FUJITSU SEMICONDUCTOR

CONTROLLER MANUAL

CM71-00324-1E

FR FAMILY

32-BIT MICROCONTROLLER

EMBEDDED C

PROGRAMMING MANUAL

FOR fcc911

FUJITSU
PREFACE

■ Objectives and Intended Reader

The FR family of microcontrollers are 32-bit microcontrollers designed for embedded systems. This manual provides information that is required for using the fcc911 FR family C compiler to create an embedded system. The manual explains how to create C programs that effectively use the FR family architecture and provides notes related to the creation of C programs. This manual is intended for engineers who will use the fcc911 to develop application programs for the FR family. Be sure to read this manual completely.

■ Trademarks

FR is an abbreviation of Fujitsu RISC controller, which is a product of Fujitsu Ltd.

Softune is a registered trademark of Fujitsu Ltd.

Other system and product names in this manual are the 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 system in which an FR family microcontroller was embedded.

CHAPTER 2 "VARIABLE DEFINITIONS AND VARIABLE AREAS"
This chapter describes variable definitions and variable areas to which the results of compilation are output. It also describes the variable area 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 VARIABLE 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. The chapter also discusses the reduction of the variable area and object efficiency for referencing when the "const" type qualifier 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 only 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 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.

PART III "USING LANGUAGE EXTENSIONS"
Part III describes the language extensions specific to the fcc911. Part III also discusses items in the extended language specifications that require special attention.

CHAPTER 9 "WHAT ARE LANGUAGE EXTENSIONS?"
This chapter describes the fcc911-specific extended language specifications, such as the qualifiers for extensions, _ _asm statements, and "#pragma."

CHAPTER 10 "NOTES ON ASSEMBLER CODE 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 11 "NOTES ON DEFINING AND ACCESSING THE I/O AREA"

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

CHAPTER 12 "NOTES ON CREATING AND REGISTERING INTERRUPT FUNCTIONS"

This chapter provides notes on using language extensions of the fcc911 to enable interrupt processing.

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 fcc896 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.

©2000 FUJITSU LIMITED Printed in Japan
HOW TO USE THIS MANUAL

■ Page Layout

The contents of each section are summarized underneath the title. You can get an overview of the product simply by reading these summaries.

In addition, higher level section headings are also provided in lower level sections so that you know to which section the text you are currently reading belongs.
CONTENTS

PART I VARIABLE DEFINITIONS AND VARIABLE AREAS ................................................. 1

CHAPTER 1 OBJECTS MAPPED INTO MEMORY AREAS ........................................... 3
  1.1 Program Components ........................................................................................................ 4
  1.2 Mapping into Memory Areas ............................................................................................ 6
  1.3 Dynamically Allocated Variables ...................................................................................... 8
  1.4 Statically Allocated Variables .......................................................................................... 10

CHAPTER 2 VARIABLE DEFINITIONS AND VARIABLE AREAS ............................... 13
  2.1 External Variables and Their Variable Area ........................................................................ 14
  2.2 Initial Values and the Variable Area for External Variables ............................................... 16
  2.3 Initialized Variables and Initialization at Execution .......................................................... 18
  2.4 Variables Declared as "static" and Their Variable Area ..................................................... 20
    2.4.1 Example of Function with Static Global Variable ....................................................... 22
    2.4.2 Example of a Function with Static Local Variable ..................................................... 23

CHAPTER 3 READ-ONLY VARIABLES AND THEIR VARIABLE AREA ................. 25
  3.1 Numeric Constants and #define Definition .................................................................. 26
  3.2 Defining Variables Using the const Type Qualifier ......................................................... 28

CHAPTER 4 USING AUTOMATIC VARIABLES TO REDUCE THE VARIABLE AREA 31
  4.1 Automatic Variables and Statically Allocated Variables .................................................. 32
  4.2 Using Automatic Variables ............................................................................................. 35
  4.3 Range of Access to Automatic Variables Through Frame Pointers (FP) ......................... 39

CHAPTER 5 ACCESSING VARIABLES THAT USE BIT FIELDS ............................... 41
  5.1 Boundary Alignment of fcc911 ....................................................................................... 42
  5.2 Bit Field Definitions and Boundary Alignment ............................................................... 43
  5.3 Accessing I/O Areas Using Bit Fields and Unions ............................................................ 45

PART II USING STACK AREA EFFICIENTLY ............................................................. 47

CHAPTER 6 FUNCTION CALLS AND THE STACK ..................................................... 49
  6.1 Areas Allocated on the Stack during Function Calls ....................................................... 50
  6.2 Stack States When Function Calls Are Nested ................................................................ 52

CHAPTER 7 REDUCING FUNCTION CALLS BY EXPANDING FUNCTIONS IN LINE 53
  7.1 Inline Expansion of Function ......................................................................................... 54
  7.2 Conditions for Inline Expansion of Function .................................................................. 57

CHAPTER 8 REDUCING ARGUMENTS TO CONSERVE STACK AREA .................... 59
  8.1 Passing Arguments During Function Calls ..................................................................... 60
  8.2 Optimization Specification and Passing of Arguments .................................................. 64
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 FR family microcontroller embedded system. It then briefly describes the relationship between variable definitions and the variable area. It concludes by describing how to efficiently create C programs.

CHAPTER 1 "OBJECTS MAPPED INTO MEMORY AREAS"
CHAPTER 2 "VARIABLE DEFINITIONS AND VARIABLE AREAS"
CHAPTER 3 "READ-ONLY VARIABLES AND THEIR VARIABLE AREA"
CHAPTER 4 "USING AUTOMATIC VARIABLES TO REDUCE THE VARIABLE AREA"
CHAPTER 5 "ACCESSING VARIABLES THAT USE BIT FIELDS"
CHAPTER 1   OBJECTS MAPPED INTO MEMORY AREAS

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

1.1 "Program Components"
1.2 "Mapping into Memory Areas"
1.3 "Dynamically Allocated Variables"
1.4 "Statically Allocated Variables"
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 Components Diagram]

  - **Code**
    This section in the program contains the machine instructions to be executed by the CPU.
    For a C program, the algorithm, which is coded as functions, is compiled and converted to machine instruction codes.
    The term "Code" refers to a set of execution instructions that are only read at execution.

  - **Data**
    Programs access a data section.
    In a C program, data includes variables, character strings, literal constants, initial values, and other items.
    Data is read and written during processing.

In addition, C programs can be classified as shown in Figure 1.1-1 "Types of Program Objects for Embedded Systems and Mapping into Memory Areas". As shown, data variables 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 use both an initial value area and the variable area.
Figure 1.1-1  Types of Program Objects for Embedded Systems and Mapping into Memory Areas

Program
---

**Code (function)**
- ROM area
  - Initial value area

**Data (variable)**
- Dynamically allocated variables
- Statically allocated variables
- RAM area
  - Without initial value
  - With initial value
  - Variable area

- Allocated to the I/O area
- I/O 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 FR family microcontroller, the memory area can be mainly classified into ROM, RAM, 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 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.
- Since the values in the RAM area are undefined when the initializing program is started, 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. Indicates a program such as startup.asm, which is provided as a sample program together with the C compiler, that performs initialization before execution of C programs.

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

- ROM area
  - Initial value area
  - Area for variables declared with the "const" type modifier
  - Code

- RAM area
  - Stack
  - Area for initialized variables
  - Area for uninitialized variables
    - Initialize with 0

- I/O area
1.3 Dynamically Allocated Variables

This section briefly describes the variables that are allocated dynamically. 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 automatic variables and the register variables that are defined within functions are allocated dynamically.

  - **Automatic variables**
    - One type of local variable
    - 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 functions
    - 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", an automatic variable is allocated in a stack area at a function call. The allocated area is released when the function terminates. Consequently, an automatic variable can be accessed only in a defined function.

During a function call, a register variable receives priority allocation to a hardware register. The register is released when the function terminates. Consequently, like an automatic variable, a register variable can be accessed only in a defined function.
Figure 1.3-1 Dynamically Allocated Variables

Dynamically allocated variables

Auto variables
• Stack area is allocated when the function is executed.
• The area is released when the function terminates.
Register variables
• Allocated to a hardware register at a function call.
• The register is released when the function terminates.

extern int init(int);  
extern void start(int, int);  
int userid = 100;  
void main(void)
{
    int i,j;
    i = init(userid);  
    
    start(userid, i);  

    
}
1.4 Statically Allocated Variables

This section briefly describes the variables that are allocated statically. In a C program, external variables that are defined outside a function and variable 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. Section 2.4 "Variables Declared as "static" and Their Variable Area", provides information about static variables.
Statically allocated variables
External variables
Static variables
- Allocated in the RAM area.
- The variables exist in the RAM area.

```c
int curspid;
int nextproc;
int semcont;
int currsem;
int nextsem;

int o = 10;
sema
extern void initproc(void);
extern int initsem(int);
extern int wait(int);

void main(void)
{
    int userid = 10;
    int a;
    initproc( );
    a = initsem(userid);
    
}
```

Variable area in RAM
The values can be read and written from all function.

Area for external variables

Area for static local variable of function

```c
void initproc(void)
{
    int flag;
    
    curspid = 0;
    nextproc = 1;

    

    return(cont);

}
```

int initsem(int num)
{
    return(cont);

}

int wait(int num)
{
    int val;

    semcont;
    val = semcont - num;

    return(val);

}
CHAPTER 2 VARIABLE DEFINITIONS AND VARIABLE AREAS

This chapter briefly describes the variable definitions and the variable area where the results of compilation are output. 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 formats.

2.1 "External Variables and Their Variable Area"
2.2 "Initial Values and the Variable Area for External Variables"
2.3 "Initialized Variables and Initialization at Execution"
2.4 "Variables Declared as "static" and Their Variable Area"
CHAPTER 2 VARIABLE DEFINITIONS AND VARIABLE AREAS

2.1 External Variables and Their Variable Area

This section provides a simple explanation of external variables and their variable area. The external variable are defined outside a function. The area for external variable is fixed allocated in RAM.

- External Variables

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

Figure 2.1-1 Definitions of External Variables

For external variables, the names of the sections to which the results of compilation are output depend on the storage class, type qualifier, and initial value specified at variable definition. Refer to the fcc911 manual for more information. Table 2.1-1 "Variables and Data Sections for Output" lists the relationship between the definition of external variables and the sections where the results of compilation are output.
### Table 2.1-1 Variables and Data Sections for Output

<table>
<thead>
<tr>
<th>Type qualifier</th>
<th>Specification of initial value</th>
<th>Variable area</th>
</tr>
</thead>
<tbody>
<tr>
<td>_ _ io</td>
<td>const</td>
<td>DATA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DATA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INIT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CONST</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IO</td>
</tr>
</tbody>
</table>

*: Specification

**[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 the Variable Area for External Variables

This section provides a simple explanation of the relationship between initial values and the variable area for external variables.

In fcc911, when an initial value is specified at definition of external variable, variable area is allocated in both the ROM and RAM area.

- **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 (There is no need to initialize the variable 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 fcc911 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).

  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.
2.2 Initial Values and the Variable Area for External Variables

Figure 2.2-1 Variable Areas and Memory Mapping

Uninitialized variable

```c
int data1;
```

Initialized variable

```c
int i_data = 123;
```

The initial value area is allocated in the ROM area. The area accessed when execution starts is allocated in the RAM area.

For an initialized variable, an area of twice the size of the defined variable is required in the ROM and RAM areas.

The startup routine transfers the initial values from ROM to the variable area in RAM.

For uninitialized variables, a variable area is allocated only in RAM. The startup routine initializes all values in this area to 0.
CHAPTER 2 VARIABLE DEFINITIONS AND VARIABLE AREAS

2.3 Initialized Variables and Initialization at Execution

This section describes initialized variables and the initialization of uninitialized variables at 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 which a variable is initialized beforehand, and an example of a function in which the value is set at the beginning of the function.

See the explanation of the function list1( ) in (1), "Definition as an initialized variable," within Figure 2.3-1 "Initialized Variables and Initial Value Assignment at Function Execution". A four-byte area is allocated for the initialization variable i_data in the INIT section. For the INIT section, area is allocated in both ROM and RAM. The startup routine transfers the initial value from ROM to the variable area in RAM.

See the function list2( ) of (2), "Assigning a value when the variable is used", in Figure 2.3-1 "Initialized Variables and Initial Value Assignment at Function Execution". Only the four-byte variable area DATA for the variable i_data is allocated in RAM. However, a code for assigning the value is required. Compared with (1) in Figure 2.3-1 "Initialized Variables and Initial Value Assignment at Function Execution", the value area is reduced by four bytes, but the code area is increased by 12 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 12-byte code is required whenever a 4-byte variable is assigned.

If we take the case of a variable that is initialized using 10 different values, code of (12 bytes x 10) = 120 bytes will be required. If a variable is defined as an initialized variable, the area for the value in ROM will increase by 40 bytes. Because the startup routine handles transfer, it is assumed that the size of the code will not increase. Thus, when the increase of 120 bytes in code and the increase in variable area of 40 bytes are compared, use of the ROM area is more economically when an initialized variable is defined.

<Notes>

Setting an initial value for a variable that does not need 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

(1) Definition as initialized variable

```c
1 int i_data = 0x1234;
2
3 int list1(int data)
4 {
5   if (!data)
6     return(i_data);
7     return(i_data + data);
9 )
```

(2) Assigning a value when the variable is used.

```c
1 int i_data;
2
3 int list2(int data)
4 {
5   i_data = 0x1234;
6   if (!data)
7     return(i_data);
8   return(i_data + data);
10 }
```

Because code is generated for assigning the value in the function, the size of the code area increases.

The variable area DINIT is output. The variable area DATA is output. The size of the object to be generated differs.
### 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 a Variable 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.
Section 2.4.1 "Scope of a Variable Declared as "static"" provides an example of a function that uses a static global variable. Section 2.4.2 "Example of a Function with 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:

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 with 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.

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.

Example of a Function with Static Global Variable

```c
1 int time; // Static global variable
4 int timet(void);
5 int list3(void)
7 { 
8    int flag = 0;
9
10    if (++count >= 60)
11       return(flag);
12 }
13
14    int timet(void)
15    { 
16       int temp;
17
18 if (++time)
19       { 
20          time = 0;
21         count = 0;
22
23         temp = time * count;
24         return(temp);
25     }
```

Area for the static global variable count is allocated via the variable LI_1, which is not declared as PUBLIC. As a result, the variable cannot be accessed from other compile units.
2.4 Variables Declared as "static" and Their Variable Area

2.4.2 Example of a Function 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," is a static local variable in the function.

