User's Manual

CA850 Ver. 3.20

C Compiler Package

C Language

Target Device
V850 Series

Document No. U18513EJ1V0UM00 (1st edition)
Date Published May 2007 CP(K)

[MEMO]

[MEMO]

INTRODUCTION

Target Devices	The V850 Series C compiler packages create the object codes for NEC Electronics's V850 Series RISC microcontrollers.
Readers	This manual is intended for user engineers who wish to develop application systems using the V850 Series C compiler package.
Purpose	This manual explains the C language specifications supported by the C compiler (ca850) included in the package.
Organization	This manual contains the following information: • OVERVIEW • BASIC LANGUAGE SPECIFICATIONS • COMPILATION ENVIRONMENT • C LANGUAGE EXPANSION • CALLING PROGRAM • STARTUP ROUTINE • LIBRARY FUNCTION • FOR EFFICIENT USE
Notes on reading this manual	• In this manual, sections on the V850 Series peculiar to "V850E" are specified by the title name or the mark " [V850E] ". Sections peculiar to other than "V850E" are specified by the title or the mark " [V850] ", etc..

Related Documents

Read this manual together with the following documents.
The related documents indicated in this publication may include preliminary versions. However, preliminary versions are not marked as such.

Documents related to development tools (user's manuals)

Document Name		Document No.
CA850 Ver. 3.20 C Compiler Package	Operation	U18512E
	C Language	This manual
	Assembly Language	U18514E
	Link Directives	U18515E
PM+ Ver. 6.30 Project Manager		U18416E
ID850 Ver. 3.00 Integrated Debugger	Operation	U17358E
ID850NW Ver. 3.10 Integrated Debugger	Operation	U17369E
ID850QB Ver. 3.20 Integrated Debugger	Operation	U17964E
SM+ System Simulator	Operation	U17246E
	User Open Interface	U18212E
SM850 Ver. 2.50 System Simulator	Operation	U16218E
SM850 Ver. 2.00 or Later System Simulator	External Part User Open Interface Specifications	U14873E
RX850 Ver. 3.20 or Later Real-Time OS	Basics	U13430E
	Installation	U17419E
	Technical	U13431E
	Task Debugger	U17420E
RX850 Pro Ver. 3.21 Real-Time OS	Basics	U18165E
	Internal Structure	U18164E
	Task Debugger	U17422E
RX850V4 Ver. 4.22 Real-Time OS	Functionalities	U16643E
	Internal Structure	U16644E
	Task Debugger	U16811E
AZ850 Ver. 3.30 System Performance Analyzer		U17423E
AZ850V4 Ver. 4.10 System Performance Analyzer		U17093E
TW850 Ver. 2.00 Performance Analysis Tuning Tool		U17241E

[MEMO]

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 BASIC LANGUAGE SPECIFICATIONS

This chapter explains the basic language specifications supported by the CA850.

The CA850 supports the language specifications stipulated by the ANSI standards. These specifications include items that are stipulated as processing definitions. This chapter explains the language specifications of the items dependent on the processing system of the V850 microcontrollers.

The differences between when options strictly conforming to the ANSI standards are used and when those options are not used are also explained.

For the expanded specifications of the CA850, refer to "CHAPTER 3 C LANGUAGE EXPANSION".

1.1 Dependent on Processing System Stipulated

This section explains items dependent on processing system stipulated by ANSI standards.

1.1.1 Data type and size

- The number of bits of 1 byte is 8.
- The number of bytes, byte order, and coding in an object are stipulated as follows.

char type	1 byte
short type	2 bytes
int, long, float, double type	4 bytes
Pointer	Same as unsigned int type

The byte order in a word is "from lower to higher". Signed integers are expressed as 2's complements. The most significant bit indicates a sign (0 if positive or 0, and 1 if negative).

1.1.2 Translation stages

The ANSI standards specify eight translation stages (known an "phases") of priorities among syntax rules for translation. In the third translation phase "decomposition of source file into preprocessing tokens and sequences of white-space characters", whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character is implementation-defined. In this implementation, each nonempty sequence of white-space characters other than new-line is retained, not replaced by one space character.

1.1.3 Diagnosis message

Error messages including the source file name and line number (only when the line number can be specified) are output in response to a translation unit that violates a syntax rule or limit. These error messages are classified into three types: "alarm", "fatal error", and "other error" messages.

1.1.4 Free-standing environment

(1) The name and type of a function called on starting program processing are not stipulated in a free-standing environment[Note]. Therefore, they are dependent on the user-own coding or target system.

(2) The effect of terminating a program in a free-standing environment is not stipulated. Therefore, it is dependent on the user-own coding or target system.

Note Environment in which a C program is executed without using the functions of the operating system.
The ANSI Standards specify two environments: a free-standing environment and a host environment.
The CA850 does not supply a host environment at present.

1.1.5 Executing program

The configuration of the interactive unit is not stipulated. Therefore, it is dependent on the uuser-own coding and target system.

1.1.6 Character set

The values of elements of the execution environment character set are ASCII codes.

1.1.7 Multi-byte characters

Multibyte characters are not supported by character constants. However, comments and character strings in Japanese are supported.

1.1.8 Meaning of character indication

The values of expanded notation are stipulated as follows:

Table 1 - 1 Expanded Notation and Meaning

Expanded Notation	Value (ASCII)	Meaning
\a	07	Alert (alarm sound)
\b	08	Back space
\f	0C	Form feed (new page)
\n	0A	New line (carriage return)
\r	0D	Carriage return (return)
\t	09	Horizontal tab
\v	0B	Vertical tab

1.1.9 Translation limit

The limit values of translation are explained below.

The values marked * are guaranteed values. These values may be exceeded in some cases, but the operation is not guaranteed.

Table 1 - 2 Translation Limit

Contents	limit Value
Number of nesting levels of compound statements, repetitive control structures, and selective control structures (however, dependent on the number of "case" labels)	127
Number of nesting levels of condition embedding	255
Number of pointers, arrays, and function declarators (in any combination) qualifying one arithmetic type, structure type, union type, or incomplete type in one declaration	16
Number of nesting levels enclosed by parentheses in a complete declarator	255(*)
Number of nesting levels of an expression enclosed by parentheses in a complete expression	255(*)
Valid number of first characters in a macro name	1023(*)
Valid number of first characters of an external identifier	1022
Valid number of first characters in an internal identifier	1023
Number of identifiers having the valid block range declared by an external identifier in one translation unit and in one basic block	4095(*)
Number of macro identifiers simultaneously defined in one translation unit^Note^	2047
Number of dummy arguments in one function definition and number of actual arguments in one function call	255
Number of dummy arguments in one macro definition	127
Number of actual arguments in one macro call	127
Number of characters in one logical source line	32768
One character string constant after concatenation, or number of characters in a wide character string constant	32768
Number of nesting levels for include (#include) files	50
Number of "case" labels for one "switch" statement (including those nested, if any)	1025
Number of members of a single structure or single union	1023(*)
Number of enumerate constants in a single enumerate type	1023(*)
Number of nesting levels of a structure or union definition in the arrangement of a single structure declaration	63(*)

Note The upper limit of the macro identifier can be changed by a C compiler option (-Xm).

1.1.10 Quantitative limit

(1) The limit values of the general integer types (limits.h file)

The limits.h file stipulates the limit values of the values that can be expressed as general integer types (char type, signed/unsigned integer type, and enumerate type).

Because multibyte characters are not supported, MB_LEN_MAX does not have a corresponding limit, and is only defined with MB_LEN_MAX as 1.

If a -Xchar=unsigned option$^{\text{Note}}$ of the CA850 is specified, CHAR_MIN is 0, and CHAR_MAX takes the same value as UCHAR_MAX.

The limit values defined by the limits.h file are as follows.

Table 1 - 3 Limit Values Defined by limits.h File

Name	Value	Meaning
CHAR_BIT	8	The number of bits (= 1 byte) of the minimum object not in bit field
SCHAR_MIN	- 128	Minimum value of signed char type
SCHAR_MAX	+ 127	Maximum value of signed char type
UCHAR_MAX	+ 255	Maximum value of unsigned char type
CHAR_MIN	- 128	Minimum value of char type
CHAR_MAX	+ 127	Maximum value of char type
SHRT_MIN	- 32768	Minimum value of short int type
SHRT_MAX	+ 32767	Maximum value of short int type
USHRT_MAX	+ 65535	Maximum value of unsigned short int type
INT_MIN	- 2147483648	Minimum value of int type
INT_MAX	+ 2147483647	Maximum value of int type
UINT_MAX	+ 4294967295	Maximum value of unsigned int type
LONG_MIN	- 2147483648	Minimum value of long int type
LONG_MAX	+ 2147483647	Maximum value of long int type
ULONG_MAX	+ 4294967295	Maximum value of unsigned long int type

Note Specify a simple char type code.

(2) The limit values of the floating-point type (float.h file)

The limit values related to the characteristics of the floating-point type are defined by the float.h file.

The limit values defined by the float.h file are as follows.

Table 1 - 4 Limit Values Defined by float.h File

Name	Value	Meaning
FLT_ROUNDS	+ 1	Rounding mode for floating-point addition. 1 for the V850 microcontrollers (rounding in the nearest direction).
FLT_RADIX	+ 2	Radix of exponent (b)
FLT_MANT_DIG	+ 24	Number of numerals (p) with FLT_RADIX of floating-point mantissa as base
DBL_MANT_DIG	+ 24	
LDBL_MANT_DIG	+ 24	
FLT_DIG	+ 6	Number of digits of a decimal number[Note 1] (q) that can round a decimal number of q digits to a floating-point number of p digits of the radix b and then restore the decimal number of q
DBL_DIG	+ 6	
LDBL_DIG	+ 6	
FLT_MIN_EXP	- 125	Minimum negative integer (e_{min}) that is a normalized floating-point number when FLT_RADIX is raised to the power of the value of FLT_RADIX minus 1.
DBL_MIN_EXP	- 125	
LDBL_MIN_EXP	- 125	
FLT_MIN_10_EXP	- 37	Minimum negative integer $\log_{10} b^{e_{min}-1}$ that falls in the range of a normalized floating-point number when 10 is raised to the power of its value.
DBL_MIN_10_EXP	- 37	
LDBL_MIN_10_EXP	- 37	
FLT_MAX_EXP	+ 128	Maximum integer (e_{max}) that is a finite floating-point number that can be expressed when FLT_RADIX is raised to the power of its value minus 1.
DBL_MAX_EXP	+ 128	
LDBL_MAX_EXP	+ 128	
FLT_MAX_10_EXP	+ 38	Maximum integer $\log_{10}((1- b^{-p}) * b^{e_{max}})$ that falls in the range of a finite floating-point number when 10 is raised to the power of its value.
DBL_MAX_10_EXP	+ 38	
LDBL_MAX_10_EXP	+ 38	
FLT_MAX	3.40282347E + 38F	Maximum number of finite floating-point numbers that can be expressed $(1 - b^{-p}) * b^{e_{max}}$
DBL_MAX	3.40282347E + 38F	
LDBL_MAX	3.40282347E + 38F	
FLT_EPSILON	1.19209290E - 07F	Difference[Note 2] between 1.0 that can be expressed by specified floating-point number type and the lowest value which is greater than 1.0, b^{1-p}
DBL_EPSILON	1.19209290E - 07F	
LDBL_EPSILON	1.19209290E - 07F	
FLT_MIN	1.17549435E - 38F	Minimum value of normalized positive floating-point number $b^{e_{min}-1}$
DBL_MIN	1.17549435E - 38F	
LDBL_MIN	1.17549435E - 38F	

Notes 1 DBL_DIG and LDBL_DIG are 10 or more in the ANSI standards but 6 in the V850 microcontrollers because both the double and long double types are 32 bits.

2 DBL_EPSILON and LDBL_EPSILON are 1E-9 or less in the ANSI standards, but 1.19209290E-07F in the V850 microcontrollers.

1.1.11 Identifier

An external name must consist of up to 1022 characters and must be able to be identified uniformly. Uppercase and lowercase characters are distinguished.

1.1.12 char type

A char type with no type specifier (signed, unsigned) specified is treated as a signed integer as the default assumption. However, a simple char type can be treated as an unsigned integer by specifying the -Xchar=unsigned option[Note] of the CA850.

The types of those that are not included in the character set of the source program required by the ANSI standards (escape sequence) is converted for storage, in the same manner as when types other than char type are substituted for a char type.

Note Specify a simple char type code.

Example

```
char c = '\777' /* Value of c is -1. */
```

1.1.13 Floating-point constants

The floating-point constants conform to IEEE754[Note].

Note IEEE: Institute of Electrical and Electronics Engineers
IEEE754 is a standard to unify specifications such as the data format and numeric range in systems that handle floating-point operations.

1.1.14 Character constants

(1) Both the character set of the source program and the character set in the execution environment are basically ASCII codes, and correspond to members having the same value.
For the character set of the source program, however, character codes in Japanese can be used (refer to "1.1.15 Character string").

(2) The last character of the value of an integer character constant including two or more characters is valid.

(3) A character that cannot be expressed by the basic execution environment character set or escape sequence is expressed as follows.

(a) An octal or hexadecimal escape sequence takes the value indicated by the octal or hexadecimal notation.

\777	511

(b) The simple escape sequence is expressed as follows.

\'	'
\"	"
\?	?
\\	\

(c) Character constants of multibyte characters are not supported.

1.1.15 Character string

The default character code is Shift JIS.

A character code can be changed by using the -Xk option of the CA850.

Option specification

```
-Xk=[e | euc | n | none | s | sjis]
```

The character codes in the output object file can be converted by the -Xkt option of the CA850.

Option specification

```
-Xkt=[e | euc | n | none | s | sjis]
```

If n or none is specified, the character code is not converted.

1.1.16 Header file name

The method to reflect the string in the two formats (< > and " ") of a header file name on the header file or an external source file name is stipulated in "1.1.33 Loading header file".

1.1.17 Comment

A comment can be described in Japanese. The character code is the same as the character string in "1.1.15 Character string".

1.1.18 Signed constants and unsigned constants

If the value of a general integer type is converted into a signed integer of a smaller size, the higher bits are truncated and a bit string image is copied.

If an unsigned integer is converted into the corresponding signed integer, the internal representation is not changed.

1.1.19 Floating-points and general integers

If the value of a general integer type is converted into the value of a floating-point type, and if the value to be converted is within a range that can be expressed but not accurately, the result is rounded to the closest expressible value[Note].

Note If the value is precisely in the middle, it is rounded to an even number (with the least significant bit of the mantissa being 0).

1.1.20 double type and float type

In the processing system of the V850 microcontrollers, a double type is expressed as a floating-point number in the same manner as a float type, and is treated as 32-bit (single-precision) data.

1.1.21 Signed type in operator in bit units

The characteristics of the shift operator conform to the stipulation in "1.1.27 Shift operator in bit units". The other operators in bit units for signed type are calculated as unsigned values (as in the bit image).

1.1.22 Members of structures and unions

If the value of a member of a union is stored in a different member, it is stored according to an alignment condition. Therefore, the members of that union are accessed according to the alignment condition (refer to "2.1.6 Structure type" and "2.1.7 Union type").

In the case of a union that includes a structure sharing the arrangement of the common first members as a member, the internal representation is the same, and the result is the same even if the first member common to any structure is referenced.

1.1.23 sizeof operator

The value resulting from the "sizeof" operator conforms to the stipulation related to the bytes in an object in "1.1.1 Data type and size". The number of bytes in a structure and union includes padding.

1.1.24 Cast operator

When a pointer is converted into a general integer type, the required size of the variable is the same as the size of the int type. The bit string is saved as is as the conversion result.

Any integer can be converted by a pointer. However, the result of converting an integer smaller than an int type is expanded according to the type.

1.1.25 Division/remainder operator

The result of the division operator ("/") when the operands are negative and do not divide perfectly with integer division, is as follows:

If either the divisor or the dividend is negative, the result is the smallest integer greater than the algebraic quotient. If both the divisor and the dividend are negative, the result is the largest integer less than the algebraic quotient.

If the operand is negative, the result of the "%" operator takes the sign of the first operand in the expression.

1.1.26 Addition and subtraction operators

If two pointers indicating the elements of the same array are subtracted, the type of the result is int type, and the size is 4 bytes.

1.1.27 Shift operator in bit units

If E1 of "E1 >> E2" is of signed type and takes a negative value, an arithmetic shift is executed.

1.1.28 Storage area class specifier

The storage area class specifier "register" is declared to increase the access speed as much as possible, but this is not always effective[Note].

Note For the registers to be allocated, refer to "7.5.1 Register specifier".

1.1.29 Structure and union specifier

(1) A simple int type bit field without signed or unsigned appended is treated as a signed field, and the most significant bit is treated as the sign bit. However, the simple int type bit field can be treated as an unsigned field by specifying the -Xbitfield option[Note] of the CA850.

Note Specify a simple int type bit field code.

(2) To retain a bit field, a storage area unit to which any address with sufficient size can be assigned can be allocated. If there is insufficient area, however, the bit field that does not match is packed into to the next unit according to the alignment condition of the type of the field.

(3) The allocation sequence of the bit field in unit is from lower to higher.

(4) Each member of the non-bit field of one structure or union is aligned at a boundary as follows.

char, unsigned char type, and its array	Byte boundary
short, unsigned short type, and its array	Halfword boundary
Others (including pointer)	Word boundary

1.1.30 Enumerate type specifier

The type of an enumeration specifier is signed int.
When the -Xenum_type=string option is specified, however, it is as follows.

char	Treated as char
uchar	Treated as unsigned char
short	Treated as short
ushort	Treated as unsigned short

1.1.31 Type qualifier

The configuration of access to data having a type qualified to be "volatile" is dependent upon the address (I/O port, etc.) to which the data is mapped.

1.1.32 Condition embedding

(1) The value for the constant specified for condition embedding and the value of the character constant appearing in the other expressions are equal.

(2) The character constant of a single character must not have a negative value.

1.1.33 Loading header file

(1) A preprocessing directive in the form of "#include <character string>"

A preprocessing directive in the form of "#include <character string>" searches for a header file from the folder specified by the -I option if "character string" does not begin with "\"Note 2, and then searches the ..\inc850 folder with a relative path from the bin folder where the ca850 is placed.

If a uniformly identified header file is searched with a character string specified between delimiters "<" and ">", the whole contents of the header file are replaced.

Notes Both "\" and "/" are regarded as the delimiters of a folder.

Example

```
#include <header.h>
```

The search order is as follows:

(1) Folder specified by -I

(2) Standard folder

(2) A preprocessing directive in the form of "#include "character string""

A preprocessing directive in the form of "#include "character string"" searches for a header file from the folder where the source file exists, and then searches the ..\inc850 folder via a relative path from the bin folder where the ca850 is placed.

If a header file uniformly identified is searched with a character string specified between delimiters (") and ("), the whole contents of the header file are replaced.

Example

```
#include "header.h"
```

The search order is as follows:

(1) Folder where source file exists

(2) Folder specified by -I

(3) Standard folder

(3) The format of "#include preprocessing character phrase string"

The format of "#include preprocessing character phrase string" is treated as the preprocessing character phrase only if the preprocessing character phrase string is a macro that is replaced to the form of <character string> or "character string".

(4) Between a string delimited (finally) and a header file name

Between a string delimited (finally) and a header file name, the length of the alphabetic characters in the string is identified, and the file name length valid in the compiler operating environment is valid. The folder that searches a file conforms to the above stipulation.

1.1.34 #pragma directives

The CA850 can specify the following #pragma directives.

(1) Description with assembler instruction

```
#pragma     asm
            assembler instruction
#pragma     endasm
```

Assembler directives can be described in a C language source program.

For the details of description, refer to "3.4 Describing Assembler Instruction".

(2) Inline expansion specification

```
#pragma inline function-name [, function-name ...]
```

A function that is expanded inline can be specified.

For the details of expansion specification, refer to "3.8 Inline Expansion".

(3) Device type specification

```
#pragma cpu device-name
```

Specify so that a device file defining the machine-dependent information of the device used is referenced.This function is the same as the device specification option (-cpu) of the CA850.

For the device file, refer to "2.6 Device File".

(4) Data or program memory allocation

```
#pragma section section-type ["section-name"] [begin | end]
#pragma text ["section-name"] function-name
```

(a) section

Allocates variables to an arbitrary section. For details about the allocation method, refer to "3.1 Allocation of Data to Section".

(b) text

A function to be allocated in a text section with an arbitrary name can be specified. For details about the allocation specification, refer to "3.2 Allocating Functions to Sections" by Specifying Section Name.

(5) Peripheral I/O register name validation specification

```
#pragma ioreg
```

The peripheral I/O registers of a device are accessed by using peripheral function register names. For the details of access, refer to "3.3 Peripheral I/O Register" by Using Register Name.

(6) Interrupt/exception handler specification

```
#pragma interrupt interrupt-request-name function-name [allocation-method]
```

Interrupt exception handlers are described in C language.

For details, refer to "3.7 Interrupt/Exception Processing Handler".

(7) Interrupt disable function specification

```
#pragma block_interrupt function-name
```

Interrupts are disabled for the entire function.

For description, refer to "3.6 Disabling Interrupts".

(8) Task specification

```
#pragma rtos_task [function-name]
```

A task that runs on an RTOS is described in C language.

For details, refer to "3.9.1 Description of task".

(9) Structure type packing specification

```
#pragma pack([1248])
```

Specifies the packing of a structure type. The packing value, which is an alignment value of the member, is specified as the numeric value. A value of 1, 2, 4, or 8 can be specified. When the numeric value is not specified, the setting is the default assumption.

For details, refer to "3.11 Structure Packing Function".

1.1.35 Predefined macro names

All the following macro names are supported.

Macros not ending with "__" are supplied for the sake of former C language specifications (K&R specifications). To perform processing strictly conforming to the ANSI standards, use macros with "__" before and after.

Table 1 - 5 A List of Supported Macros

Macro Name	Definition
__LINE__	Line number of source line at that point (decimal).
__FILE__	Name of assumed source file (character string constant).
__DATE__	Date of translating source file (character string constant in the form of "Mmm dd yyyy". The name of the month is the same as that created by the asctime function stipulated by the ANSI standards (three alphabetic characters with only the first character being uppercase) and the first character of dd is blank if its value is less than 10).
__TIME__	Translation time of source file (character string constant having format "hh:mm:ss" similar to the time created by the asctime function).
__STDC__	Decimal constant 1 (defined when -ansi option is specified)Note
__v800 __v800__	Decimal constant 1
__v850 __v850__	Decimal constant 1
__v850e __v850e__	Decimal constant 1 (defined by CA850, if V850Ex is specified as a target device)
__v850e2 __v850e2__	Decimal constant 1 (defined by CA850, if V850E2/xxx is specified as a target device)
__CA850 __CA850__	Decimal constant 1
__CHAR_SIGNED__	Decimal constant 1 (defined if signed is specified by -Xchar option or when -Xchar option is not specified)
__CHAR_UNSIGNED__	Decimal constant 1 (defined when unsigned is specified by -Xchar option)
__DOUBLE_IS_32BITS__	Decimal constant 1
_DOUBLE_IS_32BITS	Decimal constant 1
CPU macro	Decimal constant 1 of a macro indicating the target CPU. A character string indicated by "product type specification" in the device file with "__" prefixed and suffixed is defined.
Register mode macro	Decimal constant 1 of a macro indicating the target CPU. Macros defined as a register mode are as follows. 32 register mode : __reg32__ 26 register mode : __reg26__ 22 register mode : __reg22__

Note For the processing to be performed when the -ansi option is specified, refer to "1.2 ANSI Option".

1.1.36 Definition of special data type

NULL, size_t, and ptrdiff_t defined by stddef.h file are as follows.

Table 1 - 6 Definition of NULL, size_t, ptrdiff_t(stddef.h File)

NULL / size_t / ptrdiff_t	Definition
NULL	`((void *)0)`
size_t	`unsigned int`
ptrdiff_t	`int`

1.2 ANSI Option

If the -ansi option is specified by the CA850, processing strictly conforming to the ANSI standards is performed. The difference between when the -ansi option is specified and when it is not specified are as follows.

