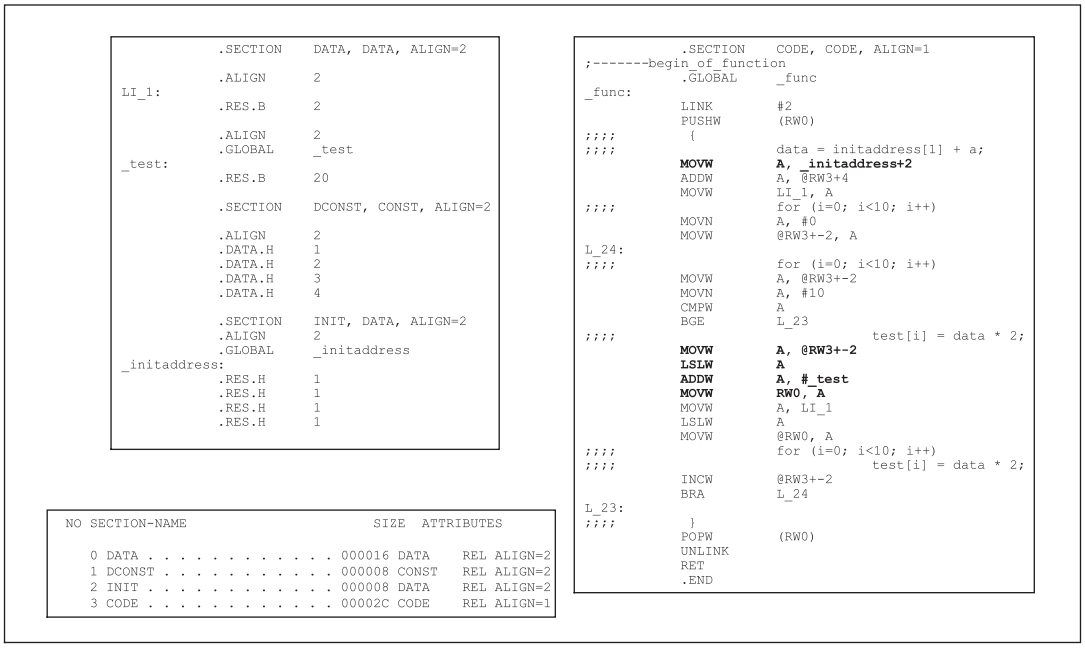


Figure 15.2-2 "Source File Compiled Using a Small Model" shows the assembler source file when s_f_dif.c is compiled using a small model. Figure 15.2-3 "Source File Compiled Using a Large Model" shows the assembler source file when s_f_dif.c is compiled using a large model.

When the source file is compiled using a small model, the code size is h'2c bytes. When the source file is compiled using a large model, the code size is h'41 bytes. The difference is that, for a small model, a code for 16-bit addressing is generated when an external variable is accessed and a code for 24-bit addressing is generated for a large model.

Figure 15.2-2 Source File Compiled Using a Small Model



```
                .SECTION    DATA, DATA, ALIGN=2

                .ALIGN      2
LI_1:
                .RES.B      2

                .ALIGN      2
                .GLOBAL     _test
_test:
                .RES.B      20

                .SECTION    DCONST, CONST, ALIGN=2

                .ALIGN      2
                .DATA.H     1
                .DATA.H     2
                .DATA.H     3
                .DATA.H     4

                .SECTION    INIT, DATA, ALIGN=2
                .ALIGN      2
                .GLOBAL     _initaddress
_initaddress:
                .RES.H      1
                .RES.H      1
                .RES.H      1
                .RES.H      1
```

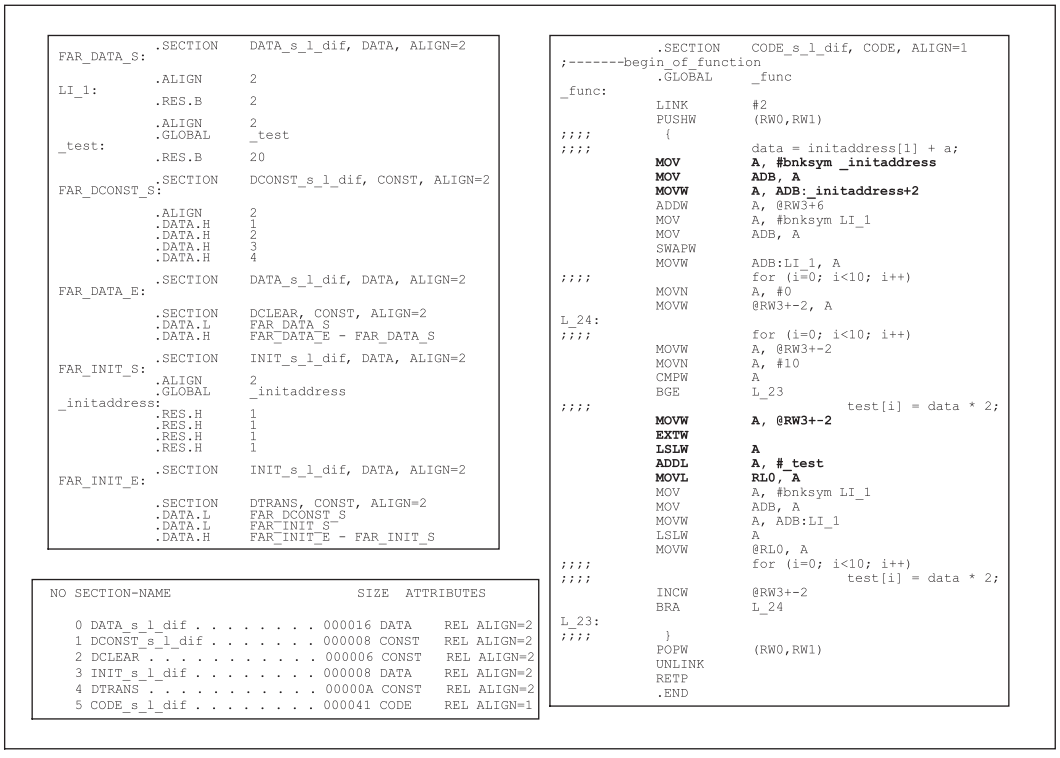
```
NO SECTION-NAME                         SIZE  ATTRIBUTES

   0 DATA . . . . . . . . . . . . 000016 DATA    REL ALIGN=2
   1 DCONST . . . . . . . . . . . 000008 CONST   REL ALIGN=2
   2 INIT . . . . . . . . . . . . 000008 DATA    REL ALIGN=2
   3 CODE . . . . . . . . . . . . 00002C CODE    REL ALIGN=1
```

```
                .SECTION    CODE, CODE, ALIGN=1
;-------begin_of_function
                .GLOBAL     _func
_func:
                LINK        #2
                PUSHW       (RW0)
;;;;             {
;;;;                        data = initaddress[1] + a;
                MOVW        A, _initaddress+2
                ADDW        A, @RW3+4
                MOVW        LI_1, A
;;;;                        for (i=0; i<10; i++)
                MOVN        A, #0
                MOVW        @RW3+-2, A
L_24:
;;;;                        for (i=0; i<10; i++)
                MOVW        A, @RW3+-2
                MOVN        A, #10
                CMPW        A
                BGE         L_23
;;;;                                    test[i] = data * 2;
                MOVW        A, @RW3+-2
                LSLW        A
                ADDW        A, #_test
                MOVW        RW0, A
                MOVW        A, LI_1
                LSLW        A
                MOVW        @RW0, A
;;;;                        for (i=0; i<10; i++)
;;;;                                    test[i] = data * 2;
                INCW        @RW3+-2
                BRA         L_24
L_23:
;;;;             }
                POPW        (RW0)
                UNLINK
                RET
                .END
```

There is no problem when creating a small model system in which the code and data areas are each within 64 Kbytes. However, for a system in which the data area or code area exceeds 64 Kbytes, even marginally, using a large, medium, or compact model can reduce program efficiency and execution speed. These problems can be avoided by planning object mapping.

Object mapping is described in CHAPTER 17 "MAPPING PROGRAMS IN WHICH THE CODE AREA EXCEEDS 64 Kbytes" and CHAPTER 18 "MAPPING PROGRAMS IN WHICH THE DATA AREA EXCEEDS 64 Kbytes".

Figure 15.2-3 Source File Compiled Using a Large Model



```
                .SECTION    DATA_s_l_dif, DATA, ALIGN=2
FAR_DATA_S:
                .ALIGN      2
LI_1:
                .RES.B      2

                .ALIGN      2
                .GLOBAL     _test
_test:
                .RES.B      20

                .SECTION    DCONST_s_l_dif, CONST, ALIGN=2
FAR_DCONST_S:
                .ALIGN      2
                .DATA.H     1
                .DATA.H     2
                .DATA.H     3
                .DATA.H     4

                .SECTION    DATA_s_l_dif, DATA, ALIGN=2
FAR_DATA_E:
                .SECTION    DCLEAR, CONST, ALIGN=2
                .DATA.L     FAR_DATA_S
                .DATA.H     FAR_DATA_E - FAR_DATA_S

                .SECTION    INIT_s_l_dif, DATA, ALIGN=2
FAR_INIT_S:
                .ALIGN      2
                .GLOBAL     _initaddress
_initaddress:
                .RES.H      1
                .RES.H      1
                .RES.H      1
                .RES.H      1

                .SECTION    INIT_s_l_dif, DATA, ALIGN=2
FAR_INIT_E:
                .SECTION    DTRANS, CONST, ALIGN=2
                .DATA.L     FAR_DCONST_S
                .DATA.L     FAR_INIT_S
                .DATA.H     FAR_INIT_E - FAR_INIT_S
```

```
NO SECTION-NAME                         SIZE  ATTRIBUTES

   0 DATA_s_l_dif . . . . . . . . 000016 DATA    REL ALIGN=2
   1 DCONST_s_l_dif . . . . . . . 000008 CONST   REL ALIGN=2
   2 DCLEAR . . . . . . . . . . . 000006 CONST   REL ALIGN=2
   3 INIT_s_l_dif . . . . . . . . 000008 DATA    REL ALIGN=2
   4 DTRANS . . . . . . . . . . . 00000A CONST   REL ALIGN=2
   5 CODE_s_l_dif . . . . . . . . 000041 CODE    REL ALIGN=1
```

```
                .SECTION    CODE_s_l_dif, CODE, ALIGN=1
;-------begin_of_function
                .GLOBAL     _func
_func:
                LINK        #2
                PUSHW       (RW0,RW1)
;;;;             {
;;;;                        data = initaddress[1] + a;
                MOV         A, #bnksym _initaddress
                MOV         ADB, A
                MOVW        A, ADB:_initaddress+2
                ADDW        A, @RW3+6
                MOV         A, #bnksym LI_1
                MOV         ADB, A
                SWAPW
                MOVW        ADB:LI_1, A
;;;;                        for (i=0; i<10; i++)
                MOVN        A, #0
                MOVW        @RW3+-2, A
L_24:
;;;;                        for (i=0; i<10; i++)
                MOVW        A, @RW3+-2
                MOVN        A, #10
                CMPW        A
                BGE         L_23
;;;;                                    test[i] = data * 2;
                MOVW        A, @RW3+-2
                EXTW
                LSLW        A
                ADDL        A, #_test
                MOVL        RL0, A
                MOV         A, #bnksym LI_1
                MOV         ADB, A
                MOVW        A, ADB:LI_1
                LSLW        A
                MOVW        @RL0, A
;;;;                        for (i=0; i<10; i++)
;;;;                                    test[i] = data * 2;
                INCW        @RW3+-2
                BRA         L_24
L_23:
;;;;             }
                POPW        (RW0,RW1)
                UNLINK
                RETP
                .END
```

CHAPTER 16 MAPPING VARIABLES QUALIFIED WITH THE TYPE QUALIFIER CONST

This chapter describes mapping of variables that have been qualified by the const type qualifier. The F^{2}MC-16L/LX/F series has a function referred to as the mirror ROM function. For small and medium models, this function enables variables mapped in the ROM area to be accessed using 16-bit addressing.

16.1 Using the Mirror ROM Function and const Type Qualifier

This section provides notes on mapping variables qualified by the const type qualifier for hardware that supports the mirror ROM function.
By mapping variables in the areas defined for the hardware, variables in the ROM area can be accessed using 16-bit addressing.

■ What Is the Mirror ROM Function?

The F^{2}MC-16L/LX/F series has a function referred to as the mirror ROM function. When area defined in the h'00 bank is accessed using 16-bit addressing, the mirror ROM function automatically accesses the same area in the ROM area of the h'ff bank using 16-bit addressing.

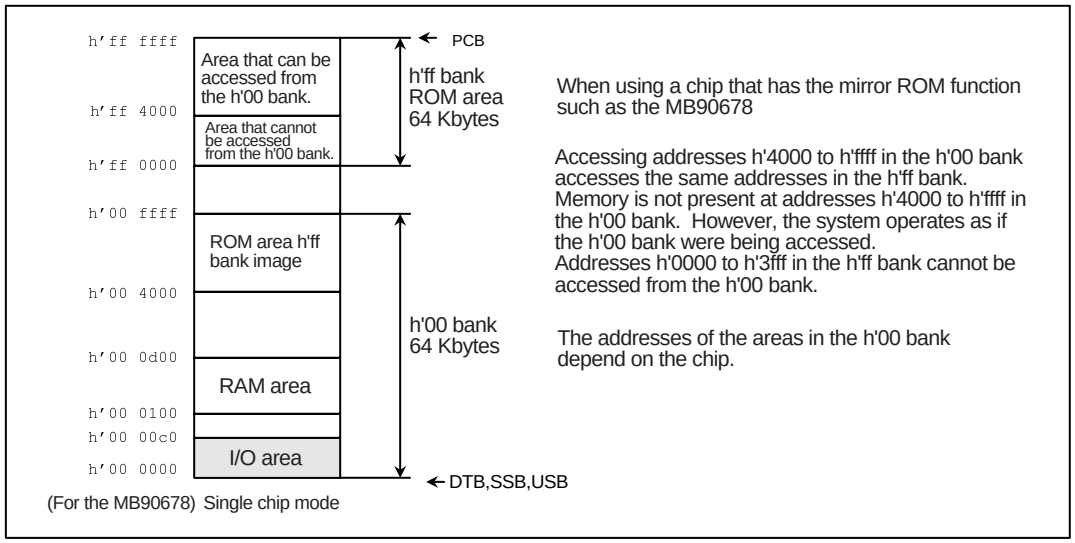
As a result, a variable qualified by the const type qualifier that has been mapped in the ROM area can be accessed using 16-bit addressing in the same way as a standard variable mapped in the h'00 bank.

Figure 16.1-1 "Accessing Variables Qualified by the const Type Qualifier for Hardware That Supports the Mirror ROM Function" shows an access image of variables qualified by a const type qualifier for hardware that supports the mirror ROM function.

Sections 16.1.1 "const Type Qualifier and Mirror ROM Function for Small and Medium Models" and 16.1.2 "const Type Qualifier and Mirror ROM Function for Compact and Large Models" provide notes on each memory model.