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 compile unit.

Example of a Function with Static Local Variable

```assembly
1. int time;
2. int timeint2(void);
3. int list4(void);
4. {
5.   int flag = 0;
6.   static int count = 0;
7.   if (*count > 50)
8.     flag = timeint2();
9.   return(flag);
10. }
11. int timeint2(void)
12. {
13.   int temp;
14.   static int count = 0;
15.   if (--time)
16.     count = 0;
17.   temp = time * count;
18.   return(temp);
19. }
```

Area for the static local variable count of function timeint2( ) is allocated via the variable LI_2, which is not declared as PUBLIC.

Area for the static local variable count of function list4( ) is allocated via the variable LI_1, which is not declared as PUBLIC.
Chapter 3  Read-Only Variables and Their Variable Area

This chapter describes how to use read-only variables. The variable values are read and written 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 these kind of variables include 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"

3.2 "Defining Variables Using the const Type Qualifier"
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 the #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 the function list5() of (1), "Defining external variables," in Figure 3.1-1 "Defining External Variables and Defining Variables Using the #define Statement". Because initialized variables have been defined for the function list5(), the variable area INIT is generated. The variable area INIT is mapped into ROM and RAM areas at linkage. The startup routine transfers the ROM initial values to the RAM area. The following variables are defined for the function list5():

- char-type variable (1 byte) c_max
- int-type variable (4 bytes) i_data
- 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 17-byte variable area will be wasted.

The external variables are accessed using the following procedure:

1. The address of an external variable is loaded.
2. A value is loaded based on this address.

As a result, an eight-byte code will be generated when the variable is accessed.

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". The variables c_max, maxaddr, pai, and d_data have been macro-defined using the #define statement for the function list6(). Because the values for the macro-defined variables have been embedded in the code, no variable area is generated. Because no code is generated for accessing external variables, the respective code remains small, resulting in faster execution speed. The code to be generated depends on the numeric constants.

Because macro-defined variables do not have a pre-defined type, their type can be determined at assignment based on the type of variable to be allocated. This can lead to unexpected results.
3.1 Numeric Constants and `#define` Definition

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

(1) Defining external variables

```c
int maxaddr = 32767;
float psi = -3.14159;
double d_data = 0xffff0000;
void list5(void)
{
    char c_max = 255;
    int maxaddr = 32767;
    float psi = 3.14159;
    double d_data = 0xffff0000;
}
```

When an initialized global variable is defined, the variable area INIT is generated. In addition, an external variable access code is generated at access.

(2) Defining numeric constants using the `#define` statement

```c
#define c_max 250
#define maxaddr 32767
#define psi 3.14159
#define d_data 0xffff0000
void list5(void)
{
    char c_max;
    int maxaddr;
    float psi;
    double d_data;
}
```

Data area is not generated when the `#define` statement is used. Because the numerical data are embedded in the code, the size of the code is smaller than when accessing external variables.

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

**Defining variables as initialized external variables**

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

An eight-byte code is generated at access of an external variable.

**Defining variables as numeric constants**

Variable area is not allocated in RAM.

Because the values are embedded directly in the code, the size of the code is smaller than when accessing 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 ROM size, 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 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 where the results of compilation are output and mapping into memory.

Normally, only the variable area CONST is output for a variable declared as "const." This CONST section is mapped into the ROM area. When a variable is accessed, the variable area in the ROM area is accessed directly.

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

The initial value area is allocated in ROM, and the area accessed at execution is allocated in ROM. Initialized variable require an area in the ROM and RAM areas with a size of twice the size for the defined variable itself. The startup routine transfers the initial value from the ROM to the RAM area.

The initial value area of the variable declared with the const type qualifier is mapped into the ROM area. When the variable is accessed, the ROM area is accessed directly.
3.2 Defining Variables Using the const Type Qualifier

Figure 3.2-2 "Defining External Variables and Defining Variables Using the "const" Type Qualifier" shows a function for which read-only values are defined as initialized external variables and a function for which variables declared as "const" are defined.

See the function list5() of (1), "Defining external variables," in Figure 3.2-2 "Defining External Variables and Defining Variables Using the "const" Type Qualifier". Because initialized variables have been defined for the function list5(), the variable area INIT is generated. The variable area INIT is mapped into ROM and RAM areas at linkage. The startup routine transfers the initial values from ROM to the RAM area. For the function list5(), the char-type variable c_max, int-type variable maxaddr, float-type variable pai, and double-type variable d_data are output to the variable area INIT.

Read-only variables are not written to at execution. From the viewpoint of economical use of the RAM area, this 17-byte variable area will be wasted.

See the function list7() of (2), "Defining variable declared with the const type qualifier," in Figure 3.2-2 "Defining External Variables and Defining Variables Using the "const" Type Qualifier". The 17-byte variable area CONST is output for the function list7(). The variable area CONST is mapped into the ROM area at linkage. Because the ROM area is accessed directly at access, the RAM area can be used economically.

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

(1) Defining external variables

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>char c_max = 350;</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>int maxaddr = 32767;</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>float pai = 3.2159;</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>double d_data = 0xffffffff;</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>void list5(void) {</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>char c_data;</td>
</tr>
<tr>
<td>9</td>
<td>int l_data;</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>float f_data;</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>double d_data;</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When an initialized global variable is defined, the variable area INIT is generated. In addition, an external variable access code is generated at access.

(2) Defining variable declared with the const type qualifiers

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>const char c_max = 350;</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>const int maxaddr = 32767;</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>const float pai = 3.2159;</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>const double d_data = 0xffffffff;</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>void list7(void) {</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>char c_data;</td>
</tr>
<tr>
<td>9</td>
<td>int l_data;</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>float f_data;</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>double d_data;</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When a variable declared with the const type qualifier is defined, the variable area CONST is generated. In addition, an external variable access code is generated.

[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.
CHAPTER 3 READ-ONLY VARIABLES AND THEIR VARIABLE AREA

• An attempt is made to change the value of a variable declared with the const type qualifier.

See the section on defining variables that are 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.
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 terminated 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"
4.2 "Using Automatic Variables"
4.3 "Range of Access to Automatic Variables Through Frame Pointers (FP)"
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.

Automatic Variables and Statically Allocated Variables

Because variable area is allocated for automatic variables dynamically, automatic variables are also referred to as dynamically allocated variables. An automatic variable cannot be accessed outside the function.

Whether area for automatic variables is allocated on a stack depends on the status when the function is called. In addition, an automatic variable is not initialized when it is allocated. 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
4.1 Automatic Variables and Statically Allocated Variables

- 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 that are allocated outside a function or variables declared as "static" are statically allocated variables. External variables can be accessed from everywhere in the program. Variables declared as "static" can be divided 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

```c
int curpid;
int nextproc;
int semcnt;
int cursem;
int nextsem;

int semno = 10;

extern void initproc(void);
extern int initsem(int);
extern int wait(int);

void main(void)
{
    int userpid = 10;
    int a;
    initproc();
    a = initsem(userpid);
    ...
}
```
CHAPTER 4 USING AUTOMATIC VARIABLES TO REDUCE THE VARIABLE AREA

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

[C source program]

```c
extern int main(void);
extern int inittime(void);
extern int init(int);
extern int numproc;
int currpid;
int semno;
int nextsem = 0;
static int nextproc = 100;

int null(void)  
{  
    int userpid = 10;
    inittime();
    currpid = init(userpid);
    nextsem++;  
    semno = 100;
    return(semno);
}

int init(int pid)  
{  
    static int num = 50;
    int i = 0;
    int j;
    return(num--);
}
```

[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

extern int init(int);
extern void start(int, int);

int userid = 100;

void main(void)
{
    int i,j;
    i = init(userid--);

    ...
}

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

Figure 4.2-2 Using an External Variable and Using an Automatic Variable

(1) Function that uses an external variable

```c
1  #define MAXPROC 100
2  "define PROC 1
3  int procno;
4  extern procno;
5  int nextproc;
6  int listp(void)
7  {
8      if (procno > 0)
9          return(ERR);
10     nextproc = currproc + 1;
11     return(nextproc);
12 }
```

The variable nextproc is defined as an external variable. The allocated variable area increased by four bytes. The code for variable area access is smaller than the one generated for stack access.

(2) Function that uses an automatic variable

```c
1  #define MAXPROC 100
2  "define ERR 1
3  int procno;
4  int currproc;
5  int listp(void)
6  {
7      if (procno > 0)
8          next = currproc + 1;
9          return(ERR);
10      next = currproc + 1;
11      return(ERR);
12     }
```

Variable "next" is defined as an automatic variable. A variable area is allocated on the stack at execution and released when the function terminates.

See the function list8( ) of (1), "Function that uses an external variable," in Figure 4.2-2 "Using an External Variable and Using an Automatic Variable". Because the variable nextproc is defined as an external variable for the function list8( ), the variable area allocated in RAM increased by four bytes. The external variable is accessed after the variable address is loaded. As a result, the size of the code will be greater than when the stack is accessed.

See the function list9( ) of (2), "Function that uses an automatic variable," in Figure 4.2-2 "Using an External Variable and Using an Automatic Variable". Because the variable "next" is defined as an automatic variable for the function list9( ), a variable area will be allocated on the stack when the function is executed. An automatic variable mapped on the stack is accessed through a frame pointer (FP). As a result, the size of the code will be smaller than when an address is loaded and an external variable accessed. 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 an External Variable and Using an Automatic Variable", the difference in the sizes of the data area is only four bytes for the external variable nextproc. In addition, the difference in the size of the code generated when a variable is accessed is only six bytes. However, as the number of accesses of external variables increases, the size of the code generated can also be expected to increase.

The amount of variable area that can be saved by reducing the number of external variables by one will only be several 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 fcc911. The Softune C Analyzer then visually displays the routes and amounts of usage. This information is useful for reducing the amount of stack usage.
4.3 Range of Access to Automatic Variables Through Frame Pointers (FP)

This section describes how to access automatic variables using frame pointers (FP).

- **Frame Pointers and Automatic Variables**
  
The fcc911 accesses automatic variables through frame pointers. Figure 4.3-1 "Accessing Automatic Variables Through Frame Pointers (FP)" shows the ranges for stack accesses through frame pointers.

**Figure 4.3-1 Accessing Automatic Variables Through Frame Pointers (FP)**

The fcc911 rearranges the local variables to increase the number of variables that can be effectively accessed from the FPs. However, if a local variable is defined that exceeds the range of stack access by a FP, this operation will also be disabled.

To enable efficient access through the frame pointers, the fcc911 rearranges the automatic variables in the order shown below and maps them on the stack:

1. Variables of the type "char"
2. Variables of the type "short"
3. Other types of variables
4. Arrays or structures

However, if a large automatic variable that exceeds the range of stack access by a FP is defined, this operation will also be disabled.
Figure 4.3-2 "Access to an Automatic Variable that Exceeds the Range of Stack Access Through a Frame Pointer" shows the example for variable access when an automatic variable that exceeds the range of stack access through a frame pointer is defined. In this example, the int-type array cont[300] and char-type array name[100] are defined. Area is allocated on the stack in the following order: First for the char-type array name[100], then for the int-type array cont[300]. Because the char-type array name[100] is within the range of (FP-128), it can be accessed through a frame pointer. However, an int-type array cont[300] has also been mapped on the stack, exceeding the range of (FP-512). If an element mapped on the stack exceeding the range of (FP-512) is accessed, more code will be generated than for accesses within the range of (FP-512).

When there is an access to a local variable on the stack that exceeds the range of FP-512, more code will be generated than for accesses within the range of FP-512.
CHAPTER 5  ACCESSING VARIABLES THAT USE BIT FIELDS

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

5.1 "Boundary Alignment of fcc911"
5.2 "Bit Field Definitions and Boundary Alignment"
5.3 "Accessing I/O Areas Using Bit Fields and Unions"
CHAPTER 5 ACCESSING VARIABLES THAT USE BIT FIELDS

5.1 Boundary Alignment of fcc911

This section briefly describes the boundary alignment of the fcc911. For the fcc911 processing, variables are allocated to memory in accordance with the variable size and boundary alignment.

- Boundary Alignment of fcc911

Table 5.1-1 "Boundary Alignment of fcc911" lists the variable types and the respective allocation sizes and their relationship to the boundary alignment of the fcc911.

The fcc911 maps variables in memory based on allocation size and boundary alignment. Defining an odd number of variables of the type char can result in the creation of unused area. 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 fcc911

<table>
<thead>
<tr>
<th>Variable type</th>
<th>Allocation size (bytes)</th>
<th>Boundary alignment (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>signed char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>unsigned char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>unsigned short</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>unsigned int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>unsigned long</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>long double</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Pointer/ address</td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>
5.2 Bit Field Definitions and Boundary Alignment

This section briefly describes memory mapping using bit field definitions and boundary alignment. Bit field 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 FR Family" to 5.2-3 "Bit Field Allocation 3 for the FR Family" show how the fcc911 allocates bit fields.

As shown in Figure 5.2-1 "Bit Field Allocation 1 for the FR Family", the fcc911 allocates contiguous bit field data starting from the most significant bit (MSB) regardless of the type.

Figure 5.2-1  Bit Field Allocation 1 for the FR Family

```c
struct tag1(
    int A:10;
    short B:3;
    char C:2;
)
```

If the bit field to be allocated extends over a type boundary, the bit field will be allocated starting from the boundary for the respective type.

Figure 5.2-2 "Bit Field Allocation 2 for the FR Family" shows an example of allocating a bit field that involves boundary alignment. In this example, a long int-type 12-bit bit field named [A] is first allocated to memory. A short-type 5-bit bit field named [B] is then allocated to memory, but one bit extends outside. In this case, boundary alignment operates so that [B] is mapped starting from a boundary appropriate for the type. In the process, a 4-bit empty space is generated. Next, when an attempt is made to allocate a char-type 5-bit bit field named [C], two bits extend outside. In this case, too, boundary alignment operates in the way that [C] is mapped starting from the boundary for the type "char." In the process, a 3-bit empty space is generated.
CHAPTER 5  ACCESSING VARIABLES THAT USE BIT FIELDS

Figure 5.2-2  Bit Field Allocation 2 for the FR Family

When defining a bit field of bit field length 0, the bit field will be forcibly allocated starting from the next storage unit.