Table 1 - 7 Processing When -ansi Option Strictly Conforming to Language Specifications Is Specified

Item	-ansi Specified	-ansi Not Specified
Trigraph series	Replaces trigraph series.	Does not replace.
Bit field	Error[Note 1] occurs if type other than int is specified for bit field.	Outputs alarm message and permits.
Scope of argument	Multiple defined error occurs if automatic variable having same name as argument of function is declared.	Outputs alarm message and validates automatic variable.
Substitution of pointer 1	Error occurs if numeric value of pointer type is substituted into general integer type[Note 2] variable.	Outputs alarm message, casts, and substitutes.
Substitution of pointer 2	Error occurs if pointers indicating different types are substituted for each other.	Outputs alarm message and permits.
Type conversion	Error occurs if conversion into pointer of array that is not left-member value is performed.	Outputs alarm message and permits.
Comparison operator	Error occurs if comparison is made between arithmetic type variable and pointer.	Outputs alarm message and permits.
Conditional operator	Error occurs if both second and third expressions are not general integer type, same structure, same union, or numeric value of pointer type to type same as substitution destination.	Outputs alarm message, casts, and substitutes.
# line number	Error occurs.	Treated in same manner as "#line line number"[Note 3].
Character # in middle of line	Error occurs if character '#' appears in middle of line.	Outputs warning message and enables the character
_asm	Outputs warning message and treats the character as function call. However, __asm is valid.	Treated as assembler insertion[Note 4]
__STDC__	Defines as macro with value of 1.	Does not define.
Binary constant	Error occurs if "0b" or "0B" is followed by one or more "0" or "1".	Treats "0b" or "0B" followed by one or more "0" or "1" as a binary constant.

Notes 1 Normal error beginning with "E". The same applies hereafter.

2 char type, signed/unsigned integer type, and enumerate type

3 Refer to the ANSI standards.

4 Refer to "3.4 Describing Assembler Instruction".

CHAPTER 2 COMPILATION ENVIRONMENT

This chapter explains how the CA850 handles data, registers, and the environment during execution.

2.1 Internal Representation and Value Area of Data

This section explains the internal representation and value area of each type for the data handled by the CA850.

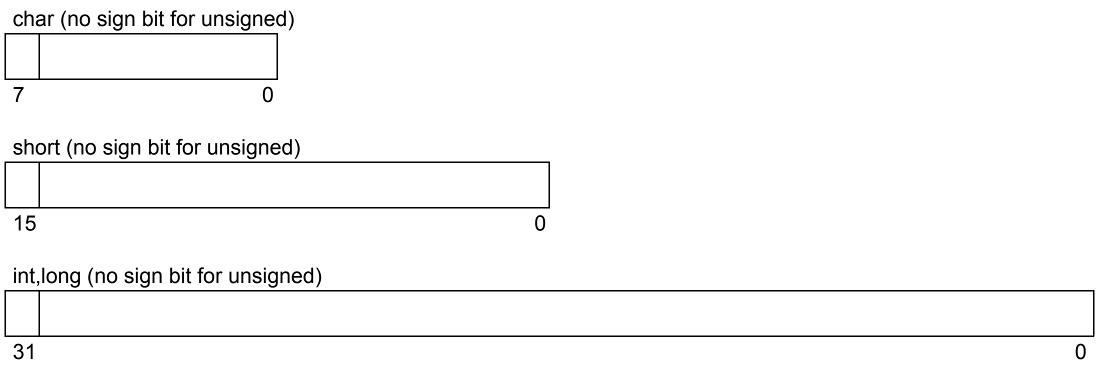
2.1.1 Integer type

(1) Internal representation

The leftmost bit in an area is a sign bit with a signed type (type declared without "unsigned"). The value of a signed type is expressed as 2's complement.

If -Xchar=unsigned is specified, however, a char type specified without "signed" or "unsigned" is assumed to be unsigned.

Figure 2 - 1 Internal Representation of Integer Type


char (no sign bit for unsigned)
7
0
short (no sign bit for unsigned)
15
0
int,long (no sign bit for unsigned)
31
0

(2) Value area

Table 2 - 1 Value Area of Integer Type

Type	Value Area
char[Note]	-128 to +127
short	-32768 to +32767
int	-2147483648 to +2147483647
long	-2147483648 to +2147483647
unsigned char	0 to 255

Table 2 - 1 Value Area of Integer Type

Type	Value Area
unsigned short	0 to 65535
unsigned int	0 to 4294967295
unsigned long	0 to 4294967295

Note The value area is 0 to 255 if "-Xchar=unsigned" is specified by the CA850.

Caution 64-bit operation cannot do the CA850.

2.1.2 Floating-point type

(1) Internal representation

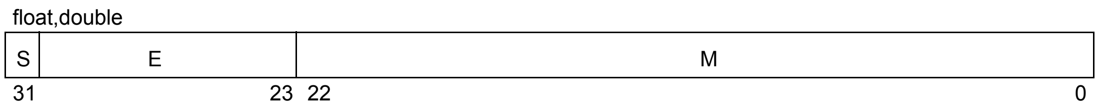
The internal representation of floating-point type data conforms to IEEE754[Note].

The leftmost bit in the area is the sign bit. If the value of this sign bit is 0, the data is a positive value; if it is 1, the data is a negative value.

A double type is a floating-point representation the same as a float type, and is handled as 32-bit (single-precision) data.

Note IEEE: Institute of Electrical and Electronics Engineers
IEEE754 is a standard to unify specifications such as the data format and numeric range in systems that handle floating-point operations.

Figure 2 - 2 Internal Representation of Floating-Point Type



S : Sign bit of mantissa

E : Exponent (8 bits)

M : Mantissa (23 bits)

(2) Value area

Table 2 - 2 Value Area of Floating-Point Type

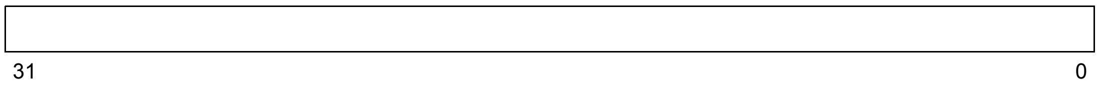
Type	Value Area of Absolute Value
float,double	1.18×10^{-38} to 3.40×10^{38}

2.1.3 Pointer type

(1) Internal representation

The internal representation of a pointer type is the same as that of an unsigned int type.

Figure 2 - 3 Internal Representation of Pointer Type

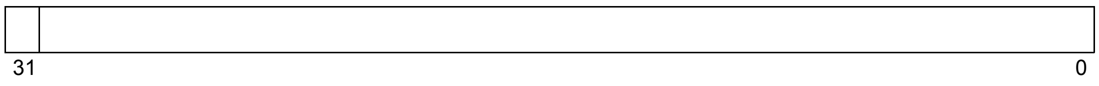


2.1.4 Enumerate type

(1) Internal representation

The internal representation of an enumerate type is the same as that of a signed int type. The leftmost bit of the area is the sign bit.

Figure 2 - 4 Internal Representation of Enumerate Type



2.1.5 Array type

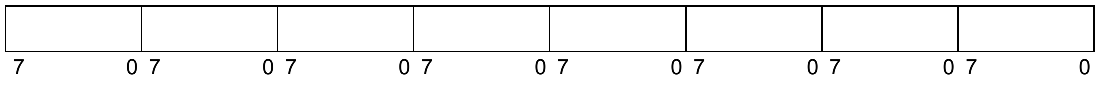
(1) Internal representation

The internal representation of an array type arranges the elements of an array in the form that satisfies the array condition(alignment) of the elements.

```
char a[8] = { 1, 2, 3, 4, 5, 6, 7, 8 }
```

The internal representation of the array shown above is as follows.

Figure 2 - 5 Internal Representation of Array Type



2.1.6 Structure type

(1) Internal representation

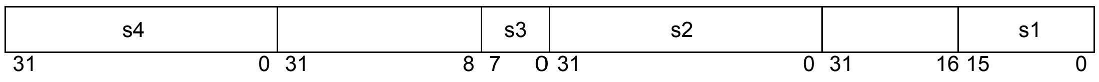
The internal representation of a structure type arranges the elements of a structure in a form that satisfies the alignment condition of the elements.

Example

```
struct {
    short   s1;
    int     s2;
    char    s3;
    long    s4;
} tag;
```

The internal representation of the structure shown above is as follows.

Figure 2 - 6 Internal Representation of Structure Type



For the internal representation when the structure type packing function is used, refer to "3.11 Structure Packing Function".

2.1.7 Union type

(1) Internal representation

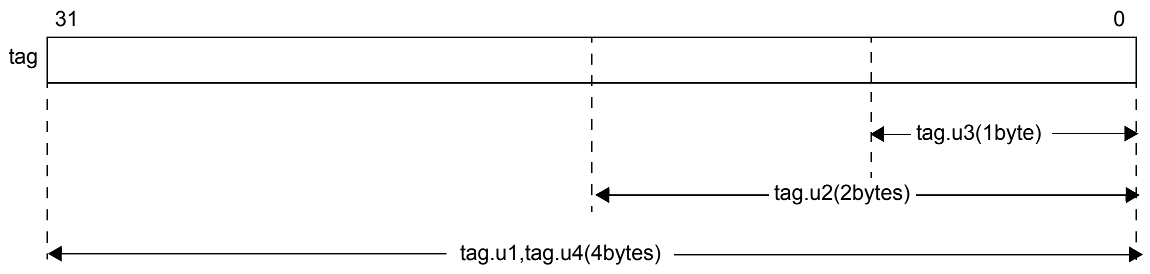
A union is considered as a structure whose members all start with offset 0 and that has sufficient size to accommodate any of its members. The internal representation of a union type is like each element of the union is placed separately at the same address.

Example

```
union {
    int     u1;
    short   u2;
    char    u3;
    long    u4;
} tag;
```

The internal representation of the union shown in the above example is as follows.

Figure 2 - 7 Internal Representation of Union



2.1.8 Bit field

(1) Internal representation

An area including the declared number of bits is reserved for a bit field. The most significant bit of the area for a bit field declared to be of signed type is a sign bit.

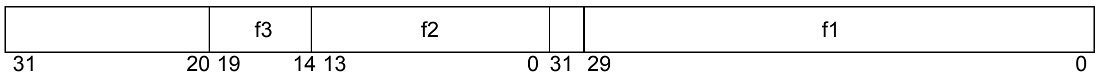
The bit field declared first is allocated from the least significant bit of a word area. If the alignment condition of the type specified in the declaration of a bit field is exceeded as a result of allocating an area that immediately follows the area of the preceding bit field to the bit field, the area is allocated starting from a boundary that satisfies the alignment condition.

Example

```
struct {
    unsigned int    f1 : 30;
    int             f2 : 14;
    unsigned int    f3 :  6;
} flag;
```

The internal representation for the bit field in the above example is as follows.

Figure 2 - 8 Internal Representation of Bit Field



The ANSI standards do not allow char and short types to be specified for a bit field, but CA850 allows this. In this case, a warning message is output, and padding is performed according to the alignment condition of the specified type[Note].

For the internal representation of bit field when the structure type packing function is used, refer to "3.11 Structure Packing Function".

Note An error occurs if -ansi is specified as an option of the CA850.

2.1.9 Alignment conditions

(1) Alignment conditions for basic type

Table 3-3 shows the alignment conditions for basic types. If -Xi of the CA850 is specified, however, all the alignment types are word boundaries.

Table 2 - 3 Alignment Condition for Basic Type

Basic Type	Alignment Condition
(unsigned) char and its array type	Byte boundary
(unsigned) short and its array type	Halfword boundary
Other basic types (including pointer)	Word boundary

(2) Alignment condition for union type

The alignment condition for the union type varies as shown in Table 3-4, depending on the maximum member size.

Table 2 - 4 Alignment Condition for Union Type

Maximum Member Size	Alignment Condition
2 bytes < size	Word boundary
Size <= 2 bytes	Maximum member size boundary

Here are examples of the respective cases:

Example 1

```
union tug1 {
    unsigned short i;   /* 2-byte member */
    unsigned char  c;   /* 1-byte member */
};                      /* The union is aligned with 2 bytes. */
```

Example 2

```
union tug2 {
    unsigned int  i;    /* 4-byte member */
    unsigned char c;    /* 1-byte member */
};                      /* The union is aligned with 4 bytes. */
```

(3) Alignment condition for structure type

The alignment condition for the structure type differs as shown in Table 3-5, depending on the size of the structure (excluding the size of the integer). If -Xi of the CA850 is specified, however, all the structure types are word boundaries.

Table 2 - 5 Alignment Condition of Structure Type

Structure Size	Alignment Condition	
2 bytes < size	Word boundary	
Size <= 2 bytes	If member of type more than int type exists	Word boundary
	If there is no member of type more than int type, and 1 byte < size <= 2 bytes	Halfword boundary
	If there is no member of type more than int type, and size <= 1 byte	Byte boundary

Here are examples of the respective cases:

Example 1

```
struct SS {
    int     i;          /* 4-byte member */
    char    c;          /* 1-byte member */
};      /* Structure is aligned with 4 bytes. */
```

Example 2

```
struct BIT_I {
    int     i1 : 5;     /* 4-byte member (size is 1 byte or less) */
};      /* Structure is aligned with 4 bytes because member type is int. */
```

Example 3

```
struct BIT_C {
    char    c1 : 5;     /* 1-byte member */
};      /* Structure is aligned with 1 byte. */
```

Example 4

```
struct BIT_CC {
    char    c1 : 5;     /* 1-byte member */
    char    c2 : 5;     /* 1-byte member */
};      /* Structure is aligned with 2 bytes because size is 2 bytes. */
```

(4) Alignment condition for function argument

The alignment condition for a function argument is a word boundary.

(5) Alignment condition for executable program

The alignment condition when an executable object file is created by linking object files is a halfword boundary.

2.2 General-Purpose Registers

Table 3-6 shows how the CA850 uses the general-purpose registers.

The general-purpose registers includes the following functions.

(1) Software register bank

The number of the work registers (r10 through r19) and register variable registers (r20 through r29) used can be reduced by the -reg option of CA850 (refer to "2.4 Software Register Bank").

(2) Mask register function

In the 32-register mode and 22-register mode, registers r20 and r21 can be used to set a mask value (refer to "2.5 Mask Register").

Table 2 - 6 Using General-Purpose Registers

Register		Usage
r0	Zero register	Used for operation as value of 0. Also used to reference data located at const section (read-only section placed in ROM area)[Note].
r1	Assembler-reserved register	Used for instruction expansion by assembler.
r2(hp)	Handler stack pointer	Reserved for system.
r3(sp)	Stack pointer	Used to indicate beginning of stack frame.
r4(gp)	Global pointer	Used to reference external variable.
r5(tp)	Text pointer	Used to indicate beginning of text section (.text section)
r6 - r9	Argument registers	Used to pass argument.
r10 - r19	Work registers	Used as work register during operation (r10 is also used to pass return value of function).
r20 - r29	Register variable registers	Used as an area for register variables.
r30(ep)	Element pointer	Used to reference external variable specified to be located in internal RAM or external RAM section[Note].
r31(lp)	Link pointer	Used to pass return address of function.

Note For the allocation of data to a section, refer to "3.1 Allocation of Data to Section".

2.3 Referencing Data

How the CA850 references data are as follows.

Table 2 - 7 Referencing Data

Type	Referencing Method
Numeric constant	Immediate
Character constant	Offset from global pointer (gp) Offset from r0 register
Automatic variable, argument	Offset from stack pointer (sp)
External variable, static variable in function	Offset from global pointer (gp) Offset from element pointer (ep) Offset from r0 register
Function address	Operated during execution by using offset from text pointer (tp)

2.4 Software Register Bank

Because the CA850 implements a register bank function by software, three register modes are provided. By specifying these register modes efficiently, the contents of some registers do not need to be saved or restored when an interrupt occurs or the task is switched. As a result, the processing speed can be improved.

The register modes are specified by using the register mode specification option (-reg) of CA850.

This function reduces the number of registers internally used by the CA850 on a step-by-step basis.

As a result, the following effects can be expected:

(1) The registers not used can be used for the application program (that is, a source program in assembly language).

(2) The overhead required for saving and restoring registers can be reduced.

Note In an application program that has many variables to be allocated to registers by the CA850, the variables so far allocated to a register are accessed from memory when a register mode has been specified. As a result, the processing speed may drop.

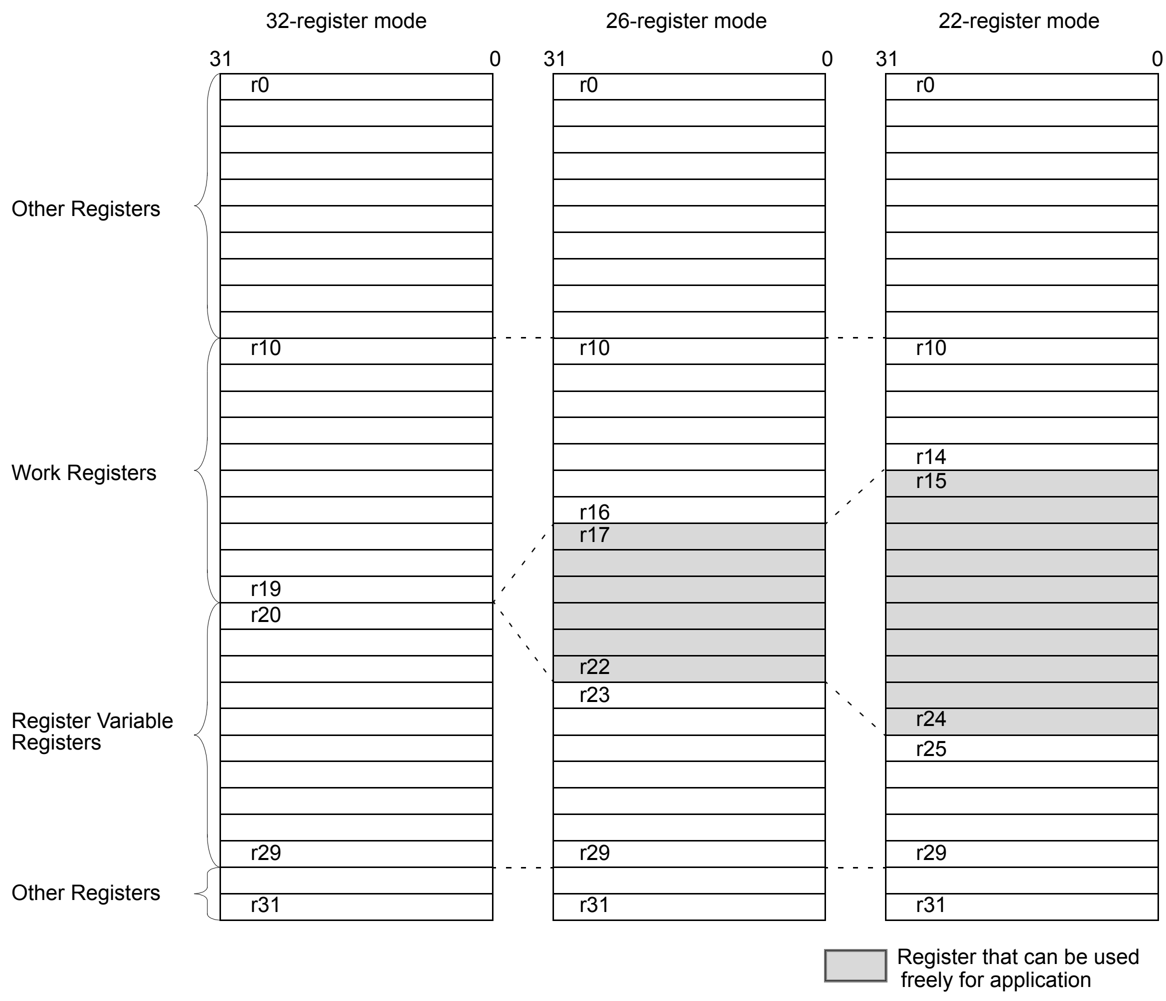
2.4.1 Register modes

Next table and next Figure show the three register modes supplied by the CA850.

Table 2 - 8 Register Modes Supplied by CA850

Mode	Work Registers	Register Variable Registers
32-register mode (default)	r10 - r19	r20 - r29
26-register mode	r10 - r16	r23 - r29
22-register mode	r10 - r14	r25 - r29

Figure 2 - 9 Register Modes and Usable Registers



Specification example on command line

```
> ca850 -cpu 3201 -reg26 file.c -- compiled in 26-register mode
```

2.4.2 Register mode and library

A library supplied by the CA850 (refer to "CHAPTER 6 LIBRARY FUNCTION") is provided for each register mode.

The standard folders that search a library are "*Install Folder*\lib850" and "*Install Folder*\lib850\r32" as the default assumption. If the 22- or 26-register mode is specified by the CA850, however, "*Install Folder* \lib850\r22" or "*Install Folder*\lib850\r26" is used as the standard folder for the library, in the place of "*Install Folder*\lib850\r32".

If ld850 is not started from the CA850 but object files are linked by directly starting ld850 from the command line, however, a library suitable for each register mode must be specified by specifying the -reg option of ld850.

2.5 Mask Register

When byte data or halfword data is loaded from the memory to a register, the V850 microcontrollers sign-extends the data to a word length according to the value of the most significant bit of the data. Therefore, mask codes of the higher bits may be generated during an unsigned char or unsigned short type data (refer to "7.8 Data Type") operation. When storing the result of an operation to a register variable, mask codes are generated to clear the higher bits if the result of the operation is unsigned byte data or unsigned halfword data.

Generation of mask codes can be prevented if word data is used. If word data cannot be used and the mask codes are generated, the code size can be reduced by using the mask register function.

However, to decide whether the mask register function is to be used or not, the following points must be carefully considered for the code where the mask register function may be used.

(1) Whether the program outputs many mask codes

(2) Two register variable registers will not be able to be used because they will be used as mask registers.Will this cause any difficulties

The CA850 uses r20 and r21 as mask registers, as shown in the example below, when the mask register function is used. Note that mask values must be set to the mask registers by program.

Mask code generation example

```
unsigned char UC;
unsigned short US;
void func(void)
{
    register unsigned char ruc;
    register unsigned short rus;
      :
    UC *= UC;
      :
    ruc = UC;
    rus = US;
      :
}
```

```
-- Normal code
    ld.b    $UC, r11
    andi    0xff, r11, r11
    mulh    r11, r11
    st.b    r11, $UC
        :
    ld.b    $UC, r29
    andi    0xff, r29, r29
    ld.h    $US, r28
    andi    0xffff, r28, r28
```

```
-- Code when mask register is used
    ld.b    $UC, r11
    and     r20, r11
    mulh    r11, r11
    st.b    r11, $UC
        :
    ld.b    $UC, r29
    and     r20, r29
    ld.h    $US, r28
    and     r21, r28
```

An instruction that executes "an operation on unsigned data" has been added to the V850Ex and the CA850 outputs a code that uses this instruction. When the V850Ex is used, therefore, setting to use the mask register may not have as much effect as expected.

2.5.1 Setting mask values

Mask values (0xff and 0xffff) must be set to r20 and r21, which are used as mask registers, via the program. The CA850 generates mask codes using the mask registers, assuming that the mask values have been set.

Example of setting of mask value

```
__start :
    mov     #__tp_TEXT, tp
    mov     #__gp_DATA, gp
        :
    mov     0xff, r20           -- Sets mask value to r20
    mov     0xffff, r21         -- Sets mask value to r21
        :
    jarl    _main, lp
```

If the program uses an RTOS, however, the mask values are automatically set according to the RTOS type.

(1) When RX850 is used

Because the mask values are automatically set by the initialization routine of the RX850, they do not have to be set by program.

(2) When RX850 Pro is used

The mask values must be set in advance by using the startup module.

(3) When real-time OS is not used

Set the mask values in advance by using the startup module[Note].

Note The startup module crtN.s (for 32-register mode) supplied with the package sets the mask values (refer to "CHAPTER 5 STARTUP ROUTINE").

2.5.2 Using mask register function

This section describes the specifications for using the mask register functions and points to be noted.

(1) To newly compile C language source file

By specifying the mask register function option (-Xmask_reg) of the CA850, an assembly language source program including the mask codes that use the mask registers and information indicating that the mask register function is used (".option mask_reg" directive) is output.

(2) Checking during linking

Once the link editor has been started by specifying the mask register function option (-Xmask_reg) of the compiler, the object file with the file name information (information specified by the ".file" directive) that indicates that the object file has been created from the .c file is checked while the object file is linked. If an object using the mask register function and an object that does not use the function exist together at this time, an error occurs.

Notes 1 Objects included in an archive file (.a file) are not checked. To use an .a file created by the user, confirm that the mask registers are not used.

2 To start ld850 alone from the command line of the DOS window, an option that performs checking during linking (-mc) must be specified.

(3) When using created assembly language source file

If the program is described in an assembly language from the beginning, check that the contents of the mask registers are not lost. The mask registers are not checked during linking because the file name information is not ".c".

Whether or not the contents of the mask registers are lost can be confirmed by a warning message that is output when the assembler is executed, if the -m option that specifies the use of the mask registers is specified in the assembler.