The mirror ROM function depends on the hardware of the F^{2}MC-16L/LX/F series. For details, refer to the hardware manual.

Figure 16.1-1 Accessing Variables Qualified by the const Type Qualifier for Hardware That Supports the Mirror ROM Function



16.1.1 const Type Qualifier and Mirror ROM Function for Small and Medium Models

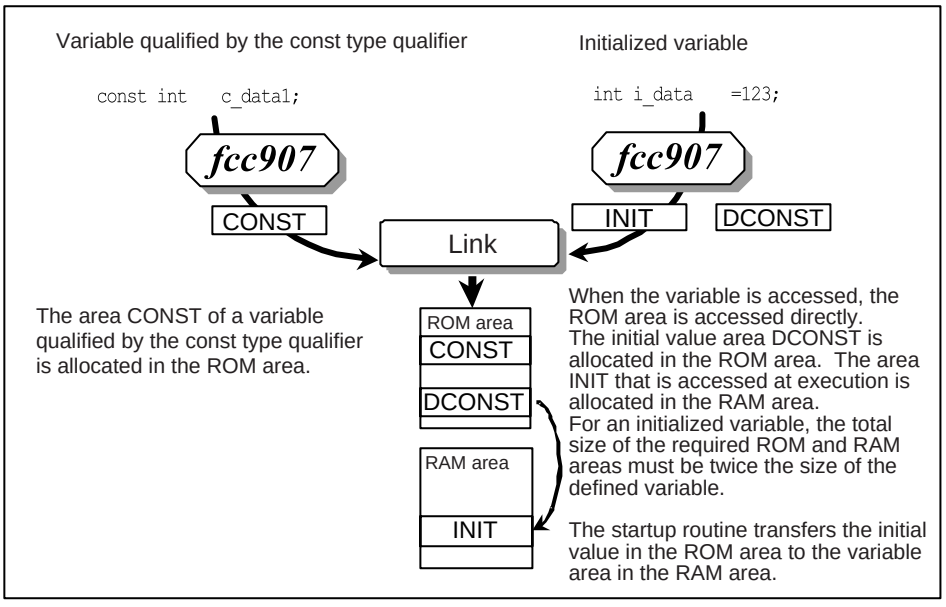
For small and medium models in which the data area is restricted to within 64 Kbytes, variables are accessed using 16-bit addressing on the premise that the variables are mapped in the bank pointed to by the DTB.
The mirror ROM function enables variables that are mapped in the h'ff bank to be accessed using 16-bit addressing.

■ Allocating Sections of Initialized Variables

For small and medium models, the data area that can be used is restricted to one bank within 64 Kbytes. As a result, variables are accessed using 16-bit addressing on the premise that the variables are in the bank pointed to by the DTB.

Figure 16.1-2 "Output Sections and Their Allocation for Small and Medium Models" shows the relationship between the output sections of variables for which initial values are specified and their allocation in memory for small and medium models.

Figure 16.1-2 Output Sections and Their Allocation for Small and Medium Models



For small and medium models, a variable qualified by the const type qualifier is output to the CONST section. At linkage, this CONST section is allocated in the ROM area of the h'ff bank. Normally, a CONST section present somewhere other than the bank pointed to by the DTB cannot be accessed using 16-bit addressing. If hardware that supports the mirror ROM function is used, however, a variable in the CONST section allocated at a defined location in the ROM area can be accessed using 16-bit addressing.

■ Notes on Using the Mirror ROM Function

The areas at which variables in the h'ff bank can be accessed by the mirror ROM function using 16-bit addressing depend on the hardware of the F^{2}MC-16L/LX/F series. Table 16.1-1 "Scope of Use of the Mirror ROM Function" lists the areas supported by the MB90670 series.

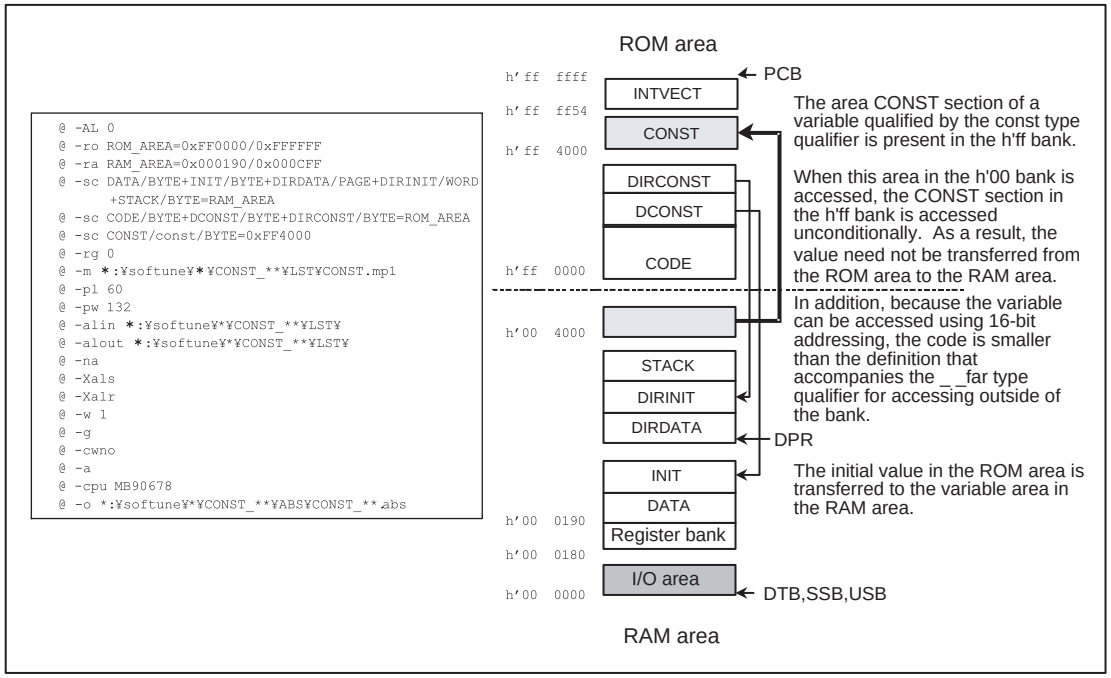
Table 16.1-1 Scope of Use of the Mirror ROM Function

Product	MB90671	MB90672	MB90673	MB90P673
Starting address	h'ffc000	h'ff8000	h'ff4000	h'ff4000
Ending address	h'ffffff	h'ffffff	h'ffffff	h'ffffff

Variables mapped within the range listed above can be accessed using 16-bit addressing in the same way as accessing other variables mapped by a function in the h'00 bank. This is possible because, when addresses h'0000 to h'ffff, h'8000 to h'ffff, or h'c000 to h'fff in the h'00 bank are accessed, the CPU unconditionally accesses the same area in the h'ff bank. Therefore, when the area CONST section of a variable qualified by the const type qualifier is allocated within the range listed above, the variable area in the ROM area can be accessed directly using 16-bit addressing without using the _ _far type qualifier. As a result, using the startup routine to transfer the initial values from the ROM area to the RAM area can be omitted for a variable qualified by the const type qualifier. Because variables qualified by the const type qualifier are present in the ROM area, initial values can of course be set at definition but the values cannot be changed at execution.

Figure 16.1-3 "Using the Mirror ROM Function and Allocating Areas of a Variable Qualified by the const Type Qualifier (for a Small Model)" shows an example of allocating areas of a variable qualified by the const type qualifier for a small model.

Figure 16.1-3 Using the Mirror ROM Function and Allocating Areas of a Variable Qualified by the const Type Qualifier (for a Small Model)



<Notes>

Note the following points regarding use of the mirror ROM function:

- Map variables qualified by the const type qualifier up to address h'ff53 in the h'ff bank.

Interrupt vectors are mapped between addresses h'ff54 and h'ffff in the h'ff bank. If a variable is mapped in this area, interrupt operations will be unpredictable.

- Allocate the area for variables qualified by the const type qualifier so that the area is accommodated in the area determined for each chip in the h'ff bank. If a variable exceeds the area, accessing using 16-bit addressing will not be possible.

- Do not allocate variable area or a stack in the area determined for each chip such as addresses h'4000 to h'ffff or h'8000 to h'ffff in the h'00 bank. Because the h'ff bank is accessed, the value of a variable or stack in this area will be unpredictable.

16.1.2 const Type Qualifier and Mirror ROM Function for Compact and Large Models

For compact and large models, variables are accessed using 24-bit addressing. Therefore, the restriction dependent on the setting of the DTB register for small and compact models does not apply.

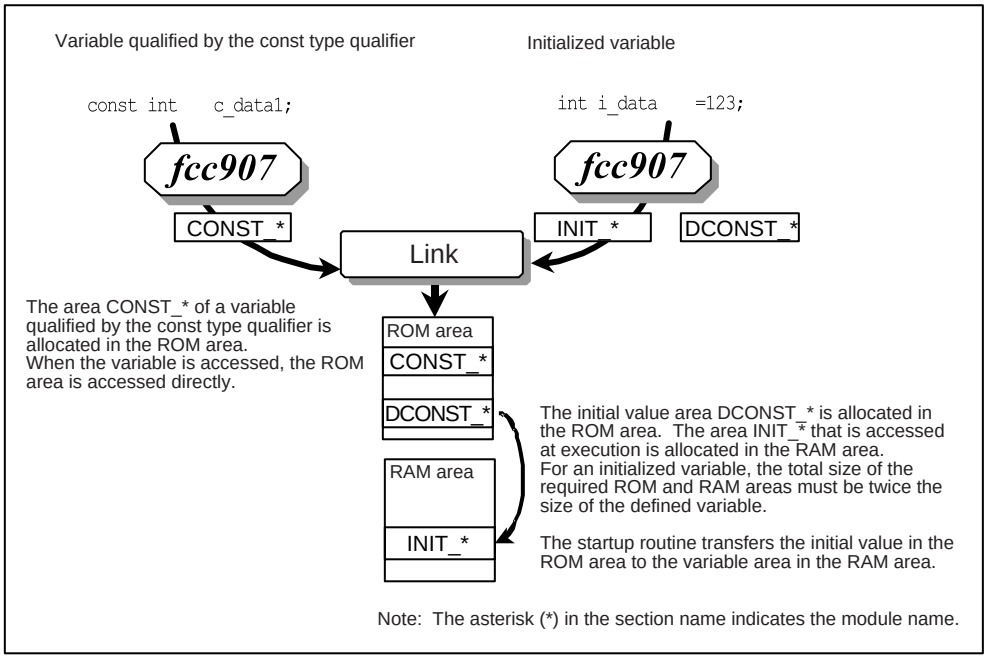
■ Allocating Sections of Initialized Variables

For compact and large models, the variable area can be allocated in multiple banks. The variables are always accessed using 24-bit addressing. Therefore, the restriction dependent on the setting of the DTB register for small and compact models does not apply. The bank pointed to by the DTB register is accessed using 16-bit addressing only when a variable qualified by the _ _near type qualifier is accessed.

When defining a variable qualified by the _ _const type qualifier, specify the _ _const type qualifier only, or specify the _ _const type qualifier and the _ _far type qualifier. For compact and large models, a variable qualified by the _ _const type qualifier is output to a section called "CONST_module name."

Figure 16.1-4 "Output Sections and Their Allocation for Compact and Large Models" shows the relationship between the output sections of variables for which initial values are specified and their allocation in memory for compact and large models.

Figure 16.1-4 Output Sections and Their Allocation for Compact and Large Models



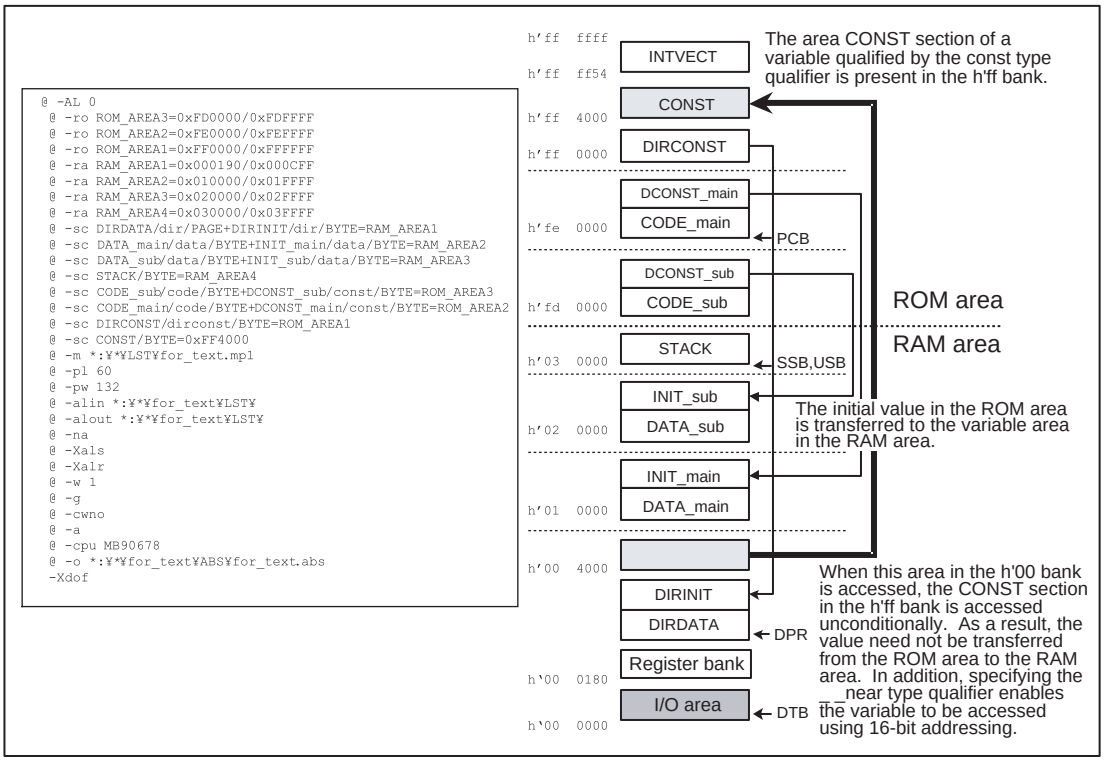
For compact and large models, a variable qualified by the const type qualifier is output to a section called "CONST_module name." At linkage, this CONST_module section is allocated in the ROM area. Because a variable is accessed using 24-bit addressing, the ROM area can be accessed directly.

■ Notes on Using the Mirror ROM Function

The areas where variables in the h'ff bank can be accessed by the mirror ROM function using 16-bit addressing depend on the hardware of the F^{2}MC-16L/LX/F series.

For compact and large models, specify the const type and _ _near type qualifiers when the mirror ROM function is used to access a variable qualified by the const type qualifier using 16-bit addressing. The variable will then be output to the CONST section that can be accessed when the bank pointed to by the DTB register is accessed using 16-bit addressing. At linkage, allocate this CONST section in an area supported by the mirror ROM function.