Figure 5.2-3 "Bit Field Allocation 3 for the FR Family" shows an example of allocation when a bit field of bit field length 0 is defined. In this example, an int-type 10-bit bit field named [A] is first allocated to memory. An int-type 5-bit bit field named [B] is then allocated to memory. Finally, an int-type six-bit bit field named [C] is allocated to memory. However, a bit field of bit field length 0 has been defined before [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 four bytes, a 17-bit empty area is generated.

Figure 5.2-3  Bit Field Allocation 3 for the FR Family

[Tip]

Softune C Checker:

The Softune C Checker outputs a warning for structure and union variables that generate empty space. If a warning is output, it is recommended to check the structure or union definitions.
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 the example shown in Figure 5.3-1 "Accessing the I/O Area with Bit Fields and Unions", the bit field structure and unsigned short-type variables are defined as unions, enabling to access either each bit individually or the unsigned short-type variable.

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

When a hardware register is mapped into an I/O area of an FR family microcontroller, both the value of each bit or the value of the entire register can be accessed. If a union is defined for this type of hardware register, a value can be assigned to the entire register as shown below:

```c
_TMCSR0.hword = 0x081b;
(TMCSR0.bit.UF = 0x01;
```

In addition, a value can be directly assigned to the bit field as shown below:

```c
(TMCSR0.bit.UF = 0x01;

This approach facilitates access to registers mapped into the I/O area.
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"
CHAPTER 7 "REDUCING FUNCTION CALLS BY EXPANDING FUNCTIONS IN LINE"
CHAPTER 8 "REDUCING ARGUMENTS TO CONSERVE STACK AREA"
CHAPTER 6 FUNCTION CALLS AND THE STACK

This chapter describes the areas allocated on the stack when a function is called. It then describes how to use the stack areas effectively. When a function is called, area, such as the areas for arguments, are allocated on the stack as necessary.

6.1 "Areas Allocated on the Stack during Function Calls"
6.2 "Stack States When Function Calls Are Nested"
6.1 Areas Allocated on the Stack during Function Calls

During a function call within a C program, such areas as argument and return address areas are allocated on the stack.

When a function is called within a C program, areas such as those shown in Figure 6.1-1 "Areas Allocated on the Stack when a Function is Called" are allocated on the stack. The following briefly describes each area:

- **Actual argument and dummy argument areas**
  These areas are used to hand over arguments when a function is called.
  - Actual argument: Argument specified by the calling function
  - Dummy argument: Argument accessed by the called function

- **Argument register save area**
  This area is used to save the argument register in a called function. The argument register is not saved if saving is unnecessary.

- **Hidden parameter save area**
  This area is used to save the beginning address of the area that stores the return value for a function that returns a structure or union.

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

- **Return address save area (previous RP)**
  This area is used to save the value of the return pointer (RP). The return address to the calling source when a function is called is stored as the RP.

- **Previous frame pointer save area**
  This area is used to save the value of the frame pointer (R14) of the source calling the function.

- **Local variable area**
  This is a local variable and work area. The size of the area allocated depends on the number of local variables. If several variables are defined in a function, an area that is appropriate for that size will be allocated.
Figure 6.1-1 Areas Allocated on the Stack when a Function is Called

- Return value area
- Dummy argument area
- Argument register save area
- Hidden parameter save area
- Register save area
- Return address storage area (RP)
- Previous frame pointer
- Local variable area
- Actual argument area

FP(R14)

Indicates the return value of a structure or union type function.

SP(R15)
6.2 Stack States When Function Calls Are Nested

The areas allocated on the stack are released when the function terminates. The deeper the nesting of function call 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 a function call is 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

```c
extern int init(int); 
extern void start(int, int);

int userpid = 100;

void main(void)
{
    int i, j;
    i = init(userpid-1);
}

Stack status when function init() starts

int init(int pid)
{
    int prev;
    int next;
    prev = pid;
    next = pid;
    start(pid, i);
    return i+next;
}

Stack status when function start() starts

void start(int pid, int count)
{
    int dummy;
    int moji[5];
    moji[0] = pid;
}

Stack status when function start() terminates

Stack status when function init() terminates
```
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"
7.2 "Conditions for Inline Expansion of Function"
7.1 Inline Expansion of Function

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

**Inline Expansion of Function**

The fcc911 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. Inline expansion is specified with "#pragma inline function-name." When the specified function is called, it is expanded inline.

```
1  extern char block01[10];
2  extern char block02[20];
3
4  int checksum(char *data, int length);{
5     int res;
6     int i;
7     res = 0;
8     for (i = 0; i < length; i++){
9         res += (int)*data;
10     }
11     return(res & 0x00ff);
12 }
13
14 #pragma inline checksum
15 int proc_block01(void)
16 {
17     int temp;
18     temp = checksum(block01, 10);
19     return(temp);
20 }
```

Because inline expansion is specified, code for the function checksum is embedded.
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.

See Figure 7.1-2 "Example in Which Inline Expansion Is Not Executed". 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

```
1 extern char block01[10];
2 extern char block02[20];
3
4 int checksum(char *data, int length)
5 {
6    int res;
7    int i;
8    res = 0;
9    for(i = 0; i < length; i++){
10        res += (int)*data;
11    }
12    return(res & 0x00ff);
13 }
14
15 #pragma inline checksum

16 extern int proc_block01(void)
17 {
18    int temp;
19    temp = checksum(block01, 10);
20    return(temp);
21 }
22
23 extern int proc_block02(void)
24 {
25    int temp;
26    temp = checksum(block02, 20);
27    return(temp);
28 }
```

<Notes>

To have the fcc911 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, inline expansion of function checksum() is specified on line 16. Because optimization using the -O 2 option has been specified for compilation, inline expansion of the function checksum() on line 22 is executed. There may be a normal function call to the function checksum(), code for 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. In addition, because the code for function checksum() is embedded in the function proc_block01(), faster processing can be expected. Because the code for function checksum() is inserted in line 22, code larger than that for an ordinary function call is generated.

```
1 extern char block01[10];
2 extern char block02[20];
3
4 int checksum(char *data, int length)
5 {
6   int res;
7   int i;
8
9   res = 0;
10  for( i = 0; i < length; i++ )
11    res += (int)data;
12 }
13
14 #pragma inline checksum
15
16 int proc_block01(void)
17 {
18   int temp;
19
20   temp = checksum(block01, 10);
21
22   return(temp);  
23 
```

When inline expansion is specified using #pragma inline and optimization (level greater than -O 1) is specified for compilation, the code for function checksum() will be embedded in function proc_block01().

```
\_proc_block01:
   \begin{verbatim}
   1      \ld t=check sum[ b l o c k 01, 1 0 ];
   2      \ld t=check sum[ b l o c k 01, 1 0 ];
   3      \ld \ld 40, 86
   4      \ld \ld \ld 40, 86
   5      \ld \ld \ld 40, 86
   6      \ld \ld \ld 40, 86
   7      \ld \ld \ld 40, 86
   8      \ld \ld \ld 40, 86
   9      \ld \ld \ld 40, 86
   10     \ld \ld \ld 40, 86
   11     \ld \ld \ld 40, 86
   12     \ld \ld \ld 40, 86
   13     \ld \ld \ld 40, 86
   14     \ld \ld \ld 40, 86
   15     \ld \ld \ld 40, 86
   16     \ld \ld \ld 40, 86
   17     \ld \ld \ld 40, 86
   18     \ld \ld \ld 40, 86
   19     \ld \ld \ld 40, 86
   20     \ld \ld \ld 40, 86
   21     \ld \ld \ld 40, 86
   22     \ld \ld \ld 40, 86
   23     \ld \ld \ld 40, 86
   24     \ld \ld \ld 40, 86
   \end{verbatim}
\end{verbatim}
```

Specification of inline expansion

Specification of optimization level 2

NO FUNCTION-NAME SIZE ATTRIBUTES
D CODE.:...:00000036 CODE XEL ALIGN:02
This section provides notes on expanding functions in line. Only the functions that were defined in the same file can be inline-expanded.

### Conditions for Inline Expansion of Function

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

The fcc911 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 option is specified.

In this example, inline expansion is specified on line 16. Because the function checksum( ) has been declared as static, the function will not be accessed 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"

```
1  extern char block01[10];
2  extern char block02[20];
3
4  static int checksum(char *data, int length)
5   {
6      int res;
7      int i;
8      res = 0;
9      for(i = 0; i < length; i++)
10         res += (int)*data;
11     }
12     return(res & 0x00ff);
13  }

15
16  #pragma inline checksum
17  int proc_block01(void)
18  {
19     int temp;
20     temp = checksum(block01, 10);
21     return(temp);
22  }

```

When inline expansion is specified using #pragma inline and optimization (level greater than -O 1) is specified for compilation, the code for function checksum( ) will be embedded in the function proc_block01( ).

```
 precinctblock01:
     :     : temp = checksum(block01, 10);
     :     : LDX:32 #block01, R6
     :     : (res = 0;
     :     : LDI $0, R5
     :     : for(i = 0; i < length; i++)
     :     :     LDAA:6 R47, R12
     :     :     LDI $0, R4
     :     :     res += (int)*data;
     :     :     LDBR 0R6, R0
     :     :     ADD R0, R5
     :     :     ADD $4, R4
     :     :     for(i = 0; i < length; i++)
     :     :     CMP $10, R4
     :     :     BNE:32 LDAA:6, R12
     :     :     LDI $255, R0
     :     :     MOV R5, R4
     :     :     temp = checksum(block01, 10);
     :     :     return(temp);
     :     :     )
     :     : RFT:3 ADD R0, R4
```
<Notes>

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

- Optimization using the -O option has not been specified for compilation.
- Inline expansion has been 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 module in which the setjmp function is called.
- Inline expansion was specified in a file containing the __asm statement.
- Argument between functions do not match.

[Tip]

For the fcc911:

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, optimization must be specified using the -O option (level greater than -O 1).

```
-K ADDSP-option
```

Specifying the above option can slightly reduce the overhead of function calls and generate faster 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.
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," described how to use inline expansion to conserve stack area. However, some function calls cannot be eliminated because of the size of the function or the nature of the processing. This chapter describes how to use fewer arguments as a second method of conserving stack area.

8.1 "Passing Arguments During Function Calls"

8.2 "Optimization Specification and Passing of Arguments"
8.1 Passing Arguments During Function Calls

This section describes how to pass arguments during function calls. During a function call, the fcc911 stores arguments in the argument registers (R4 to R7) for passing the arguments to the called function.

**Argument Registers**

As shown in Figure 8.1-1 "Storing Arguments in Argument Registers and Function Calls", the fcc911 stores arguments in the argument registers for passing the arguments to the called function. The fcc911 supports four argument registers. During a function call, the fcc911 stores arguments in the argument registers (R4 to R7) and passes the arguments to the called function. Function calls are fast if no more than four arguments are involved.

**Figure 8.1-1 Storing Arguments in Argument Registers and Function Calls**

```c
void func1(int, int);
void func2(char, int, int);
void func3(long, int, int);
void func4(double);
void func5(int, int, int, int, int);

void main(void)
{
    int a=1, b=2, c=3, d=4, e=5, f=6;
    char char_1='A';
    long long_1=0x0fff;
    double dou_1=1.23;