(4) Supplied library restrictions

Although the object files in the archive file are not checked during linking, almost all the libraries in the package do not destroy the contents of the mask registers. The bsearch function in the standard library, however, may destroy the contents of the mask registers because it calls an application function. Therefore, do not use the bsearch function when the mask register function is used (the compiler does not output an error even if the bsearch function is used)[Note].

Note The bsearch function in the standard library, however, may destroy the contents of the mask registers because it calls an application function. Therefore, do not use the bsearch function when the mask register function is used (The CA850 does not output an error even if the bsearch function is used).

2.6 Device File

A device file is a binary file that contains information dependent upon the device type. One device file is available for each device or group of target devices as a package. The compiler references a device file to generate object codes corresponding to the target system that is used in the application system. Therefore, place the device file to be used under the standard folder for the device file. If the device is placed under any other folder, specify the folder using a compiler option; otherwise an error occurs during compilation because the device file is not found.

2.6.1 Specifying device file

A device file that is referenced by a program in C language can be specified in the following two ways.

(1) Specifying device name using compiler option (-cpu *device-name*)

Example

```
> ca850 -cpu 3201 file.c
```

When building a program with PM+, specifying a device has an effect equivalent to specifying this option.

(2) Specifying device name using #pragma directive (#pragma cpu *device-name*) in C language source file

Example

```
#pragma cpu 3201
```

In this example, the device name is "3201" (V850ES/SA2).

The character strings that can be specified as "device name" are common to option specification and the #pragma directive. Uppercase and lowercase characters are not distinguished.

For the character strings that can be specified as a device name, refer to the Architecture User's Manual of each device.

Notes 1 When specifying a device name using the #pragma directive, device specification must be described in all source files.

2 Specify a device name at the beginning of a source file when using the #pragma directive. Only preprocessing that has nothing to do with C language syntax and comments can be described before specification of the device name. If a device name is specified in C language syntax, the compiler outputs the following error message and stops processing.

```
F2625: illegal placement ' #pragma cpu '
```

Example of incorrect specification

```
#include <stdio.h>
int     i;
#pragma cpu 3201
    :
```

2.6.2 Notes on specifying device file

(1) If no device name is specified

If a device file is specified by neither the #pragma directive nor the -cpu option, and if neither the -cn option nor the -cnv850e option[Note] is specified, the compiler outputs the following error message and stops compiling.

```
F2620: unknown cpu type, cannot compile
```

Note A device file is necessary during linking even if the -cn or -cnv850e option is specified.

(2) If device is specified by both option and #pragma directive

The compiler outputs a warning message and continues processing, giving priority to the option. If different device names are specified by two or more options or #pragma directives, the compiler outputs the following message and stops processing.

```
F2622: duplicated cpu type
```

(3) Program described in assembler instructions

In this case also, a device must be specified by an assembler option or the .option quasi directive when an object file that can be linked is created.

CHAPTER 3 C LANGUAGE EXPANSION

This chapter explains the language specifications expanded by the CA850.

The expanded specifications include how to specify section location of data and access the internal peripheral function registers of the device, how to insert assembler code in a C language source program, how to specify inline expansion for each function, how to define a handler when an interrupt or exception occurs, how to disable interrupts at the C language level, the valid RTOS functions when a real-time OS is used for the target environment, and how to embed instructions in a C language source program.

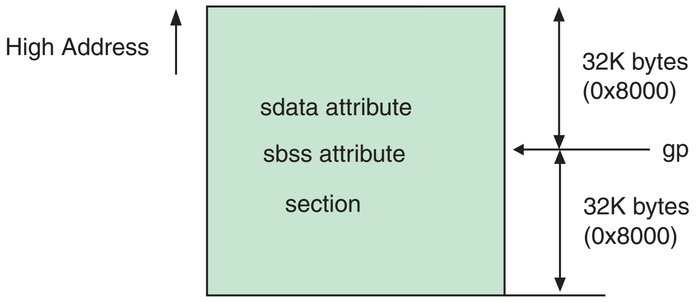
3.1 Allocation of Data to Section

When external variables and data are defined in a source, the CA850 allocates them to memory.

The memory location to which the variables and data are allocated is, basically, an area that can be referenced by an offset from the address pointed to by the global pointer (gp). If the variables or data are accessed in the program, therefore, the CA850 attempts to output a code that accesses the area using gp, by default.

At this time, the CA850 attempts to output a code that allocates data to an area that can be referenced from gp by one instruction, in order to enhance the object efficiency and execution efficiency as much as possible. Since the range that can be referenced by one instruction from gp must be within ±32 K bytes (a total of 64 K bytes) from gp according to the V850 architecture, the CA850 has dedicated sections in the ±32 K bytes area from gp. These sections are called the sdata and sbss attribute sections.

Figure 3 - 1 sdata and sbss Attribute Sections



In many cases, however, variables exceed in this range when using an application that uses many variables. In this case, the variables must be allocated to other sections. The CA850 has many sections to which variables and data can be allocated, in addition to the sdata and sbss attribute sections. Each of these sections has its own feature and sections to which variables that must be accessed quickly can be allocated are also available, so that the sections can be selected depending on the usage. The sections that can be used in the CA850 are explained below.

(1) sdata and sbss attribute sections

These sections can be referenced from gp with one instruction and must be allocated within ±32 K bytes from gp.

Data with initial values is allocated to the sdata attribute section, and data without initial values is allocated to the sbss attribute section.

The CA850 first attempts to generate a code that is to be allocated to these sections. If the code exceeds the upper limit of the section of these attributes, the compiler generates a code that allocates data to a data or bss attribute section.

To increase the amount of data to be allocated to the sdata or sbss attribute sections, the upper size limit for the data to be allocated can be specified with the "-G" option of the CA850 so that data in excess of this upper limit is not allocated to the sdata or sbss attribute sections (refer to CA850 for Operation User’s Manual for details of this option).

Use the #pragma section directive to specify a variable to be allocated to the sdata or sbss attribute section in the program (refer to "3.1.1 #pragma section directive" for details).

```
#pragma section sdata begin
int     a=1;        /* allocated to sdata attribute section */
int     b;          /* allocated to sbss attribute section */
#pragma section sdata end
```

(2)   data and bss attribute sections

These sections can be referenced from gp with two instructions. Since access is performed after address generation, the code becomes correspondingly longer and the execution speed also drops, but the entire 32-bit space can be accessed. In other words, these sections can be allocated anywhere as long as they are in RAM.

Use the #pragma section directive to specify a variable to be allocated to the data or bss attribute section in the program (refer to "3.1.1 #pragma section directive" for details).

```
#pragma section data begin
int     a=1;        /* allocated to data attribute section */
int     b;          /* allocated to bss attribute section */
#pragma section data end
```

(3)   sconst attribute section

This is a section that can be referenced from r0, in other words from address 0, with 1 instruction, and must be allocated within +32 bytes from address 0. Basically, data that can be fixed to ROM is allocated to this section.

In the case of V850 devices with internal ROM, in many cases the internal ROM is assigned from address 0, and data that should be referenced with 1 instruction and that can be fixed to ROM is allocated to the sconst attribute section. Variables/data declared by adding the const qualifier are subject to allocation to the sconst attribute section. If the data exceeds the upper limit of this attribute section, it is allocated to the const attribute section.

To increase the amount of data to be allocated to the sconst attribute section, the upper size limit for the data to be allocated can be specified with the "-Xsconst" option of the CA850 so that data in excess of this upper limit is not allocated to the sconst attribute section (refer to CA850 for Operation User’s Manual for details of this option).

Use the #pragma section directive to specify a variable to be allocated to the sconst attribute section in the program (refer to "3.1.1 #pragma section directive" for details).

```
#pragma section sconst begin
const int a=1;              /* allocated to sconst attribute section */
#pragma section sconst end
```

(4) const attribute section

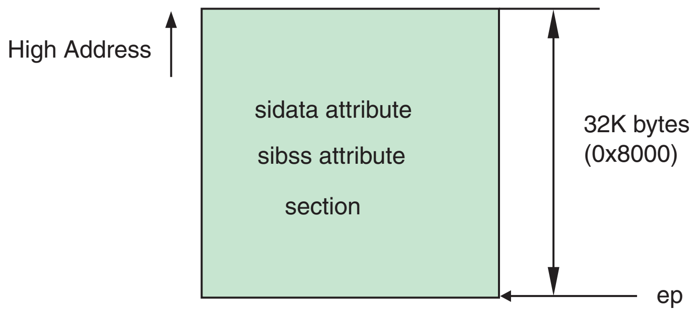
This is a section that can be referenced from r0, in other words from address 0, with two instructions. Data that can be fixed to ROM that exceeds the upper limit of the sconst attribute section, or data that should be allocated to external ROM in the case of ROMless devices of the V850 microcontrollers, is allocated to the const attribute section. Variables/data declared by adding the const qualifier are subject to allocation to the const attribute section. The variables declared by adding the const qualifier are allocated to the const attribute section, string literal even if allocation to the .const section is not specified by the #pragma section directive. Since access is performed after address generation, the code becomes correspondingly longer and the execution speed also drops, but the entire 32-bit space can be accessed. In other words, the const attribute section can be allocated anywhere as long as it is in the 32-bit space. Use the #pragma section directive to specify a variable to be allocated to the const attribute section in the program (refer to "3.1.1 #pragma section directive" for details).

```
#pragma section const begin
const int a=1;          /* allocated to const attribute section */
#pragma section const end
```

(5) sidata and sibss attribute sections

These sections can be referenced from ep (element pointer) with 1 instruction toward higher addresses. In other words, these sections are allocated in the 32 K bytes space toward higher addresses from ep.

Figure 3 - 2 sidata and sibss Sections



Data with initial values is allocated to the tidata attribute section, and data without initial values is allocated to the tibss attribute section. If variables that exceed the upper limit of the sdata and sbss attribute sections that can be accessed from gp with 1 instruction, but which need to be accessed with 1 instruction still exist, they can be allocated in the range that can be accessed with 1 instruction using ep.The sidata and sibss attribute sections are sections for access toward higher addresses from ep; the sedata and sebss attribute sections are sections for access toward lower addresses from ep.

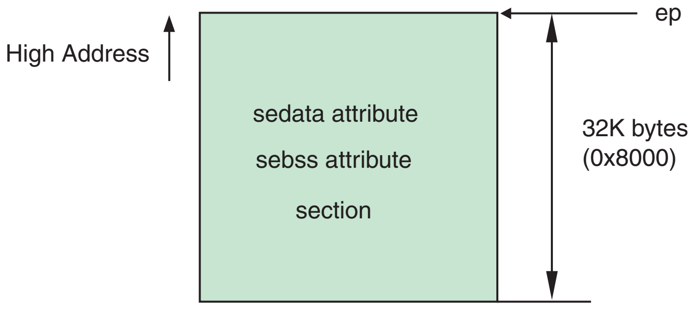
Use the #pragma section directive to specify a variable to be allocated to the sidata or sibss attribute section in the program (refer to "3.1.1 #pragma section directive" for details).

```
#pragma section sidata begin
int     a=1;        /* allocated to sidata attribute section */
int     b;          /* allocated to sibss attribute section */
#pragma section sidata end
```

(6) sedata and sebss attribute sections

These sections can be referenced from ep (element pointer) with 1 instruction toward lower addresses. In other words, these sections are allocated within 64 K bytes toward lower addresses from ep.

Figure 3 - 3 sedata and sebss Sections



Data with initial values is allocated to the sedata attribute section, and data without initial values is allocated to the sebss attribute section. If variables that exceed the upper limit of the sdata and sbss attribute sections that can be accessed from gp with 1 instruction, but which need to be accessed with 1 instruction still exist, they can be allocated in the range that can be accessed with 1 instruction using ep. The sidata and sibss attribute sections are sections for access toward higher addresses from ep; the sedata and sebss attribute sections are sections for access toward lower addresses from ep.

Use the #pragma section directive to specify a variable to be allocated to the sedata or sebss attribute section in the program (refer to "3.1.1 #pragma section directive" for details).

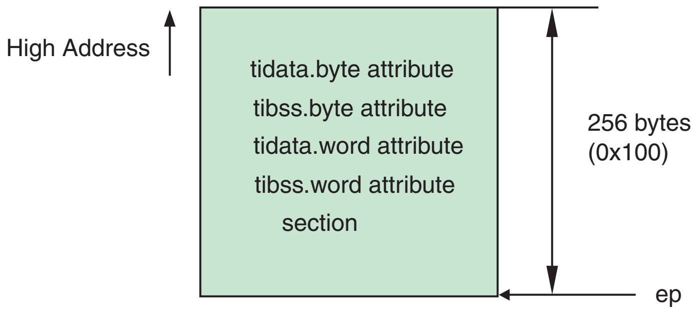
```
#pragma section sedata begin
int     a=1;        /* allocated to sedata attribute section */
int     b;          /* allocated to sebss attribute section */
#pragma section sedata end
```

(7) tidata (tidata.byte, tidata.word) and tibss (tibss.byte, tibss.word) attribute sections

These sections can be referenced from ep (element pointer) with 1 instruction toward higher addresses. These sections are accessed with 1 instruction in the same manner as the sidata and sibss attribute sections, but differ in terms of the assembler instruction to be used.

The sidata and sibss attribute sections use the 4-byte st/ld instruction for store/reference, whereas the tidata and tibss attribute sections use the 2-byte sst/sld instruction to perform access. In other words, the code efficiency of the tidata and tibss attribute sections is better than that of the sidata and sibss attribute sections. However, the range in which sst/sld instructions can be applied is small, so it is not possible to allocate a large number of variables.

Figure 3 - 4 tidata and tibss Sections



Data with initial values is allocated to the tidata (tidata.byte, tidata.word) attribute section, and data without initial values is allocated to the tibss (tibss.byte, tibss.word) attribute section. Specify the tidata.byte/tibss.byte attribute to allocate byte data, and specify the tidata.word/tibss.word attribute to allocate word data. To select automatic byte/word judgment by the CA850, specify the tidata/tibss attribute.

The tidata and tibss attribute sections are used to allocate data that must be accessed the fastest in the system. However, the data to be allocated to these sections must be carefully selected because the quantity of data that can be allocated to these sections is limited. Use the #pragma section directive to specify variables to be allocated to the tidata.byte/tibss.byte or tidata.word/tibss.word attribute section in the program (refer to "3.1.1 #pragma section directive" for details).

```
#pragma section tidata_byte begin
char            a=1;    /* allocated to tidata.byte attribute section */
unsigned char   b;      /* allocated to tibss.byte attribute section */
#pragma section tidata_byte end
```

```
#pragma section tidata_word begin
int     a=1;    /* allocated to tidata.word attribute section */
short   b;      /* allocated to tibss.word attribute section */
#pragma section tidata_word end
```

```
#pragma section tidata begin
int     a=1;    /* allocated to tidata.word attribute section */
char    b;      /* allocated to tibss.byte attribute section */
#pragma section tidata end
```

The efficiency can be enhanced in terms of execution speed if variables or data that are especially frequently used in the system are selected and allocated to the tidata (tidata.byte, tidata.word) or tibss (tibss.byte or tibss.word) attribute section.

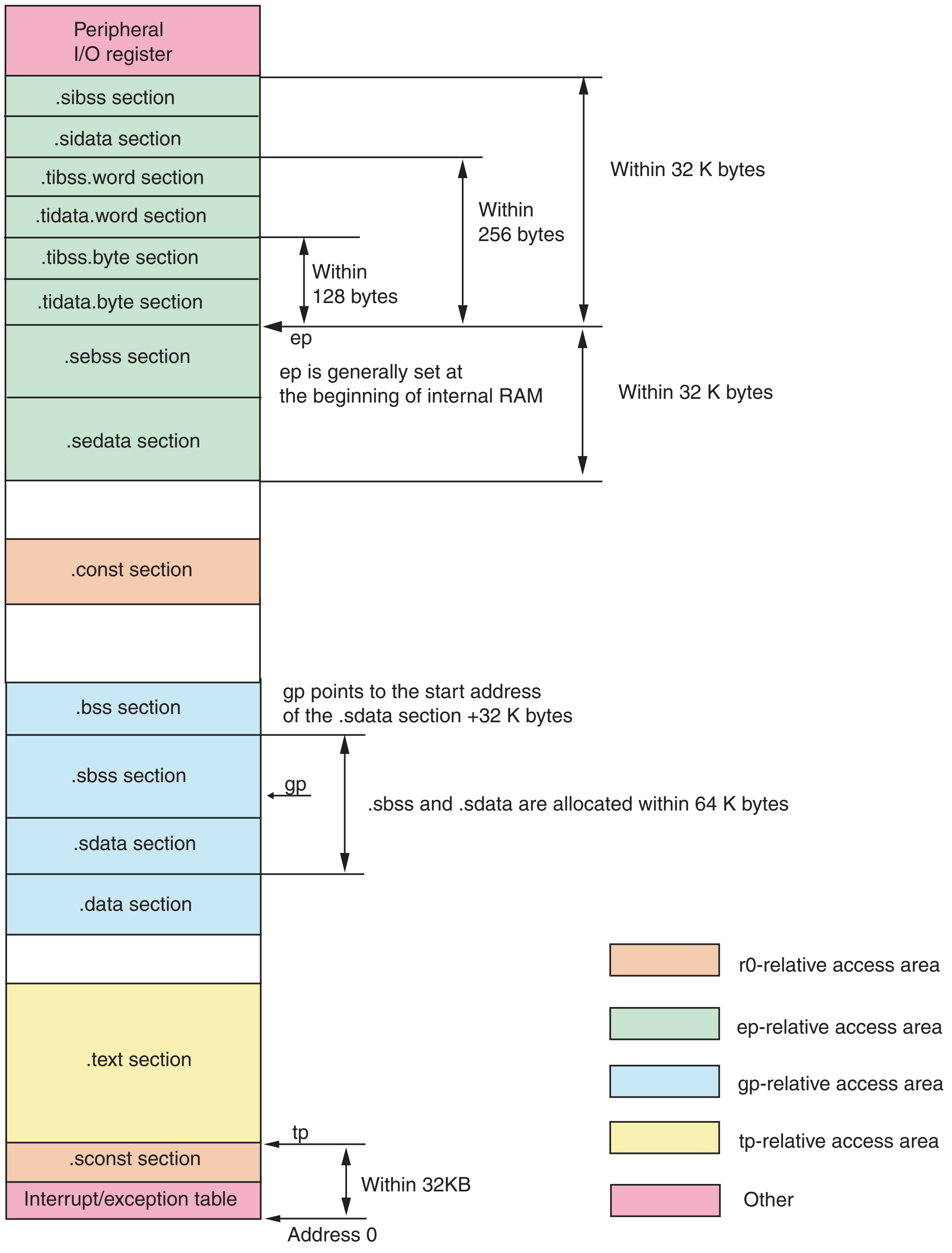
The CA850 has a section file generator that investigates the frequency of reference.

The frequency information obtained as a result of the investigation is output as a frequency information file. The code that allocates data to the tidata (tidata.byte, tidata.word) or tibss (tibss.byte, tibss.word) attribute section is output based on this information. The user can edit the frequency information file to select variables that should be allocated to the tidata (tidata.byte, tidata.word) or tibss (tibss.byte, tibss.word) attribute section by priority. The variables can then be allocated to these sections without qualifying the source.

Refer to CA850 for Operation User's Manual for details of the section file generator and frequency information file.

Figure 3 - 5 shows an example of memory allocation of each section.

Figure 3 - 5 Image of Memory Allocation of Each Section



3.1.1 #pragma section directive

How to allocate data to a section using the #pragma section directive is explained below.

(1) To use default section name as is

Describe the #pragma section directive in the following format when using the section name defined by the CA850 as is.

```
#pragma section section-type begin
variable-declaration/definition
#pragma section section-type end
```

The following can be specified as the section-type.

- data
- sdata
- sedata
- sidata
- tidata
- tidata.word
- tidata.byte
- sconst
- const

The name of the bss attribute section must not be specified as the section type. The CA850 automatically allocates declared or defined data with initial values to the data attribute section, and data without initial values to the bss attribute section.

```
#pragma section sdata begin
int     a=1;        /* allocated to sdata attribute section */
int     b;          /* allocated to sbss attribute section */
#pragma section sdata end
```

In the above case, "variable a" is allocated to the data-attribute .sdata section because it has an initial value, and "variable b" is allocated to the sbss-attribute sbss section because it does not have an initial value.

Two or more variable declarations or definitions can be described between "#pragma section *section-type* begin" and "#pragma section *section-type* end". Enumerate variables to be allocated for each section type.

Use "_" (underscore) instead of "." (period) to specify tidata.word or tidata.byte as the section type, as shown below.

- tidata_word
- tidata_byte

(2) To assign original section name

The user can assign a specific name to the sections with the following attributes, and can allocate variables and data to those sections.

- data
- sdata
- sconst
- const

In this case, describe the #pragma section directive in the following format.

```
#pragma section section-type "created-section-name" begin
Variable declaration/definition
#pragma section section-type "created-section-name" end
```

However, "*.section-type*" is appended to a section name actually generated by this method as follows.

```
created-section-name.section-type
```

This is to prevent a section with another attribute and having the same name from being created because the section attribute is classified into data or bss depending on whether the data has an initial value or not. Specify a generated section name when specifying a section in a link directive file. Refer to "3.1.2 Specifying link directive of specific data section" for an example of specification in a link directive file.

The following table shows specific examples of section names specified by the user and generated section names.

Table 3 - 1 Section Names Specified by User and Generated Section Names

Section Name Specified by User	Section Type	Character String Appended	Generated Section Name
mydata	data attribute	.data/.bss	mydata.data/mydata.bss
mysdata	sdata attribute	.sdata/.sbss	mysdata.sdata/mysdata.sbss
myconst	const attribute	.const	myconst.const
mysconst	sconst attribute	.sconst	mysconst.sconst

If the name is specified as follows, "variable a" is allocated to the mysdata.sdata section because it has an initial value, and "variable b" is allocated to the mysdata.sbss section because it does not have an initial value.

```
#pragma section sdata "mysdata" begin
int     a=1;        /* allocated to mysdata.sdata attribute section */
int     b;          /* allocated to mysdata.sbss attribute section */
#pragma section sdata "mysdata" end
```

3.1.2 Specifying link directive of specific data section

When a specific section is created using the #pragma section directive, describe that section in a link directive file as explained below.

If "variable a" and "variable b" are specified as follows in a C language source, "variable a" is allocated to the mysdata.sdata section because it has an initial value, and "variable b" is allocated to the mysdata.sbss section because it does not have an initial value.

```
#pragma section sdata "mysdata" begin
int     a=1;        /* allocated to mysdata.sdata attribute section */
int     b;          /* allocated to mysdata.sbss attribute section */
#pragma section sdata "mysdata" end
```

At this time, the variable can be allocated to a specific section if the mapping directive in the link directive file is described as follows.

```
.data = $PROGBITS ?AW .data;
.bss = $NOBITS ?AW .bss;

mysdata.data = $PROGBITS ?AW mysdata.data;
mysdata.bss = $NOBITS ?AW mysdata.bss;
```

Since the variables are allocated in the order in which they are described, change the description order to change the allocation order. It is also possible to specify the start address of the section directly (generally, a segment is created first and a mapping directive, which specifies the start address of a section in segment units, is then described in that segment).

Because the attribute of mysdata.data is "$PROGBITS?AW" and that of mysdata.bss is "$NOBITS?AW", do not omit the input section (".data", ".bss", "mysdata.data", and "mysdata.bss" on the rightmost side of the mapping directive in the above example) from mapping directives that have the same attribute as these.

Example

```
.data = $PROGBITS ?AW;
.bss = $NOBITS ?AW;
```

If an input section is omitted from a mapping directive having the same "$PROGBITS?AW" or "$NOBITS?AW" attribute, the linker links and locates all the sections having that attribute. Consequently, data is not allocated to the specific section created by the user. This means that the data that should be allocated to the mysdata.data section is allocated to the .data section, and the data that should be allocated to the mysdata.bss section is allocated to the .bss section.

Refer to CA850 for Link Directive User’s Manual for details of the format of the link directive file.

3.1.3 Notes on section allocation

Notes below must be noted when sections are allocated by the #progma section directive, the const qualifier, or the section file.

(1) An error occurs during compilation if the #pragma section directive is specified as follows.

- Section allocation is nested.
- begin and end of #pragma section cross.
- Either begin or end of #pragma section is missing.

Example of incorrect specification: "Nesting of sections"

```
#pragma section data begin
int     a=1

#pragma section sdata begin
short   b;
char    c=0x10
#pragma section sdata end

int     d;
#pragma section data end
```

Example of incorrect specification: "Crossing sections"

```
#pragma section data begin
int     a=1

#pragma section sdata begin
short   b;
char    c=0x10
#pragma section data end

int     d;
#pragma section sdata end
```

(2) If a section is specified for an automatic variable, the specification is ignored. Section specification is a function for external variables.