Figure 16.1-5 Using the Mirror ROM Function and Allocating Areas of a Variable Qualified by the const Type Qualifier (for a Large Model)



16.2 const Type Qualifier When the Mirror ROM Function Cannot Be Used

This section provides notes on mapping variables qualified by the const type qualifier for hardware that does not support the mirror ROM function.
The -ramconst option can be specified to output a section allocated to the ROM and RAM areas. In addition, specifying the const type and _ _far type qualifiers enables the variable area in the ROM area to be accessed directly using 24-bit addressing.

■ Mapping Variables Qualified by the const Type Qualifier for Hardware That Does Not Support the Mirror ROM Function

The F^2MC-16L/LX/F series includes hardware that does not support the mirror ROM function. For systems that use such hardware, the ROM area in the h'ff bank cannot be accessed using 16-bit addressing. This applies to small and compact models where the data area is accessed using 16-bit addressing.

For these types of systems, the following two methods are available for mapping variables qualified by the const type qualifier:

- Specify the -ramconst option at compilation.
- Specify the const type and _ _far type qualifiers at definition.

Sections 16.2.1 "Mapping Variables Qualified by the const Type Qualifier to RAM Area" and 16.2.2 "Specifying the const Type and _ _far Type Qualifiers at Definition" describe these two methods.

For compact and large models where the data area is accessed using 24-bit addressing, because the variable area in the ROM area can be accessed directly, the above problem does not occur.

16.2.1 Mapping Variables Qualified by the const Type Qualifier to RAM Area

For small and medium models in which the mirror ROM function cannot be used because the data area is restricted to within 64 Kbytes, specify the -ramconst option at compilation. Specifying the -ramconst option will enable the area of a variable qualified by the const type qualifier to be mapped in the RAM area in the same way as a normal variable.

■ Specification of the -ramconst Option and Output Sections

If hardware not supporting the mirror ROM function is used, a method is available for mapping a variable qualified by the const type qualifier in the RAM area in the same way as a normal variable.

In this case, specify the -ramconst option at compilation. Specifying the -ramconst option will output the areas of a variable qualified by the const type qualifier to the CONST and CINIT sections. The CONST section is allocated in the ROM area. The CINIT section is allocated in the RAM area. The startup routine transfers the initial value in the CONST section to the CINIT section. The CINIT section in the RAM area is accessed from a function. When a program is executed, this CINIT section becomes read-only.

Figure 16.2-1 "Specifying the -ramconst Option (for a Small Model)" shows the relationship between the output sections when the -ramconst option is specified for a small model.

Figure 16.2-1 Specifying the -ramconst Option (for a Small Model)

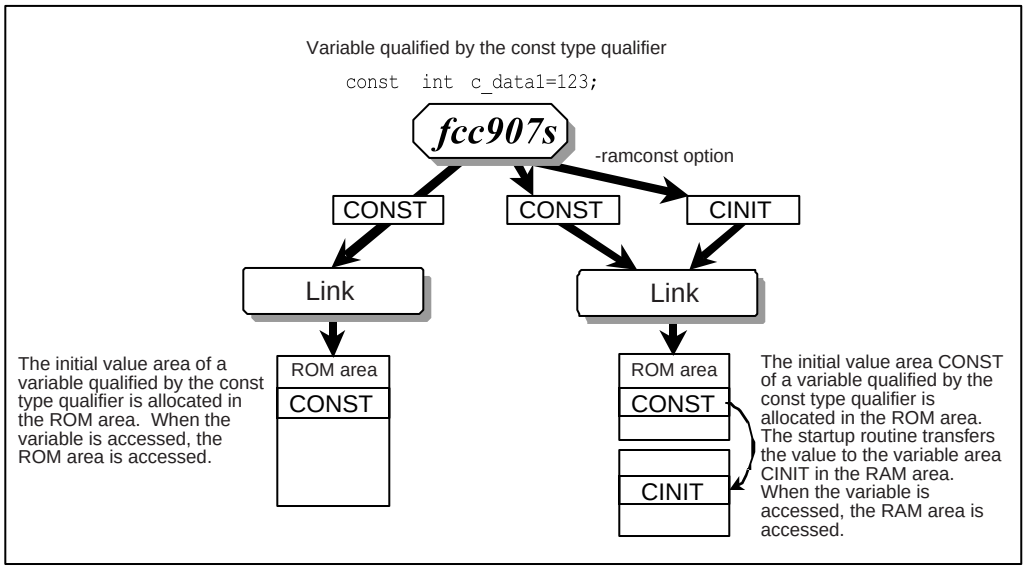
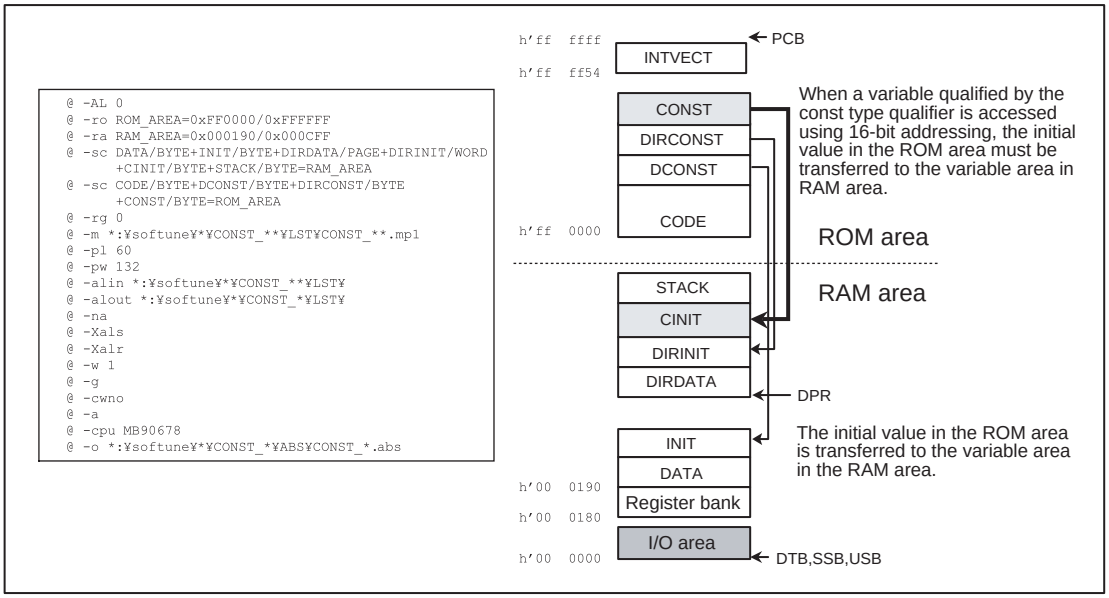


Figure 16.2-2 "Mapping a Variable Qualified by the const Type Qualifier to RAM Area (for a Small Model)" is an example of mapping when the -ramconst option is specified for a small model.

In this example, the CONST section is allocated in the h'ff bank of the ROM area and the CINIT section is allocated in the h'00 bank of the RAM area at linkage. The startup routine transfers the value from the CONST section to the CINIT section.

Figure 16.2-2 Mapping a Variable Qualified by the const Type Qualifier to RAM Area (for a Small Model)



16.2.2 Specifying the const Type and _ _far Type Qualifiers at Definition

For small and medium models where the data area is restricted to within 64 Kbytes, variables are accessed using 16-bit addressing on the premise that the variables are mapped in the bank pointed to by the DTB.
If hardware not supporting the mirror ROM function is used, specifying the const type and _ _far type qualifiers will enable the variable area in the ROM area to be accessed directly using 24-bit addressing.

■ Output Sections of Variables Qualified by the const Type and _ _far Type Qualifiers

For small and medium models, the available data area is restricted to within one bank (64 Kbytes). Variables are therefore accessed using 16-bit addressing on the premise that the variables are mapped in the bank pointed to by the DTB.

For hardware not supporting the mirror ROM function, specify the const type and _ _far type qualifiers to enable direct access of a variable qualified by the const type qualifier in the ROM area. A code for 24-bit addressing will then be generated only when a variable qualified by the const type qualifier that has been mapped in the ROM area is accessed.

Figure 16.2-3 "Output Sections of a Variable Qualified by the const Type and _ _far Type Qualifiers (for a Small Model)" shows the relationship between the output sections of a variable for which an initial value has been specified and the sections of a variable qualified by const type and _ _far type qualifiers. This applies to small and medium models.

Figure 16.2-3 Output Sections of a Variable Qualified by const Type and _ _far Type Qualifiers (for a Small Model)

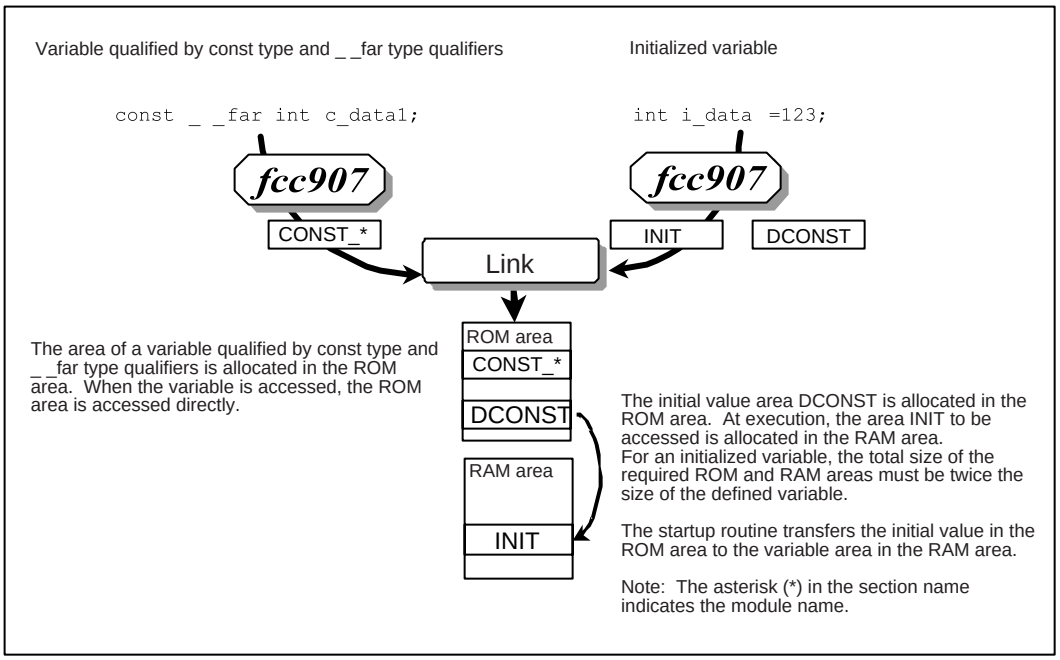
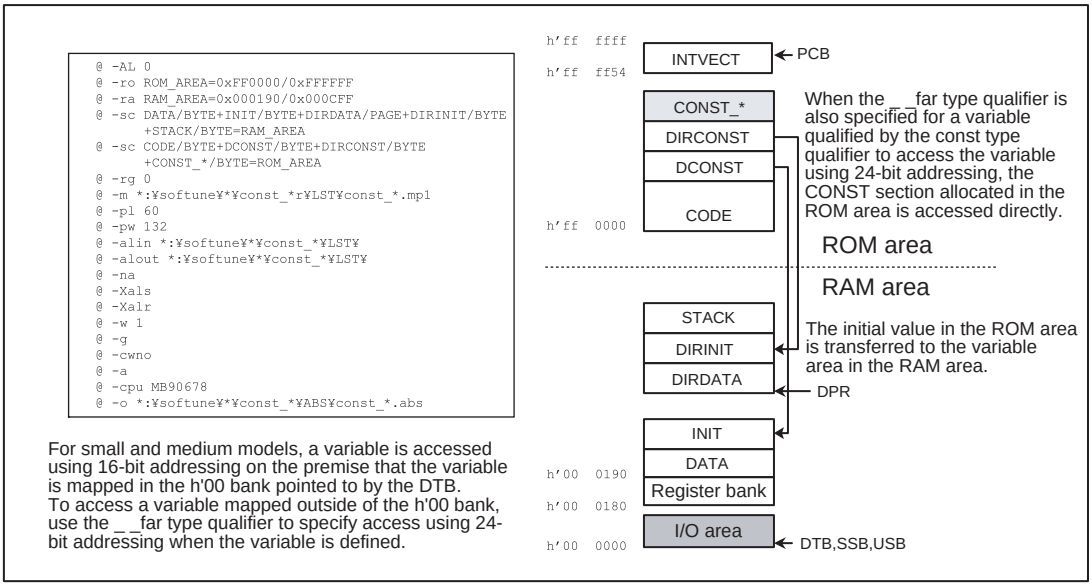


Figure 16.2-4 "Mapping a Variable Qualified by the const Type and _ _far Type Qualifiers (for a Small Model)" is an example of mapping when a variable qualified by const type and _ _far type qualifiers is defined for a small model.

In this example, the const_* section is allocated in the h'ff bank of the ROM area at linkage. A code for 24-bit addressing is generated only when a variable mapped in the const_* section is accessed.

Figure 16.2-4 Mapping a Variable Qualified by const Type and _ _far Type Qualifiers (for a Small Model)



CHAPTER 17 MAPPING PROGRAMS IN WHICH THE CODE AREA EXCEEDS 64 Kbytes

This chapter describes how to map programs in which the code area of the program to be created exceeds 64 Kbytes.
For a system in which the code area exceeds 64 Kbytes, it is recommended that a small or compact model be used and that the _ _far type qualifier be specified in the functions.

17.1 Functions Calls of Programs in Which the Code Area Exceeds 64 Kbytes

When creating a system in which the code area exceeds 64 Kbytes, use a medium or large model in which the functions are called using 24-bit addressing.
For a system in which the code area exceeds 64 Kbytes, however, function calls using 24-bit addressing can increase the code size.

■ Function Calls Using 24-Bit Addressing

Table 17.1-1 "Type Qualifiers, Memory Models, and Code Section Names" lists the relationship between the output code section names for type qualifiers and memory models of the functions.

Table 17.1-1 Type Qualifiers, Memory Models, and Code Section Names

Type qualifier specification	Small or compact model	Large or medium model
None	CODE	CODE_module name
_ _near	CODE	CODE_module name
_ _far	CODE_module name	CODE_module name

As listed in Table 15.1-1 "fcc907 Memory Models" the fcc907 uses a medium or large model when creating a program in which the code area for the entire system exceeds 64 Kbytes.

For a medium or large model, a code for 24-bit addressing is generated unconditionally when a function is called. When multiple banks are used and there are frequent calls between the banks, a problem will not occur even if a code for 24-bit addressing is generated. For a system in which the code area exceeds one bank (64 Kbytes), accessing functions using 24-bit addressing can increase the size of the code area.

For small or compact models in which function calls are accessed using 16-bit addressing, a function qualified by the _ _far type qualifier can be accessed using 24-bit addressing. Section 17.2 "Using Calls for Functions Qualified by the _ _far Type Qualifier" explains how to define and map functions qualified by the _ _far type qualifier for small and compact models.