    func1(a, b);
    func2(char_1, e, d);
    func3(long_1, e, f);
    func4(dou_1);
    func5(1, b, c, d, e, f);
}
```

The arguments are stored in argument registers R4 and R5 and passed to function `func1()`, which is the calling destination.
8.1 Passing Arguments During Function Calls

Figure 8.1-2 "Storing All Arguments in Argument Registers and Passing the Arguments to Functions" shows an example for passing arguments during function calls.

In Figure 8.1-2 "Storing All Arguments in Argument Registers and Passing the Arguments to Functions", two arguments of type int are specified when function func1( ) is called. One argument of type char and two arguments of type int are specified when function func2( ) is called. One argument of type long and two arguments of type int are specified when function func3( ) is called. In all of these function calls, the number of arguments does not exceed four. As a result, all of the arguments are stored in the argument registers and passed to the called function. The arguments are stored in the argument registers in order starting from R4.

Two argument registers are used when an argument of the type double or type long double is specified. This means that the number of arguments that can be stored in the argument registers is reduced. Figure 8.1-2 "Storing All Arguments in Argument Registers and Passing the Arguments to Functions" shows an example of storing an argument of type double of function func4( ) in argument registers and passing the arguments to the called function. The high-order word of argument dou_1 of type double is stored in argument register R4, the low-order word is stored in argument register R5, and function func4( ) is called.

Figure 8.1-2 Storing All Arguments in Argument Registers and Passing the Arguments to Functions

<table>
<thead>
<tr>
<th>Storing arguments in argument registers R4 to R7 and passing the arguments to the called function</th>
</tr>
</thead>
</table>
| void func1(int, int);  
void func2(char, int, int);  
void func3(long, int, int);  
void func4(double);  
void main(void)  
{  
    int a=1, b=2, c=3, d=4, e=5, f=6;  
    char char_1='A';  
    long long_1=0x123;  
    double dou_1=1.23;  
    func1(a, b);  
    func2(char_1, c, d);  
    func3(long_1, e, f);  
    func4(dou_1);  
}  

When function void func1(int, int) is called  
| R4 | a | The first argument, a, of type int is stored in R4. The second argument, b, of type int is stored in R5. The arguments are then passed to function func1( ). |
| R5 | b |

When function void func2(char, int, int) is called  
| R4 | char_1 | The first argument, char_1, of type char is stored in R4. The second argument, c, of type int is stored in R5. The arguments are then passed to function func2( ). |
| R5 | c  
| R6 | d |

When function void func3(long, int, int) is called  
| R4 | long_1 | The first argument, long_1, of type long is stored in R4. The second argument, e, of type int is stored in R5. The third argument, f, of type int is stored in R6. The arguments are then passed to function func3( ). |
| R5 | c  
| R6 | d |

When function void func4(double) is called  
| R4 | High-order word of dou_1 | The high-order word of the first argument, dou_1, of type double is stored in R4. The low-order word is stored in R5. The argument is then passed to function func4( ). |
| R5 | Low-order word of dou_1 |
CHAPTER 8 REDUCING ARGUMENTS TO CONSERVE STACK AREA

 Calling Functions When Arguments Are Not Stored in Argument Registers

Figure 8.1-3 "When Arguments Are Not Stored in Argument Registers" shows the passing of arguments when arguments are not stored in argument registers R4 to R7.

Six arguments of type int are specified when function func1( ) in Figure 8.1-3 "When Arguments Are Not Stored in Argument Registers" is called. The arguments are stored in the argument registers in order starting from R4. The fifth argument (variable e) and sixth argument (variable f) are not stored in the argument registers, but are stored on the stack before being passed to function func1( ).

Two arguments of type int and one argument of type double are specified in the function call for function func2( ) in Figure 8.1-3 "When Arguments Are Not Stored in Argument Registers". The arguments are stored in the argument registers in order starting from R4. In this case, the low-order word of argument dou_1 of type double is stored on the stack and passed to the function without being stored in an argument register.

Figure 8.1-3 When Arguments Are Not Stored in Argument Registers

| Passing arguments without storing them in argument registers R4 to R7 |
|---|---|
| When function void func1(int, int, int, int, int) is called |
| R4 | a |
| R5 | b |
| R6 | c |
| R7 | d |

The first argument, a, of type int is stored in R4. The second argument, b, of type int is stored in R5. The third argument, c, of type int is stored in R6. The fourth argument, d, of type int is stored in R7. The fifth argument, d, and the sixth argument, e, which are not stored in the argument registers, are stored on the stack. The arguments are then passed to function func1( ).

| When function void func2(int, int, char, double) is called |
|---|---|
| R4 | a |
| R5 | b |
| R6 | char_1 |
| R7 | High-order word of dou_1 |

The first argument, a, of type int is stored in R4. The second argument, b, of type int is stored in R5. The third argument, c har_1, of type char is stored in R6. The arguments are then passed to function func2( ). The high-order word of the fourth argument, dou_1, of type double is stored in R7. The low-order word is stored on the stack. The argument is then passed to function func2( ).
Argument Registers and Calling Functions That Return a Structure or a Union

Figure 8.1-4 "Passing Arguments When a Structure or a Union Is Returned" shows the passing of arguments when functions that return a structure or a union are called.

When a function that returns a structure or a union is called, the address of the area that returns the structure or union is stored in argument register R4 and passed to the called function. Therefore, only the three registers R5 to R7 can actually store arguments.

When function func1( ) in Figure 8.1-4 "Passing Arguments When a Structure or a Union Is Returned" is called, the address of the area that returns the structure containing the return values of the function is stored in R4. The arguments are stored in R5 and R6 and passed to function func1( ).

When function func2( ) in Figure 8.1-4 "Passing Arguments When a Structure or a Union Is Returned" is called, the address of the area that returns the structure containing the return values of the function is passed to R4 in the same way as for function func1( ). The arguments are then stored in the argument registers in order. In this case, the argument of type int (variable d) is stored on the stack and passed to the function without being stored in an argument register.

![Figure 8.1-4 Passsing Arguments When a Structure or a Union Is Returned](image)

Passing arguments when a function that returns a structure or a union is called

When function struct data func1(int, int) is called

```
struct data(
    int a, b;
    int data_1;
);
```

```
struct data func1(int, int);
struct data func2(char, char, int, int);
void main(void)
{
    struct data f_data[5];
    int a=1, b=2, c=3, d=4;
    char char_1='A';
    char char_2='B';
    f_data[0] = func1(a, b);
    f_data[1] = func2(char_1, char_2, c, d);
}
```

The address of the area that returns structure data is stored in R4. The first argument, a, of type int is stored in R5. The second argument, b, of type int is stored in R6. The arguments are then passed to function func1( ).

When the function struct data func2(char, char, int, int) is called

```
struct data f_data[5];
    int a=1, b=2, c=3, d=4;
    char char_1='A';
    char char_2='B';
    f_data[0] = func1(a, b);
    f_data[1] = func2(char_1, char_2, c, d);
}
```

The address of the area that returns structure data is stored in R4. The first argument, char_1, of type char is stored in R5. The second argument, char_2, is stored in R6. The third argument, c, of type int is stored in R7. The arguments are then passed to function func2( ). The fourth argument, of type int, which is not stored in the argument register, is stored on the stack and then passed to function func2( ).
CHAPTER 8 REDUCING ARGUMENTS TO CONSERVE STACK AREA

8.2 Optimization Specification and Passing of Arguments

How the arguments stored in the argument registers are handled depends on the specified optimization level. This section briefly describes the optimization specification and handling of arguments.

- Optimization Specification and Handling of Arguments

The fcc911 stores arguments in the argument registers for passing the arguments to the called function. However, if there are too many arguments to be stored in the argument registers, the remaining arguments are passed via the stack. When there are many arguments, the amount of stack area used increases by that amount. Using fewer arguments during a function call can conserve stack area.

How the arguments are handled during a function call depends on the specified optimization level.

If optimization is not specified, the arguments passed by the argument registers are copied to the stack before the function is executed. When optimization is specified, the arguments in the argument registers are accessed as far as possible.

Figure 8.2-1 Optimization Specification and Passing of Arguments

Figure 8.2-1 "Optimization Specification and Passing of Arguments" shows the optimization specification and the function call to a function that passes three arguments.
Function func_sub1( ) is called from function func_main( ). Three arguments are stored in the argument registers and are passed. If optimization is not specified, the called function func_sub1( ) first copies the arguments in the argument registers to the stack.

Optimization level 4 is specified in Figure 8.2-1 "Optimization Specification and Passing of Arguments". In this case, the arguments stored in the argument registers are accessed directly. If enabled by the optimization specification, the called function will directly access the arguments stored in the argument registers.

In the cases below, fast function calls are enabled. Moreover, the amount of stack area used for the arguments during function calls is be reduced.

- All arguments are stored in the argument registers (R4 to R7).
  - There are no more than four four-byte arguments.
  - Eight-byte arguments are included (4 bytes x 4 registers).
  - An argument does not contain a structure or union.
- Optimization is specified.

[Tip]

Softune C Checker:

If an argument of a called function is not accessed at all, the Softune C Checker will output a warning for the argument. In addition, if a structure or union is specified in an argument, the Softune C Checker will output a warning to the effect that performance may deteriorate. These warnings are useful for investigating how arguments are passed.
Part III describes the language extensions of the fcc911. The fcc911 supports specifications for using the FR family architecture. These specifications are referred to as the language extensions. Part III begins with an overview of these language extensions. It then provides notes on including assembler code in a C program and on the specification and placement of the _io type qualifier. This part also provides notes on creating and registering interrupt functions.

CHAPTER 9 "WHAT ARE THE LANGUAGE EXTENSIONS?"
CHAPTER 10 "NOTES ON ASSEMBLER PROGRAM IN C PROGRAMS"
CHAPTER 11 "NOTES ON DEFINING AND ACCESSING THE I/O AREA"
CHAPTER 12 "NOTES ON CREATING AND REGISTERING INTERRUPT FUNCTIONS"
CHAPTER 9  WHAT ARE THE LANGUAGE EXTENSIONS?

The fcc911 provides the following functions as language extensions:
• __io type qualifier
• __interrupt type qualifier
• __asm statement
• Pragma (#pragma)
• Interrupt function
This chapter describes these functions.

9.1 "Extended Type Qualifiers"
9.2 "__asm Statement"
9.3 "Extended Functions Using #pragma"
9.4 "Built-in Functions"
CHAPTER 9  WHAT ARE THE LANGUAGE EXTENSIONS?

9.1 Extended Type Qualifiers

This section describes the extended type qualifiers, which are one part of the language extensions. In addition to the ordinary type qualifiers (const and volatile), the fcc911 provides the __io and __interrupt type qualifiers as extended type qualifiers. These two types of qualifiers are dependent on the FR family architecture.

- Extended Type Qualifiers
  The fcc911 provides the following extended type qualifiers:

  Type qualifiers specific to the fcc911

  \[
  \begin{align*}
  \text{__io type qualifier} \\
  \text{__interrupt type qualifier}
  \end{align*}
  \]

  Sections 9.1.1 "__io Type Qualifier" and 9.1.2 "__interrupt Type Qualifier" provide brief notes on using each of these qualifiers.
This section describes the _io type qualifier, which is an extended fcc911 type qualifier. The _io type qualifier is specified for variables mapped into the I/O area.

**Variables with _io Qualifier**

The _io type qualifier is specific to the fcc911.

In the fcc911, the _io type qualifier is specified for variables mapped into the I/O area (addresses h'0000' to h'03ff'). I/O addressing is used to access variables modified with the _io type qualifier. I/O addressing is specific to access to the I/O area. Compared with ordinary variable access, I/O addressing generates shorter code.

Chapter 11 "NOTES ON DEFINING AND ACCESSING THE I/O AREA", describes in detail the mapping of variables into the I/O area.

Figure 9.1-1 _io Type Qualifier Specification and Access

```
1  static unsigned char _adr0000;
2  static unsigned char _adr0033;
3  static unsigned char _adr0066;

4  extern unsigned char _pdr2;

5  unsigned char a = 0x01;
6  unsigned char b = 0x02;
7  void func_io(void)
8  {
9      char test1,test2,test3;
10     test1 = a + b;
11     test2 = _pdr2 + _pdr3;
12     test3 = _pdr2 - _pdr3;
13    }  
```

Figure 9.1-1 "_io Type Qualifier Specification and Access" shows an example of _io type qualifier specification and access.

In this example, _io type qualifiers are specified when the variable _pdr2 is defined. Because an absolute address in the I/O area is also specified in the #pragma section, area is allocated in section PDR2 mapped from address h'0000' for the variable _pdr2. Moreover, because the storage class "extern" is also specified in the definition, the variable _ddr2 is
handled as an external access variable. When accessed, both variables are accessed via I/O addressing.

When external variables `a` and `b` are accessed, code using 32-bit addressing is generated. When variables modified by the `__io` type qualifier are accessed, code shorter than that used for variable access will be generated using I/O addressing.

[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.
9.1.2  __interrupt Type Qualifier

This section describes the __interrupt type qualifier, which is an fcc911 extended type qualifier. The __interrupt type qualifier is specified for interrupt functions.

■ Functions with __interrupt Type Qualifier

The __interrupt type qualifier is specific to the fcc911.

In the fcc911, the __interrupt type qualifier is specified for interrupt variables.

For a function qualified by the __interrupt type qualifier, the registers used by the function (including the work registers R12 and R13) when the function is called are saved. The multiply-divide registers (MDH and MDL), and the return pointer (RP) are saved as well. The interrupt function is then processed. When the function terminates, all of the saved registers are restored. Processing is then resumed at the location where the interrupt occurred. Use of this type qualifier facilitates coding of interrupt functions in C.

Figure 9.1-2 "Specification of the __interrupt Type Qualifier" shows an example of an interrupt function for which the __interrupt type qualifier is specified. In this example, when an interrupt occurs and the interrupt function timeint() is executed, the work registers R12 and R13 and the registers R0, R1, R2, R3, R4, R5, R6, and R7 used by the function are saved to the stack. In addition, the multiply-divide registers (MDH and MDL) and the return pointer (RP) are saved to the stack.

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

See Chapter 12 "NOTES ON CREATING AND REGISTERING INTERRUPT FUNCTIONS", for more information about functions that use the __interrupt type qualifier.
CHAPTER 9 WHAT ARE THE LANGUAGE EXTENSIONS?

Figure 9.1-2 Specification of the __interrupt Type Qualifier

#define OK 1
int count0;
int def;
int *empty;
int *empty;
int okiff;
int *stop;
void wake(void);
void irex(void);

interrupt void timeint(void)
{
    if(!--count0)
    count0 = 10;
    if(def <= 0)
    if(!--empty && (--*stop) <=0 ) wake();
    if(!--iren) irex();
    else okiff++;
}

When the function starts, the registers used by the function and the work registers are saved. For the function timeint(), the registers R0, R1, R2, R3, R4, R5, R6, and R7 and the work registers R12 and R13 are saved to the stack. In addition, the multiply-divide registers (MDH and MDL) and return pointer (RP) are saved.

When the function terminates, the return pointer (RP), multiply-divide registers (MDH and MDL), registers R0, R1, R2, R3, R4, R5, R6, and R7, and work registers R12 and R13 saved when the function started are restored. The reti instruction restores the values of the PC and PS saved to the system stack and restarts processing from the location where the interrupt occurred.

[Tip]

Softune C Checker:

The Softune C Checker outputs a warning if the __interrupt type qualifier, a language extension, is used in a definition and declaration. The fcc911 supports the same function for coding interrupt functions qualified by the __interrupt type as the fcc896 and fcc907. This check function is useful for porting programs between the fcc896 or fcc907 and fcc911.
This section briefly describes how to use the _asm statement to include Assembler instructions in C programs.
The _asm statement is used to include Assembler instructions in C programs.

Using the _asm Statement to Write Assembler Instructions

The _asm statement is used to include Assembler instructions in C programs. Write the _asm statement as shown below:

```
__asm ("Assembler instruction") ;
```

C programs cannot directly set values in the CPU registers. In addition, some operations in C programs are not fast enough. For these operations, you can use the _asm statement to include instead Assembler instructions in the C program.

The fcc911 uses the _asm statement to include Assembler instructions both inside a function or outside functions.

Figure 9.2-1 Functions in Which the _asm Statement Is Used

The assembler executes the code assuming that the character string coded starting in column 2 is an instruction. A tab code or null character string must be included at the beginning of the character string.

Figure 9.2-1 "Functions in Which the _asm Statement Is Used" shows an example of code including the _asm statement. At the location where an _asm statement is included, an Assembler instruction is expanded.

See Chapter 10 "NOTES ON ASSEMBLER PROGRAM IN C PROGRAMS", for information about including assembler programs using the _asm statement.
CHAPTER 9 WHAT ARE THE LANGUAGE EXTENSIONS?

9.3 Extended Functions Using #pragma

This section describes #pragma as used in the fcc911. The fcc911 provides the following five #pragma types as extended functions:

- `asm/endasm`
- `inline`
- `ilm/noilm`
- `section`
- `intvect/defvect`

Control statements that begin with #pragma specify operations specific to the fcc911. Sections 9.3.1 "Coding #pragma asm/endasm" to 9.3.5 "Changing the Output Section Using #pragma Section" provide brief notes on the use of each function.
9.3.1 Inserting Assembler Programs Using #pragma asm/endasm

This section describes how #pragma asm/endasm is used to include Assembler program in C programs.

The #pragma asm directive specifies the start of an inserting Assembler program.