(3) When specifying a specific section name, keep the length of the name to within 256 characters.

(4) A variable declaration that is not set with an initial value is usually treated as a tentative definition. When a section is specified, however, it is treated as a "definition". Do not allow variable declarations which do not have their initial values set to get mixed in with definitions.

`/* Variable declaration not using`
`   #pragma section */`
 	
`int i;      /* tentative definition */`
`int i=10;   /* definition */`
 	
`/* Error does not occur. */`
	`/* Variable declaration using`
	`   #pragma section */`
	
	`#pragma section sedata begin`
	`int     i;      /* definition */`
	`int     i=10;   /* definition */`
	`#pragma section sedata end`
	
	`/* Duplicated definition error */`

If a section is specified for the tentative definition of an ordinary external variable, it is treated as a "definition". Be sure to make extern declaration in files that reference an external variable. In the example below, a duplicated definition error occurs if extern is missing in the tentative definition of the variable in file1.c.

`[file1.c]`
`#pragma section sedata begin`
`extern int  i;`
`#pragma section sedata end`
	`[file2.c]`
	`#pragma section sedata begin`
	`int     i;`
	`#pragma section sedata end`

[Duplicated definition error occurs if extern is not declared]	

(5) When a variable specified by a section is referenced by another file, the section must be specified with the same section type for the extern declaration of that variable. An error occurs if a type of section different from that of the section specified when a variable is defined is specified.

For example, if "#pragma section data begin - #pragma section data end" is specified on the definition side and "#pragma section data begin - #pragma section data end" is not specified on the tentative definition side (extern declaration), it is assumed on the tentative definition side that the variable is allocated to the sdata section. This means that a code that accesses the variable from gp with two instructions is output on the definition side and that a code that accesses the variable from gp with one instruction is output on the tentative definition side. In other words, a contradiction occurs. Consequently, the following error message is output during linking.

```
F4163: output section ".data" overflowed or illegal label reference for symbol
"symbol" in file "file" (value: value, input section: section, offset: offset,
type:R_V850_GPHWLO_1). "symbol" is allocated in section ".data" (file: file).
```

Example of correct specification

```
[file1.c]
#pragma section sedata begin
int     i=1;
#pragma section sedata end
```

```
[file2.c]
#pragma section sedata begin
extern int  i;
#pragma section sedata end
```

Example of incorrect specification 1

```
[file1.c]
int     i=1;
```

```
[file2.c]
#pragma section sedata begin
extern int  i;
#pragma section sedata end
```

"variable i" defined by file1.c is allocated to the sbss or bss attribute section. However, file2.c outputs a code that accesses the sebss attribute section for "variable i". As a result, the linker outputs the following error message.

```
F4163: output section ".sebss" overflowed or illegal label reference forsymbol
"_i" in file "file2.o" (value: value, input section: section, offset: offset,
type: type). "_i" isallocated in section ".sbss" (file: file1.o).
```

Example of incorrect specification 2

[file1.c]	[file2.c]
#pragma section sedata begin	extern int i;
int i=1;	
#pragma section sedata end	

f"variable i" defined by file1.c is allocated to the sebss attribute section but file2.c outputs a code that accesses the sbss attribute section or bss attribute section to "variable i". Consequently, the linker outputs the following mismatch error.

```
F4156: can not find GP-symbol in segment "*DUMMY*" or illegal labelreference
for symbol "_i" in file "file2.o" (section: section, offset: offset,
type:R_V850_GPHWLO_1). "_i" is allocated in section ".sedata" (file: file1.o).
```

(6) When defining a variable with the sconst or const attribute using the #pragma section directive, be sure to make a const specification for the variable. A const specification is also necessary at the location of the tentative definition made by extern declaration.

If the const declaration is missing when a variable is declared, the variable is not allocated to the sconst section or const section (the #pragma section directive is ignored) even if "#pragma section sconst begin - #pragma section sconst end" or "#pragma section const begin - #pragma section const end" is specified, but to a gp-relative section such as the sdata section or data section. In other words, allocation is not performed as intended.

[file1.c]	[file2.c]
#pragma section sconst begin	#pragma section sconst begin
const int i=1;	int i;
#pragma section sconst end	#pragma section sconst end

A code that allocates "variable i" to the sconst section is output in file1.c. In file2.c, however, the #pragma section specification is ignored because the const specification is missing from "variable i", and therefore the variable is treated as a gp-relative variable. In other words, a code that allocates the variable to the sdata or data section is output. Consequently, "variable i" is not allocated to the sconst section during linking. A const specification is also necessary at the location of the tentative definition with extern declaration, as shown below.

[file1.c]	[file2.c]
#pragma section sconst begin	#pragma section sconst begin
const int i=10;	extern const int i;
#pragma section sconst end	#pragma section sconst end

3.1.4 Example of #pragma section directive

Here are some examples of using the #pragma section directive.

(1) Allocating "variable a" to tidata.word section and "variable b" to tibss.word section

```
#pragma section tidata_word begin
int     a=1;        /* allocated to tidata.word attribute section */
short   b;          /* allocated to tibss.word attribute section */
#pragma section tidata_word end
```

(2) Allocating "variable c" to tidata.byte section and "variable d" to tibss.byte section

```
#pragma section tidata_byte begin
char    c=0x10;     /* allocated to tidata.byte attribute section */
char    d;          /* allocated to tibss.byte attribute section */
#pragma section tidata_byte end
```

In the tidata section, word data or halfword data is allocated to the tidata_word or tibss_word section, and byte data is allocated to the tidata_byte or tibss_byte section. If char-type arrays are declared in the C language source, however, they are allocated to the tidata.word section. The tidata.word section can be used up to 256 bytes. Because the arrays are of char type, a code using sld.b or sst.b is output. However, the sld.b and sst.b instructions cannot access more than 128 bytes. Therefore, if a char-type array is declared and if the array itself is of more than 128 bytes or is located at a place exceeding 128 bytes relatively from ep, an error occurs during linking. Take this point into consideration when allocating char-type arrays to the tidata section.

(3) Allocating "variable e" specified by const to the sconst section and character string constant data indicated by pointer p to sconst section

```
#pragma section sconst begin
const int   e=0*10;
const char  *p="Hello World";
#pragma section sconst end
```

In the above description, "Hello World" indicated by pointer p is allocated to the sconst section, and pointer variable "p" itself is allocated to the sdata section or data section. The allocation position of the pointer variable and the contents indicated by the pointer vary depending on how const is specified.

Example 1

```
const char  *p="Hello World";
```

If this declaration is made, the pointer variable and character sting constant indicated by the pointer are allocated as follows.

Pointer variable "p"	Can be rewritten ("p = 0" can be compiled).
Character string constant "Hello World"	Cannot be rewritten ("*p = 0" cannot be compiled).

Describe as shown below to allocate what the pointer variable indicates to a section with the const attribute. This description is used when the pointer itself is fixed to ROM.

```
#pragma section sconst begin
const char  *p="Hello World";
#pragma section sconst end
```

With the above definition, the pointer and character string constant are allocated to the following sections.

Pointer variable "p"	sdata/data section
Character string constant "Hello World"	sconst section

Example 2

```
char    *const p;
```

Pointer variable "p"	Cannot be rewritten ("p = 0" cannot be compiled).

Describe as shown below to allocate the pointer variable to a section with the const attribute. This is used to fix the pointer itself to ROM.

```
char *const p="Hello World";
```

The above description allocates both the pointer variable and character string constant "Hello World" to a section with the const attribute.

```
#pragma section sconst begin
char *const p="Hello World";
#pragma section sconst end
```

The above definition allocates the pointer variable and constant to the following sections.

Pointer variable "p"	sconst section
Character string constant "Hello World"	sconst section

Example 3

```
const char  *const p;
```

Pointer variable "p"	Cannot be rewritten ("p = 0" cannot be compiled).

Describe as shown above to allocate the pointer variable and the destination it indicates to a section with the const attribute. Both the above descriptions are used to fix the pointer to ROM.

```
const char *const   p="Hello World";
```

The above description allocates both the pointer variable and character string constant "Hello World" to a section with the const attribute.

```
#pragma section sconst begin
const char *const   p="Hello World";
#pragma section sconst end
```

The above definition allocates the pointer variable and constant to the following sections.

Pointer variable "p"	sconst section
Character string constant "Hello World"	sconst section

In addition to the #pragma section directive, the compiler option "-Xconst" can be used to allocate a variable specified by const to the sconst section.

(4) Make the extern declaration of the #pragma section directive in a commonly used header file and include it in the C language source.

```
[header.h]
#pragma section sidata begin
extern int  k;
#pragma section sidata end
```

```
[file1.c]
#include "header.h"
#pragma section sidata begin
int     k;
#pragma section sidata end
```

```
[file2.c]
#include "header.h"
void func1(void)
{
    k = 0x10;
}
```

If the extern declaration of the #pragma section directive is made in a header file as shown above, the errors decrease and the source is visually simplified.

3.2 Allocating Functions to Sections

The CA850 allocates the functions of a C language source program, i.e., program codes, to the .text section by default. When the .text section allocation address is specified in the link directive file, the program is allocated from that address. However, it may be necessary to change the allocation address for each function or distribute the allocation address because of the layout of the memory. To satisfy these necessities, the CA850 has the #pragma text directive. Using this directive, any name can be given to a section with the text attribute, and the allocation address can be changed in the link directive file.

3.2.1 #pragma text directive

Using the #pragma text directive, any name can be given to a section with the text attribute. The #pragma text directive can be used in the following two ways.

- Specifying the function name to be allocated to a section to be created using the #pragma text directive
- Describing the #pragma text directive before the main body of a function (function definition) but not specifying a function name

(1) Specifying the function name to be allocated to a section to be created using the #pragma text directive

```
#pragma text "created section name" function-name
```

Describe functions that are described in the C language. In the case of a function, "void func1() {}", specify "func1". The created section name can be omitted. In this case, a function specified by "function name" is allocated to the default .text section.

(2) Describing the #pragma text directive before the main body of a function (function definition) but not specifying a function name

```
#pragma text "created section name"
```

The created section name can be omitted. In this case, specification of the name of section to be created by "#pragma text" specified immediately before is canceled, and the subsequent functions are allocated to the default .text section. However, ".text" is appended to a section name actually generated by this method as follows.

```
section-name.text
```

Specify the generated section name when specifying a section in a link directive file. Refer to "3.2.2 Specifying link directive of specific text section" for an example of specifying in a link directive file.

The following table shows specific examples of section names specified by the user and generated section names.

Table 3 - 2 Section Names Specified by User and Generated Section Names (text)

Section Name Specified by User	Section Type	Character String Appended	Generated Section Name
mytext	text attribute	.text	mytext.text

If the name is specified as follows, "func1" is allocated to the mytext1.text section, and "func2" is allocated to the .text section by default, because the #pragma text directive is not used.

```
#pragma text "mytext1" func1
void func1(void)
{
    :
}
void func2(void)
{
    :
}
```

If the name is specified as follows, "func1" and "func2" are allocated to the mytext2.text section, "func3" to the "mytext3.text section", and "func4" to the default .text section because the #pragma text "mytext3" immediately before is canceled.

```
#pragma text "mytext2"
void func1(void)
{
    :
}
void func2(void)
{
    :
}
#pragma text "mytext3"
void func3(void)
{
    :
}
#pragma text
void func4(void)
{
    :
}
```

3.2.2 Specifying link directive of specific text section

When a specific section is created using the #pragma section directive, describe that section in a link directive file as explained below.

```
#pragma text "mytext2"
void func1(void)
{
    :
}
void func2(void)
{
    :
}
#pragma text "mytext3"
void func3(void)
{
    :
}
#pragma text
void func4(void)
{
    :
}
```

If the #pragma text directive is specified in a C language source as shown above, "func1" and "func2" are allocated to the mytext2.text section, "func3" to the mytext3.text section, and "func4" to the default .text section because the #pragma text "mytext3" immediately before is canceled.

```
.text    = $PROGBITS  ?AX .text;
mytext2 = $PROGBITS  ?AX mytext2.text;
mytext3 = $PROGBITS  ?AX mytext3.text;
```

Since the functions are allocated in the order in which they are described, change the description order to change the allocation order. It is also possible to specify the start address of the function directly (generally, a segment is created first and a mapping directive, which specifies the start address of a function in segment units, is then described in that segment).

Because the attribute of mytext2.text and mytext3.text is "$PROGBITS ?AW", do not omit the input section (".text", "mytext2.text", and "mytext3.text" on the rightmost side of the mapping directive in the above example) from mapping directives that have the same attribute as these.

Example

```
.text   = $PROGBITS ?AX;
```

If an input section is omitted from a mapping directive having the same "$PROGBITS ?AX" attribute, the linker links and locates all the sections having that attribute. Consequently, data is not allocated to the specific section created by the user. This means that the program that should be allocated to the mytext2.text or mytext3.text section is allocated to the .text section.

Refer to CA850 for Link Directive User's Manual for details of the format of the link directive file.

3.2.3 Notes on #pragma text directive

Note the following points when using the #pragma text directive.

(1) Describe the #pragma text directive before the function definition in the same file; otherwise a warning message is output and the directive is ignored. However, the order of prototype declaration of a function is not affected.

(2) If a function specified by the #pragma text directive is an interrupt handler specified as direct allocation, a warning message is output and the #pragma text directive is ignored. Refer to "3.7 Interrupt/Exception Processing Handler" for details of direct allocation specification.

(3) A function specified by #pragma text cannot be expanded inline by a #pragma inline specification or an optimization option. Inline expansion specification is ignored.

(4) When specifying a section name, keep the length of the name to within 256 characters.

3.3 Peripheral I/O Register

Peripheral I/O registers are used to control the internal peripheral functions of a device.

By using the peripheral I/O register name defined by the device, the internal I/O can be accessed at C language level. The peripheral I/O register names can be treated in the C language source program as if they were normal unsigned external variables.

For the register names and attributes that can be specified, refer to the Relevant Device’s Hardware User’s Manual of each device.

3.3.1 Accessing

A peripheral function register name is validated by describing the following pragma directive.

```
#pragma ioreg
```

In a C language source file in which "#pragma ioreg" directive is described, the peripheral function register name described after the pragma directive can be used.

If this directive is not used or if a peripheral function register name is used without specifying an applicable device name, an "undefined variable" error occurs. An error also occurs if the access attribute peculiar to the specified register is violated.

Of the examples as follows, Example 1 is correct, but Examples 2 and 3 cause an error.

P0, P1, P2, RXB0, and OVF0 in the following examples indicate the peripheral I/O registers of the V850 microcontrollers.In this way, symbols defined by the device file are specified as "register names".

Next shows specification examples.

Example 1

```
#pragma ioreg
void func1(void)
{
    int     i;
    P0 = 1;         /* Writes to P0 */
    i = RXB0;       /* Reads from RXB0 */
}

void func2(void)
{
    P1 = 0;         /* Writes to P1 */
}
```

Example 2

```
void func(void)
{
    P1 = 0;         /* Undefined error */
}
```

Example 3

```
#pragma ioreg
void functorial)
{
    RXB0 = 1;   /* Error that occurs if attribute of RXB0 is read-only */
}
```

3.3.2 Bit access

The CA850 can access each bit of a peripheral function register.

"bit number" is specified as 0 to 31 in the case of a 32-bit register.

```
(register name).(bit number) = ...
```

(1) Cautions of case of bit access

(a) If a value other than 0 or 1 is substituted in accessing a bit, the binary least significant value of that value is set (In this case, no message is output.).

Specification examples 1

```
#pragma ioreg
void func(void)
{
    P0.1 = 1;       /* Sets bit 1 of P0 to 1 */
    P2.3 = 0;       /* Resets bit 3 of P2 to 0 */
}
```

(b) The bits of the flag of each register can be accessed by using a bit name.Specify a name defined by the device file as the bit name

Specification examples 2

```
#pragma ioreg
void func(void)
{
    OVF0 = 1;       /* Sets bit name OVF0 to 1 */
}
```

3.4 Describing Assembler Instruction

With the CA850, assembler instruction can be described in the functions of a C language source program in the following format.

- asm declaration
- #pragma directive

To use registers with an inserted assembler, save or restore the contents of the registers in the program because they are not saved or restored by the CA850.

It is advisable to insert assembler in a function. It the instructions are described outside a function, the following restrictions apply and a warning message is output.

- The output sequence of the function and code is not guaranteed.
- The code is not output in a file where the function does not exist.

(1) asm declaration

```
__asm(character string constant); or _asm(character string constant);
```

[Cautions]

(a) The _asm format is provided to maintain compatibility with the conventional language specifications. If the -ansi option is specified, the compiler outputs a warning message to the _asm format and treats the option as a function call. When specifying the -ansi option, use the __asm format.

(b) If the asm declaration is specified, the compiler suffixes a new-line character (\n) to the specified character string constant[Note] and passes it to the assembler.

Note The specified character string constant is unlike the normal character string constant, "\" followed by a character other than a new line indicates the following character itself ("\" followed by a new line causes an error).

Example

```
__asm("nop ");
__asm(".str \"string\\0\"");
```

(c) __asm or __asm is a declaration and is not treated as a statement. Therefore, because of the syntax of the C language source, an error occurs in a structure that does not allow the use of a declaration only, as shown in Example 1 below. Therefor, enclose the statement in "{ }" as shown in Example 2 to make it a compound statement.

Example 1

```
if(i == 0)
    __asm("mov r11, r10"); /* Error occurs because only declaration is made. */
```

Example 2

```
if(i == 0){
    __asm("mov r11, r10"); /* Can be used because this is compound statement. */
}
```

(2) #pragma directive

In the range enclosed by the above #pragma directives, assembler instructions can be described as is. This is useful for using two or more assembler instructions.

```
#pragma asm
    assembler instruction
#pragma endasm
```

A description of example 1 to show next is same to a description of example 2.

Example 1

```
extern int i;
void f(void)
{
#pragma asm
    mov     r0, r10
    st.w    r10, $_i
        :
#pragma endasm
}
```

Example 2

```
extern int  i;
void f(void)
{
    __asm("mov r0, r10");
    __asm("st.w r10, $_i");
    :
}
```

The description from "#pragma asm" to "#pragma endasm" is passed to the assembler as is. In other words, the CA850 internally creates an assembler instruction and starts the assembler. Therefore, a quasi directive of the assembler can be used after the #pragma asm declaration. A local variable in a C language source must not be used with the assembler. Because the local variable is allocated to the stack or a register by the CA850, it cannot be used with an inline assembler.

A local variable in a C language source must not be used with the assembler. Because the local variable is allocated to the stack or a register by the CA850, it cannot be used with an inline assembler. A symbol defined using #define in the C language source file cannot be used in the description from "#pragma asm" to "#pragma endasm", therefore expand a macro defined by #define in a file by an assembler instruction, as follows.

- Define the macro by using the .macro instruction in the #pragma asm - #pragma endasm directives.
- Call an assembler instruction from the C language source program by means of a function call.

Another method is to write an assembler instruction without making a macro definition.

3.5 Controlling Interrupt Level

3.5.1 __set_il function

The CA850 can manipulate the interrupts of the V850 microcontrollers as follows in a C language source.

- By controlling interrupt level
- By enabling or disabling acknowledgment of maskable interrupts (by masking interrupts)

In other words, the interrupt control register can be manipulated. For this purpose, the "__set_il" function is used. Specify this function as follows to manipulate the interrupt priority level.

```
__set_il(interrupt-priority-level, "interrupt-request-name");
```

The "interrupt request name" that can be specified is the "maskable interrupt request name" defined in the device file. Because a request name defined in the device file is used, the #pragma ioreg directive must be described in the C language source that uses this function. Integer values 1 to 8 can be specified as the interrupt priority level. With the V850, eight steps, from 0 to 7, can be specified as the interrupt priority level. To set the interrupt priority level to "5", therefore, specify the interrupt priority level as "4" by this function.

Example

```
__set_il(2, "INTP0");
```

This specification sets the interrupt priority level of interrupt INTP0 to 1.

Specify the __set_il function as follows to enable or disable acknowledgment of a maskable interrupt.

```
__set_il(enables/disables maskable interrupt, "interrupt request name");
```

"-1" or "0" can be specified to enable or disable the maskable interrupt.

Table 3 - 3 Enabling or Disabling Maskable Interrupt

Set Value	Operation
-1	Disables acknowledgment of maskable interrupt (masks interrupt).
0	Enables acknowledgement of maskable interrupt (unmasks interrupt).

Example

```
__set_il(-1, "INTP0");
```

If the function is specified as shown above, acknowledging maskable interrupt INTP0 is disabled (INTP0 is masked). Note that the __set_il function does not manipulate the ep flag (that indicates that exception processing is in progress) in the program status word (PSW).

3.5.2 __set_il function and interrupt control register

The interrupt control register of the V850 microcontrollers is configured as follows.

7	6	5	4	3	2	1	0
xxIFn	xxMKn	0	0	0	xxPRn2	xxPRn1	xxPRn0

If the __set_il function is used, either "priority level" or "interrupt mask flag" is set. This means that the __set_il function cannot set an interrupt request flag.

To set the interrupt priority level to 6 when the interrupt request name is "INTP000" and the interrupt control register name is "P00IC0", for example, describe the function as follows.

```
__set_il(7, "INTP000");
```

The following codes will be output.

```
    ld.b    P00IC0, r1
    andi    0xf8, r1, r1
    ori     0x6, r1, r1
    st.b    r1, P00IC0
```

Therefore, codes that change only the lower 3 bits (xxxPR02 to xxxPR00) of the setting of the priority level are output.

Describe the __set_il function as follows to enable a maskable interrupt when the interrupt request name is "INTP000" and the interrupt control register name is "P00IC0".

```
__set_il(0, "INTP000");
```

The following code will then be output.

```
    clr1    6, P00IC0
```

A code that changes only the interrupt mask flag is output.

If a value is directly written to the interrupt control register, values are set to the priority level, interrupt mask flag, and interrupt request flag.

Example

When the interrupt control register name is "P00IC0"

```
P00IC0=0x6;
```

The above description outputs the following codes.

```
    mov      0x6, r29
    st.b     r29, P00IC0
```

The meanings of these codes are as follows.

- Sets the priority level to 6.
- Enables the maskable interrupt.
- Clears the interrupt request flag.

3.6 Disabling Interrupts

The CA850 can disable the maskable interrupts in a C language source. This can be done in the following two ways.

- Locally disabling interrupt in function
- Disabling interrupts in entire function

3.6.1 Locally disabling interrupt in function

The "di instruction" and "ei instruction" of the assembler instruction can be used to disable an interrupt locally in a function described in C language. However, the CA850 has functions that can control the interrupts in a C language source.

Table 3 - 4 Interrupt Control Functions

Interrupt Control Function	Operation	Processing by CA850
`__DI();`	Disables interrupt.	Generates di instruction.
`__EI();`	Enables interrupt.	Generates ei instruction.

Example (How to use the __DI() and __EI() functions and the codes to be output are shown below.)

```
[C language source]
void func1(void)
{
        :
    __DI();
    /* describe processing to be performed with interrupt disabled */
    __EI();
        :
}
```

```
[output code of C language source above]
_func1:
    -- prologue code
        :
    di
    -- processing to be performed with interrupt disabled
    ei
        :
    -- epilogue code
    jmp [lp]
```

3.6.2 Disabling interrupts in entire function

The CA850 has a "#pragma block_interrupt" directive that disables the interrupts of an entire function.

This directive is described as follows.

```
#pragma block_interrupt function-name
```

Describe functions that are described in the C language. In the case of a function, "void func1() {}", specify "func1".

The interrupt to the function specified by "function-name" above is disabled.

As explained in "3.6.1 Locally disabling interrupt in function", "__DI()" can be described at the beginning of a function and "__EI()", at the end. In this case, however, an interrupt to the prologue code and epilogue code output by the CA850 cannot be disabled or enabled, and therefore, interrupts in the entire function cannot be disabled.

Using the #pragma block_interrupt directive, interrupts are disabled immediately before execution of the prologue code, and enabled immediately after execution of the epilogue code. As a result, interrupts in the entire function can be disabled.

Example (How to use the #pragma block_interrupt directive and the code that is output are shown below.)

```
[C language source]
#pragma block_interrupt func1
void func1(void)
{
        :
    /* describe processing to be performed with interrupt disabled */
        :
}
```

```
[output code of C language source above]
_func1:
    di
    -- prologue code
        :
    -- processing to be performed with interrupt disabled
        :
    -- epilogue code
    ei
    jmp [lp]
```

3.6.3 Notes on disabling interrupts in entire function

Note the following points when disabling interrupts in an entire function.