17.2 Using Calls For Functions Qualified by the _ _far Type Qualifier

**This section describes how to specify the _ _far type qualifier in a function for small and compact models in which functions are accessed using 16-bit addressing.
It is recommended that the _ _far type qualifier be specified for functions that are not frequently called or functions that are called from all functions.**

■ Specifying the _ _far Type Qualifier in a Function for Small and Compact Models

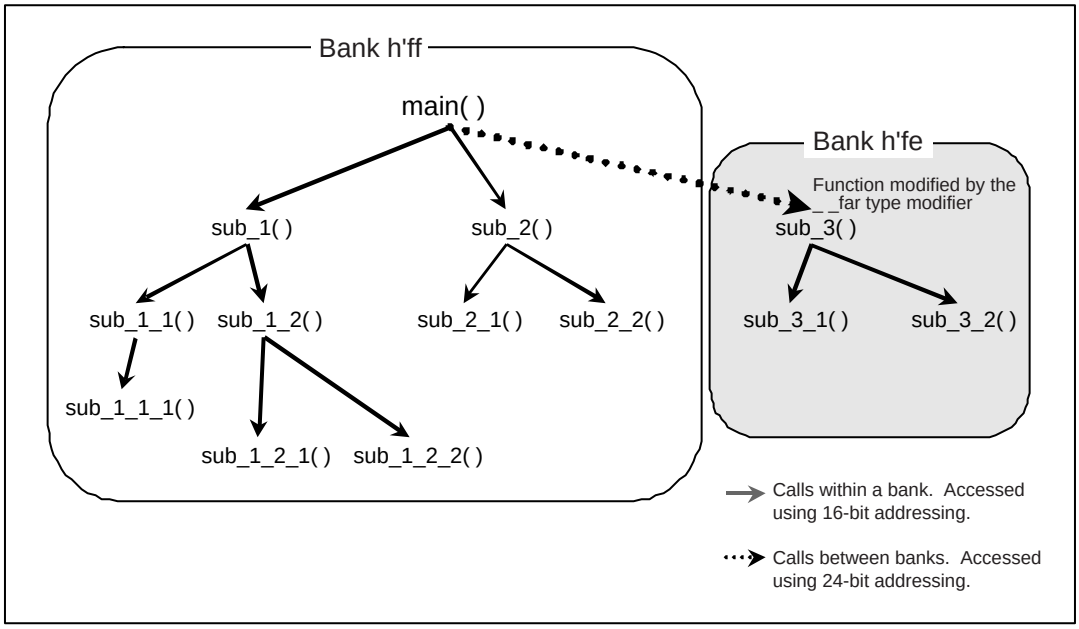
When creating a system in which the code area exceeds 64 Kbytes, a code for 24-bit addressing will be generated if a medium or large model in which all functions are accessed using 24-bit addressing is used. Even for a small model in which calling within a bank is a default, the _ _far type qualifier can be specified to generate a code for 24-bit addressing.

When creating a system in which the code area exceeds 64 Kbytes, it is recommended that the _ _far type qualifier be specified for some of the functions at compilation for a small or compact model.

■ Dividing Modules and Specifying the _ _far Type Qualifier in a Function

The tree structure shown in Figure 17.2-1 "Function Call Relationship and Mapping Image 1" is assumed for the relationship of all function calls in the system to be developed.

Figure 17.2-1 Function Call Relationship and Mapping Image 1



In this example, function main( ) calls the three functions sub_1( ), sub_2( ), and sub_3( ). For subsequent functions sub_1_xx( ), it is assumed that functions sub_2_xx( ) and sub_3_xx( ) are also called using the same route via sub_1( ). In addition, function sub_3( ) is not frequently called.

The relationship of these calls is used to divide the banks in which the functions are to be

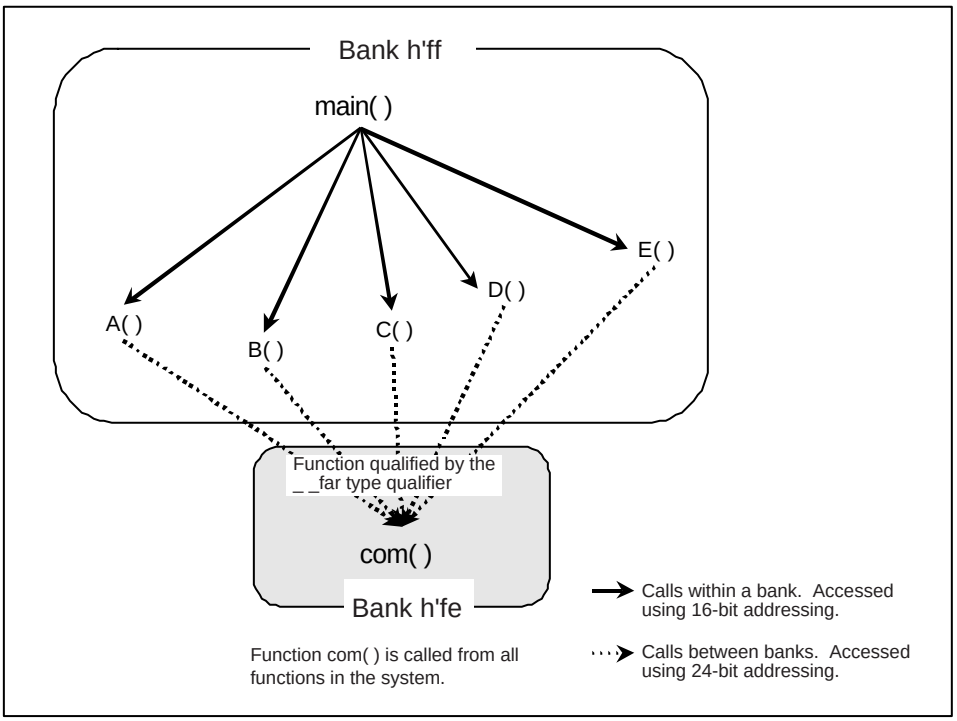
mapped. In this example, function sub_3( ) not called frequently and function sub_3( ) are mapped in bank h'fe. The other functions are mapped in bank h'ff.

When a system in which the calls have this type of relationship is compiled using a medium or large model, a code for 24-bit addressing will be generated for all function calls. Even when function sub_1_1( ) is called from function sub_1( ), a code for 24-bit addressing will be generated in the same way as when function sub_1( ) is called in the same bank from function main( ).

Assume that the _ _far type qualifier is specified in function sub_3( ) for compilation using a small or compact model. Then, when the function is mapped as shown in Figure 17.2-1 "Function Call Relationship and Mapping Image 1" a call for outside the bank using 24-bit addressing will be generated only when function sub_3( ) is called from function main( ). For all other functions, the functions will be called using 16-bit addressing within the bank.

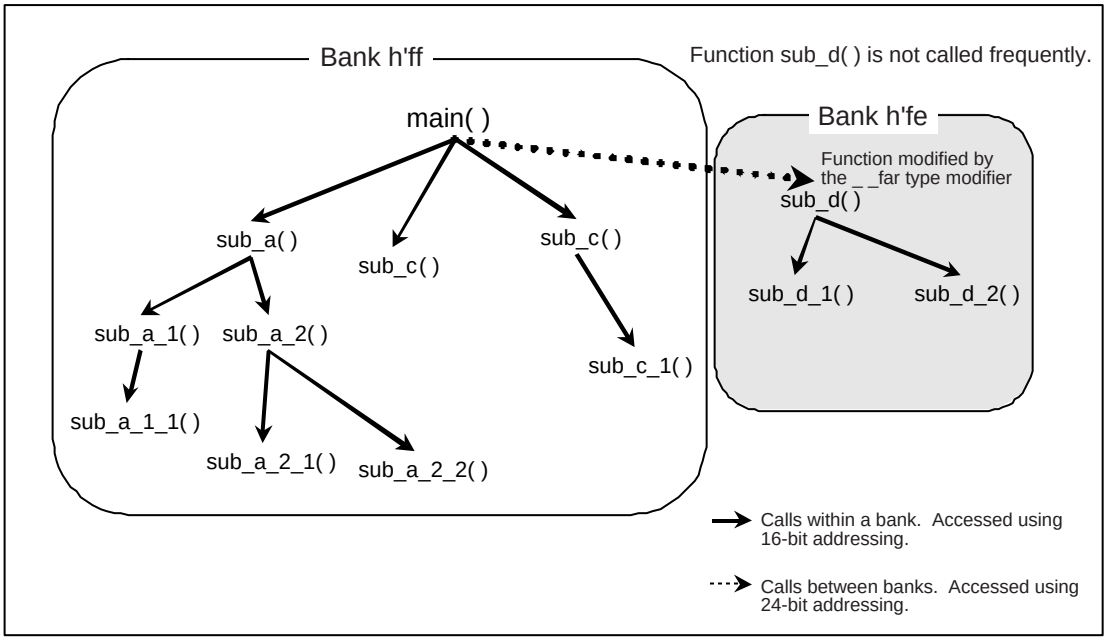
As shown in this example, it is recommended that the _ _far type qualifier be specified for small and compact models in which processing of the functions can be easily divided. Using the _ _far type qualifier can reduce the size of the code and increase execution speed.

Figure 17.2-2 Function Call Relationship and Mapping Image 2



Two methods are available if processing of the functions cannot be easily divided. In one method, as shown in Figure 17.2-2 "Function Call Relationship and Mapping Image 2" specify the _ _far type qualifier to map a common function into a separate bank because the common function can be called from all locations in a system. In the other method, as shown in Figure 17.2-3 "Function Call Relationship and Mapping Image 3" specify the _ _far type qualifier to map a function that is not called frequently into a separate bank. Determine the functions to be qualified by the _ _far type qualifier based on the system to be created.

Figure 17.2-3 Function Call Relationship and Mapping Image 3



[Tip]

Softune C Analyzer:

The Softune C Analyzer displays mutual calls of the analyzed functions. The relationship of the displayed function calls is helpful in determining the functions to be qualified by the _ _far type qualifier.

17.3 Mapping Functions Qualified by the _ _far Type Qualifier

This section provides notes on mapping functions qualified by the _ _far type qualifier. The output section name of a function is dependent on the memory model specified at compilation. A function qualified by the _ _far type qualifier is always output to a section called "CODE_module name."

■ Memory Models and Output Sections of Functions Qualified by the _ _far Type Qualifier

The output section name of a function is dependent on the memory model specified at compilation. The output section of a function qualified by the _ _far type qualifier, however, is not dependent on the memory model. The function is always output to a section called "CODE_module name." Sections 17.3.1 "Functions Qualified by the _ _far Type Qualifier for Small and Compact Models" and 17.3.2 "Functions Qualified by the _ _far Type Qualifier for Medium and Large Models" provide notes on mapping functions qualified by the _ _far type qualifier for each memory model.

17.3.1 Functions Qualified by the _ _far Type Qualifier for Small and Compact Models

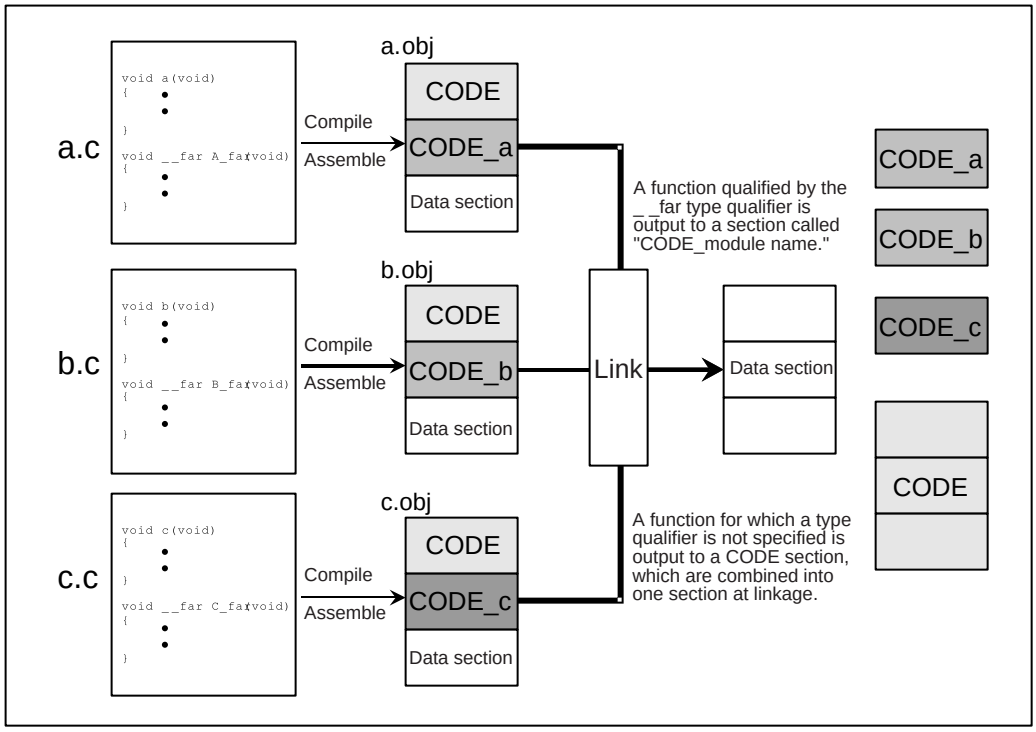
This section provides notes on mapping functions qualified by the _ _far type qualifier for small and compact models in which functions are accessed using 16-bit addressing. For small and compact models, a function qualified by the _ _far type qualifier is output to a section called "CODE_module name" as a result of compilation.

■ Code Sections of Small and Compact Models

Figure 17.3-1 "Linkage of Functions Qualified by the _ _far Type Qualifier for Small and Compact Models" shows an image of linkage of function qualified by the _ _far type qualifier for small and compact models.

For small and compact models, a function for which a type qualifier is not specified is output to a CODE section as a result of compilation. At linkage, this CODE section is allocated in the ROM area pointed to by the PCB. This CODE section is always allocated in the area of bank h'ff. A function qualified by the _ _far type qualifier is output to a section called "CODE_module name" as a result of compilation. Because a function output to this section is accessed using 24-bit addressing, a section called "CODE_module name" can be allocated in a ROM area other than the ROM area pointed to by the PCB.