```
#pragma asm
```

The #pragma endasm directive specifies the end of inserting Assembler program.

```
#pragma endasm
```

C programs cannot directly set CPU registers. Moreover, some of the operations in C programs are not fast enough. For these operations, you can use #pragma asm/endasm to include Assembler program in C programs.

Figure 9.3-1 "Coding #pragma asm/endasm" shows an example of coding #pragma asm/endasm. At the location where #pragma asm/endasm is included, Assembler program is expanded.

See Chapter 10 "NOTE ON ASSEMBLER PROGRAM IN C PROGRAMS", for information about including Assembler program using #pragma asm/endasm.

```
1 void main(void)
2 {
3       #pragma asm
4         BRA32:D L_32, R12
5         LDI #0, R4
6       L_30:
7         ADD #1, R4
8         CMP #10, R4
9         BRZ32 L_32, R12
10      #pragma endasm
11     }
12 }
```

The assembler executes the program assuming that the character string coded starting in column 2 is an instruction. A tab code or null character string must be included at the beginning of the character string.
CHAPTER 9  WHAT ARE THE LANGUAGE EXTENSIONS?

9.3.2 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.

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

Figure 9.3-2 "Inline Expansion of a Function Using #pragma inline" shows an example of using #pragma inline.

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

**Figure 9.3-2 Inline Expansion of a Function Using #pragma inline**

```cpp
extern char block01[10];
extern char block02[20];

int checksum(char *data, int length)
{
  int res;
  int i;
  res = 0;
  for(i = 0; i < length; i++)
    res ^= (*data);
  return(res & 0x00ff);
}

#pragma inline checksum

int proc_block01(void)
{
  int temp;
  temp = checksum(block01, 10);
  return(temp);
}
```

See Chapter 7 "REDUCING FUNCTION CALLS BY EXPANDING 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 the -O option is not specified, inline expansion will not be executed.
[Tip]

For the fcc911:
The following option can be used to specify the function that is to be expanded in line during compilation:

- **-x function name option**

In addition, the following option can be used to specify the number of lines of the function to be expanded in line during compilation:

- **-xauto size option**

In this case also, optimization must be specified using the -O option.
This section describes #pragma ilm/noilm. The #pragma ilm/noilm directive is used to specify the function interrupt level.

Specifying the Interrupt Level Using #pragma ilm/noilm

The #pragma ilm directive is used to specify the function interrupt level and can set the interrupt level for each function.

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

In addition, #pragma noilm releases the switched interrupt level.

```
#pragma noilm
```

Figure 9.3-3 "Using #pragma ilm/noilm to Set Function Interrupt Levels" shows an example of functions that use #pragma ilm.

In this example, the interrupt level when function p_ilm1() is executed is set to 16 on line 1. In addition, specifying #pragma noilm on line 15 releases the interrupt level specified using #pragma ilm(16). As a result, the interrupt level is switched to 16 when function p_ilm1() is called. However, the interrupt level is not switched when function sub_ilm1() is called.
The interrupt level of function sub_ilm1( ) depends on the state when function sub_ilm1( ) is called. Therefore, when function sub_ilm1( ) is executed, processing is executed using the interrupt level of the function that called function sub_ilm1( ).

When creating a system where the interrupt level is switched for each function, use #pragma ilm to specify the interrupt level for each function as shown in Figure 9.3-4 "Using #pragma ilm to Set the Interrupt Level for Each Function".

The minimum unit for which #pragma ilm/noilm can specify the interrupt level is a function. To switch the interrupt level inside a function, use the built-in function __set_il( ).

Figure 9.3-4 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( ).

The #pragma noilm directive only releases the specified #pragma ilm. This function does not return to the previously specified interrupt level.
CHAPTER 9 WHAT ARE THE LANGUAGE EXTENSIONS?

9.3.4 Changing Section Names Using #pragma Section and Specifying Mapping Addresses

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

The #pragma section directive can change the default section names output by the fcc911 to user-specified section names. In addition, #pragma section can change the section attributes.

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

The fcc911 can specify the sections listed in Table 9.3-1 "Default Sections That Can Be Specified Using #pragma Section" for the default section. In addition, the fcc911 can specify the default section attributes listed in Table 9.3-2 "Default Section Attributes That Can Be Specified Using #pragma Section" for attr.

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

Table 9.3-1  Default Sections That Can Be Specified Using #pragma Section

<table>
<thead>
<tr>
<th>Section name</th>
<th>Section type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE</td>
<td>Code area</td>
</tr>
<tr>
<td>INIT</td>
<td>Variable area that specifies an initial value</td>
</tr>
<tr>
<td>DATA</td>
<td>Variable area that does not specify an initial value</td>
</tr>
<tr>
<td>CONST</td>
<td>Area for variables qualified by the const type qualifier</td>
</tr>
<tr>
<td>IO</td>
<td>Area for variables qualified by the _ _io type qualifier</td>
</tr>
</tbody>
</table>

Table 9.3-2  Default Section Attributes That Can Be Specified Using #pragma Section

<table>
<thead>
<tr>
<th>Section attribute name</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE</td>
<td>Program code area</td>
</tr>
<tr>
<td>DATA</td>
<td>Variable area</td>
</tr>
<tr>
<td>STACK</td>
<td>Stack area</td>
</tr>
<tr>
<td>CONST</td>
<td>Variable area with initial value</td>
</tr>
<tr>
<td>COMMON</td>
<td>Area linked by the same address</td>
</tr>
</tbody>
</table>

Figure 9.3-5 "Changing the Output Section Using #pragma Section" shows an example of using #pragma section. In this example, #pragma section is specified on line 1 to change the default IO section to the PDR2 section. In addition, the mapping address is specified so that the PDR2
section is mapped starting from address 0x0000. As a result, the variable qualified by the \_\_io type qualifier is output to the PDR2 section mapped from address 0x0000.

**Figure 9.3-5 Changing the Output Section Using #pragma Section**

The #pragma section changes the default IO section to the PDR2 section. The locate specification specifies mapping to start from address 0x00000000.

### Tip
For the fcc911:
The following option can be used to perform the same processing as that of #pragma section during compilation:

-s default-section-name=new-section-name[,attribute][,mapping-address] option
CHAPTER 9  WHAT ARE THE LANGUAGE EXTENSIONS?

9.3.5 Generating Interrupt Vector Tables Using #pragma intvect/defvect

This section describes #pragma intvect/defvect, which is used to generate interrupt vector tables.

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 a function to be mapped to an interrupt vector that has not been specified using #pragma intvect.

#pragma defvect interrupt-function-name

Figure 9.3-6 Example of Using #pragma intvect

The #pragma section specifies the interrupt vector section INTVECT mapped from address 0xffc00.

Figure 9.3-6 "Example of Using #pragma intvect" shows an example of using #pragma intvect. In this example, startup routine start( ) is registered for interrupt vector number 0. The 16-bit reload timer interrupt processing function timer_int( ) is registered for interrupt vector number 35. All vectors other than vector numbers 0 and 35 of the INTVECT section are set to 0.
Figure 9.3-7 Example of Using #pragma defvect

The #pragma section specifies the interrupt vector section INTVECT mapped from address 0xffc00.

```c
.extern __interrupt void start(void);
.extern __interrupt void timer_int(void);
.extern __interrupt void dummy(void);

#pragma intvect start 0
#pragma intvect timer_int 35
#pragma defvect dummy
```

The function dummy specified using #pragma defvect is registered for vectors for which an interrupt function has not been specified using #pragma intvect.

Figure 9.3-7 “Example of Using #pragma defvect” shows an example of using #pragma defvect. In this example, the interrupt function dummy( ) is registered for all vectors except interrupt vector numbers 0 and 35, which were specified using #pragma intvect.

See Chapter 12 “NOTES ON CREATING AND REGISTERING INTERRUPT FUNCTIONS”, for information about interrupt functions.

<Notes>

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

The interrupt vector tables defined using #pragma intvect/defvect are output to an independent section named INTVECT. When #pragma defvect is executed, the specified interrupt function is set for all interrupt vectors that have not been specified using #pragma intvect among the INTVECT sections.

When #pragma defvect is specified, define all interrupt vector tables using the same compilation unit.
CHAPTER 9 WHAT ARE THE LANGUAGE EXTENSIONS?

9.4 Built-in Functions

This section briefly describes the fcc911 built-in functions. The fcc911 provides the following four types of built-in functions:

- `_DI()`
- `_EI()`
- `_set_il()`
- `_wait_nop()`

Using the Built-in Functions to Add Functions

The fcc911 adds the following built-in functions:

fcc911 built-in functions

{ __DI( )
  __EI( )
  __set_il( )
  __wait_nop( )
}

Sections 9.4.1 “Disabling Interrupts Using `_DI()`” to 9.4.4 “Outputting a Nop Instruction Using `_wait_nop()`” provides brief notes on using each of the built-in functions.
This section describes ___DI(), which is used to disable interrupts. ___DI() is used to disable interrupt in the entire system.

Disabling Interrupts Using ___DI()

The ___DI() directive expands code that masks interrupts. The ___DI() directive can thus disable interrupts for the entire system.

```c
void ___DI(void)
```

Figure 9.4-1 "Using ___DI() to Disable System Interrupts" shows an example of using ___DI() to code a function that disables system interrupts. See Chapter 12 "NOTES ON CREATING AND REGISTERING INTERRUPT FUNCTIONS", for information about interrupt processing.

```
#define ___interrupt void timer_int(void)
  int i;
  ___set_bits(1,1);

  ___DI();

  switch (flag){
    case 0x01: PDR1 = LDB_pr[1];
      break;
      
    case 0x08: PDR1 = LDB_pr[8];
      break;
      
    case 0x0f:
      PDR1 = flag;
      if(((flag << 1))
        flag = 0x01;
      
    TIMEX00 = 0x86b;
    
  ___EI();
```

The ___DI() directive outputs code that disables interrupts. Interrupts are disabled until code that, using ___EI(), enables them is output.
This section describes `__EI()` , which is used to enable interrupts. The `__EI()` direction 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 can thus enable interrupts for the entire system.

```c
void __EI(void)
```

Figure 9.4-2 "Using `__EI()` to Enable System Interrupts" shows an example of using `__EI()` to code a function that enables system interrupts.

See Chapter 12 "NOTES ON CREATING AND REGISTERING INTERRUPT FUNCTIONS", for information about interrupt processing.

![Figure 9.4-2 Using `__EI()` to Enable System Interrupts](image-url)
9.4.3 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. The __set_il() directive can thus determine the interrupt level allowed for the entire system.

```c
void __set_il (interrupt-level)
```

Figure 9.4-3 "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 12 "NOTES ON CREATING AND REGISTERING INTERRUPT FUNCTIONS", for information about interrupt processing.

The __set_il(16) directive sets the interrupt level to 16. Thereafter the interrupt level during execution of an interrupt function is set to 16.
9.4.4 Outputting a Nop Instruction Using __wait_nop( )

This section briefly describes 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.

The __wait_nop( ) directive expands one nop instruction at the location of the function call. Code __wait_nop( ) functions only for the required number of nop instructions.

```c
void __wait_nop (void)
```

Figure 9.4-4 "Using __wait_nop() to Output a Nop Instruction" shows an example of coding a function that uses __wait_nop() to output a nop instruction.

A nop instruction is output at the location where the function call is coded.

```c
1  void wait(void)
2  {
3    __wait_nop();
4  }
```

<Notes>
The fcc911 outputs one nop instruction at the location where __wait_nop( ) is coded. Code __wait_nop( ) functions only for the required number of nop instructions.

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

__wait_nop( ) can control the timing so as to minimize the side effects on optimization.
CHAPTER 10  NOTES ON ASSEMBLER PROGRAM IN C PROGRAMS

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

10.1 "Including Assembler Program in C Programs"

10.2 "Differences Between Using the _asm Statement and #pragma asm/endasm"
10.1 Including Assembler Program in C Programs

This section briefly describes how to code assembler programs. This section also provides notes on the differences between the _ _asm statement and #pragma asm/endasm when a program is written.

Coding Assembler Programs

Assembler source programs consist of the following fields:

<table>
<thead>
<tr>
<th>Symbol field</th>
<th>Instruction field</th>
<th>Operand field</th>
<th>Comment field</th>
<th>Line-feed field</th>
</tr>
</thead>
</table>

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

As listed in Table 10.1-1 "Coding Assembler Programs", the fcc911 enables using the _ _asm statement or #pragma asm/endasm to include Assembler program in C programs.

Table 10.1-1 Coding Assembler Programs

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>_ _asm statement</td>
<td>Only one Assembler instruction can be coded per _ _asm statement.</td>
</tr>
<tr>
<td>#pragma asm/endasm</td>
<td>More than one Assembler instruction can be coded.</td>
</tr>
</tbody>
</table>

As listed in Table 10.1-2 "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 10.1-2 Location for Including Assembler Programs

<table>
<thead>
<tr>
<th>Coding location</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coding inside a function</td>
<td>The Assembler instruction is included as part of the function.</td>
</tr>
<tr>
<td>Coding outside a function</td>
<td>Because the instruction is expanded as an independent section, the section must be defined using a section definition pseudo-instruction.</td>
</tr>
</tbody>
</table>
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 external 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 10.1-1 "Referencing External 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 a C program. In function func1(), the variable b is referenced as _b from the assembler program coded using #pragma asm/endasm.

Figure 10.1-1  Referencing External Variables in a C Program from an Assembler Program

![Assembler Code Example]

The external variable b defined in the C program is referenced from the assembler program as _b (with a prefixed underscore).

Figure 10.1-2 "Referencing Functions and External Variables in a C Program from an Assembler Program" shows an example of referencing functions and variables defined in a C program from an assembler program. In this example, function wait() is called after a value is assigned to external variable cont outside the function in the C program. External variable cont and function wait() are referenced from the assembler program as _cont and _wait, with a prefixed underscore.

![Assembler Code Example]
<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 general-purpose registers R0 to R3, R12, and R13 can be used without special precautions. If other registers are used, it is the user’s responsibility to save and restore them.

- Include only one Assembler instruction per __asm statement.

- When several Assembler instructions are included, use either as many __asm statements as there are Assembler instructions, or use #pragma asm/endasm.

- When an __asm statement or #pragma asm/endasm is included in a C program, optimization might be suppressed. This applies regardless of whether the code optimization option -O is specified for compilation.

- The fcc911 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 conversion from the fcc896 or fcc907 to the fcc911.
10.2 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 10.1-1 "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 10.2-1 "Using the _asm Statement to Include Assembler Program in a Function" shows an example of using the _ _asm statement to include three Assembler instructions in a function.

**Figure 10.2-1 Using the _asm Statement to Include Assembler Program in a Function**