(1) If an interrupt handler and a #pragma block_interrupt directive are specified for the same interrupt, the interrupt handler takes precedence, and the setting of disabling interrupts is ignored.

(2) If the following functions are called in a function in which an interrupt is disabled, the interrupt is enabled when execution has returned from the call.

- Function specified by #pragma block_interrupt
- Function that disables interrupt at the beginning and enables interrupt at the end

(3) Describe the #pragma block_interrupt directive before the function definition in the same file; otherwise an error occurs during compilation. However, the order of prototype declaration of a function is not affected.

(4) Neither #pragma inline nor inline expansion can be specified by an optimization option for the function specified by a #pragma block_interrupt directive. The inline expansion specification is ignored.

(5) A code that manipulates the ep flag (that indicates exception processing is in progress) in the program status word (PSW) is not output even if #pragma block_interrupt is specified.

3.7 Interrupt/Exception Processing Handler

The CA850 can describe an interrupt handler or exception handler that is called if an interrupt or exception occurs. This section explains how to describe these handlers.

3.7.1 Occurrence of interrupt/exception

If an interrupt or exception occurs in the V850, the program jumps to a handler address corresponding to the interrupt or exception. An interrupt source and a handler address correspond one by one. A collection of handler addresses is called an interrupt/exception table. For example, the interrupt/exception table of the V850ES/SG2 is as shown below (only the top part is shown).

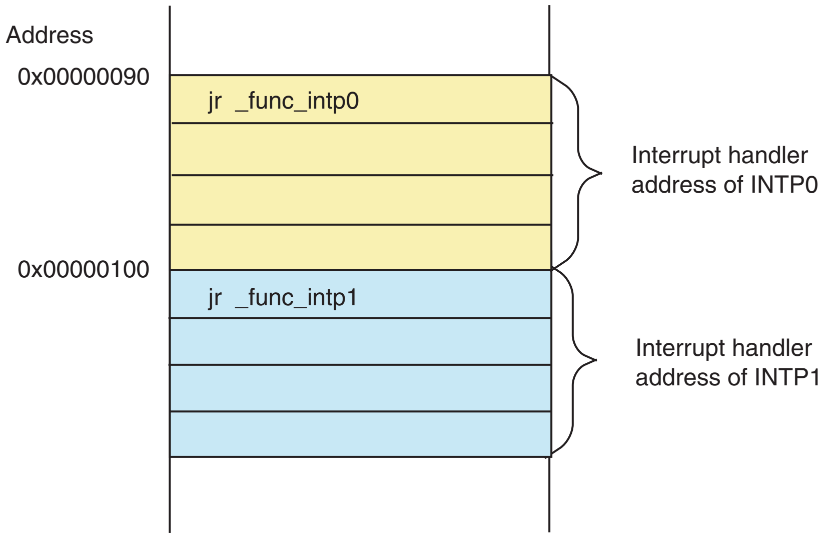
Table 3 - 5 Interrupt/Exception Table (V850ES/SG2)

Address	Interrupt Name	Interrupt Trigger
0x00000000	RESET	RESET pin input/reset by internal source
0x00000010	NMI	Valid edge input to NMI pin
0x00000020	INTWDT2	Overflow of WDT2
0x00000040	TRAP0n	TRAP instruction
0x00000050	TRAP1n	TRAP instruction
0x00000060	LGOP/DBG0	Illegal instruction code/DBTRAP instruction
0x00000080	INTLVI	Low voltage detection
0x00000090	INTP0	Detection of input edge of external interrupt pin (INTP0)
0x000000A0	INTP1	Detection of input edge of external interrupt pin (INTP1)
0x000000B0	INTP2	Detection of input edge of external interrupt pin (INTP2)
0x000000C0	INTP3	Detection of input edge of external interrupt pin (INTP3)
:		
:		

The arrangement of the handler addresses and the available interrupts vary depending on the device of the V850. Refer to the Relevant Device's Hardware User's Manual of each device for details.

Each handler address has a 16-byte area. If an interrupt occurs, an instruction written in that 16-byte area is executed. This means that, if the processing code does not exceed 16 bytes, it is performed only in the handler address. If not, an instruction that branches to a function (interrupt handler) where the processing is written is described.

Figure 3 - 6 Image of Interrupt Handler Address


Address
0x00000090
jr _func_intp0
Interrupt handler address of INTP0
0x00000100
jr _func_intp1
Interrupt handler address of INTP1

If the INTP0 interrupt occurs in the V850ES/SG2, the program jumps to address 0x90 and executes the code written at that address.
In this example, the program jumps to the func_intp0 function because a code that branches to that function is written.
This means that func_intp0 is the interrupt handler of INTP0.

The above description is at an assembly language source level. With the CA850, users do not have to pay much attention to this when describing interrupt servicing in C language source. How to describe interrupt servicing is explained specifically in "3.7.3 Describing interrupt/exception handler".

3.7.2 Processing necessary in case of interrupt/exception

If an interrupt/exception occurs while a function or a task is being executed, interrupt/exception processing must be immediately executed. When the interrupt/exception processing is completed, execution must return to the function or task that was interruptedNote. Therefore, the register information at that time must be saved when an interrupt/exception occurs, and the register information must be restored when interrupt/exception processing is complete.

Note When a real-time OS is used, execution may not return to a task that is interrupted if a system call is issued during interrupt servicing. Refer to the User's Manual of each real-time OS for details.

The prologue and epilogue codes of an ordinary function save and restore the registers for register variables.

The registers for register variables are shown below. Those that must be saved and restored are saved and restored.

Table 3 - 6 Registers for Register Variables

Register Mode	Registers for Register Variables
22-register mode	r25, r26, r27, r28, r29
26-register mode	r23, r24, r25, r26, r27, r28, r29
32-register mode	r20, r21, r22, r23, r24, r25, r26, r27, r28, r29

When execution shifts to an interrupt/exception handler, the following registers that must be saved, in addition to the registers shown in the above table, are also saved as a stack frame for the interrupt/exception handler.

Table 3 - 7 Stack Frame for Interrupt/Exception Handler

Register Mode	Registers Saved/Restored in Case of Interrupt/Exception
22-register mode	r1, r6, r7, r8, r9, r10, r11, r12, r13, r14, r31(lp), CTPC **[V850E]**, CTPSW **[V850E]**
26-register mode	r1, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15, r16, r31(lp), CTPC **[V850E]**, CTPSW **[V850E]**
32-register mode	r1, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15, r16, r17, r18, r19, r31(lp), CTPC **[V850E]**, CTPSW **[V850E]**

When execution shifts to an interrupt/exception handler, the following registers that must be saved, in addition to the registers shown in the above table, are also saved as a stack frame for the interrupt/exception handler.

Table 3 - 8 Stack Frame for Multiple Interrupt/Exception Handler

Register Mode	Registers Saved/Restored in Case of Multiple Interrupts/Exceptions
22-register mode	r1, r6, r7, r8, r9, r10, r11, r12, r13, r14, r31(lp), EIPC, EIPSW, CTPC **[V850E]**, CTPSW **[V850E]**
26-register mode	r1, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15, r16, r31(lp), EIPC, EIPSW, CTPC **[V850E]**, CTPSW **[V850E]**
32-register mode	r1, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15, r16, r17, r18, r19, r31(lp), EIPC, EIPSW, CTPC **[V850E]**, CTPSW **[V850E]**

The usage of the above registers is as follows.

Table 3 - 9 Usage of Registers

Register	Usage
r1	Assembler-reserved register
r6 - r9	Registers for arguments (registers to set arguments of function)
r10 - r19	Work registers (registers used by CA850 to generate codes)
r31	Link pointer
CTPC **[V850E]**	Program counter (PC) when CALLT instruction is executed
CTPSW **[V850E]**	Program status word (PSW) when CALLT instruction is executed
EIPC	Program counter (PC) during interrupt/exception processing
EIPSW	Program status word (PSW) during EIPSW interrupt/exception processing

When interrupt/exception processing is completed, the code which restores saved registers is output, the reti instruction is output. This instruction notifies the V850 that the interrupt servicing is completed.

If codes for saving/restoring registers or outputting the reti instruction are described as explained in "3.7.3 Describing interrupt/exception handler", the CA850 automatically outputs the relevant code. The code for saving/restoring registers is output as explained in Table 3 - 10. The user therefore does not have to pay much attention to this and can concentrate on describing the processing of the main body of the interrupt handler.

Table 3 - 10 Processing for Saving/Restoring Registers During Interrupt

Register Name		Register	Explanation
r1 register		r1	Always saved/restored at interrupt.
Argument registers		r6 - r9	r6 is always saved/restored when the interrupt source is TRAP0/ TRAP1. Saved/restored when a function call (including runtime functions) exists. Saved/restored if a function call does not exist but is used with an interrupt function.
Work registers	22-register mode	r10 - r14	Saved/restored when a function call exists. Saved/restored if a function call does not exist but is used with an interrupt function.
	26-register mode	r10 - r16	
	32-register mode	r10 - r19	
Register variable registers	22-register mode	r25 - r29	Saved/restored as necessary, as with ordinary functions.
	26-register mode	r23 - r29	
	32-register mode	r20 - r29	
Link pointer		r31(lp)	Saved/restored when a function call (including runtime functions) exists. Saved/restored if a function call does not exist.
Interrupt-related system registers		EIPCE, EIPSW	Saved/restored with functions using the multiple interrupt qualifier __multi_interrupt. Not saved/restored with the __interrupt qualifier.
callt instruction-related system registers **[V850E]**		CTPC, CTPSW	Always saved/restored with interrupt functions being compiled with a V850E/V850ES/V850E2 core device specified.

3.7.3 Describing interrupt/exception handler

The format in which an interrupt/exception handler is described does not differ from ordinary C functions, but the functions described in C must be recognized as an interrupt/exception handler by the CA850. With the CA850, an interrupt/exception handler is specified using the #pragma interrupt directive and __interrupt qualifier, or #pragma interrupt directive and __multi_interrupt qualifier.

(1) When specifying interrupt handler

```
#pragma interrupt Interrupt-request-name Function-name Allocation-method
```

```
__interrupt Function-definition, or Function-declaration
```

(2) When specifying multiple-interrupt handler

```
#pragma interrupt Interrupt-request-name Function-name Allocation-method
```

```
__multi_interrupt Function-definition, or Function-declaration
```

Describe functions that are described in the C language. In the case of a function, "void func1() {}", specify "func1".

"Specifying multiple-interrupt handler" means to "specify a function that can be interrupted more than once" and does not mean "to specify a function that interrupts more than once".

(a) Interrupt request name

Interrupt request names registered in the device file can be specified. Refer to the interrupt request names described in the Relevant Device’s Architecture User’s Manual of each device; they are the interrupt request names registered in the device file.

A non-maskable interrupt (NMI) can also be specified in this way, but a reset interrupt (RESET) cannot be specified. Processing after reset must be described with assembler instructions. Processing after reset is generally described in the startup routine, so refer to "CHAPTER 5 STARTUP ROUTINE" for details.

(b) Function name

Specify the names of functions that are used as an interrupt handler. Describe the function name in C language source. When specifying the function "void func1(void)", specify the function name as "func1".

(c) Allocation method

Specify whether the main body of the function is directly allocated to the handler address, or only the instruction that branches to the interrupt handler function is allocated. Specify "direct" when the main body of the function is directly allocated; otherwise describe nothing as "allocation method".

By specifying "direct", all functions are allocated from the handler address of the specified interrupt source. Note, however, that the areas for the subsequent handler address are also used.

When specifying "direct", be sure to describe the #pragma interrupt directive before the function definition; otherwise an error occurs during compilation.

Next, the roles of the #pragma interrupt directive, __interrupt qualifier, and __multi_interrupt qualifier are explained.

(1) #pragma interrupt directive

Allocates an instruction (jr) that branches to the specified function to a handler address corresponding to the interrupt request name specified by the #pragma interrupt directive. When the -Xj option is specified, this directive allocates an instruction that saves the r1 register contents to the stack and an instruction (jmp) that branches to the specified function.

(2) __interrupt qualifier

Adds processing to save/restore the register contents by an interrupt/exception handler to a function with the __interrupt qualifier and adds the reti instruction at the end. When the -Xj option is specified, processing to save the r1 register contents is output to the handler address, so only restore processing is output for the function.

(3) multi_interrupt qualifier

Adds processing to save/restore the register contents by an interrupt handler and processing to save/ restore the contents of the EIPC and EIPSW registers to a function with the __multi_interrupt qualifier. This directive also adds the reti instruction at the end. When the -Xj option is specified, processing to save the r1 register contents is output to the handler address, so only restore processing is output for the function.

When the #pragma interrupt directive, __interrupt qualifier, and __multi_interrupt qualifier are specified at the same time, the following codes are output and the handler completes the interrupt servicing routine.

- Allocation of an instruction branching to the specified interrupt handler to the handler address
- Addition of processing to save/restore the register contents by an interrupt handler (and processing to save/restore the contents of EIPC and EIPSW if the __multi_interrupt qualifier is specified)
- Addition of the reti instruction at the end of the handler

In this case, function definition and the #pragma interrupt directive can be described in separate files in any order. If "direct" is specified for the allocation method, however, they cannot be described in separate files.The following codes are output if only the __interrupt qualifier or __multi_interrupt qualifier is specified.

- Addition of processing to save/restore the register contents by an interrupt handler (and processing to save/restore the contents of EIPC and EIPSW if the __multi_interrupt qualifier is specified)
- Addition of the reti instruction at the end of the handler

Therefore, the function can be started as an interrupt handler but the processing to allocate "an instruction to branch to the interrupt handler to the handler address" output by the #pragma interrupt directive is not performed.

Example

The #pragma interrupt is specified as follows when the interrupt handler "void intp0_func(void)" is used for the interrupt request name "INTP0" without "direct" being specified and multiple interrupts being enabled.

```
#pragma interrupt INTP0 intp0_func

__interrupt
void intp0_func(void)
{
        :
    /* main body of interrupt servicing */
        :
}
```

Next, the function type that can be specified as an interrupt handler is explained.

(1) Function type

The type of a handler that handles a maskable interrupt or NMI is as follows.

`void func(void) type`

The argument and return value of this function are void type.

The type of a software exception processing (trap) handler is as follows.

`void func(unsigned int) type`

EICC (exception code) of the interrupt source register (ECR) is set as the argument. Unless the function is specified by this type, an error occurs during compilation. Refer to the next paragraph for the software exception processing function.

(2) Software exception processing (trap processing) handler

When software exception processing (trap processing) is used, two entry points, TRAP0 (address 0x40) and TRAP1 (address 0x50), are used according to the specifications of the V850 microcontrollers.

When the software exception "trap 0x00 to trap 0x0f" occurs, execution branches to TRAP0 (address 0x40); if "trap 0x10 to trap0x1f" occurs, it branches to TRAP1 (address 0x50). At this time, the value "0x40 to 0x4f" is set to the interrupt source register (ECR) as a software exception code in the case of TRAP0. In the case of TRAP1, the value "0x50 to 0x5f" is set to the ECR.

Table 3 - 11 Trap Instructions and Software Exception Codes

Trap Instruction	Software Exception Code
trap 0x00	0x40
trap 0x01	0x41
trap 0x02	0x42
: :	: :
trap 0x0a	0x4a
trap 0x0b	0x4b

Table 3 - 11 Trap Instructions and Software Exception Codes

Trap Instruction	Software Exception Code
: :	: :
trap 0x10	0x50
trap 0x11	0x51
trap 0x12	0x52
: :	: :
trap 0x1e	0x5e
trap 0x1f	0x5f

When software exception processing for TRAP0 or TRAP1 is described, that function has one argument and the type of the variable is "unsigned int". The software exception code set to the interrupt source register (ECR) is set as the argument. In the case of TRAP0, the value is "0x40 to 0x4f". In the case of TRAP1, it is "0x50 to 0x5f". Processing must be described in the handler depending on these values.

```
#pragma interrupt TRAP0 trap0_func

__interrupt
void trap0_func(unsigned int codenum)
{
        :
    /* describe processing of each exception code. */
        :
}
```

3.7.4 Notes on describing interrupt/exception handler

(1) "Specifying multiple-interrupt handler" with the __multi_interrupt qualifier means to "specify a function that can be interrupted more than once" and does not mean "to specify a function that interrupts more than once".

(2) Even if a handler that enables multiple interrupts is specified by __multi_interrupt, interrupts are not enabled when the interrupt handler is activated. Therefore, be sure to issue an interrupt enabling instruction (such as __EI()) in the interrupt handler, and issue an interrupt disabling instruction (such as __DI()) at the end of the handler. If the interrupt disabling instruction is not issued at the end of the handler, an interrupt may be acknowledged while the contents of a register are being restored, which may cause a hang-up.

(3) The reset interrupt cannot be specified by the #pragma interrupt directive.

```
#pragma interrupt RESET reset_func /* error */
```

If the above description is made, an error occurs during compilation. Processing after reset must be described with assembler instructions. Processing after reset is generally described in the startup routine, so refer to "CHAPTER 5 STARTUP ROUTINE" for details.

(4) The #pragma interrupt directive and __multi_interrupt qualifier do not support multiple exceptions and multiple NMIs. To use multiple exceptions or multiple NMI, add a code that saves or restores the necessary system registers (such as FEPC and FEPSW). Refer to the Relevant Device’s Hardware User’s Manualt of each device for the necessary system registers.

(5) The user is not required to additionally describe an interrupt handler address in the link directive file; it is output internally by the CA850.

(6) The same interrupt request name must not be specified for two or more functions.

(7) Both the __interrupt qualifier and __multi_interrupt qualifier must not be specified for the same function.

(8) An error occurs during compilation if a function is declared with the __interrupt qualifier or __multi_interrupt qualifier after the function is defined without the __interrupt qualifier or __multi_interrupt qualifier being specified.

(9) A function specified as an interrupt/exception handler cannot be expanded inline. The #pragma inline directive is ignored even if specified.

(10) An interrupt to a function specified as an interrupt/exception handler is disabled. Therefore, the #pragma block_interrupt directive is ignored even if specified.

(11) A function specified as an interrupt/exception handler cannot be called by an ordinary function call. If it is called from another file, the compiler cannot check it.

(12) When an assembler program is called from an interrupt/exception handler and the registers for register variables and the registers shown in Table 3 - 6 and Table 3 - 7 are used, processing to save/restore the register contents must be described. Processing to save/restore the register contents must also be described when sp (r3), gp (r4), tp (r5), and ep (r30) are rewritten.

(13) The #pragma interrupt directive, __interrupt qualifier, and __multi_interrupt qualifier do not issue a processing end report (EOI command) to the external interrupt controller. The user should therefore execute this directive, if necessary.

(14) Disable interrupts at the end of multiple interrupts because a code that restores EIPC and EIPSW must be described.

(15) If "direct" is not specified, an instruction to branch to the interrupt/exception handler is allocated to the handler address. In this case, the CA850 outputs the jr instruction to enhance the code efficiency. However, the range in which the jr instruction can branch execution is limited to ±21 bits from the jr instruction. If the main body of the interrupt handler is not within the range in which the jr instruction can branch execution, an error occurs during linking. In this case, specify the compilation option "-Xj" to replace the jr instruction with the jmp instruction.

3.7.5 Description example of interrupt/exception handler

Examples of describing interrupt/exception handlers are shown below. Note that the interrupt request name differs depending on the device. Refer to the Relevant Device's Architecture User's Manual of each device.

Example 1 (Non-maskable interrupt)

```
#pragma interrupt NMI func1     /* non-maskable interrupt */

__interrupt
void func1(void)
{
    :
}
```

Example 2 (Trap)

```
#pragma interrupt TRAP0 func2   /* trap 0 */

__interrupt
void func2(unsigned int num)
{
    switch(num){                /* for every exception code */
        :
    }
}
```

Example 3 (#pragma interrupt and __interrupt qualifier in separate files)

```
[a. c]
__interrupt             /* __interrupt specification */
void func1(void)
{
    :
}
[b. c]
#pragma interrupt NMI func1 /* can be described after definition or in separate file */
```

Example 4 (Specification of multiple interrupts)

```
#pragma interrupt INTP0 func1
__multi_interrupt       /* multiple-interrupt function specified */
void func1(void)
{
    :
}
```

3.8 Inline Expansion

The CA850 allows inline expansion of each function. This section explains how to specify inline expansion.

3.8.1 Inline expansion

Inline expansion is used to expand the main body of a function at a location where the function is called. This decreases the overhead of function call and increases the possibility of optimization. As a result, the execution speed can be increased. If inline expansion is executed, however, the object size increases.

Specify the function to be expanded inline using the #pragma inline directive.

```
#pragma inline function-name [,function-name...]
```

Describe functions that are described in the C language. In the case of a function, "void func1() {}", specify "func1". Two or more function names can be specified with each delimited by "," (comma).

```
#pragma inline func1,func2

void func1(){ ... }
void func2(){ ... }

void func(void)
{
    func1();  /* function subject to inline expansion */
    func2();  /* function subject to inline expansion */
}
```

3.8.2 Conditions of inline expansion

At least the following conditions must be satisfied for inline expansion of a function specified using the #pragma inline directive. If optimization other than "size priority optimization (-Os)" and "execution speed priority optimization (-Ot)" is specified, however, inline expansion may not be executed even if the following conditions are satisfied, because of the internal processing of the CA850.

(1) A function that expands inline and a function that is expanded inline are described in the same file.

A function that expands inline and a function that is expanded inline, i.e., a function call and a function definition must be in the same file. This means that a function described in another C language source cannot be expanded inline. If it is specified that a function described in another C language source is expanded inline, the CA850 does not output a warning message and ignores the specification.

(2) The #pragma inline directive is described before function definition.

If the #pragma inline directive is described after function definition, the CA850 outputs a warning message and ignores the specification. However, prototype declaration of the function may be described in any order. Here is an example.

Example

Valid Inline Expansion Specification	Invalid Inline Expansion Specification

`#pragma inline func1,func2`
 	
`/* prototype declaration */`
`void func1();`
`void func2();`
 	
`/* function definition */`
`void func1() { /* ... */ }`
`void func2() { /* ... */ }`
	`/* prototype declaration */`
	`void func1();`
	`void func2();`
	
	`/* function definition */`
	`void func1() { /* ... */ }`
	`void func2() { /* ... */ }`
	
	`#pragma inline func1,func2`

(3) The number of arguments is the same between "call" and "definition" of the function to be expanded inline.

If the number of arguments is different between "call" and "definition" of the function to be expanded inline, the CA850 outputs a warning message and ignores the specification.

(4) The types of return value and argument are the same between "call" and "definition" of the function to be expanded inline.

If the return value type and argument type are different between "call" and "definition" of the function to be expanded inline, the CA850 outputs a warning message and ignores the specification. If the type can be converted, however, it is converted as follows and the function is expanded inline.

- The return value type is the type of the "calling side".
- The argument type is the type of the "function definition".

If the "-ansi" option is specified, however, the type is not converted and an error is output.

(5) The size of the function to be expanded inline and the stack size are not too large.

If the size of the function to be expanded inline and the stack size are too large, neither an error nor warning is output, and the inline expansion specification is ignored. This "size" means the size in the intermediate language and is different from the size of the actual object. The upper limit of the size can be changed in the CA850. The function size in the intermediate language can be changed by this option.

The function size in the intermediate language can be changed by this option.

```
-Wp,-Nnum
```

The stack size used by the function in the intermediate language can be changed by this option.

```
-Wp,-Gnum
```

In addition, the size of each function and stack size used in the intermediate language can be checked by using this option.

```
-Wp,-l
```

This option can be used to determine the size for specification.

(6) The number of arguments of the function to be expanded inline is not variable.

If inline expansion is specified for a function with a variable number of arguments, the CA850 outputs neither an error nor warning message and ignores the specification.

(7) Recursive function is not specified to be expanded inline.

If a recursive function that calls itself is specified for inline expansion, the CA850 outputs neither an error nor warning message and ignores the specification. If two or more function calls are nested and if a code that calls itself exists, however, inline expansion may be executed.

(8) An interrupt handler is not specified to be expanded inline.

A function specified by the #pragma interrupt, __interrupt, or t__multi_interrupt directive is recognized as an interrupt handler. If inline expansion is specified for this function, the CA850 outputs a warning message and ignores the specification.

(9) A task of a real-time OS is not specified to be expanded inline.

A function specified by the #pragma rtos_task directive is recognized as a task of a real-time OS. If inline expansion is specified for this function, the CA850 outputs a warning message and ignores the specification.

(10) Interrupts are not disabled in a function by the #pragma block_interrupt directive.

If inline expansion is specified for a function in which interrupts are declared by the #pragma block_interrupt directive to be disabled, the CA850 outputs a warning message and ignores the specification.