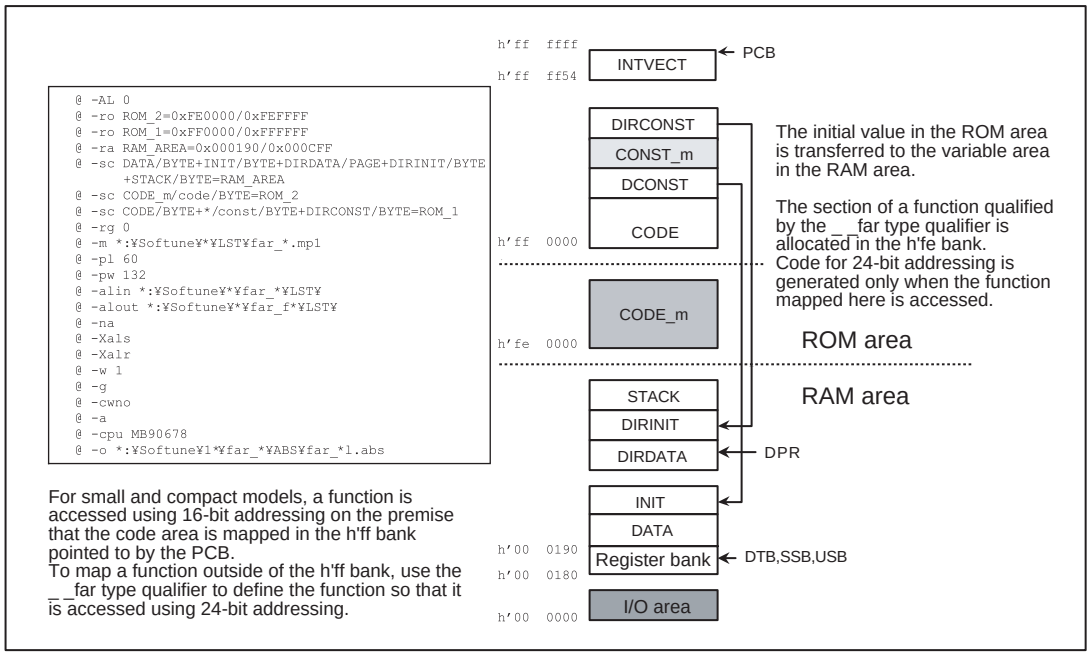
Figure 17.3-1 Linkage of Functions Qualified by the _ _far Type Qualifier for Small and Compact Models



■ Example of Mapping Functions Qualified by the _ _far Type Qualifier (for a Small Model)

Figure 17.3-2 "Example of Mapping Functions Qualified by the _ _far Type Qualifier (for a Small Model)" shows an example of mapping functions qualified by the _ _far type qualifier compiled using a small model.

Figure 17.3-2 Example of Mapping Functions Qualified by the _ _far Type Qualifier (for a Small Model)



In this example, the h'ff and h'fe banks are a ROM area. The following sections are allocated in the h'ff bank:

- CODE (code area of a function for which a type qualifier is not specified)
- DCONST (initial value area of a variable)
- CONST_m (area of a variable qualified by the const type and _ _far type qualifiers for module m)
- DIRCONST (initial value area of a variable qualified by the _ _direct type qualifier)

The section CODE_m of a function qualified by the _ _far type qualifier for module m is allocated in the h'fe bank.

The h'00 bank is a RAM area. The following sections are allocated in the h'00 bank:

- IO_REG (I/O register variable area)
- DATA (variable area)
- INIT (area of an initialized variable)
- DIRDATA (area of a variable qualified by the _ _direct type qualifier)
- DIRINIT (area of an initialized variable qualified by the _ _direct type qualifier)
- STACK (user stack and system stack)

Refer to this example to allocate a section based on the system to be created.

17.3.2 Functions Qualified by the _ _far Type Qualifier for Medium and Large Models

This section provides notes on mapping functions qualified by the _ _far type qualifier for medium and large models in which functions are accessed using 24-bit addressing. For medium and large models, functions for which a type qualifier is not specified and functions that are qualified by the _ _far type qualifier are output to sections called "CODE_module name."

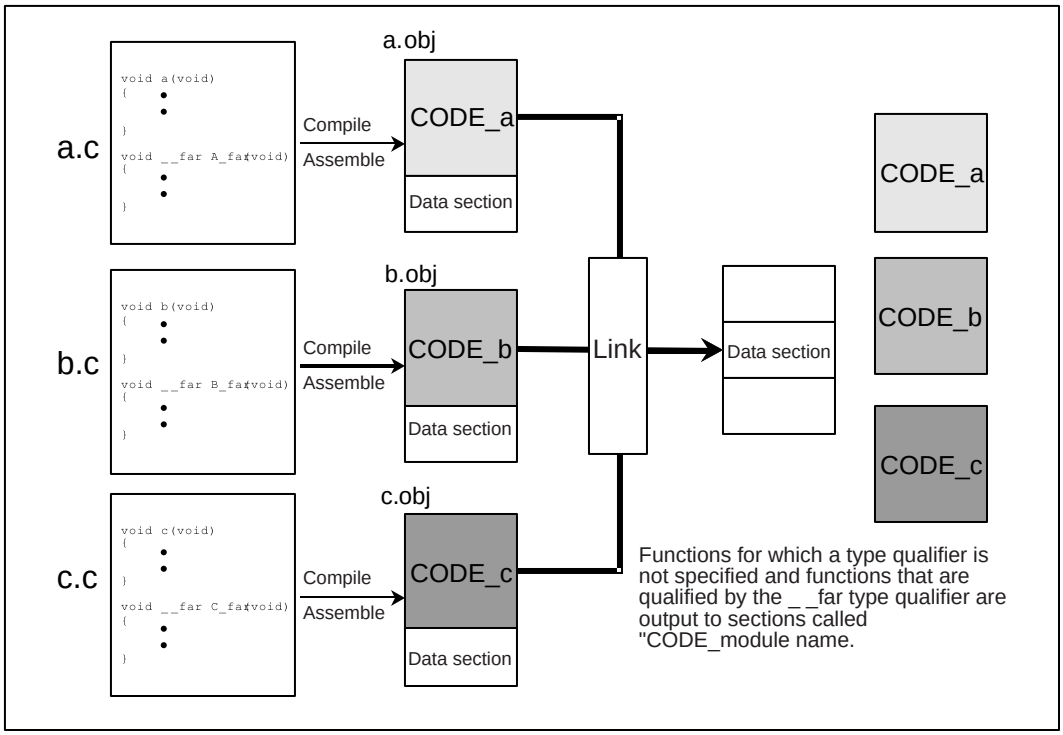
■ Code Sections of Medium and Large Models

Figure 17.3-3 "Linkage of Functions Qualified by the _ _far Type Qualifier for Medium and Large Models" shows an image of linkage of functions qualified by the _ _far type qualifier for medium and large models.

For medium and large models, a function for which a type qualifier is not specified is output to a section called "CODE_module name" as a result of compilation. A function qualified by the _ _far type qualifier is also output to a section called "CODE_module name." As a result, a function qualified by the _ _far type qualifier is output to the same section as a function for which a type qualifier is not specified.

The functions output to these sections are accessed using 24-bit addressing. As a result, a section called "CODE_module name" can be allocated in a ROM area other than the ROM area pointed to by the PCB.

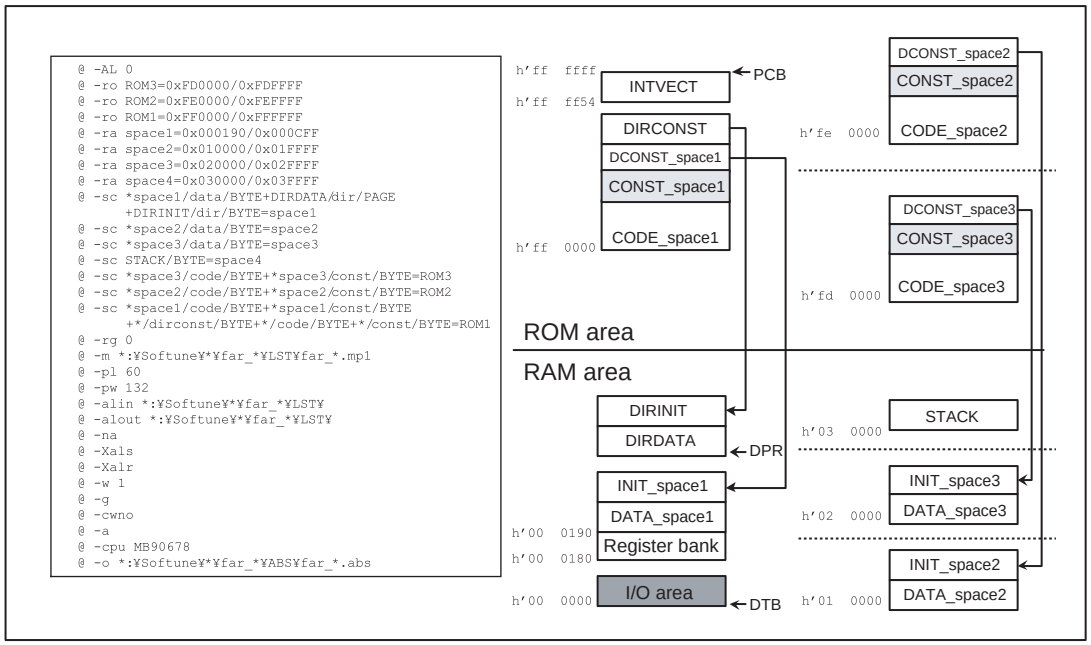
Figure 17.3-3 Linkage of Functions Qualified by the _ _far Type Qualifier for Medium and Large Models



■ Example of Mapping Functions Qualified by the _ _far Type Qualifier (for a Large Model)

Figure 17.3-4 "Example of Mapping Functions Qualified by the _ _far Type Qualifier (for a Large Model)" is an example of mapping functions qualified by the _ _far type qualifier compiled using a large model.

Figure 17.3-4 Example of Mapping Functions Qualified by the _ _far Type Qualifier (for a Large Model)



A function qualified by the _ _far type qualifier is output to a section called "CODE_module name."

In this example, the h'fd, h'fe, and h'ff banks are ROM area. The following sections are allocated in the h'fd bank:

- CODE_space3 (code area of module space3)
- CONST_space3 (variable area of a variable qualified by the const type qualifier of module space3)
- DCONST_space3 (initial value area of a variable of module space3)

The following sections are allocated in the h'fe bank:

- CODE_space2 (code area of module space2)
- CONST_space2 (variable area of a variable qualified by the const type qualifier of module space2)
- DCONST_space2 (initial value area of a variable of module space2)

The following sections are allocated in the h'ff bank:

- CODE_space1 (code area of module space1)
- CONST_space1 (variable area of a variable qualified by the const type qualifier of module space1)
- DCONST_space1 (initial value area of a variable of module space1)
- DIRCONST (initial value area of a variable qualified by the _ _direct type qualifier)

In this example, the h'00, h'01, h'02, and h'03 banks are in a RAM area. The following sections are allocated in the h'00 bank:

- IO_REG (I/O register variable area)
- DATA_space1 (variable area of module space1)
- INIT_space1 (area of an initialized variable of module space1)
- DIRDATA (variable area of a variable qualified by the _ _direct type qualifier)
- DIRINIT (variable area of an initialized variable qualified by the _ _direct type qualifier)

The following sections are allocated in the h’01 bank:

- DATA_space2 (variable area of module space2)
- INIT_space2 (area of an initialized variable of module space2)

The following sections are allocated in the h’02 bank:

- DATA_space3 (variable area of module space3)
- INIT_space3 (area of an initialized variable of module space3)

The following section is allocated in the h’03 bank:

- STACK (user stack and system stack)

Refer to this example to allocate each section based on the system to be created.

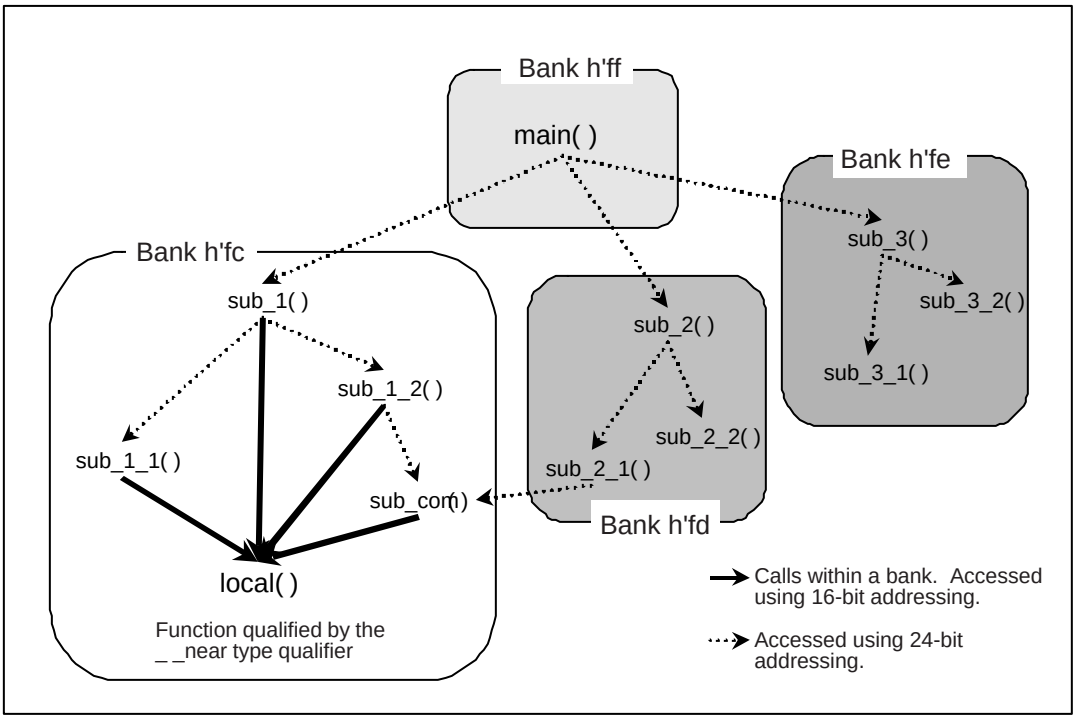
17.4 Using Calls for Functions Qualified by the _ _near Type Qualifier

This section describes how to specify the _ _near type qualifier in a function for medium and large models in which functions are accessed using 24-bit addressing. Specifying the _ _near type qualifier enables functions mapped in the same bank to be accessed using 16-bit addressing.

■ Specifying the _ _near Type Qualifier in Functions for Medium and Large Models

A medium or large model in which functions are accessed using 24-bit addressing is used for a system in which most functions are called between banks. Even in a system such as this, however, there are functions called only from functions mapped in the same bank and not called from functions mapped outside of the bank. These functions are shown in Figure 17.4-1 "Function Call Relationship and Mapping Image 4". Because the scope of a variable declared as static is within the module, this is equivalent to a function called within a bank. To access functions called within a bank, 16-bit addressing will be sufficient. However, when functions are compiled using a medium or large model, a code for 24-bit addressing will be generated for all function calls. Therefore, the _ _near type qualifier can be specified for these functions so that they will be mapped in the same bank as the function calling them. As a result, a code for calling within a bank using 16-bit addressing can be generated even for medium and large models in which accessing functions outside the bank are default.

Figure 17.4-1 Function Call Relationship and Mapping Image 4



[Tip]

Softune C Analyzer:

The Softune C Analyzer displays mutual calls of the analyzed functions. The relationship of the displayed function calls is helpful in determining the functions to be qualified by the _ _near type qualifier.

17.5 Mapping Functions Qualified by the _ _near Type Qualifier

This section provides notes on mapping functions qualified by the _ _near type qualifier for medium and large models in which functions are accessed using 24-bit addressing.
For medium and large models, a function qualified by the _ _near type qualifier is output to a section called "CODE_module name" in the same way as other functions.