```
1 char c;
2 int a, b;
3
4 void main(void)
5 {
6     a = 0xff;
7     _asm("LDI:32 \#b, R12");
8     _asm("ST \#R0, \#x12");
9     ...
10   }
11
```

Only one Assembler instruction can be used per __asm statement. To include more than one Assembler instruction, include as many _ _asm statements as there are Assembler instructions.

Figure 10.2-2 "Using #pragma asm/endasm to Include Assembler Programs in a Function" shows how #pragma asm/endasm is used to rewrite the function shown in Figure 10.2-1 "Using the _asm Statement to Include Assembler Program in a Function".

There is no difference in the two coding methods. However, when only one Assembler instruction is to be included in a function, it is recommended that the _ _asm statement be used.
Figure 10.2-2 Using \#pragma asm/endasm to Include Assembler Programs in a Function

Including \#pragma
asm/endasm in a function

```c
#include <stdio.h>

int main() {
    int a, b;
    a = 0xff;
    #pragma asm
    LDI #1, R0
    LDI:32 #b, R12
    ST R0, R12
    #pragma endasm
    return 0;
}
```

More than one Assembler instruction can be included in the area enclosed by \#pragma asm/endasm.
10.2 Differences Between Using the _ _asm Statement and #pragma asm/endasm

■ Coding an Assembler Program Outside a Function

When coding an assembler program outside a function, the coded assembler program will be expanded as an independent section. When coding an assembler program outside a function, use the section pseudo-instruction to define the section. If the section has not been defined, the operation result of the coded Assembler instructions will be unpredictable.

Figure 10.2-3 "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, the section pseudo-instruction is used outside the function to define the four-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, it is recommended that #pragma asm/endasm be used.

Figure 10.2-3 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 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 conversion from the fcc896 or fcc907 to the fcc911.
CHAPTER 11  NOTES ON DEFINING AND ACCESSING THE I/O AREA

This chapter describes definition and accessing of resources mapped into the I/O area. This chapter uses as examples the I/O area of the FR30 series of microcontrollers, which belong to the FR family of microcontrollers, to explain how resources mapped into the I/O area are defined and accessed.

11.1 "FR Family I/O Areas"
11.2 "Defining and Accessing Variables Mapped into I/O Areas"
11.1 FR Family I/O Areas

This section briefly describes the I/O areas of the FR family of microcontroller.

FR Family Memory Mapping

Figure 11.1-1 "FR Family Memory Mapping" shows memory mapping of the FR family microcontrollers.

For the FR30 series, the I/O area is between addresses h'0000' and h'07ff'. Individual resource registers are provided for the I/O area. Within this area, the direct addressing area is between addresses h'0000' and h'03ff'. The internal RAM area starts at address h'1000'. The size of the internal RAM area depends on the product. Refer to the manual of the product you are using for more information.

Figure 11.1-2 "MB91101 Series I/O Register Mapping" lists the resource registers that are between addresses h'0000' and h'03ff' of the FR30 series MB91101. Refer to the hardware manual for information about the registers.
11.1 FR Family I/O Areas

Figure 11.1-2 MB91101 Series I/O Register Mapping
CHAPTER 11 NOTES ON DEFINING AND ACCESSING THE I/O AREA

11.2 Defining and Accessing Variables Mapped into I/O Areas

This section describes how to define and access the I/O area of the MB91101 series of microcontrollers.

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 the variable to be mapped into the area.
3. Specify the __io type qualifier to declare access to the variable mapped into the area.

Sample I/O Register Files Provided by the fcc911

When the fcc911 is installed, the files required for defining and accessing the I/O registers of these products are created in the directory below. The following describes how to define and access the I/O area using the MB91101 series as an example.

Figure 11.2-1 Sample I/O File Directory

```
Directory containing tools, e.g., fcc911

Directory containing tools, e.g., C Checker, C Analyzer

Directory containing FR library files

Directory containing standard header file

Sample I/O register file
```
Table 11.2-1 "Sample I/O Register Definition Files" lists the C source files in the directory that contains this sample I/O register file.

**Table 11.2-1 Sample I/O Register Definition Files**

<table>
<thead>
<tr>
<th>File name</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>_io_bsd.c</td>
<td>Defines a bit search module from address 0x3f0 to address 0x3ff when the MB91191 is used.</td>
</tr>
<tr>
<td>_io_clk.c</td>
<td>Defines a clock control register mapped from address 0x480 to address 0x485.</td>
</tr>
<tr>
<td>_io_dicr.c</td>
<td>Defines a delay interrupt control register at address 0x430.</td>
</tr>
<tr>
<td>_io_dmad.c</td>
<td>Defines a DMA descriptor from address 0x1000 in internal RAM when the MB91101 or MB91106 is used.</td>
</tr>
<tr>
<td>_io_ebus.c</td>
<td>Defines an external bus interface register from address 0x600 to address 0x62f.</td>
</tr>
<tr>
<td>_io_hrlc.c</td>
<td>Defines a hold request cancel level register at address 0x431.</td>
</tr>
<tr>
<td>_io_lcr.c</td>
<td>Defines a little endian register at address 0x7fe.</td>
</tr>
<tr>
<td>_io_modr.c</td>
<td>Defines a mode register at address 0x7ff.</td>
</tr>
<tr>
<td>_io_pctr.c</td>
<td>Defines a PLL control register at address 0x488.</td>
</tr>
<tr>
<td>_io_port.c</td>
<td>Defines an I/O port register from address 0x600 to address 0x60b.</td>
</tr>
<tr>
<td>_io_ppg.c</td>
<td>Defines a programmable pulse generator register from address 0x3c0 to address 0x3c7 when the MB91191 is used.</td>
</tr>
<tr>
<td>_io_sio.c</td>
<td>Defines a serial I/O register from address 0x3c8 to address 0x3d3 when the MB91191 is used.</td>
</tr>
<tr>
<td>_io_waitc.c</td>
<td>Defines a wait control register at address 0x7c4 when the MB91191 is used.</td>
</tr>
</tbody>
</table>
Defining the MB91101 I/O Registers

Specifying the following compilation options for the files in the directory that contains the sample I/O files can define all of the I/O registers available in the MB91101 hardware.

```
f911 -cpu mb91101 -c *.c
```

The MB number specified by the -CPU option for compilation has already been defined in the predefined macro `__CPU_MB-number__`. The number is used to select the required files and define the I/O area.

For the definition file `_fr.c`, the following processing procedure is used:

1. `__IO_DEFINE` is defined by `#define` and `_fr.h` is included.

For `_fr.h`, the following processing procedure is used:

1. The `_mb91101.h` file is included by the `-cpu mb91101` option specified during compilation.

For `_mb91101.h`, the following procedure is used to actually define the I/O registers from address 0x0000 to address 0x042f:

1. The file `_frr.h`, declared as a structure of the I/O registers, is included.
2. Because `__IO_DEFINE` has been defined by the `_fr.c`, a macro instruction to replace `__IO_EXTERN` with blanks is defined by `#define` for the `_mb91101.h`.
3. Because `__IO_DEFINE` has been defined by the `_fr.c`, `#pragma section` specifies mapping of an IO_REG section from address 0x0000.
4. Because `__IO_DEFINE` has been defined by the `_fr.c`, a static declaration and the `__io` type qualifier are specified to allocate dummy areas that cannot be accessed from other functions. These areas are allocated for areas that do not have I/O registers between addresses 0x0000 and 0x042f.
5. `__IO_EXTERN` and an `__io` type qualifier are specified to define the registers from address 0x0000 to address 0x03ff. Because `__IO_EXTERN` is replaced with blanks, areas are actually allocated.
6. `__IO_EXTERN` and the volatile type qualifier are specified to define the interrupt controller from addresses 0x0400 to 0x042f. Because `__IO_EXTERN` is replaced with blanks, areas are actually allocated.
7. Access is declared for the I/O registers mapped from address 0x0430 to address 0x07ff.
11.2 Defining and Accessing Variables Mapped into I/O Areas

Figure 11.2-2 Defining Variables Mapped into the I/O Area

The definitions of the registers from address 0x0430 to address 0x07ff include files other than the _fr.c listed in Table 11.2-1 “Sample I/O Register Definition Files”. Specifying the -cpu mb91101 option only defines the registers available in the MB91101.
### Accessing the MB91101 I/O Registers

To access the registers mapped into the I/O area, include \_fr.h without using \#define to define \_\_IO_DEFINE. The following describes how to declare access when the MB91101 is used.

For \_fr.h, the following processing procedure is used:

1. The \_mb91101.h file is included by the \-cpu mb91101 option specified during compilation.

For \_mb91101.h, the following procedure is used to declare access to the I/O registers:

1. The file \_fr.r.h, declared as a structure of the I/O registers, is included.
2. The macro \_\_IO_EXTERN is defined depending on the state of \_\_IO_DEFINE. Because \_\_IO_DEFINE has not been defined, a definition is made to replace \_\_IO_EXTERN with extern.
3. \_\_IO_EXTERN and the \_\_io type qualifier are specified to declare access to the registers from address 0x0000 to address 0x03ff.
4. \_\_IO_EXTERN and the volatile type qualifier are specified to declare access to the interrupt controller from address 0x0400 to address 0x042f.
5. Access is declared for the I/O registers mapped from address 0x0430 to address 0x07ff.

The above operations can declare access to the variables defined in the I/O area.

---

**Figure 11.2-3 Accessing Variables Mapped into the I/O Area**

---

The include file \_fr.h is read without \_\_IO_DEFINE being defined.

The MB91101 I/O register file mb91101.h is read by specifying the \-cpu mb91101 option for compilation.

The file \_fr.r.h, declared as a structure of the I/O registers, is read.

Because \_\_IO_DEFINE has not been defined, \_\_IO_EXTERN is replaced with extern.

Access to the I/O registers is declared.
<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 0x3ff. The I/O area can be accessed using highly efficient dedicated instructions.

• Specify the volatile type qualifier to define I/O variables starting at address 0x400.

• 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 12  NOTES ON CREATING AND REGISTERING INTERRUPT FUNCTIONS

This chapter provides notes on creating and registering interrupt functions. The FR family of microcontrollers has various resources for generating interrupts. The generation and processing of interrupts requires to set initial values for hardware and software.

12.1 "FR Family Interrupts"
12.2 "Required Hardware Settings for Interrupts"
12.3 "Using the __interrupt Type Qualifier to Define Interrupt Functions"
12.4 "Setting of Interrupt Vectors"
CHAPTER 12 NOTES ON CREATING AND REGISTERING INTERRUPT FUNCTIONS

12.1 FR Family Interrupts

This section briefly describes FR family EITs. EIT is an abbreviation of exception, interrupt, and trap.

■ FR Family EITs

EIT is an abbreviation of exception, interrupt, and trap. When any of these events occurs for an FR family microcontroller, the currently executing program is halted and another program starts execution.

FR family EITs are caused as follows:

Reset
User interrupt
(external resource and external interrupt)
Delay interrupt
Undefined instruction exception
Trap instructions (INT and INTE)
Step trace trap
Coprocessor not present trap
Coprocessor error trap

■ Interrupt Handling in the FR Family

The following discussion, centered around internal resource interrupts, describes the interrupt processing handler for FR family microcontrollers.

When an internal resource or external interrupt is requested during program execution on an FR family microcontroller, control passes to an interrupt handler if interrupts are enabled. After the required processing has been executed, the reti instruction is issued to resume execution at the location where the interrupt occurred. Processing restarts from that point.

Figure 12.1-1 "FR Family Interrupt Handling" illustrates the interrupt processing handler for FR family microcontrollers.

The following preparations are required to enable interrupt processing initiated by FR family internal resources and external interrupts:

☐ Hardware settings
  • Setting of system stack area
  • Initialization of the internal resources that can generate interrupt requests
  • Setting of interrupt levels in the interrupt control registers (ICR) in the interrupt controller
  • Setting of the interrupt enable bit of a resource to the enable state and starting of internal resource operation
  • Setting of the ILM and I flag in the CPU to the state that enables acceptance of interrupts

☐ Creation of an interrupt function
12.1 FR Family Interrupts

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

  If 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 12.2 "Required Hardware Settings for Interrupts", to 12.4 “Setting of Interrupt Vectors”, briefly describe the preparations for interrupt processing.

**Figure 12.1-1 FR Family Interrupt Handling**