3.8.3 Controlling inline expansion via options

Inline expansion can be controlled using options when inline expansion by the compiler should be suppressed. The cases in which inline expansion can be controlled and the options are as follows. If execution speed priority optimization (-Ot) is specified, however, refer to "3.8.4 Execution speed priority optimization and inline expansion".

(1) To expand inline all static functions that are referenced only once

```
-Wp,-S
```

If this option is specified, a static function that is referenced only once is expanded inline, regardless of optimization specification and the presence or absence of a #pragma inline specification. If optimization other than the size priority optimization (-Os) is specified, however, inline expansion may not be executed even if the -Wp,-S option is specified, because of the internal processing of the CA850.

(2) To suppress inline expansion of all functions

```
-Wp,-no_inline
```

In this case, inline expansion is suppressed even if the -Wp,-S option or the #pragma inline directive is specified.

3.8.4 Execution speed priority optimization and inline expansion

If the "execution speed priority optimization (-Ot)" option of the CA850 is specified, the CA850 uses inline expansion as one of the means of optimization. If the -Ot option is specified, the CA850 selects an appropriate function and expands it inline as long as the inline expansion conditions in "3.8.2 Conditions of inline expansion" are satisfied, even if the function is not specified for inline expansion by the #pragma inline directive. Inline expansion can be controlled using options when inline expansion by the compiler should be suppressed. The items that can be controlled and the options are as follows.

(1) To suppress inline expansion of all functions even though the -Ot option is specified

```
-Wp,-no_inline
```

In this case, inline expansion is suppressed even if the -Wp,-S option or the #pragma inline directive is specified.

(2) To expand inline only the function specified by the #pragma inline directive even though the -Ot option is specified

```
-Wp,-inline
```

In this case, the function for which inline expansion is specified must meet the conditions explained in "3.8.2 Conditions of inline expansion".

3.8.5 Examples of differences in inline expansion operation depending on option specification

Here are examples of differences in inline expansion operation depending on whether the #pragma inline directive or an option is specified.

When -Os (size priority optimization) is specified (other than -Ot)

```
#pragma inline func0

/* expanded if inline expansion conditions are satisfied because
    #pragma inline is specified #pragma inline */
void func0(){...}
void func1(){...} /* not expanded */
void func2(){...} /* not expanded */
```

When -Ot (execution speed priority optimization) is specified

```
#pragma inline func0

/* expanded if inline expansion conditions are satisfied because -Ot is
   specified */
void func0(){...}
/* expanded if inline expansion conditions are satisfied because -Ot is
   specified */
void func1(){...}
/* expanded if inline expansion conditions are satisfied because -Ot is
   specified */
void func2(){...}
```

When -Ot (execution speed priority optimization)
+ -Wp,-inline (inline expansion of only function specified by #pragma inline) are specified

```
#pragma inline func0

/* expanded if inline expansion conditions are satisfied because
   #pragma inline is specified */
void func0(){...}
/* not expanded because -Wp,-inline is specified but #pragma inline is not
   specified */
void func1(){...}
/* not expanded because -Wp,-inline is specified but #pragma inline is not
   specified */
void func2(){...}
```

Remarks 1 The CA850 does not treat a function specified for inline expansion by the #pragma inline directive as a static function. To use such a function as a static function, static must be explicitly specified.

2 When executing debugging, a breakpoint cannot be specified for a function specified for inlineexpansion in the C language source.

3.9 Real-Time OS Support Function

The CA850 has functions to improve programming description and to reduce the number of codes, making allowances for organizing a system using the V850 microcontrollers real-time OS RX850 or RX850 Pro.

3.9.1 Description of task

An application using a real-time OS performs processing in task units. The real-time OS schedules a task using a system call issued in that task or interrupt servicing. Register contents are saved and restored by the real-time OS when the task is switched (when the context is switched). Therefore, prologue and epilogue processing are different from those of an ordinary function. In other words, the prologue and epilogue processing generated by the CA850 when a function is called are not executed by a task.

To use a function described as a task, the code can be reduced by deleting the prologue and epilogue processing that are executed when a function is called. However, ordinary functions and tasks are not distinguished according to the description method of C language. Therefore, the CA850 has the following #pragma directive so that a function can be recognized as a task of a real-time OS.

```
#pragma rtos_task [function-name]
```

Consequently, the function specified by "function-name" can be recognized as a task of a real-time OS. A function name described in C is specified as "function-name". The following description is made, for example, to use the function "void func1(int inicode){}" as a task.

```
#pragma rtos_task func1
```

"*function-name*" can also be omitted. If omitted, the function following the #pragma rtos_task directive in that file is recognized as a task.

Specifying the #pragma rtos_task directive has the following effect.

(1) The prologue/epilogue processing output by an ordinary function is not performed. Specifically, the following codes are not output.

(a) Saving/restoring of register contents for register variables

(b) Saving/restoring of link pointer (lp)

(c) Jump to return address

(2) The system call "ext_tsk" can be used as a defined function.

This system call can be used even if a prototype declaration is not made in the application. Functions other than the one specified as a task can be called in the same manner as long as they are described after the #pragma rtos_task directive. When this system call is called, a code using the jr instruction is output to reduce the code size. If the main body of system call "ext_tsk" is not in the range in which the jr instruction can branch execution, the linker (ld850) outputs an error. In this case, take the following actions.

(a) Change the memory allocation by the link directive.

(b) Replace the jr instruction with the jmp instruction in the assembly language source.

(c) Specify far jump

Note the following points when the #pragma rtos_task directive is specified.

- A task cannot be called in the same manner as calling a function. A task called from a separate file is not checked. A task cannot be expanded inline because it cannot be called as a function. That is, even if the #pragma inline directive is specified for a function specified by the #pragma rtos_task directive, the #pragma inline specification is ignored.
- An error occurs if "#pragma rtos_task function-name" is described after the function definition in the same file. If the function is not defined after "#pragma rtos_task function-name" is described in the file, the #pragma directive for that function is ignored.
- A function specified by the #pragma rtos_task directive cannot be specified as an ordinary interrupt/exception handler (refer to "3.7 Interrupt/Exception Processing Handler").

Refer to the User's Manual of each real-time OS for the real-time OS functions.

3.10 Embedded Functions

In the CA850 some of the instructions can be described in C language source as embedded functions.

Table 3 - 12 shows the instructions that can be described as functions.

Table 3 - 12 Embedded Functions

Assembler Instruction	Function	Description

di
ei
	Interrupt control (DI/EI)	__DI()
		__EI()

nop	nop	__nop()
halt	halt	__halt()
satadd	Saturated addition (satadd)	long a, b; long __satadd(a, b)
satsub	Saturated subtraction (satsub)	long a, b; long __satsub(a, b)
bsh	Halfword data byte swap (bsh) [V850E]	long a; long __bsh(a)
bsw	Word data byte swap (bsw) [V850E]	long a; long __bsw(a)
hsw	Word data halfword swap (hsw) [V850E]	long a; long __hsw(a)
sxb	Byte data sign extension (sxb) [V850E]	char a; long __sxb(a)
sxh	Halfword data sign extension (sxh) [V850E]	short a; long __sxh(a)
mul	Instruction that assigns higher 32 bits of multiplication result to variable using mul instruction [V850E]	long a; long b; long __mul32(a, b)
mulu	Instruction that assigns higher 32 bits of unsigned multiplication result to variable using mulu instruction [V850E]	unsigned long a, b; unsigned long __mul32u(a, b)

sasf	Flag condition setting with logical left shift (sasf) [V850E]	long a;
		unsigned int b;
		long __sasf(a, b)

Remark [V850E] mark indicates that only V850Ex core is available.

Caution Even if a function is defined with the same name as an embedded function, it cannot be used. If an attempt is made to call such a function, processing for the embedded function provided by the compiler takes precedence.

3.10.1 Interrupt control (DI/EI)

An example of describing the interrupt control (DI/EI) instruction is shown below.

Example

```
void func(void)
{
        :
    __DI();  /* Describe the processing to be executed while interrupts are disabled. */
    __EI();
        :
}
```

Compiler output of above example

```
_func:
    -- Prologue code

    di
    -- Describe the processing to be executed while interrupts are disabled.
    ei
        :
    -- Epilogue code
    jmp [lp]
```

3.10.2 nop

An example of describing the nop instruction is shown below.

Example

```
void func(void)
{
        :
    __nop();
        :
}
```

Compiler output of above example

```
_func:
     :
    nop
     :
```

3.10.3 halt

An example of describing the halt instruction is shown below.

Example

```
void func(void)
{
        :
    __halt();
}
```

Compiler output of above example

```
_func:
        :
    halt
```

3.10.4 Saturated addition (satadd)

An example of describing the saturated addition instruction is shown below.

Example

```
void func(void)
{
    long    a, b, c;
        :
    c = __satadd(a, b);     /* The result of the saturated operation of a */
                            /* and b is stored in c. */
        :
}
```

Compiler output of above example

```
_func:
        :
    ld.w    -4 +.A2[sp], r10    -- Load variable a
    ld.w    -8 +.A2[sp], r11    -- Load variable b
    satadd  r11, r10            -- Saturated subtraction (a + b)
    st.w    r10, -12 +.A2[sp]
    -- The result of the saturated operation is stored in variable c.
        :
```

3.10.5 Saturated subtraction (satsub)

An example of describing the saturated subtraction instruction is shown below

Example

```
void func(void)
{
    long    a, b, c;
        :
    c = __satsub(a, b);     /* The result of the saturated operation of a */
                            /* and b is stored in c (c = a - b). */
        :
}
```

Compiler output of above example

```
_func:
            :
    ld.w    -4 +.A2[sp], r10    -- Load variable a
    ld.w    -8 +.A2[sp], r11    -- Load variable b
    satsub  r11, r10            -- Saturated subtraction (a - b)
    st.w    r10, -12 +.A2[sp]
    -- The result of the saturated operation is stored in variable c.
            :
```

3.10.6 Halfword data byte swap (bsh) [V850E]

An example of describing the halfword data byte swap (bsh) instruction is shown below.

Example

```
void func(void)
{
    long    a, b;

        :
    b = __bsh(a);   /* Halfword data of a is byte-swapped */
                    /* and the result is stored in b. */
        :
}
```

Compiler output of above example

```
_func:
         :
    ld.w    -4+.A2[sp], r10     -- Load variable a
    bsh     r10, r10            -- Halfword data byte swap
    st.w    r10, -8+.A2[sp]
    -- The result of halfword data byte swap is stored in variable b.
         :
```

3.10.7 Word data byte swap (bsw) [V850E]

An example of describing the word data byte swap (bsw) instruction is shown below.

Example

```
void func(void)
{
    long    a, b;
        :
    b = __bsw(a);   /* Word data of a is byte-swapped and the result is stored in b.*/
        :
}
```

Compiler output of above example

```
_func:
        :
    ld.w    -8+.A2[sp], r10     -- Load variable a
    bsw     r10, r10            -- Word data byte swap
    st.w    r10, -12+.A2[sp]    -- Stored in variable b
        :
```

3.10.8 Word data halfword swap (hsw) [V850E]

An example of describing the word data halfword swap (hsw) instruction is shown below.

Example

```
void func(void)
{
    long    a, b;
        :
    b = __hsw(a);   /* Word data of a is halfword-swapped and the result is stored in b. */
        :
}
```

Compiler output of above example

```
_func:
        :
    ld.w    -8+.A2[sp], r10     -- Load variable a
    hsw     r10, r10            -- Word data halfword swap
    st.w    r10, -12+.A2[sp]    -- Stored in variable b
        :
```

3.10.9 Byte data sign extension (sxb) [V850E]

An example of describing the byte data sign extension (sxb) instruction is shown below.

Example

```
void func(void)
{
    char    a;
    long    b;
        :
    b = __sxb(a);   /* Sign extension of the byte data of a is performed */
                    /* and the result is  stored in b. */
        :
}
```

Compiler output of above example

```
_func:
         :
    ld.w    -8+.A2[sp], r10      -- Load variable a
    sxb     r10, r10             -- Sign extension of byte data
    st.w    r10, -12+.A2[sp]     -- Stored in variable b
         :
```

3.10.10 Halfword data sign extension (sxh) [V850E]

An example of describing the halfword data sign extension (sxh) instruction is shown below.

Example

```
void func(void)
{
    short   a;
    long    b;
        :
    b = __sxh(a);   /* Sign extension of the halfword data of a is */
                    /* performed and the result is stored in b. */
        :
}
```

Compiler output of above example

```
_func:
        :
    ld.w    -8+.A2[sp], r10      -- Load variable a
    sxh     r10                  -- Sign extension of halfword data
    st.w    r10, -12+.A2[sp]     -- Stored in variable b
        :
```

3.10.11 Instruction that assigns higher 32 bits of multiplication result to variable using mul instruction [V850E]

An example of describing the instruction that assigns variable for the higher 32 bits of the multiplication result using the mul instruction is shown below.

Example

```
void func(void)
{
    long    a, b, c;
        :
    c = __mul32(a, b);  /* The higher 32 bits of the result of a * b are stored in c. */
        :
}
```

Compiler output of above example

```
_func:
        :
    ld.w    -4+.A2 [sp], r10    -- Load variable a
    ld.w    -8+.A2 [sp], r11    -- Load variable b
    mul     r11, r10, r12       -- a * b
    st.w    r12, -12+.A2 [sp]   -- Stored in variable c
        :
```

3.10.12 Instruction that assigns higher 32 bits of unsigned multiplication result to variable using mulu instruction [V850E]

An example of describing the instruction that assigns the higher 32 bits of the unsigned multiplication result to variable using mulu instruction is shown below.

Example

```
void func(void)
{
    unsigned long a, b, c;
        :
    c = __mul32u(a, b); /* The higher 32 bits of the result of a * b are stored in c. */
        :
}
```

Compiler output of above example

```
_func:
        :
    ld.w    -4+.A2 [sp], r10     -- Load variable a
    ld.w    -8+.A2 [sp], r11     -- Load variable b
    mulu    r11, r10, r12        -- a * b
    st.w    r12, -12+.A2 [sp]    -- Stored in variable c
        :
```

3.10.13 Flag condition setting with logical left shift (sasf) [V850E]

An example of describing the flag condition setting instruction with logical left shift when a conditional expression is written in the second argument is shown in Example 1.

An example of describing the flag condition setting instruction with logical left shift when a variable is written in the second argument is shown in Example 2.

Example 1

```
/* When a conditional expression is written in the second argument */
void func(void)
{
    unsigned long a, b, c;
        :
    c = __sasf(c, a == b);  /* If a == b is true, c is shifted left logically by */
                            /* 1 bit and 1 is added. */
                            /* If a == b is not true, c is shifted left logically */
                            /* by 1 bit. The result is stored in c. */
         :
}
```

Compiler output of above Example 1

```
_func:
        :
    ld.w     -4+.A2 [sp], r10   -- Load variable a
    ld.w     -8+.A2 [sp], r11   -- Load variable b
    cmp      r11, r10           -- Compare variable a and b.
    ld.w     -12+.A6 [sp], r12  -- Load variable c
    sasf     0x2, r12
  -- If a == b is true, c is shifted left logically by 1 bit and 1 is added.
  -- If a == b is not true, c is shifted left logically by 1 bit.
    st.w     r12, -12+.A2 [sp]  -- Stored in variable c.
        :
```

Example 2

```
/* When a variable is written in the second argument */
void func(void)
{
    unsigned long a, b;
        :
    b = __sasf(b, a);   /* If a is not 0, b is shifted left logically by 1 bit */
                        /* and 1 is added. */
                        /* If a is other than 0, b is shifted left logically */
                        /* by 1 bit. */
                        /* The result is stored in b. */
        :
}
```

Compiler output of above Example 2

```
_func:
        :
    ld.w    -4+.A2 [sp], r10    -- Load variable a
    cmp     r0, r10             -- Compare variable a and 0.
    ld.w    -8+.A2 [sp], r11    -- Load variable b
    sasf    0xa, r11
    -- If a is not 0, b is shifted left logically by 1 bit and 1 is added.
    -- If a is 0, b is shifted left logically by 1 bit.
    st.w    r11, -8+.A2 [sp]    -- Stored in variable b
        :
```

3.11 Structure Packing Function

In the CA850, the alignment of structure members can be specified at the C language level. This function is equivalent to the -Xpack option, however, the structure-packing directive can be used to specify the alignment value in any location in the C language source.

Note The data area can be reduced by packing a structure, but the program size increases and the execution speed is degraded.

3.11.1 Structure packing specified

The structure packing function is specified in the following format.

```
#pragma pack([1248])
```

#pragma pack changes to an alignment value of the structure member upon the occurrence of this directive. The alignment value is called the packing value and the specifiable numeric values are 1, 2, 4, and 8. When the packing value is not specified, the default alignment (1)[Note] is specified. Since this directive becomes valid upon occurrence, several directives can be described in the C language source.

Example

```
/* Structure member aligned using 1-byte alignment */
#pragma pack(1)
struct TAG {
    char    c;
    int     i;
    short   s;
};
```

Note Alignment values "4" and "8" are treated as the same in Ver. 2.70.

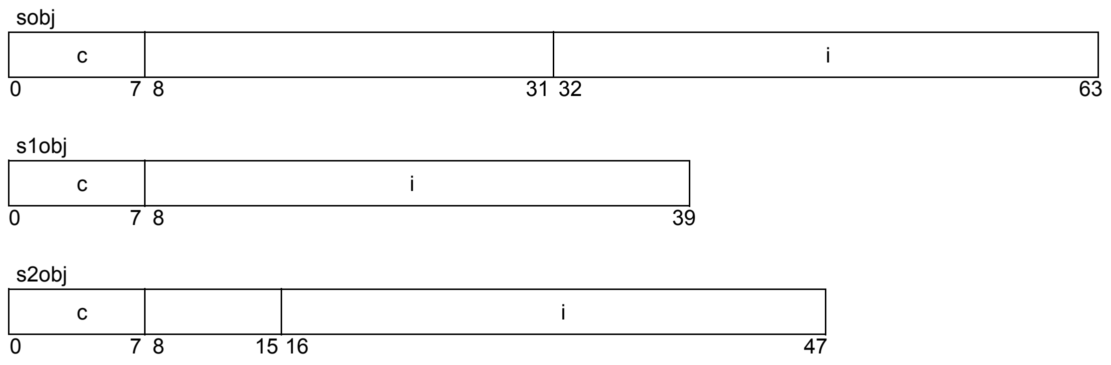
3.11.2 Rules of structure packing

The structure members are aligned in a form that satisfies the condition whereby members are aligned according to whichever is the smaller value: the structure packing value or the member's alignment value.

For example, if the structure packing value is 2 and member type is int type, the structure members are aligned in 2-byte alignment.

Example

```
struct S {
    char    c;      /* Satisfies 1-byte alignment condition */
    int     i;      /* Satisfies 4-byte alignment condition */
};
#pragma pack(1)
struct S1 {
    char    c;      /* Satisfies 1-byte alignment condition */
    int     i;      /* Satisfies 1-byte alignment condition */
};
#pragma pack(2)
struct S2 {
    char    c;      /* Satisfies 1-byte alignment condition */
    int     i;      /* Satisfies 2-byte alignment condition */
};
struct S    sobj;   /* Size of 8 bytes */
struct S1   s1obj;  /* Size of 5 bytes */
struct S2   s2obj;  /* Size of 6 bytes */
```



3.11.3 Union

A union is treated as subject to packing and is handled in the same manner as structure packing.

Example 1

```
union U {
    struct S {
        char c;
        int i;
    } sobj;
};
#pragma pack(1)
union U1 {
    struct S1 {
        char c;
        int i;
    } s1obj;
};
#pragma pack(2)
union U2 {
    struct S2 {
        char c;
        int i;
    } s2obj;
};
union   U   uobj;       /* Size of 8 bytes */
union   U1  u1obj;      /* Size of 5 bytes */
union   U2  u2obj;      /* Size of 6 bytes */
```

Example 2

```
union U {
    int 7:i;
};
#pragma pack(1)
union U1 {
    int 7:i;
};
#pragma pack(2)
union U2 {
    int 7:i;
};
union   U   uobj;       /* Size of 4 bytes */
union   U1  u1obj;      /* Size of 1 byte */
union   U2  u2obj;      /* Size of 2 bytes */
```

3.11.4 Bit field

Data is allocated to the area of the bit field element as follows.

(1) When the structure packing value is equal to or larger than the alignment condition value of the member type

Data is allocated in the same manner as when the structure packing function is not used. That is, if the data is allocated consecutively and the resulting area exceeds the boundary that satisfies the alignment condition of the element type, data is allocated from the area satisfying the alignment condition.

(2) When the structure packing value is smaller than the alignment condition value of the element type

(a) If data is allocated consecutively and results in the number of bytes including the area becoming larger than the element type

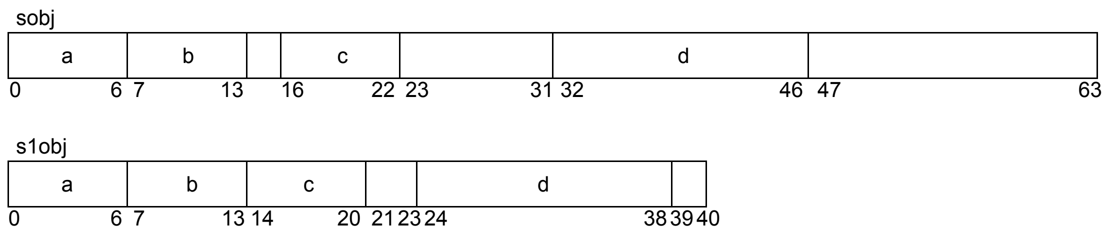
The data is allocated in a form that satisfies the alignment condition of the structure packing value.

(b) Other conditions

The data is allocated consecutively

Example

```
struct S {
    short   a : 7;      /* 0 to 6th bit */
    short   b : 7;      /* 7 to 13th bit */
    short   c : 7;      /* 16 to 22nd bit (aligned to 2-byte boundary) */
    short   d : 7;      /* 32 to 46th bit (aligned to 2-byte boundary) */
} sobj;
#pragma pack (1)
struct S1 {
    short   a : 7;      /* 0 to 6th bit */
    short   b : 7;      /* 7 to 13th bit */
    short   c : 7;      /* 14 to 20th bit */
    short   d : 15;     /* 24 to 38th bit (aligned to byte boundary) */
} s1obj;
```



3.11.5 Alignment condition of top structure object

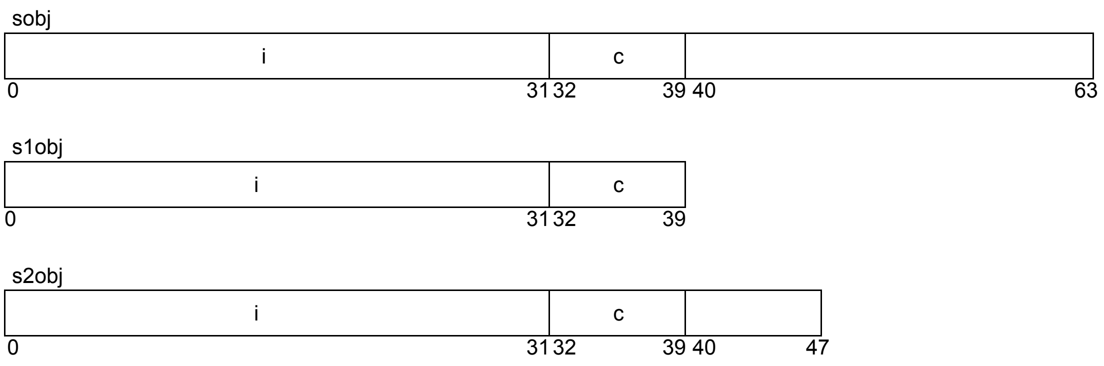
The alignment condition of the top structure object is the same as when the structure packing function is not used.

3.11.6 Size of structure objects

Perform padding so that the size of structure objects becomes a multiple value of whichever is the smaller value: the structure alignment condition value or the structure packing value. The alignment condition of the top structure object is the same as when the structure packing function is not used.