■ Memory Models and Output Sections of Functions Qualified by the _ _near Type Qualifier

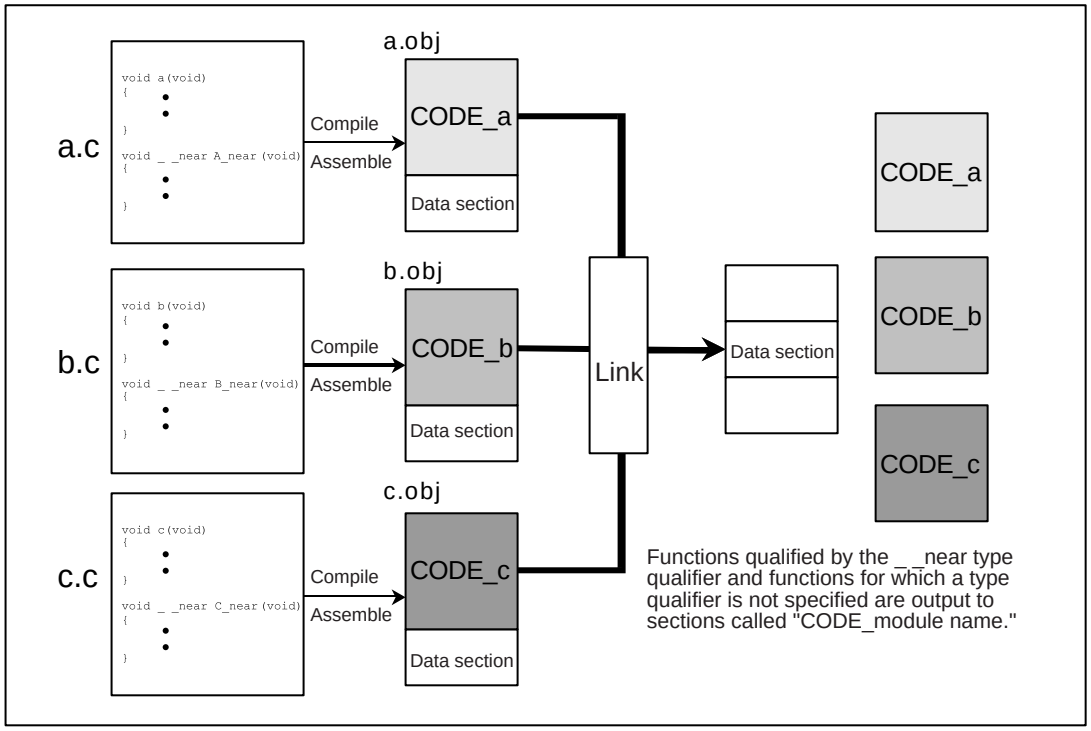
The output section name of a function is dependent on the memory model specified at compilation. For small and medium models, a function qualified by the _ _near type qualifier is output to a CODE section in the same way as a function for which a type qualifier is not specified.

For medium and large models, a function for which a type qualifier is not specified is output to a section called "CODE_module name." A function qualified by the _ _near type qualifier is also output to a section called "CODE_module name."

Figure 17.5-1 "Linkage of Functions Qualified by the _ _near Type Qualifier for Medium and Large Models" shows an image of linkage of functions qualified by the _ _near type qualifier for medium and large models.

The function A_near( ) defined in module a is output to the CODE_a section. The function B_near( ) defined in module b is output to the CODE_b section. In the same way, the function C_near( ) defined in module c is output to the CODE_c section. At linkage, these sections are allocated in the same bank as the module in which the function is defined.

Figure 17.5-1 Linkage of Functions Qualified by the _ _near Type Qualifier for Medium and Large Models

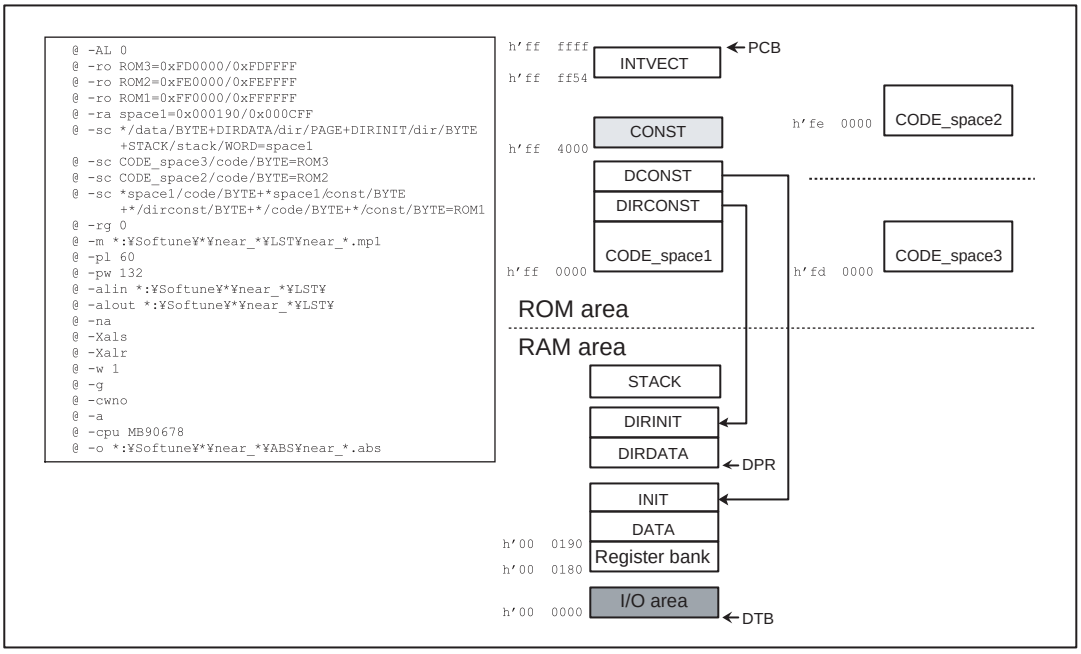


■ Example of Mapping Functions Qualified by the _ _near Type Qualifier (for a Medium Model)

For medium and large models, functions are accessed using 24-bit addressing using the PCB register. For medium and large models, when a function qualified by the _ _near type qualifier is called from a function for which a type qualifier is not specified, the calling function and called function must be mapped in the same bank. The PCB that is set when calling a function for which a type qualifier is not specified is used as is for calling a function qualified by the _ _near type qualifier.

Figure 17.5-2 "Example of Mapping Functions Qualified by the _ _near Type Qualifier (for a Medium Model)" is an example of mapping functions qualified by the _ _near type qualifier compiled using a medium model. The functions qualified by the _ _near type qualifier are output to sections called "CODE_module name" in the same way as functions for which a type qualifier is not specified.

Figure 17.5-2 Example of Mapping Functions Qualified by the _ _near Type Qualifier (for a Medium Model)



In this example, the h'fd, h'fe, and h'ff banks are ROM area. The following section is allocated in the h'fd bank:

- CODE_space3 (code area of module space3)

The following section is allocated in the h'fe bank:

- CODE_space2 (code area of module space2)

The following sections are allocated in the h'ff bank:

- CODE_space1 (code area of module space1)
- DIRCONST (initial value area of a variable qualified by the _ _direct type qualifier)
- DCONST (initial value area of a variable)
- CONST (variable area of a variable qualified by the const type qualifier)

In this example, the h'00 bank is RAM area. The following sections are allocated in the h'00 bank:

- IO_REG (I/O register variable area)
- DATA (variable area)
- INIT (area of an initialized variable)
- DIRDATA (variable area of a variable qualified by the _ _direct type qualifier)
- DIRINIT (variable area of an initialized variable qualified by the _ _direct type qualifier)
- STACK (user stack and system stack)

Refer to this example to allocate each section based on the system to be created.

CHAPTER 18 MAPPING PROGRAMS IN WHICH THE DATA AREA EXCEEDS 64 Kbytes

---

This chapter describes how to map programs in which the data area of the program to be created exceeds 64 Kbytes.
For a system in which the data area exceeds 64 Kbytes even slightly, it is recommended that a small or compact model be used and that the _ _far type qualifier be specified in the functions.

18.1 Function Calls of Programs Where the Data Area Exceeds 64 Kbytes

When creating a system in which the data area exceeds 64 Kbytes, a compact or large model in which variables are accessed using 24-bit addressing is used.
For a system in which the data area exceeds 64 Kbytes, however, accessing variables using 24-bit addressing can increase the size of the code area.

■ Accessing Variables Using 24-Bit Addressing

Table 18.1-1 "Data Section Names for Small and Medium Models" and Table 18.1-2 "Data Section Names for Compact and Large Models" list the type qualifiers of variables and the memory model specifications and output section names at compilation.

Table 18.1-1 Data Section Names for Small and Medium Models

Type qualifier specification					Initial value specification	Variable area name	Initial value area
_ _io	_ _direct	const	_ _near	_ _far			
						DATA	
			o			DATA	
				o		DATA_module name	
					o	INIT	DCONST
			o		o	INIT	DCONST
				o	o	INIT_module name	DCONST_module name
		o			o	CONST	CINIT
		o	o		o	CONST	CINIT
		o		o	o	CONST_module name	CINIT_module name
	o					DIRDATA	
	o				o	DIRINIT	DIRCONST
o						IO	

Table 18.1-2 Data Section Names for Small and Medium Models

Type qualifier specification					Initial value specification	Variable area name	Initial value area
_ _io	_ _direct	const	_ _near	_ _far			
						DATA_module name	
			o			DATA	
				o		DATA_module name	
					o	INIT_module name	DCONST_module name
			o		o	INIT	DCONST
				o	o	INIT_module name	DCONST_module name
		o			o	CONST_module name	CINIT_module name
		o	o		o	CONST	CINIT
		o		o	o	CONST_module name	CINIT_module name
	o					DIRDATA	
	o				o	DIRINIT	DIRCONST
o						IO	

As listed in Table 15.1-1 "fcc907 Memory Models" the fcc907 uses a compact or large model when creating a program in which the data area for the entire system exceeds 64 Kbytes.

For a compact or large model, a code for accessing variables using 24-bit addressing is generated. When multiple banks are used in the data area for accessing variables, there is no problem even if a code for 24-bit addressing is generated. In the same way as described above for the functions, accessing variables using 24-bit addressing can increase the size of the code area. This applies for a system in which the data area exceeds one bank (64 Kbytes).

Even for a small or medium model in which variables are accessed using 16-bit addressing, a variable qualified by the _ _far type qualifier can be accessed using 24-bit addressing. Section 18.2 "Using Calls for Variables Qualified by the _ _far Type Qualifier" describes how to define and map variables that have been qualified by the _ _far type qualifier for small and medium models.

18.2 Using Calls For Variables Qualified by the _ _far Type Qualifier

This section describes how to specify the _ _far type qualifier in a variable for small and medium models where variables are accessed using 16-bit addressing.
It is recommended that the _ _far type qualifier be specified for variables not accessed frequently or variables called from all functions.

■ Specifying the _ _far Type Qualifier in a Variable for Small and Medium Models

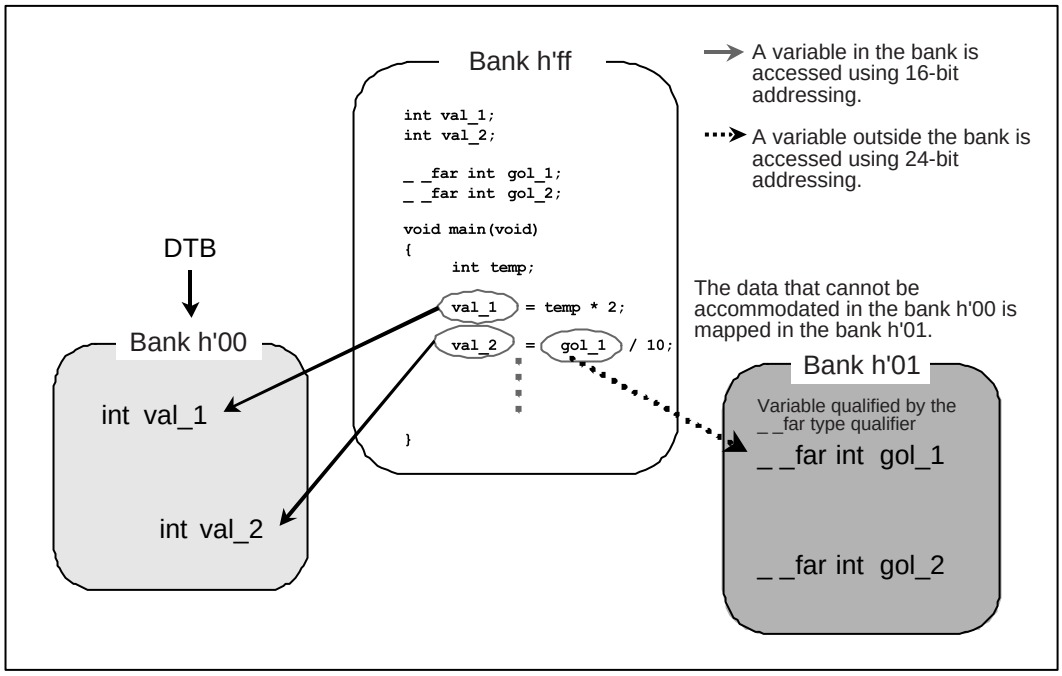
When creating a system in which the data area exceeds 64 Kbytes, a code for 24-bit addressing will be generated even when variables mapped in the bank pointed to by the DTB are accessed. This applies when a compact or large model is used in which all variables are accessed using 24-bit addressing. Even for a small model in which variables mapped in the bank pointed to by the DTB are accessed using 16-bit addressing, the _ _far type qualifier can be specified so that the variables outside of the bank pointed to by the DTB can be accessed using 24-bit addressing.

When creating a system in which the data area exceeds 64 Kbytes, it is recommended that the _ _far type qualifier be specified for some of the variables at compilation for a small or medium model.

■ Specifying the _ _far Type Qualifier in Variables Depending on Access Frequency

The access frequency of the variables in the entire system to be developed is not defined. Some variables are accessed frequently while others are accessed infrequently. When creating a system in which the data area exceeds 64 Kbytes, the variables frequently accessed are mapped in the bank pointed to by the DTB as shown in Figure 18.2-1 "Variable Access Relationship and Mapping Image 1". The _ _far type qualifier can be specified for a variable that exceeds 64 Kbytes so that the variable is mapped outside the bank pointed to by the DTB. As a result, code for 24-bit addressing is generated only when a variable qualified by the _ _far type qualifier is accessed.

Figure 18.2-1 Variable Access Relationship and Mapping Image 1



[Tip]

Softune C Analyzer:

The Softune C Analyzer displays the functions that access the external variables in the analyzed program. The access relationship of the displayed variables is helpful in determining the variables to be qualified by the _ _far type qualifier.