```c
void main(void)
{
    int data1;
    long ldata;
    long ladd;
    data1 = sub(ldata);
    if (ldata > ladd)(
        ...
        ...
    )
}
```
12.2 Required Hardware Settings for Interrupts

This section describes the required hardware settings for interrupt handling.

- Required Hardware Settings for Interrupts
  Execution of interrupt processing for FR family microcontrollers requires to make the following hardware settings:
  - Set the system stack area.
  - Initialize resources.
  - Set the interrupt control registers.
  - Start operation of internal resources.
  - Enable CPU interrupts.

Sections 12.2.1 "Setting the System Stack Area" to 12.2.5 "Enabling CPU Interrupts" describe the required initializations.
12.2.1 Setting the System Stack Area

This section briefly describes how to set the system stack areas used for interrupt handling.

- Setting the System Stack

  When an allowed FR family interrupt occurs, the CPU saves the contents of the following registers to the system stack and then executes interrupt processing. The interrupt functions use the system stack.

  **Figure 12.2-1 Registers Saved to the System Stack When Interrupt Occurs**

  
<table>
<thead>
<tr>
<th>Registers saved to the system stack when an interrupt occurs</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS register</td>
</tr>
<tr>
<td>PC register (address of the instruction to be executed next)</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
  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 system stack allocation for the linker

  Since register values cannot be directly set in a C program, assembler must be used to code 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.
12.2.2 Initializing Resources

This section briefly describes how to initialize the resources that generate interrupt requests. This initialization value must be defined dependent on the used resources.

- Initializing the 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 FR family microcontroller have an interrupt enable flag and interrupt request flag in a register. First, the resources that 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 12.2-2 "Initializing the Internal Resources (for Interrupts Using the 16-Bit Reload Timer)" shows the registers for the 16-bit reload timer, which is an internal resource. These registers are initialized for interrupt processing using the 16-bit reload timer. See Figure 12.2-9 "Example of Initializing Interrupt Processing", for an example of an initialization program for interrupt processing that uses the 16-bit reload timer.

Figure 12.2-2 Initializing the Internal Resources (for Interrupts Using the 16-Bit Reload Timer)

Interrupt processing using the 16-bit reload timer

<table>
<thead>
<tr>
<th>TMCSR (timer control status register)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt request enable bit</td>
</tr>
<tr>
<td>1: Interrupts enabled</td>
</tr>
<tr>
<td>0: Interrupts disabled</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Timer interrupt request flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>If the counter underflows from 0 to h’ffff’, 1 is set.</td>
</tr>
</tbody>
</table>

If the INTE bit has been set to 1 and interrupts enabled, an interrupt request is issued when the UF bit is set to 1.

<table>
<thead>
<tr>
<th>TMR (16-bit timer register)/TMRLR (16-bit reload register)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMR (at read): Count value of the 16-bit timer</td>
</tr>
<tr>
<td>TMRLR (at write): The initial value of the counter is saved.</td>
</tr>
</tbody>
</table>

Refer to the hardware manual for the specific product for information about the registers for each of its internal resources.
12.2.3 Setting Interrupt Control Registers

The interrupt control registers are set after the resources that generate interrupt requests have been initialized.

- Setting Interrupt Control Registers

  The interrupt control registers are set after the resources that generate interrupt requests have been initialized.

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

  Figure 12.2-3 "Bit Configuration of an FR Family Interrupt Control Register" shows the bit configuration of an FR family interrupt control register.

  When a reset occurs, the interrupt control registers are initialized to interrupts-disabled level 31, which is the interrupts-disabled state. Because ICR4 is always 1, an interrupt level between 16 and 31 can be set, with 16 being the highest and 31 being the lowest. Because the interrupt controller sends the value corresponding to an interrupt to the CPU when an interrupt request is generated by a resource, set a value that matches the system to be created.

  The interrupt control registers of FR family microcontrollers have been mapped at addresses 0x0400 to 0x042f of the I/O area. (See Figure 11.1-2 "MB91101 Series I/O Map").

  Table 12.2-1 "Relationship Between Interrupt Sources, Interrupt Control Registers, and Interrupt Vectors for the MB91101" shows the relationship between interrupt sources and interrupt control registers. Also refer to the hardware manual of the specific product for information about the interrupt control registers.

Figure 12.2-3 Bit Configuration of an FR Family Interrupt Control Register

<table>
<thead>
<tr>
<th>ICR (interrupt control register)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="" /></td>
</tr>
<tr>
<td>ICR4 is always 1.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interrupt level bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
</tr>
<tr>
<td>01110</td>
</tr>
<tr>
<td>01111</td>
</tr>
<tr>
<td>10000</td>
</tr>
<tr>
<td>10001</td>
</tr>
<tr>
<td>11110</td>
</tr>
<tr>
<td>11111</td>
</tr>
</tbody>
</table>
## Table 12.2-1 Relationship Between Interrupt Sources, Interrupt Control Registers, and Interrupt Vectors for the MB91101

<table>
<thead>
<tr>
<th>Interrupt source</th>
<th>Interrupt control register</th>
<th>Corresponding interrupt vector</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number</td>
<td>Address</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Decimal</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>External interrupt 0</td>
<td>ICR00</td>
<td>0x400</td>
</tr>
<tr>
<td>External interrupt 1</td>
<td>ICR01</td>
<td>0x401</td>
</tr>
<tr>
<td>External interrupt 2</td>
<td>ICR02</td>
<td>0x402</td>
</tr>
<tr>
<td>External interrupt 3</td>
<td>ICR03</td>
<td>0x403</td>
</tr>
<tr>
<td>UART0 receive completed</td>
<td>ICR04</td>
<td>0x404</td>
</tr>
<tr>
<td>UART1 receive completed</td>
<td>ICR05</td>
<td>0x405</td>
</tr>
<tr>
<td>UART2 receive completed</td>
<td>ICR06</td>
<td>0x406</td>
</tr>
<tr>
<td>UART0 send completed</td>
<td>ICR07</td>
<td>0x407</td>
</tr>
<tr>
<td>UART1 send completed</td>
<td>ICR08</td>
<td>0x408</td>
</tr>
<tr>
<td>UART2 send completed</td>
<td>ICR09</td>
<td>0x409</td>
</tr>
<tr>
<td>DMAC0</td>
<td>ICR10</td>
<td>0x40a</td>
</tr>
<tr>
<td>DMAC1</td>
<td>ICR11</td>
<td>0x40b</td>
</tr>
<tr>
<td>DMAC2</td>
<td>ICR12</td>
<td>0x40c</td>
</tr>
<tr>
<td>DMAC3</td>
<td>ICR13</td>
<td>0x40d</td>
</tr>
<tr>
<td>DMAC4</td>
<td>ICR14</td>
<td>0x40e</td>
</tr>
<tr>
<td>DMAC5</td>
<td>ICR15</td>
<td>0x40f</td>
</tr>
<tr>
<td>DMAC6</td>
<td>ICR16</td>
<td>0x410</td>
</tr>
<tr>
<td>DMAC7</td>
<td>ICR17</td>
<td>0x411</td>
</tr>
<tr>
<td>A/D</td>
<td>ICR18</td>
<td>0x412</td>
</tr>
<tr>
<td>Reload timer 0</td>
<td>ICR19</td>
<td>0x413</td>
</tr>
<tr>
<td>Reload timer 1</td>
<td>ICR20</td>
<td>0x414</td>
</tr>
<tr>
<td>Reload timer 2</td>
<td>ICR21</td>
<td>0x415</td>
</tr>
<tr>
<td>PWM0</td>
<td>ICR22</td>
<td>0x416</td>
</tr>
<tr>
<td>PWM1</td>
<td>ICR23</td>
<td>0x417</td>
</tr>
<tr>
<td>PWM2</td>
<td>ICR24</td>
<td>0x418</td>
</tr>
<tr>
<td>PWM3</td>
<td>ICR25</td>
<td>0x419</td>
</tr>
<tr>
<td>U-TIMER0</td>
<td>ICR26</td>
<td>0x41a</td>
</tr>
<tr>
<td>U-TIMER1</td>
<td>ICR27</td>
<td>0x41b</td>
</tr>
<tr>
<td>U-TIMER2</td>
<td>ICR28</td>
<td>0x41c</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>Reserved for system</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>Delayed interrupt source bit</td>
<td>ICR47</td>
<td>0x42f</td>
</tr>
</tbody>
</table>
12.2.4 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 12.2-4 "Starting Operation of Internal Resources (for Interrupt Processing Using the 16-Bit Reload Timer)" shows how to start operation of the 16-bit reload timer, which is an internal resource. See Figure 12.2-9 "Example of Initializing Interrupt Processing", for an example of an initialization program for interrupt processing that uses the 16-bit reload timer.

**Figure 12.2-4 Starting Operation of Internal Resources (for Interrupt Processing Using the 16-Bit Reload Timer)**

<table>
<thead>
<tr>
<th>Interrupt processing using the 16-bit reload timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMCSR (timer control status register)</td>
</tr>
<tr>
<td>bits: 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0</td>
</tr>
<tr>
<td>Timer count enable bit: 1: Wait for activation trigger, 0: Stop count operation</td>
</tr>
<tr>
<td>Software trigger bit:</td>
</tr>
</tbody>
</table>

There are also resources that immediately generate an interrupt request when the bit for starting the resource is set. If an interrupt occurs before initialization of interrupt processing is completed for these resources, the results can be unpredictable. Therefore, initialize the resources and start operations that match the system to be created.

Refer to the hardware manual of the specific product for information about the registers of each resource.
12.2.5 Enabling CPU Interrupts

This section briefly describes how to set CPU interrupts to be enabled.

### Enabling CPU Interrupts

The CPU that accepts interrupts is set after the resources that execute interrupt processing have been set up.

For FR family microcontrollers, the values of the interrupt enable flag (I flag) and interrupt level mask register (ILM) in the program status register (PS) are used to set the hardware interrupt level of the entire system.

Figure 12.2-5 "Bit Configuration of the PS Register" shows the bit configuration of the PS register.

ILM represents the interrupt level that is currently allowed. If an interrupt request that exceeds the level indicated by the ILM occurs, interrupt processing will be executed. An interrupt level between 0 and 31 is possible, with 0 being the highest and 31 being the lowest, but only a value between 16 and 31 can actually be set. A reset sets the interrupt level to the lowest interrupt level of 31.

**Figure 12.2-5 Bit Configuration of the PS Register**

For the fcc911, the function `_set_il()` and `#pragma ilm/noilm` can be used to set the interrupt level.

The ILM value determines the interrupt level allowed for the entire system. If the level of the interrupt that occurs exceeds the ILM value and the I flag has been set to 1 to enable interrupts, interrupt processing will be executed.
12.2 Required Hardware Settings for Interrupts

- **Using \_\_set_il(\_) to Set the Interrupt Level in a Function**

The function \_\_set_il(\_) changes the value of the ILM register to a value set as a variable, enabling the interrupt level to be set at an optional location in a function.

\_\_set_il (interrupt-level)

Figure 12.2-6 "Using Function \_\_set_il() to Set the Interrupt Level in a Function" shows a function for which the interrupt level has been set using the function \_\_set_il(\_). In function main(\_), function \_\_set_il(\_) is called on line 13. Because 31 has been specified for the variable, function \_\_set_il(\_) generates code that sets 31 in the ILM register.

Function \_\_set_il(\_) can generate code that changes the interrupt level at an optional location in a function.

Figure 12.2-6 Using \_\_set_il() to Set the Interrupt Level in a Function

```
#include "_icr.h"
#include "_tmsr0.h"

extern void initLed(void);
extern void init_time16(void);

void main(void)
{
  initLed();
  init_time16();

  \_\_set_il(31);

  while(1){}
}

void init_time16(void)
{
  _ICR[19].bit.ICR = 17;
  _THMSR0 = 0x6499;
  _TMCSR0.word = 0x001b;
}
```

Function \_\_set_il(\_), which sets the interrupt level, is called in function main(\_). As a result, the interrupt level can be changed at an optional location in the function.
Using #pragma ilm/noilm to Set the Interrupt Level in a Function

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

Use #pragma ilm to specify the interrupt level before the function whose interrupt level will be changed.

#pragma ilm (interrupt-level)

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

#pragma noilm

Figure 12.2-7 "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 31 has been specified on line 7. When function main() starts, code that sets 31 in the ILM register is generated. Because #pragma noilm has been specified on line 20 after function main(), code that sets an interrupt level will not be generated when function init_timer16() defined on line 22 is started.

<Notes>

The interrupt level setting function _set_il() and #pragma ilm/noilm of the fcc911 are able set a value between 0 and 31, but only a value between 16 and 31 can actually be set. If a value between 0 and 15 is set, the value obtained by adding 16 to the set value will be set.
12.2 Required Hardware Settings for Interrupts

**[Tip]**

Softune C Checker:

The Softune C Checker will output the message "The interrupt level setting function has been used" at the location where the function _set_il() or #pragma ilm/noilm has been specified. The fcc896 and fcc907 also support the _set_il() function and #pragma ilm/noilm. When conversion from the fcc896 or fcc907 to the fcc911 is done, this message is useful for investigating what should be applied to the system to be converted.

**Using the I flag to Enable Interrupts for the Entire System**

Finally, the I flag is set after all of the functions related to interrupts have been initialized.

When the I flag is 1, interrupts are enabled for the entire system. Resetting clears the I flag to 0. Whether interrupts that are higher than the interrupt level set by the ILM register are enabled and whether interrupt processing is actually executed depend on the status of the I flag.

The fcc911 uses _DI( ) to set the I flag to 0 to disable interrupts.

```
void _DI (void)
```

The fcc911 uses _EI( ) to set the I flag to 1 to enable interrupts.