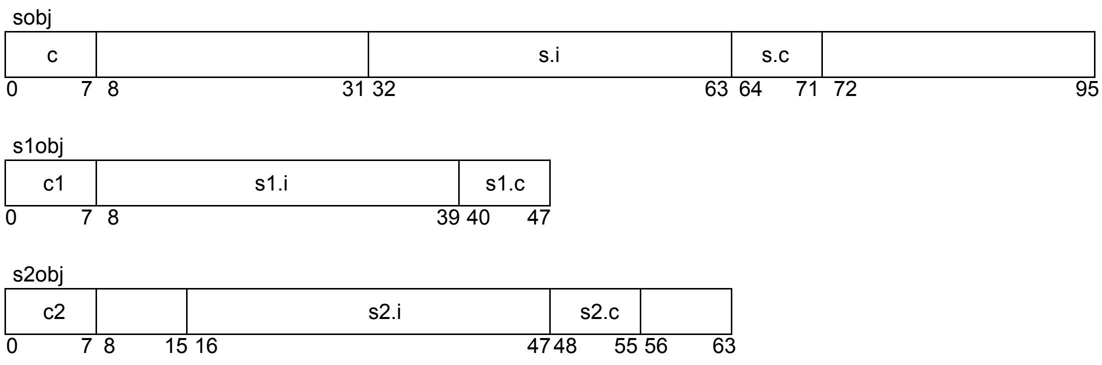
Example 1

```
struct S {
    int     i;
    char    c;
};
#pragma pack(1)
struct S1 {
    int     i;
    char    c;
};
struct S    sobj;       /* Size of 8 bytes */
struct S1   s1obj;      /* Size of 5 bytes */
struct S2   s2obj;      /* Size of 6 bytes */
```



Example 2

```
struct S {
    int     i;
    char    c;
};
struct T {
    char    c;
    struct S s;
};
#pragma pack(1)
struct S1 {
    int     i;
    char    c;
};
struct T1 {
    char    c;
    struct S1 s1;
};
#pragma pack(2)
struct S2 {
    int     i;
    char    c;
};
struct T2 {
    char    c;
    struct S2 s2;
};
struct T    tobj;       /* Size of 12 bytes */
struct T1   t1obj;      /* Size of 6 bytes */
struct T2   t2obj;      /* Size of 8 bytes */
```

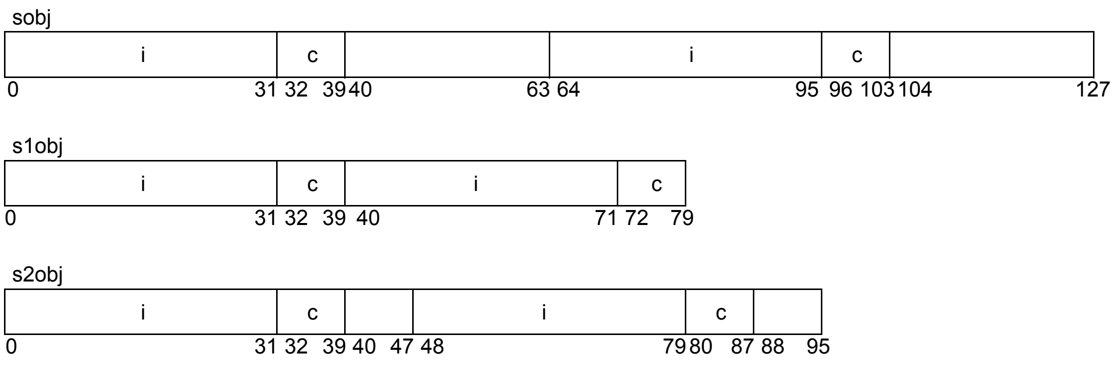


3.11.7 Size of structure array

The size of the structure object array is a value that is the sum of the number of elements added to the size of structure object.

Example

```
struct S {
    int     i;
    char    c;
};
#pragma pack(1)
struct S1 {
    int     i;
    char    c;
};
#pragma pack(2)
struct S2 {
    int     i;
    char    c;
};
struct S    sobj[2];        /* Size of 16 bytes */
struct S1   s1obj[2];       /* Size of 10 bytes */
struct S2   s2obj[2];       /* Size of 12 bytes */
```

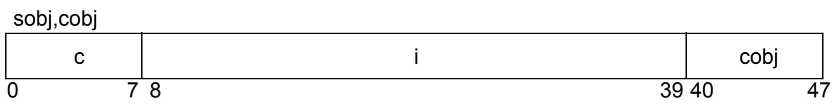


3.11.8 Area between objects

For example, sobj.c, sobj.i, and cobj may be allocated consecutively without a gap in the following source program (the allocation order of sobj and cobj is not guaranteed).

Example

```
#pragma pack(1)
struct S {
    char    c;
    int     i;
} sobj;
char    cobj;
```



3.11.9 Notes concerning structure packing function

(1) Specification of -Xpack option and #pragma pack directive at the same time

If the -Xpack option is specified when structure packing is specified with the #pragma directive in the C language source, the specified option value is applied to all the structures until the first #pragma pack directive appears. After this, the value of the #pragma directive is applied.

Even after the #pragma directive appears, however, the specified option value is applied to the area specified as default

Example

```
/* When specify -Xpack = 2 */

struct S2 { ... };  /* Packing value is specified as 2 in option */
                    /* Option -Xpack = 2 is valid: packing value is 2 */

#pragma pack(1)     /* Packing is specified as 1 in #pragma directive */

struct S1 { ... };  /* Pragma pack (1) is valid: packing value is 1 */

#pragma pack( )     /* Packing value is specified as default in #pragma directive */

struct S2_2 { ... };/* Option -Xpack = 2 is valid: packing value is 2 */
```

(2) Restrictions

When using the V850 microcontrollers and a CPU that is set to disable misalign access for V850Ex products, the following restrictions apply.

(a) Access using the structure member address cannot be executed correctly.

As shown in the following example, the structure member address is acquired, and the access to that address is then performed with the address masked in accordance with the data alignment of the device. Therefore, some data may disappear or be rounded off.

Example

```
struct test {
    char    c;          /* offset 0 */
    int     i;          /* offset 1-4 */
} test;

int *ip ,i;

void func(void)
{
    i = *ip;            /* Accessed with address masked */
}

void func2(void)
{
    ip = &(test.i);     /* Acquire structure member address */
}
```

(b) In bit field access, an area with no data to be read using the member's type is also accessed.

If the width of the bit field is smaller than the member's type as shown in the following example, access occurs outside the object because reading is performed using the member's type.

Generally, there is no problem with the function, but if I/O are mapped, an illegal access may occur.

Example

```
struct S {
    int x : 21;
} sobj;             /* 3byte */

sobj.x = 1
```

3.12 Binary Constants

The CA850 can handle integer constants in binary form.

A binary constant is a string that consists of "0b" or "0B" followed by one or more "0" or "1".

Example

```
0b00010110111101010111111010010111
```

Note Binary constants cannot be used when the -ansi option is specified.

CHAPTER 4 CALLING PROGRAM

This chapter explains how to handle arguments when a program is called by the CA850.

4.1 Calling Between C Functions

- Normal function call

  → jarl instruction

- Function call using a pointer indicating a function (and returning from function call)

  → jmp instruction (dispose instruction **[V850E]**)

When a C function is called from another C function, a 4-word argument is stored in the argument registers (r6 to r9). An argument in excess of 4 words is stored in the stack frame of the calling function. Control is then transferred (jumps) to the called function and the value in the argument registers stored when the function was called is stored in the stack frame of the calling function.

The stack frame is generated when the prologue code of the function, i.e., the code that is executed before the code of the main body of the function is called (processing (4) to (7) in Figure 4 - 3 and Figure 4 - 5 is the prologue code), is executed and the stack pointer (sp) is shifted by the necessary size. The stack frame disappears when the epilogue code of the function, i.e., the code that is executed after the code of the main body of the function is executed and until control returns to the calling function, is executed and the stack pointer (sp) is returned.

4.1.1 Stack frame/function call

This section explains the stack frame format and how the stack frame is generated and disappears when a function is called.

(1) Stack frame format

The CA850 allocates the argument register area to either the beginning of the stack or center of the stack in the stack frame, according to the argument condition. The argument conditions are as follows.

(a) When the argument register area is allocated to the beginning of the stack

The argument register area is allocated to the beginning of the stack when the area is accessed successively, exceeding the area for the 4-word argument, in the following two cases.

- If the number of arguments is variable
- If the argument is the entity of a structure and its area extends over a 4-word area

(b) When the argument register area is allocated to the center of the stack

The argument register area is allocated to the center of the stack under conditions other than (a).

Figure 4 - 1 shows the stack frame when the argument register area is at the beginning of the stack, and Figure 4 - 2 shows the stack frame when the argument register area is at the center of the stack.

Figure 4 - 1 Stack Frame (When Argument Register Area Is Located at Center of Stack)

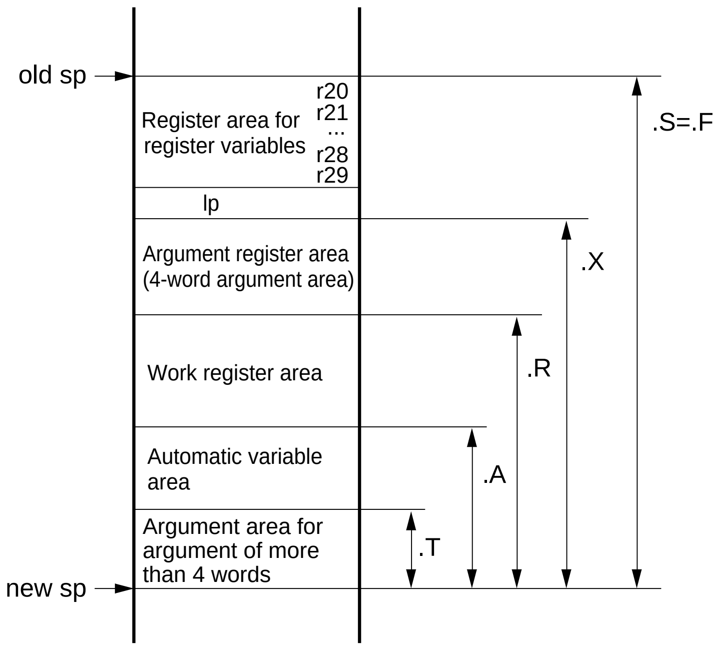
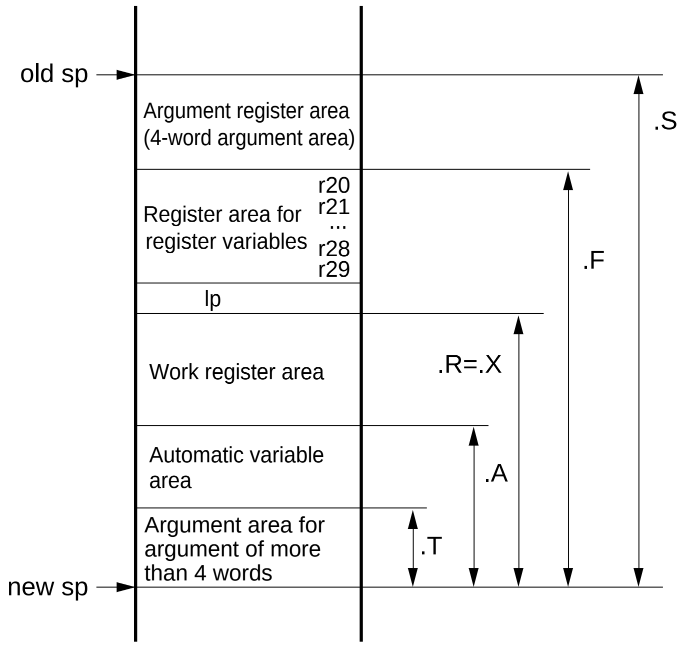

old sp
Register area for register variables
r20
r21
...
r28
r29
lp
Argument register area (4-word argument area)
Work register area
Automatic variable area
Argument area for argument of more than 4 words
new sp
.S=.F
.X
.R
.A
.T

Figure 4 - 2 Stack Frame (When Argument Register Area Is Located at Beginning of Stack)


old sp
Argument register area (4-word argument area)
Register area for register variables
r20
r21
...
r28
r29
lp
Work register area
Automatic variable area
Argument area for argument of more than 4 words
new sp
.S
.F
.R=.X
.A
.T

".S, .F, .X, .R, .A, and .T" in the figure are macros for functions output by the compiler internally. These macros are used for a specific purpose, as shown in Table 4 - 1.

Table 4 - 1 Meanings of Macros for Functions

Macro Name	Meaning
.S	Stack size
.F	Stack size - Size of argument register area (if at the beginning of the stack)
.X	Size of argument register area (if at the center of the stack) + .R
.R	Size of work register area + .A + .T
.A	Size of automatic variable area + .T
.T	Size of area for arguments of function to be called in excess of 4 words
.P	Always 0 (macro for code generation)Note

Note .P is not shown in Figure 4 - 1 and Figure 4 - 2 because it is always 0.

These macros are used to access the stack area. Table 4 - 2 shows specific access methods (access codes to be output).

Table 4 - 2 Method of Accessing Stack Area

Stack Area	Access Method (Displacement [sp])
Register area for register variables (including lp)	-offset+.Fxx[sp]
Work register area	-offset+.Rxx[sp]
Automatic variable area	-offset+.Axx[sp]
Area for arguments in excess of 4 words	offset+.Pxx[sp]
Argument register area (if at the beginning of the stack)	offset+.Fxx[sp]
Argument register area (if at the center of the stack)	offset+.Rxx[sp]

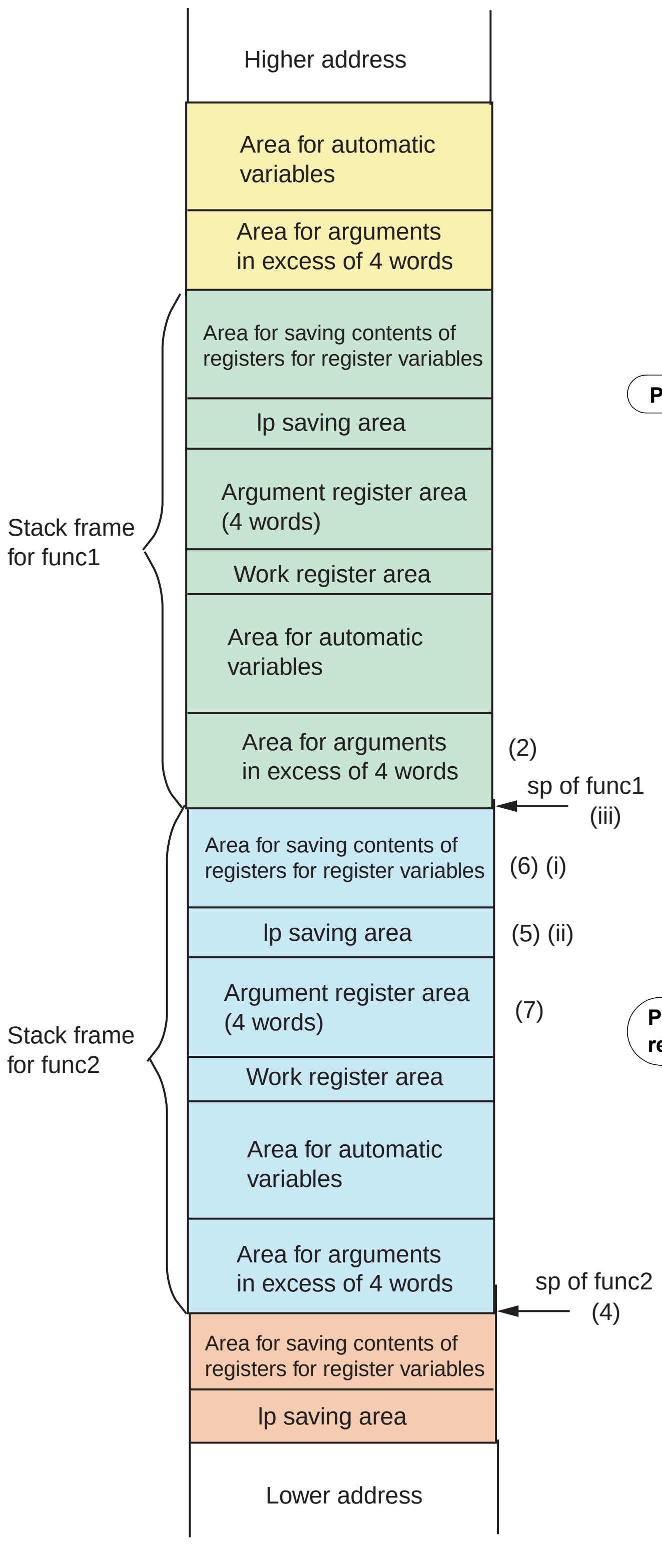
"offset" in this table is a positive integer and means the offset in each area. "xx" after a macro is a positive integer and indicates the frame number of the function.

(2) Generation/disappearance of stack frame when function is called (when argument register area is at center of stack)

The following explains the generation and disappearance of the stack frame when a function is called if the argument register area is at the center of the stack.

This case applies to most function calls. Figure 4 - 3 shows an example of the generation/disappearance of the stack frame when the function "func2()" is called from the function "func1()" and then execution returns to "func1()".

Figure 4 - 3 Generation/Disappearance of Stack Frame
(When Argument Register Area Is Located at Center of Stack)



Higher address
Area for automatic variables
Area for arguments in excess of 4 words
Area for saving contents of registers for register variables
lp saving area
Argument register area (4 words)
Work register area
Area for automatic variables
Area for arguments in excess of 4 words
Stack frame for func1
(2)
sp of func1
(iii)
Area for saving contents of registers for register variables
(6) (i)
lp saving area
(5) (ii)
Argument register area (4 words)
(7)
Work register area
Area for automatic variables
Area for arguments in excess of 4 words
Stack frame for func2
sp of func2
(4)
Area for saving contents of registers for register variables
lp saving area
Lower address

Processing on func1() side when func2() is called

(1) The arguments are stored in the argument registers.
The arguments of func2 to be called are stored in r6 to r9.

(2) The arguments in excess of 4 words are stored in the stack.
The excess arguments that cannot be stored in r6 to r9 are stored in the stack. This processing is performed if the number of arguments is five or more.

(3) Execution branches to func2() by the jarl instruction.

Processing on func2() side when called by func1

(4) sp is shifted.
The stack pointer moves to the stack to be used by func2.

(5) lp is saved.
The return address of func1() is stored.

(6) Register variable registers are saved.
These registers are saved because the register values used by func1 must be retained when func2 also uses the register variable registers.

(7) Arguments in argument register area are stored.
The values of r6 to r9 are stored. The current argument values are stored in the stack because when another function is called from func2, the arguments at that time are stored in registers r6 to r9.

[Note]Since the V850Ex can perform processing (4), (5), and (6) with the prepare instruction, the CA850 outputs the prepare instruction.

Processing on func2() side when execution returns from func2() to func1()

(i) The contents of the registers for register variables are restored.
The values of the register variable registers of func1() is restored to registers.

(ii) lp is restored.
The return address of func1() is restored.

(iii) sp is returned.
The stack pointer moves back to the stack to be used by func1().

(iv) Execution is returned by the jmp [lp] instruction.

[Note]Since the V850Ex can perform processing (i), (ii), (iii), and (iv) with the dispose instruction, the CA850 outputs the dispose instruction.

The items saved to the stack frame and the stack frame to be used are summarized below.

(a) Calling side - func1() -

- The values of the excess arguments are called if the arguments of func2() to be called exceed 4 words

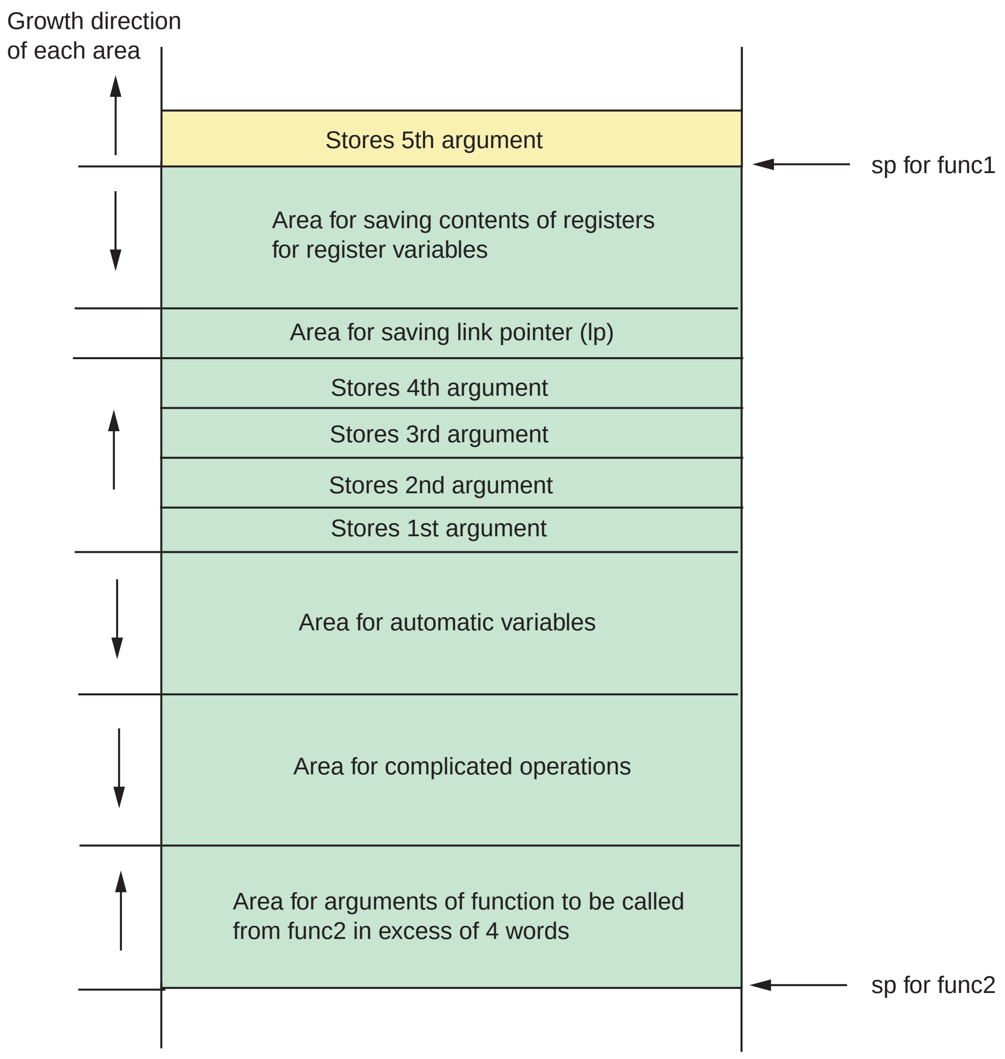
(b) Called side - func2() -

- Passing the arguments stored in the argument registers
  (The calling side (func1()) stores the argument in the register.)
- Saving the link pointer (lp) (= return address of func1()) of the calling side (func1())
- Saving the contents of the register variable registers
- The register variable registers are allocated as follows.
  In 22-register mode: r25, r26, r27, r28, r29
  In 26-register mode: r23, r24, r25, r26, r27, r28, r29
  In 32-register mode: r20, r21, r22, r23, r24, r25, r26, r27, r28, r29
  Of these registers, those that are used are saved.
- Area for automatic variables
- Allocating an area used for operation if a very complicated expression is used in a function
  Although this area is not shown in Figure 4 - 3, it is allocated at the lower address of the area for automatic variables if it is necessary.

If the function has a return value, that value is stored in r10.

The location of each area of the stack frame and the image of the stack growth direction of each area are illustrated below (it is assumed that func2() to be called has five arguments).

Figure 4 - 4 Stack Growth Direction of Each Area of Stack Frame


Growth direction of each area
Stores 5th argument
sp for func1
Area for saving contents of registers for register variables
Area for saving link pointer (lp)
Stores 4th argument
Stores 3rd argument
Stores 2nd argument
Stores 1st argument
Area for automatic variables
Area for complicated operations
Area for arguments of function to be called from func2 in excess of 4 words
sp for func2

An example of a source calling a C function from a C function and an assembly source when that source is compiled is shown below.