18.3 Mapping Variables Qualified by the _ _far Type Qualifier

This section provides notes on mapping variables qualified by the _ _far type qualifier. The output section name of a variable depends on the memory model specified at compilation. A variable qualified by the _ _far type qualifier, however, is output to a section called "XXXX_module name" regardless of the specified memory model.

■ Memory Models and Output Sections of Variables Qualified by the _ _far Type Qualifier

The output section name of a variable is dependent on the memory model specified at compilation. A variable qualified by the _ _far type qualifier, however, is always output to a section called "XXXX_module name" regardless of the specified memory model. Sections 18.3.1 "Variables Qualified by the _ _far Type Qualifier for Small and Medium Models" and 18.3.2 "Variables Qualified by the _ _far Type Qualifier for Compact and Large Models" provide notes on mapping variables qualified by the _ _far type qualifier for each memory model.

18.3.1 Variables Qualified by the _ _far Type Qualifier for Small and Medium Models

This section provides notes on mapping variables qualified by the _ _far type qualifier for small and medium models in which variables are accessed using 16-bit addressing. For small and medium models, a variable qualified by the _ _far type qualifier is output to a section called "XXXX_module name."

■ Code Sections of Small and Medium Models

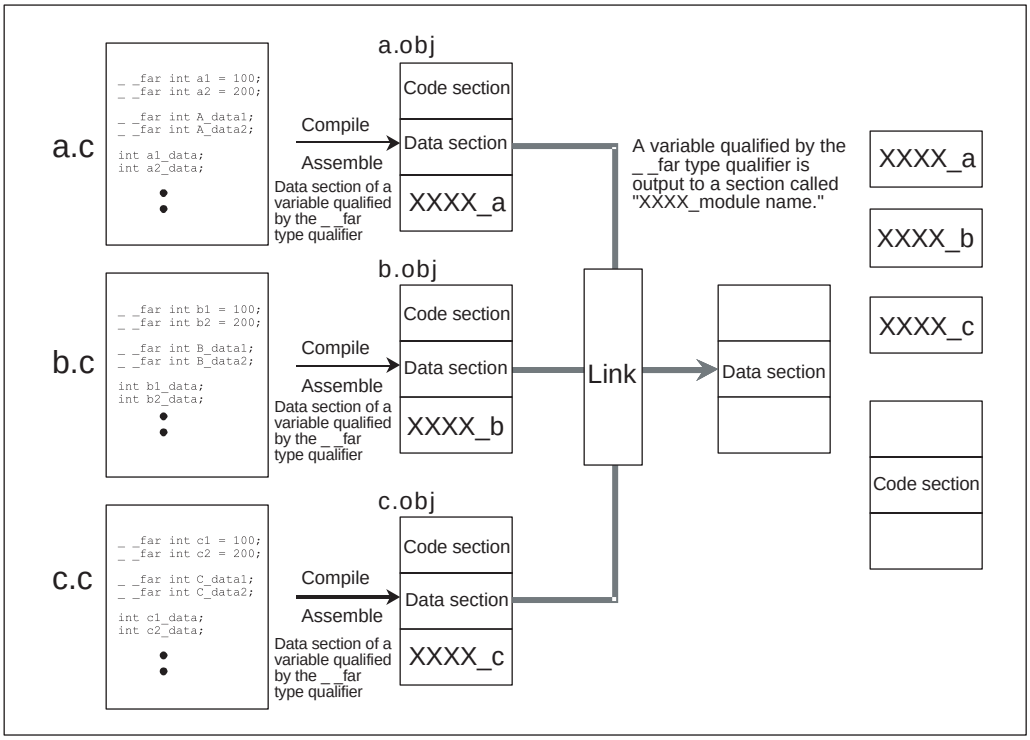
Figure 18.3-1 "Linkage of Variables Qualified by the _ _far Type Qualifier for Small and Medium Models" shows an image of linkage of variables qualified by the _ _far type qualifier for small and medium models.

For small and medium models, the output section name as a result of compilation is different for a variable for which a _ _far type qualifier is not specified than it is for a variable for which the _ _far type qualifier is specified.

For a variable for which a type qualifier is not specified, the variable is output to a DATA, INIT, DCONST, or CONST section depending on the nature of the variable. Among these sections, the variable areas (DATA and INIT) are allocated in the bank pointed to by the DTB. Normally, these variable areas are allocated in the bank h'00. As a result, a variable output in the DATA or INIT section is accessed using 16-bit addressing.

A variable qualified by the _ _far type qualifier is output to a section in which "_module name" has been added to the section name. That is, a variable qualified by the _ _far type qualifier is output to a section called "DATA_module name," INIT_module name," "DCONST_module name," or "CONST_module name." These variables are accessed using 24-bit addressing. As a result, a section called "XXXX_module name" can be allocated in an area outside of the bank pointed to by the DTB.

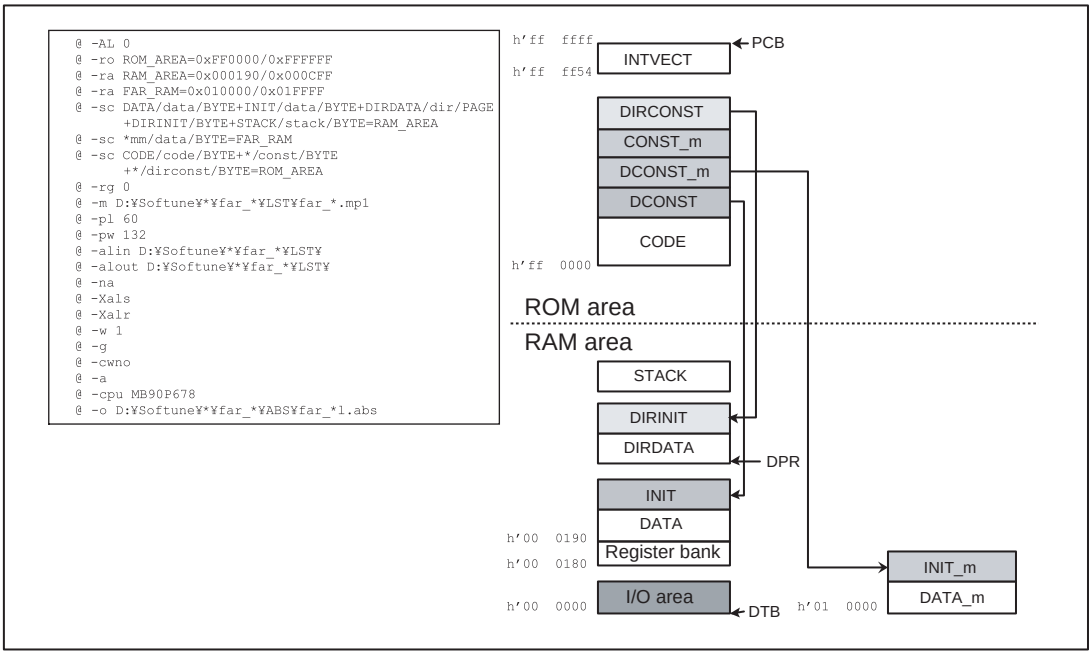
Figure 18.3-1 Linkage of Variables Qualified by the _ _far Type Qualifier for Small and Medium Models



■ Example of Mapping Variables Qualified by the _ _far Type Qualifier (for a Small Model)

Figure 18.3-2 "Example of Mapping Variables Qualified by the _ _far Type Qualifier (for a Small Model)" is an example of mapping variables qualified by the _ _far type qualifier compiled using a small model.

Figure 18.3-2 Example of Mapping Variables Qualified by the _ _far Type Qualifier (for a Small Model)



A variable qualified by the _ _far type qualifier is output to a section called "XXXX_module name."

In this example, the h’ff bank is a ROM area. The following sections are allocated in the h’ff bank:

- CODE (code area)
- DCONST (initial value area of a variable)
- DCONST_m (initial value area of a variable qualified by the _ _far type qualifier for module m)
- CONST_m (variable area of a variable qualified by the _ _far type and const type qualifiers for module m)
- DIRCONST (initial value area of a variable qualified by the _ _direct type qualifier)

The h’00 bank and h’01 banks are a RAM area. The following sections are allocated in the h’00 bank:

- IO_REG (I/O register variable area)
- DATA (variable area)
- INIT (area of an initialized variable)
- DIRDATA (area of a variable qualified by the _ _direct type qualifier)
- DIRINIT (area of an initialized variable qualified by the _ _direct type qualifier)
- STACK (user stack and system stack)

The sections of the following variables qualified by the _ _far type qualifier for module m are allocated in the h’01 bank:

- DATA_m (variable area)
- INIT_m (area of an initialized variable)

Refer to this example to allocate each section based on the system to be created

18.3.2 Variables Qualified by the _ _far Type Qualifier for Compact and Large Models

This section provides notes on mapping variables qualified by the _ _far type qualifier for compact and large models in which variables are accessed using 24-bit addressing.
For compact and large models, variables for which the _ _far type or _ _near type qualifier has not been specified and variables qualified by the _ _far type qualifier are output to sections called "XXXX_module name."

■ Data Sections of Compact and Large Models

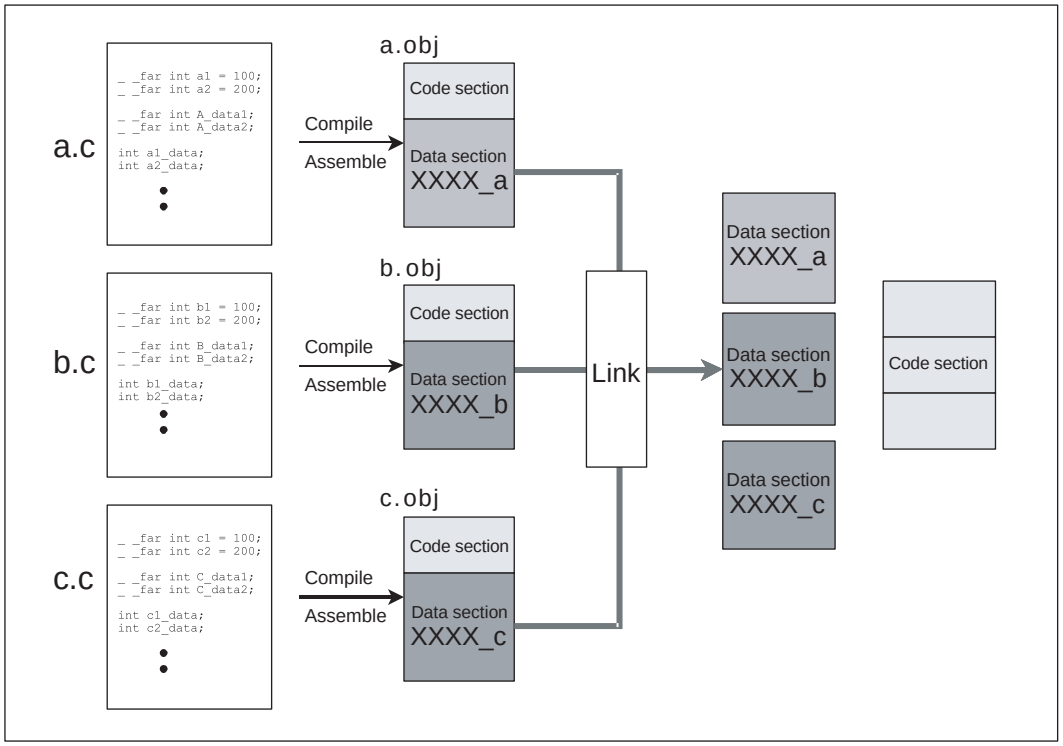
Figure 18.3-3 "Linkage of Variables Qualified by the _ _far Type Qualifier for Compact and Large Models" is an image of linkage of variables qualified by the _ _far type qualifier for compact and large models.

For compact and large models, a variable for which the _ _far type or _ _near type qualifier has not been specified is output to a section called "XXXX_module name" as a result of compilation. A variable qualified by the _ _far type qualifier is also output to a section called "XXXX_module name" in the same way.

Therefore, a variable qualified by the _ _far type qualifier is output to the same section of the same module as a variable for which the _ _far type qualifier has not been specified.

The variables output to these sections are accessed using 24-bit addressing. As a result, a section called "XXXX_module name" can be allocated in RAM area other than the RAM area pointed to by the DTB.

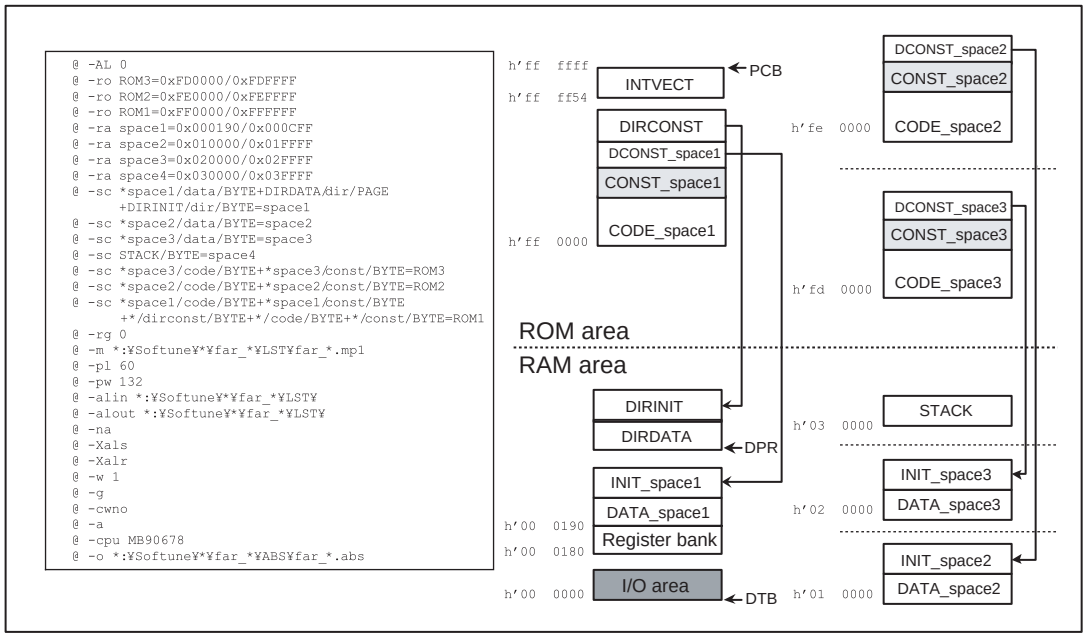
Figure 18.3-3 Linkage of Variables Qualified by the _ _far Type Qualifier for Compact and Large Models



■ Example of Mapping Variables Qualified by the _ _far Type Qualifier (for a Large Model)

Figure 18.3-4 "Example of Mapping Variables Qualified by the _ _far Type Qualifier (for a Large Model)" is an example of mapping variable qualified by the _ _far type qualifier compiled using a large model.

Figure 18.3-4 Example of Mapping Variables Qualified by the _ _far Type Qualifier (for a Large Model)



A variable qualified by the _ _far type qualifier is output to a section called "XXXX_module name."