```
void _EI (void)
```

Figure 12.2-8 "Example of a Function Using _EI( ) to Enable Interrupts" shows an example of a function that uses _EI( ) to enable system interrupts.

**Figure 12.2-8 Example of a Function Using _EI( ) to Enable Interrupts**

```c
#include "_icr.h"

extern void init_led(void);
extern void init_timer16(void);

void main(void)
{
    init_led();
    init_timer16();
    _set_il(31);
    while(1());
}

void init_timer16(void)
{
    _ICR[19].bit.ICR = 17;
    _MR0  = 62499;
    _MCUSR0.hword = 0x081b;
}
```
CHAPTER 12  NOTES ON CREATING AND REGISTERING INTERRUPT FUNCTIONS

Figure 12.2-9 “Example of Initializing Interrupt Processing” shows an example of an initialization program of interrupt processing using the 16-bit reload timer.

For this program, the function main( ) calls the function init_timer16( ) that initializes the 16-bit reload timer. For the function init_timer16( ), interrupt level 17 is set in the 16-bit reload timer interrupt control register. The reload value is then set in the _TMR register. Finally, operation is started at the same time that the _TMRLR0 register is initialized. The timer thus starts operation.

When initialization of the 16-bit reload timer terminates, control is returned to the function main( ). System interrupts are then enabled after _set_il(31) sets the interrupt level of the entire system. Interrupts using the 16-bit reload timer are thus enabled.

**Figure 12.2-9  Example of Initializing Interrupt Processing**

```assembly
#include "_icr.h"
#include "tmrlr0.h"

extern void init_led(void);
extern void init_timer16(void);

void main(void)
{
  init_led();
  init_timer16();
  _set_il(31);
  while(1);
}

void init_timer16(void)
{
  _ICR[19].bit.ICR = 17;
  _TMRLR0 = 62499;
  _TMCSR0.lword = 0x081b;
}
```

<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.
12.2 Required Hardware Settings for Interrupts

[Tip]

Softune C Checker:

The Softune C Checker will output the messages "The interrupt mask setting has been used." and "The interrupt mask release function has been used." at the locations where _ _EI( ) and _ _DI( ) are used. The fcc896 and fcc907 also support _ _EI( ) and _ _DI( ). When converting from the fcc896 or fcc907 to the fcc911, these messages are useful for investigating the contents to be applied to the system to be converted.
12.3 Using the __interrupt Type Qualifier to Define Interrupt Functions

Sections 12.2.1 "Setting the System Stack Area" to 12.2.4 "Starting Resource Operation" 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 FR family microcontroller occurs, the following procedure is used to execute interrupt processing:

1. The PS and PC (eight bytes total) are saved to the system stack.
2. The contents of the ILM register are received and updated to the interrupt level.
3. The S flag of the PS register is cleared (the system stack is used).
4. The instruction at the address that indicates the corresponding interrupt vector is executed.

Figure 12.3-1 Executing an Interrupt Processing Function

As shown in Figure 12.3-1 "Executing an Interrupt Processing Function", the hardware automatically saves the contents of registers and passes control to an interrupt routine when an interrupt occurs.

When an interrupt processing routine is coded using the assembler, the reti instruction is issued at the end of the interrupt processing routine. The reti instruction restores the PS and PC register values saved to the system stack, and restarts execution from the location where the interrupt occurred.

When an interrupt processing function is coded using the fcc911, the __interrupt type qualifier
is used for the interrupt function as shown in Figure 12.3-2 "Using the _ _interrupt Type Qualifier to Define an Interrupt Function". Using this coding, the fcc911 compiles the specified function as an interrupt function.

Figure 12.3-2 Using the _ _interrupt Type Qualifier to Define an Interrupt Function

```
[storage-class] _ _interrupt void function-name ( void ) {
  
  The interrupt program is coded in C.
  
}
```

When a function qualified by the _ _interrupt type qualifier is executed, all of the registers used by the function, including the work registers (R12 and R13), and return pointer, are saved. When the interrupt function terminates, the saved registers are restored and the reti instruction is then issued. The reti instruction restores the values of the registers saved to the system stack, and restarts execution from the location where the interrupt occurred.

Figure 12.3-3 "Example of an Interrupt Function Using the _ _interrupt Type Qualifier" shows an example of an interrupt processing function.

When function timeint( ) qualified by the _ _interrupt type qualifier is called, the registers R0 to R7 used by the function, work registers R12 and R13, and the return pointer are saved to the system stack.

When the function terminates, the saved registers are restored and the reti instruction is issued.
For a function qualified by the __interrupt type qualifier, always specify void as the function type.

When interrupt processing terminates, the reti instruction restores the registers that were saved to the system stack when the interrupt occurred. Saving of the registers enables to resume the interrupted processing. Because the registers are restored after the interrupt function returns the return values and terminates, the return values of the function cannot be accessed. In addition, even though the return values on the system stack are returned, the locations cannot be checked from the function restored from the interrupt because the reti instruction switches the stack. As a result, the return values cannot be accessed even though they are expressly returned. To eliminate wasteful operations, always specify type void for interrupt functions. If the results of an interrupt function are required, define an external variable where the results can be saved, then access the external variable as required.

---

### Notes

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

When interrupt processing terminates, the reti instruction restores the registers that were saved to the system stack when the interrupt occurred. Saving of the registers enables to resume the interrupted processing. Because the registers are restored after the interrupt function returns the return values and terminates, the return values of the function cannot be accessed. In addition, even though the return values on the system stack are returned, the locations cannot be checked from the function restored from the interrupt because the reti instruction switches the stack. As a result, the return values cannot be accessed even though they are expressly returned. To eliminate wasteful operations, always specify type void for interrupt functions. If the results of an interrupt function are required, define an external variable where the results can be saved, then access the external variable as required.

### Tip

Softune C Checker:

The Softune C Checker will output the message "The type qualifier for coding an interrupt function has been used" at the location where the __interrupt type qualifier is used. The fcc896 and fcc907 also support the __interrupt type qualifier. When conversion from the fcc896 or fcc907 to the fcc911 is done, this message is useful for investigating the what should be applied to the system to be converted.

---

### Example of an Interrupt Function Using the __interrupt Type Qualifier

```c
# define OK 1

#define OK 1
int count10;
void wake(void);

@ 14
interrupt void timeint(void)
{
    if(!--count10)
        count10 = 10;
    if(0 <= count10 && --*stop) <= 0 ) wake();
    if(!--pree) lres();
    else clkoff++;;
}

@end

@end

// Define a function that uses the __interrupt type qualifier.
// When the function is started, the work registers, registers R0, R1, R2, R3, R4, R5, R6, and R7, and work registers R12 and R13 saved when the function started are restored. The reti instruction is then issued to stop processing of the interrupt function. The reti instruction restores the values of the PC and PS saved to the system stack, and restarts processing from the location where the interrupt occurred.

<Notes>

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

When interrupt processing terminates, the reti instruction restores the registers that were saved to the system stack when the interrupt occurred. Saving of the registers enables to resume the interrupted processing. Because the registers are restored after the interrupt function returns the return values and terminates, the return values of the function cannot be accessed. In addition, even though the return values on the system stack are returned, the locations cannot be checked from the function restored from the interrupt because the reti instruction switches the stack. As a result, the return values cannot be accessed even though they are expressly returned. To eliminate wasteful operations, always specify type void for interrupt functions. If the results of an interrupt function are required, define an external variable where the results can be saved, then access the external variable as required.

<table>
<thead>
<tr>
<th>Tips</th>
</tr>
</thead>
<tbody>
<tr>
<td>Softune C Checker:</td>
</tr>
<tr>
<td>The Softune C Checker will output the message &quot;The type qualifier for coding an interrupt function has been used&quot; at the location where the __interrupt type qualifier is used. The fcc896 and fcc907 also support the __interrupt type qualifier. When conversion from the fcc896 or fcc907 to the fcc911 is done, this message is useful for investigating the what should be applied to the system to be converted.</td>
</tr>
</tbody>
</table>

```
12.4 Setting of Interrupt Vectors

This section briefly describes how to use #pragma intvect/defvect to register interrupt vectors for interrupt functions.

### Using #pragma intvect/defvect to Register Interrupt Functions

The hardware settings for executing interrupt processing and the definitions of the interrupt functions for actually executing processing have been completed. As the last step, the created interrupt functions must be registered.

The FR family provides interrupt vectors at addresses 0xFFC00 to 0xFFFFF. Registering the required interrupt processing functions in this area enables the required interrupt processing to be executed when an interrupt occurs.

See Table 12.2-1 "Relationship Between Interrupt Sources, Interrupt Control Registers, and Interrupt Vectors for the MB91101", for the relationship between the interrupt sources, interrupt control registers, and interrupt vectors.

The fcc911 can use #pragma intvect to register interrupt functions.

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

Figure 12.4-1 "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 as interrupt vector number 0. The 16-bit reload timer interrupt processing function timer_int( ) is registered as interrupt vector number 35.
When #pragma intvect is executed, a 256-byte interrupt vector table is created in a section named INTVECT. When the table is created, the interrupt vectors that have not been assigned a vector number by #pragma intvect are filled with zeros. In addition, when #pragma defvect is executed, the specified interrupt function can be set for all interrupt vectors that have been filled with zeros as described above.

In this example, the #pragma section has specified mapping of INTVECT at address 0xffc00. As a result, the interrupt vector table INTVECT is mapped from address 0xffc00 to address 0xfffff.

In addition, access that involves the __interrupt type qualifier has been declared on line 3 for a startup routine coded in assembler.

A startup routine named start has been registered for the reset vector of interrupt number 0 and will therefore be executed when a reset occurs.

<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 fcc911 will output a warning message if the specification of the __interrupt type qualifier is omitted.

The #pragma section can set the mapping addresses of the interrupt vector table INTVECT created by #pragma intvect. Set the mapping addresses based on the system to be created.
The index follows on the next page.
This is listed in alphabetic order.
Symbols
"const" type modifier, defining variable using .......... 28
"static" and its variable area .................................. 20
#define definition and numeric constant .......... 26
#pragma asm/endasm, inserting assembler code using .................................. 77
#pragma ilm/noilm used for setting interrupt level in function .................. 120
#pragma ilm/noilm, specifying interrupt level using .................. 80
#pragma inline specification, executing inline expansion using ........ 56
#pragma inline, inline expansion using .................. 78
#pragma intvect/defvect used for registering interrupt function .................. 127
#pragma intvect/defvect, interrupt vector table generating using ......... 84
#pragma, extended function that use ..................... 76
__asm statement used for writing assembler instruction .......... 75
__DI( ), disabling interrupt using ..................... 87
__EI( ), enabling interrupt using ..................... 88
__interrupt type modifier used for coding interrupt function .................. 124
__interrupt type modifier, function with .............. 73
__io modifier, variable with ................................ 71
__set_il( ) function used for setting interrupt level in function .... 119
__set_il( ), setting interrupt level using ............ 89
__wait_nop( ), outputting nop instruction using .... 90

A
accessing variable and function in C program from assembler program .... 93
actual argument and dummy argument areas ........... 50
area allocated to stack during function call ............ 50
Argument .......................................................... 63
argument register .................................................. 60
argument register and calling function that return structure or union .......... 63
argument register and function call, storing argument in ................ 61
argument register save area .................................. 50
assembler code insertion using #pragma asm/endasm .......... 77
assembler instruction writing, using __asm statement ........ 75
assembler program having multiple instructions in function, coding .......... 95
assembler program outside function, coding .......... 97
assembler program, coding .................................. 92
automatic variable ........................................... 8
automatic variable and external variable ............ 35
automatic variable and frame pointer .................. 39
automatic variable and statically allocated variable .... 32
automatic variable and statically allocated variable, definition and scope of .... 34

B
bit field definition and boundary alignment .......... 43
bit field length 0, bit field of ................................ 44
boundary alignment of fcc911 ................................ 42
built-in function, function addition by ............... 86

C
C program’s variable and function accessing from assembler program .......... 93
calling function and argument register that return structure or union .......... 63
calling function when argument not being stored in argument register .......... 62
coding assembler program .................................. 92
coding assembler program having multiple instructions in function .......... 95
coding assembler program outside function ........... 97
coding interrupt function using __interrupt type modifier .......... 124
CPU interrupt enable .......................................... 118

disabling interrupt using __DI( ) ..................... 87
dynamically allocated variable ................................ 8

e
EITS, FR family ........................................ 110
enable interrupt of entire system using I flag .......... 121
enabling interrupt using __EI( ) ..................... 88
executing inline expansion using #pragma inline specification .......... 56
extended function that use #pragma .................... 76
extended type modifier ........................................ 70
external variable ............................................ 10, 14
external variable and automatic variable .............. 35
external variable, initial value and
variable area for ........................................ 16

F
fcc911, bounday alignment .............................. 42
fcc911, sample I/O register file provided by .......... 102
FR family EITS ............................................ 110
FR family memory mapping ............................ 100
FR family microcontroller, interrupt processing procedure for .... 110
frame pointer and automatic variable .................. 39
function addition by built-in function .................. 86
function call, area allocated to stack during ....... 50

H
hardware setting for interrupt .......................... 112
hidden parameter save area ............................. 50

I
I flag used for enabling interrupt of entire system 121
I/O area accessing using bit field and union .......... 45
I/O area register accessing operation
as variable from C program ........................... 102
I/O area register as variable from
C program, operation for accessing ............. 102
I/O register file provided by fcc911, sample .......... 102
initial value and variable area for external variable 16
initialized variable and initialization at execution .... 18
inline expansion is not executed even though
#pragma inline is specified ................................ 55
inline expansion of function ............................. 54
inline expansion using #pragma inline .............. 78
interrupt control register setting ...................... 115
interrupt function registering
using #pragma intvect/defvect to .................... 127
interrupt level specifying using #pragma ilm/noilm 80
interrupt processing function in
interrupt vector table, registration of ............ 111
interrupt processing procedure for
FR family microcontroller .......................... 110
interrupt routine, creation of .......................... 110
interrupt vector table generating
using #pragma intvect/defvect ....................... 84
interrupt vector table, registration of
interrupt processing function in .................... 111
interrupt, required hardware setting for ........... 112

L
Line, note on expanding function in .............. 57
local variable area ......................................... 50

M
mapping into memory area .............................. 6
MB91101 I/O register accessing ....................... 106
MB91101 I/O register, defining ....................... 104
memory mapping, FR family .......................... 100
modifier, extended type ............................... 70

N
nop instruction using __wait_nop(), outputting ..... 90
numeric constant and #define definition .......... 26

O
old frame pointer ....................................... 50
optimization specification and
handling of argument .................................. 64

P
program structure element ............................ 4

R
register save area ...................................... 50
register variable ........................................ 8
resource initializing ................................... 114
resource operation starting ........................... 117
return address storage area (old RP) ............... 50

S
section name changing using #pragma section and
specifying mapping address .......................... 82
setting interrupt level in function
using #pragma ilm/noilm ............................ 120
setting interrupt level in function
using function __set_il( ) ............................ 119
setting interrupt level using __set_il( ) .............. 89
specifying interrupt level using #pragma ilm/noilm .80
specifying mapping address and changing section
name using #pragma section .......................... 82
stack state when function call being nested ....... 52
static global variable, example of function that has 22
static local variable, example of function that has 23
static variable ........................................... 10
statically allocated variable .......................... 10
statically allocated variable and automatic variable 32
statically allocated variable and automatic variable,
definition and scope of .............................. 34
statically allocated variable and
variable area in RAM .................................. 33
INDEX

storing argument in argument register and function call.................................61
system stack setting.....................................................................................113

V

V

variable defining using "const" type modifier ............ 28
variable with __io modifier ......................................................... 71
variable declared as "static" and variable area .......... 20