In this example, the h'fd, h'fe, and h'ff banks are ROM area. The following sections are allocated in the h'fd bank:

- CODE_space3 (code area of module space3)
- CONST_space3 (variable area of a variable qualified by the const type qualifier of module space3)
- DCONST_space3 (initial value area of a variable of module space3)

The following sections are allocated in the h'fe bank

- CODE_space2 (code area of module space2)
- CONST_space2 (variable area of a variable qualified by the const type qualifier of module space2)
- DCONST_space2 (initial value area of a variable of module space2)

The following sections are allocated in the h'ff bank:

- CODE_space1 (code area of module space1)
- CONST_space1 (variable area of a variable qualified by the const type qualifier of module space1)
- DCONST_space1 (initial value area of a variable of module space1)
- DIRCONST (initial value area of a variable qualified by the _ _direct type qualifier)

In this example, the h'00, h'01, h'02, and h'03 banks are RAM area. The following sections are allocated in the h'00 bank:

- IO_REG (I/O register variable area)
- DATA_space1 (variable area of module space1)
- INIT_space1 (area of an initialized variable of module space1)
- DIRDATA (variable area of a variable qualified by the _ _direct type qualifier)
- DIRINIT (variable area of an initialized variable qualified by the _ _direct type qualifier)

The following sections are allocated in the h'01 bank:

- DATA_space2 (variable area of module space2)
- INIT_space2 (area of an initialized variable of module space2)

The following sections are allocated in the h'02 bank:

- DATA_space3 (variable area of module space3)
- INIT_space3 (area of an initialized variable of module space3)

The following section is allocated in the h'03 bank:

- STACK (user stack and system stack)

Refer to this example to allocate each section based on the system to be created.

18.4 Using Calls For Variables Qualified by the _ _near Type Qualifier

This section describes how to specify the _ _near type qualifier in a variable for compact and large models where variables are accessed using 24-bit addressing. Specifying the _ _near type qualifier enables a variable mapped in the bank pointed to by the DTB to be accessed using 16-bit addressing.

■ Specifying the _ _near Type Qualifier in Variables for Compact and Large Models

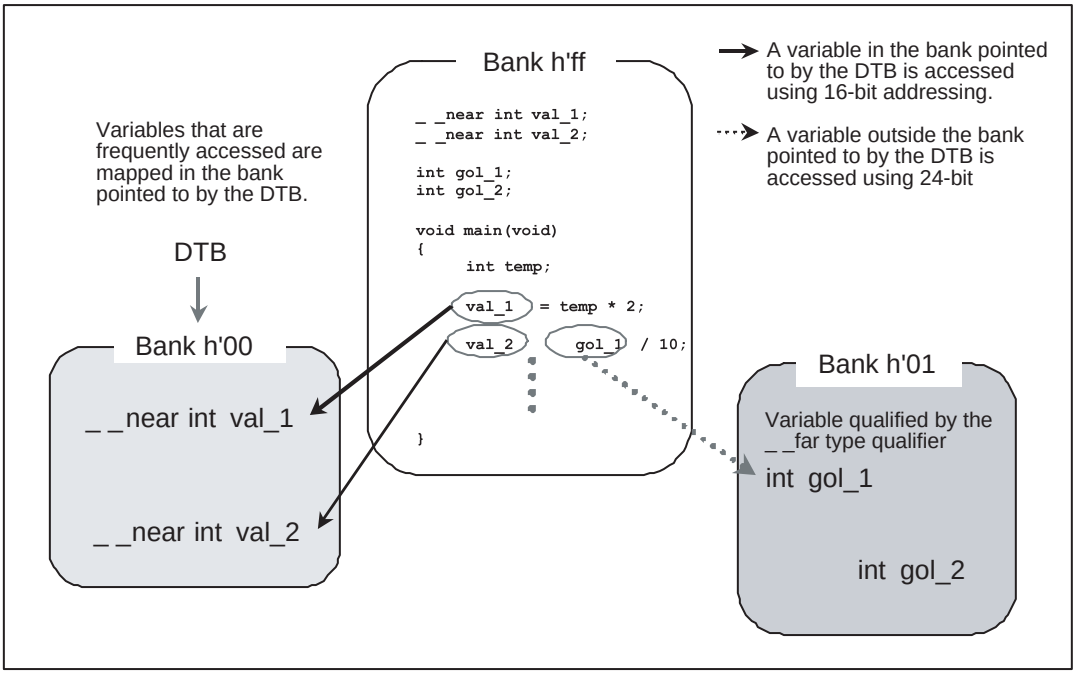
A compact or large model in which all variables are accessed using 24-bit addressing is used for systems in which large numbers of variables are accessed. That is, the data area exceeds 64 Kbytes. When a compact or large model is used, the variables are accessed using 24-bit addressing.

This does not mean, however, that all of the variables are accessed with the same frequency. Some variables are accessed very frequently while others are accessed infrequently. When 24-bit addressing is used to access a variable that is accessed with high frequency, the code size is increased and execution speed at access is reduced.

As shown in Figure 18.4-1 "Variable Access Relationship and Mapping Image 2" specifying the _ _near type qualifier when compiling a variable that is frequently accessed will generate code for accessing the variable using 16-bit addressing. The variable area of the variables qualified by the _ _near type qualifier will then be set in the bank pointed to by the DTB.

As a result, a variable mapped in the bank pointed to by the DTB can be accessed using 16-bit addressing. This also applies for compact and large models in which variables are accessed using 24-bit addressing by default.

Figure 18.4-1 Function Call Relationship and Mapping Image 2



[Tip]

Softune C Analyzer:

The Softune C Analyzer displays the functions that access the external variables in the analyzed program. The access relationship of the displayed variables is helpful in determining the variables to be qualified by the _ _near type qualifier.

18.5 Mapping Variables Qualified by the _ _near Type Qualifier

This section provides notes on mapping variables qualified by the _ _near type qualifier for compact and large models in which variables are accessed using 24-bit addressing.
A variable qualified by the _ _near type qualifier is output to the DATA, INIT, or DCONST section regardless of the specified memory model.

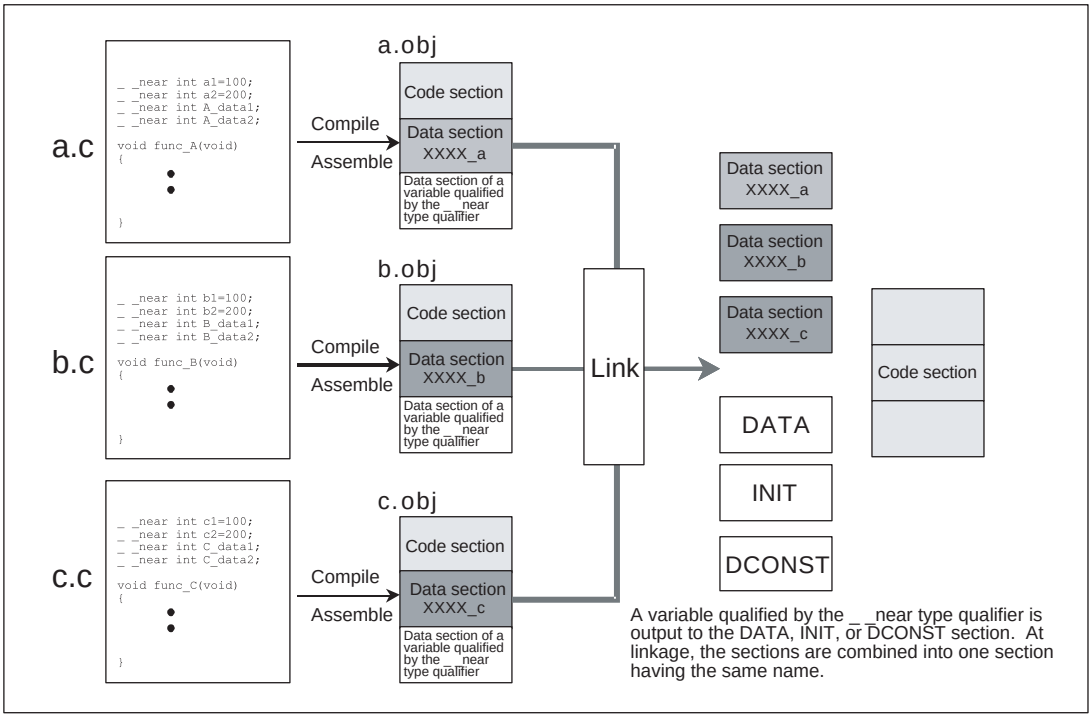
■ Memory Models and Output Sections of Variables Qualified by the _ _near Type Qualifier

The output section name of a variable is dependent on the memory model specified at compilation. A variable qualified by the _ _near type qualifier, however, is output to the DATA, INIT, or DCONST section regardless of the specified memory model.

Figure 18.5-1 "Linkage of Variables Qualified by the _ _near Type Qualifier for Compact and Large Models" shows an image of linkage of variables qualified by the _ _near type qualifier for compact and large models.

When the variable qualified by the _ _near type qualifier defined in module A, the variable qualified by the _ _near type qualifier defined in module B, and the variable qualified by the _ _near type qualifier defined in module C are linked, the variables are combined into one section regardless of the defined module. As a result, variables qualified by the _ _near type qualifier can be mapped in the same bank even if the variables are in different modules. However, these sections cannot be divided and mapped. To divide the sections of variables qualified by the _ _near type qualifier into a section different for each module, the output section name must be changed.

Figure 18.5-1 Linkage of Variables Qualified by the _ _near Type Qualifier for Compact and Large Models

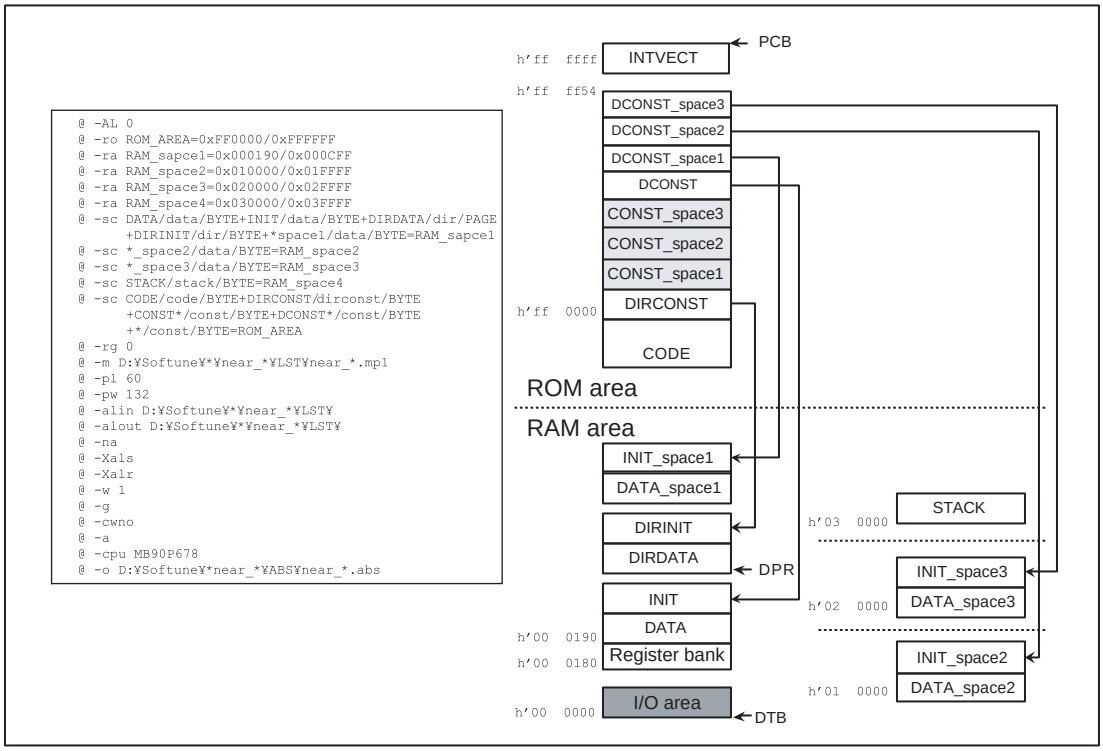


■ Example of Mapping Variables Qualified by the _ _near Type Qualifier (for a Compact Model)

For compact and large models, variables are accessed using 24-bit addressing. Variables qualified by the _ _near type qualifier, however, are accessed using 16-bit addressing on the premise that the variables are mapped in the bank pointed to by the DTB.

The DATA and INIT sections must be allocated in the h'00 bank pointed to by the DTB.

Figure 18.5-2 Example of Mapping Variables Qualified by the _ _near Type Qualifier (for a Compact Model)



In this example, the h'ff bank is ROM area. The following sections are allocated in the h'ff bank:

- CODE (code area)
- DIRCONST (initial value area of a variable qualified by the _ _direct type qualifier)
- CONST_space1 (area of a variable qualified by the const type qualifier for module space1)
- CONST_space2 (area of a variable qualified by the const type qualifier for module space2)
- CONST_space3 (area of a variable qualified by the const type qualifier for module space3)
- DCONST (initial value area of a variable qualified by the _ _near type qualifier)
- DCONST_space1 (initial value area of a variable of module space1)
- DCONST_space2 (initial value area of a variable of module space2)
- DCONST_space3 (initial value area of a variable of module space3)

In this example, the h'00, h'01, h'02, and h'03 banks are RAM area. The following sections are allocated in the h'00 bank:

- IO_REG (I/O register variable area)
- DATA (variable area of a variable qualified by the _ _near type qualifier)
- INIT (variable area of an initialized variable qualified by the _ _near type qualifier)
- DIRDATA (variable area of a variable qualified by the _ _direct type qualifier)

- DIRINIT (variable area of an initialized variable qualified by the _ _direct type qualifier)
- DATA_space1 (variable area of module space1)
- INIT_space1 (variable area of an initialized variable of module space1)

The following sections are allocated in the h’01 bank:

- DATA_space2 (variable area of module space2)
- INIT_space2 (variable area of an initialized variable of module space2)

The following section is allocated in the h’02 bank:

- DATA_space3 (variable area of module space3)
- INIT_space3 (variable area of an initialized variable of module space3)

The following section is allocated in the h’03 bank:

- STACK (user stack and system stack)

Refer to this example to allocate a section based on the system to be created.

INDEX

The index follows on the next page.
This is listed in alphabetic order.

Index

F

G

H

I

L

M

N

O

P

R

S

U

V

W

CM42-00327-1E

FUJITSU SEMICONDUCTOR • CONTROLLER MANUAL

F^2MC-16 FAMILY
16-BIT MICROCONTROLLER
EMBEDDED C PROGRAMMING MANUAL
FOR fcc907

October 2000 the first edition

Published **FUJITSU LIMITED** Electronic Devices

Edited Technical Communication Dept.

FUJITSU

CM42-00327-1E

FUJITSU SEMICONDUCTOR F^{2}MC-16 FAMILY 16-BIT MICROCONTROLLER EMBEDDED C PROGRAMMING MANUAL FOR fcc907