Example

```
void func1(void)
{
    int     a, b, c, d, e;
    func2(a, b, c, d, e);
        :

}
int func2(int a, int b, int c, int d, int e)
{
    register int    i;
        :

    return i;
}
```

Assembler instructions generated when func2 is called in example above

[V850]

```
_func1:
    jbr .L3
.L4:
    ld.w    -8+.A3 [sp], r6
    ld.w    -12+.A3 [sp], r7
    ld.w    -16+.A3 [sp], r8 --- (1)
    ld.w    -20+.A3[sp], r9
    ld.w    -24+.A3[sp], r10
    st.w    r10, [sp]    --- (2)
    jarl    _func2, lp  --- (3)
        :
    -- epilogue for main
    -- processing from (ii) to (iv)
.L3:
    -- prologue for main
    -- processing from(4) and (5)
        :
jbr .L4

_func2:
    jbr .L5
.L6:
    st.w    r6, .R2[sp]
    st.w    r7, 4+.R2[sp]
    st.w    r8, 8+.R2[sp] --- (7)
    st.w    r9, 12+.R2[sp]
    st.w    r29, -4+.A2[sp]
        :
    jbr     .L2
.L2:
    ld.w    -4+.A2[sp], r10
    ld.w    -4+.F2[sp], r29 ---  (i)
    ld.w    -8+.F2[sp], lp  --- (ii)
    add     .F2, sp         ---  (iii)
    jmp [lp]        ---  (iv)

.L5:
    add     -.F2, sp    ---  (4)
    st.w    lp, -8+.F2[sp] --- (5)
    st.w    r29, -4+.F2[sp] ---  (6)
    jbr     .L6
```

[V850E]

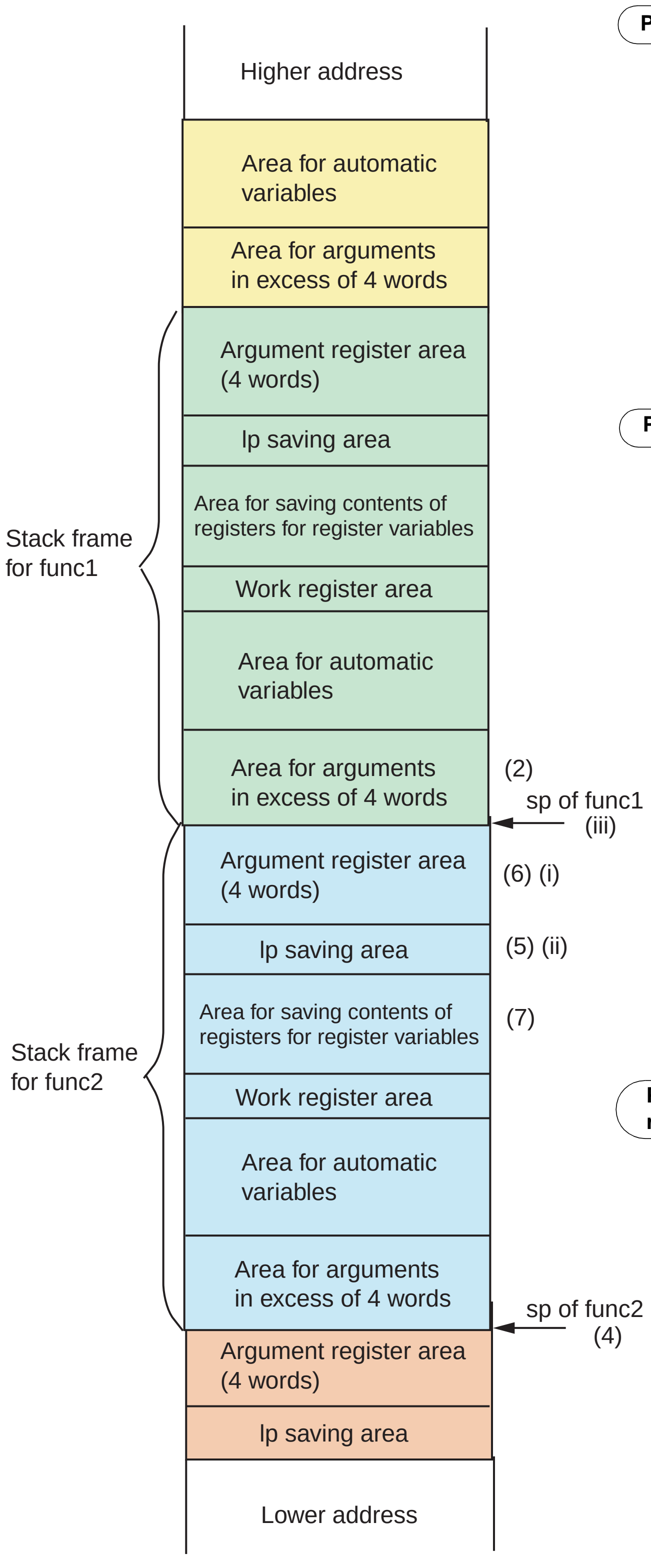
```
_func1:
    jbr .L3
.L4:
    ld.w    -8+.A3[sp], r6
    ld.w    -12+.A3[sp], r7
    ld.w    -16+.A3[sp], r8 ---  (1)
    ld.w    -20+.A3[sp], r9
    ld.w    -24+.A3[sp], r10
    st.w    r10, [sp]    --- (2)
    jarl    _func2, lp  ---  (3)
        :
    -- epilogue for main
    -- processing from (ii) to (iv)
.L3:
    -- prologue for main
    -- processing from(4) and (5)
        :
    jbr .L4

_func2:
    jbr .L5
.L6:
    st.w    r6, .R2[sp]
    st.w    r7, 4+.R2[sp]
    st.w    r8, 8+.R2[sp] --- (7)
    st.w    r9, 12+.R2[sp]
    st.w    r29, -4+.A2[sp]
        :
    jbr     .L2
.L2:
    ld.w    -4+.A2[sp], r10
    dispose .X2, 0x3, [lp]
    --(i), (ii), (iii), (iv)



.L5:
    prepare 0x3, .X2
    --(4), (5), (6)
    jbr     .L6
```

(3) Generation/disappearance of stack frame when function is called (when argument register area is at beginning of stack)

The following explains the generation and disappearance of the stack frame when a function is called if the argument register area is at the beginning of the stack.

Figure 4 - 5 shows an example of the generation/disappearance of the stack frame when the function "func2()" is called from the function "func1()" and then execution returns to "func1()".

Figure 4 - 5 Generation/Disappearance of Stack Frame
(When Argument Register Area Is Located at Beginning of Stack)



Higher address
Area for automatic variables
Area for arguments in excess of 4 words
Argument register area (4 words)
lp saving area
Area for saving contents of registers for register variables
Work register area
Area for automatic variables
Area for arguments in excess of 4 words
Stack frame for func1
(2)
sp of func1
(iii)
Argument register area (4 words)
(6) (i)
lp saving area
(5) (ii)
Area for saving contents of registers for register variables
(7)
Work register area
Area for automatic variables
Area for arguments in excess of 4 words
Stack frame for func2
sp of func2
(4)
Argument register area (4 words)
lp saving area
Lower address

Processing on func1() side when func2() is called

(1) The arguments are stored in the argument registers.
The arguments of func2 to be called are stored in r6 to r9.

(2) The arguments in excess of 4 words are stored in the stack.
The excess arguments that cannot be stored in r6 to r9 are stored in the stack. This processing is performed if the number of arguments is five or more.

(3) Execution branches to func2() by the jarl instruction.

Processing on func2() side when called by func1

(4) sp is shifted.
The stack pointer moves to the stack to be used by func2.

(5) lp is saved.
The return address of func1() is stored.

(6) Register variable registers are saved.
These registers are saved because the register values used by func1 must be retained when func2 also uses the register variable registers.

(7) Arguments in argument register area are stored.
The values of r6 to r9 are stored. The current argument values are stored in the stack because when another function is called from func2, the arguments at that time are stored in registers r6 to r9.

[Note]Since the V850Ex can perform processing (4), (5), and (6) with the prepare instruction, the CA850 outputs the prepare instruction.

Processing on func2() side when execution returns from func2() to func1()

(i) The contents of the registers for register variables are restored.
The values of the register variable registers of func1() is restored to registers.

(ii) lp is restored.
The return address of func1() is restored.

(iii) sp is returned.
The stack pointer moves back to the stack to be used by func1().

(iv) Execution is returned by the jmp [lp] instruction.

[Note]Since the V850Ex can perform processing (i), (ii), (iii), and (iv) with the dispose instruction, the CA850 outputs the dispose instruction.

The items that are saved to the stack frame and the stack frame to be used are summarized below.

(a) Calling side - func1() -

- The values of the excess arguments are called if the arguments of func2() to be called exceed 4 words

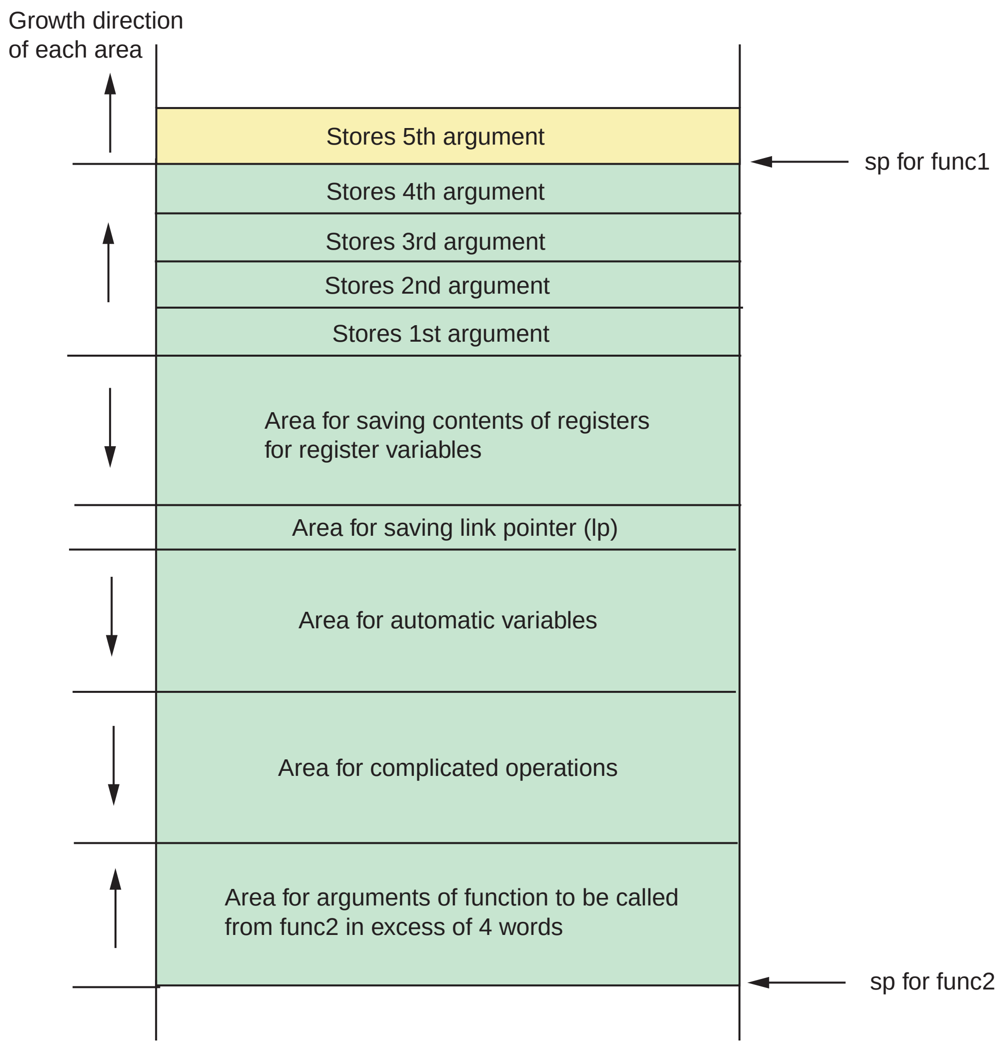
(b) Called side - func2() -

- Passing arguments stored in argument registers
  (The calling side (func1()) stores the arguments in the registers.)
- Saving the link pointer (lp) (= return address of func1()) of the calling side (func1())
- Saving contents of register variable registers
- Area for automatic variables
- Allocating an area used for operation if a very complicated expression is used in a function
  Although this area is not shown in Figure 4 - 3, it is allocated at the lower address of the area for automatic variables if it is necessary.

If the function has a return value, it is stored in r10.

The location of each area of the stack frame and the image of the stack growth direction of each area are illustrated below (it is assumed that func2() to be called has five arguments).

Figure 4 - 6 Stack Growth Direction of Each Area of Stack Frame


Growth direction
of each area
Stores 5th argument
sp for func1
Stores 4th argument
Stores 3rd argument
Stores 2nd argument
Stores 1st argument
Area for saving contents of registers for register variables
Area for saving link pointer (lp)
Area for automatic variables
Area for complicated operations
Area for arguments of function to be called from func2 in excess of 4 words
sp for func2

An example of a source calling a C function from a C function and an assembly source when that source is compiled is shown below.

Example

```
void func1(void)
{
    int a, b, c, d, e;

    func2(a, b, c, d, e);
        :
}

int func2(int a, int b, int c, int d, int e)
{
    register int    i;
        :

    return i;
}
```

Assembler instructions generated when func2 is called in example above

[V850]

```
_func1:
    jbr .L3
.L4:
    ld.w    -8+.A3[sp], r6
    ld.w    -12+.A3[sp], r7
    ld.w    -16+.A3[sp], r8 --- (1)
    ld.w    -20+.A3[sp], r9
    ld.w    -24+.A3[sp], r10
    st.w    r10, [sp]        --- (2)
    jarl    _func2, lp       --- (3)
        :
    -- epilogue for main
    -- processing from (ii) to (iv)
.L3:
    -- prologue for main
    -- processing (4) and (5)
        :
    jbr .L4

_func2:
    jbr .L5
.L6:
    st.w    r6, .F2[sp]
    st.w    r7, 4+.F2[sp]
    st.w    r8, 8+.F2[sp] --- (7)
    st.w    r9, 12+.F2[sp]
        :
    st.w    r29, -4+.A2[sp]
    jbr .L2
.L2:
    ld.w    -4+.A2[sp], r10
    ld.w    -4+.F2[sp], r29 --- (i)
    ld.w    -8+.F2[sp], lp --- (ii)
    add     .S2, sp          --- (iii)
    jmp     [lp]             --- (iv)
.L5:
    sub     -.S2, sp         --- (4)
    st.w    lp, -8+.F2[sp] --- (5)
    st.w    r29, -4+.F2[sp] --- (6)
    jbr     .L6
```

[V850E]

```
_func1:
    jbr .L3
.L4:
    ld.w    -8+.A3[sp], r6
    ld.w    -12+.A3[sp], r7
    ld.w    -16+.A3[sp], r8 --- (1)
    ld.w    -20+.A3[sp], r9
    ld.w    -24+.A3[sp], r10
    st.w    r10, [sp]        --- (2)
    jarl    _func2, lp       --- (3)
        :
    -- epilogue for main
    -- processing from (ii) to (iv)
.L3:
    -- prologue for main
    -- processing (4) and (5)
        :
    jbr .L4

_func2:
    jbr .L5
.L6:
    st.w    r6, .F2[sp]
    st.w    r7, 4+.F2[sp]
    st.w    r8, 8+.F2[sp] --- (7)
    st.w    r9, 12+.F2[sp]
        :
    st.w    r29, -4+.A2[sp]
    jbr .L2
.L2:
    ld.w    -4+.A2[sp], r10
    dispose .X2, 0x3
    -- (i), (ii), (iii)
    add     .S2-.F2, sp      --- (iii)
    jmp     [lp]             --- (iv)
.L5:
    add     .F2 -.S2, sp     --- (4)
    prepare 0x3, .X2
    --- (4), (5), (6)
    jbr .L6
```

4.2 Calling Between C Function and Assembler Function

This section explains the points to be note when an assembler function is called by a C function or vice versa.

4.2.1 Calling assembler function from C function

Note the following points when calling an assembler function from a C function.

(1) Identifier

If external names, such as functions and external variables, are described in the C language source by the CA850, they are prefixed with "_" (underscore) when they are output to the assembler.

Table 4 - 3 Identifier

C	Assembler
`func1()`	`_func1`

Prefix "_" to the identifier when defining functions and external variables with the assembler. Remove "_" when referencing them from a C function.

(2) Stack frame

The CA850 outputs codes on the assumption that the stack pointer (sp) always indicates the lowest address of the stack frame. Therefore, the address area lower than the address indicated by sp can be freely used in the assembler function after branching from a C language source to an assembler function. Conversely, if the contents of the higher address area are changed, the area used by a C function may be lost and the subsequent operation cannot be guaranteed. To avoid this, change sp at the beginning of the assembler function before using the stack. At this time, however, make sure that the value of sp is retained before and after calling. When using a register variable register in an assembler function, make sure that the register value is retained before and after the assembler function is called. In other words, save the value of the register variable register before calling the assembler function, and restore the value after calling.

The register variable registers that can be used differ depending on the register mode.

Table 4 - 4 Registers for Register Variables

Register Mode	Registers for Register Variables
22-register mode	r25, r26, r27, r28, r29
26-register mode	r23, r24, r25, r26, r27, r28, r29
32-register mode	r20, r21, r22, r23, r24, r25, r26, r27, r28, r29

(3) Argument passed to assembler function

The CA850 stores 4-word arguments in argument registers r6 to r9 and arguments in excess of 4 words in the stack frame on the calling side (refer to "4.1.1 Stack frame/function call" for details). Reference each stored value when using an argument value in an assembler function. An argument value in a C function is the value itself that is specified as an argument. The operation of the C function is not affected even if this value is changed in an assembler function.

(4) Return value returned from assembler function

The CA850 generates codes on the assumption that the return value of a function is stored in the r10 register. Store the value returned from an assembler function in r10. If the function returns a structure, the return value, i.e., the structure, is stored in the stack frame of the calling function.

(5) Return address passed to C function

The CA850 generates codes on the assumption that the return address of a function is stored in link pointer lp (r31). When execution branches to an assembler function, the return address of the function is stored in lp. Execute the jmp [lp] instruction to return to a C function.

4.2.2 Calling C function from assembler function

Note the following points when calling a C function from an assembler function.

(1) Stack frame

The CA850 generates codes on the assumption that the stack pointer (sp) always indicates the lowest address of the stack frame. Therefore, set sp so that it indicates the higher address of an unused area of the stack area when execution branches from an assembler function to a C function. This is because the stack frame is allocated toward the lower addresses.

(2) Work register

The CA850 retains the values of the register variable registers before and after a C function is called but does not retain the values of the work registers. Therefore, do not leave a value that must be retained assigned to a work register.

The register variable registers and work registers that can be used differ depending on the register mode.

Table 4 - 5 Registers for Register Variables

Register Mode	Registers for Register Variables
22-register mode	r25, r26, r27, r28, r29
26-register mode	r23, r24, r25, r26, r27, r28, r29
32-register mode	r20, r21, r22, r23, r24, r25, r26, r27, r28, r29

Table 4 - 6 Work Registers

Register Mode	Work Registers
22-register mode	r10, r11, r12, r13, r14
26-register mode	r10, r11, r12, r13, r14, r15, r16
32-register mode	r10, r11, r12, r13, r14, r15, r16, r17, r18, r19

(3) Argument passed to C function

The CA850 stores 4-word arguments in argument registers r6 to r9 and arguments in excess of 4 words in the stack frame of the calling function (refer to "4.1.1 Stack frame/function call" for details). Store the arguments in excess of 4 words upward from the address indicated by sp.

(4) Return value returned from C function

The CA850 generates codes on the assumption that the return value of a function is stored in the r10 register. Reference the r10 register when using the value returned from a C function. If the function returns a structure, a value is stored in an area for the return value of the calling function, and a code that passes the address of that area as an argument is output. An area for the return value must be allocated in advance on the calling side.

(5) Return address returned to assembler function

The CA850 generates codes on the assumption that the return address of a function is stored in link pointer lp (r31). When execution branches to a C function, therefore, the return address of the function must be stored in lp. Execution is generally branched to a C function using the jarl instruction.

4.3 Prologue/Epilogue Processing of Function

The CA850 can reduce the object size in part of the prologue/epilogue processing of a function by calling a runtime library. Because the prologue/epilogue processing of a function is predetermined, it is prepared as runtime library functions and these functions are called when a function is called or execution returns to a function.

An example of the assembler code of the prologue/epilogue processing of a function is shown below. Numbers in parentheses in this example correspond to those in Figure 4 - 3.

Example

```
int func(int a, int b, int c, int d, int e)
{
    register int    i;
        :
    return i;
}
```

Assembler instruction in prologue/epilogue processing of function "func" in above example

[Code when runtime library function is not used]

```
_func :
    jbr .L5
.L6 :
    st.w    r6, .R2[sp]
    st.w    r7, 4+.R2[sp]
    st.w    r8, 8+.R2[sp] --- (7)
    st.w    r9, 12+.R2[sp]
        :
    st.w    r29, -4+.A2[sp]
    jbr     .L2
.L2 :
    ld.w    -4+.A2[sp], r10
    ld.w    -4+.F2[sp], r29 --- (i)
    ld.w    -8+.F2[sp], lp  --- (ii)
    add     .F2, sp         --- (iii)
    jmp     [lp]            --- (iv)
.L5:
    add     -.F2, sp        ---  (4)
    st.w    lp, -8+.F2[sp]  ---  (5)
    st.w    r29, -4+.F2[sp] ---  (6)
    jbr     .L6
```

[Code when runtime library function is used]

```
_func :
    jbr .L5
.L6 :
    st.w    r6, .R2[sp]
    st.w    r7, 4+.R2[sp]
    st.w    r8, 8+.R2[sp] --- (7)
    st.w    r9, 12+.R2[sp]
        :
    st.w    r29, -4+.A2[sp]
    jbr     .L2
.L2 :
    ld.w    -4+ .A2[sp], r10
    add     .R2, sp          --- (iii)
    jarl    ___pop2904, lp
    -- (i), (ii), (iii), (iv)
.L5 :
    jarl    ___push2904, r10
    -- (4), (5), (6)
    add     -.R2, sp         --- (4)
    jbr     .L6
```

4.3.1 Specifying use of runtime library function for prologue/epilogue of function

Specify the compiler option "-Xpro_epi_runtime=on" to call the runtime library for prologue/epilogue. Specify "-Xpro_epi_runtime=off" if the runtime library is not called. When an optimization option other than "-Ot (execution speed priority optimization)" is specified, however, the runtime library is automatically called for the prologue/epilogue of a function. That is, the compiler option "-Xpro_epi_runtime=on" is automatically specified.

If an option other than "-Ot" is specified and if a runtime library should not be called, specify the "-Xpro_epi_runtime=off" option. The "-Xpro_epi_runtime" option can be specified in each source file, so a file for which the runtime library is called and a file for which the runtime library is not called can be used together.

When a runtime library is called for the prologue/epilogue of a function by specifying the "-Xpro_epi_runtime=on" option, a dedicated section ".pro_epi_runtime" is necessary. Consequently, the following definition must be described by a link directive.

```
.pro_epi_runtime = $PROGBITS ?AX .pro_epi_runtime;
```

Table information of the prologue/epilogue runtime function is allocated to this section.

4.3.2 Calling runtime library for prologue/epilogue of function in V850Ex

When the V850Ex is used, the following instruction is used to call the prologue/epilogue runtime function of a function.

```
CALLT instruction
```

The CALLT instruction is a 2-byte instruction. The code size can be reduced by using this instruction for calling a function. The CALLT instruction requires a pointer that indicates that the table of the function subject to the CALLT instruction is set to the CTBP (Callt Base Pointer) register. If processing of the setting is missing from the program, the following error message is output during linking.

```
F4414: CallTBasePointer(CTBP) is not set. CTBP must be set when compileroption
"-Ot" (or "-Xpro_epi_runtime=off") is not specified.
```

Since setting a value to the CTBP register is performed by an assembler instruction, it should be performed in the startup routine. Add the following instruction to the startup routine.

```
    mov     #___PROLOG_TABLE, r12 -- three underscores "_" before "PROLOG"
    ldsr    r12, 20
```

At this time, ___PROLOG_TABLE is the first symbol of the function table of the runtime function of the prologue/epilogue of a function, and the function table itself is allocated to the ".pro_epi_runtime" section. The r12 register is used in the above example, but it is not always necessary to use r12. If the CALLT instruction provided in the CA850 is used for any purpose other than calling a runtime library for the prologue/epilogue of a function, the CTBP register contents must be saved/restored. If the CALLT instruction is used by another object, such as middleware or a user-created library, and if a code that saves/restores the CTBP register contents is missing or cannot be inserted in that object, a runtime library for the prologue/epilogue of a function cannot be called. In this case, suppress calling the runtime library by specifying the "-Xpro_epi_runtime=off" option.

Refer to the Relevant Device’s Architecture User’s Manual of each device for details of the CALLT instruction and the CTBP register.

4.3.3 Notes on calling runtime library for prologue/epilogue of function

Note the following points when calling a runtime library for the prologue/epilogue of a function.

(1) Calling a runtime library for the prologue/epilogue of a function degrades the execution speed because a function is called. Specify the "-Xpro_epi_runtime=off" option to avoid this. Specifying this option in file units is effective.

(2) In the case of a program in which few functions are called, the code size may not be reduced even if a runtime library is called for the prologue/epilogue. If no real effect can be expected, specify the "-Xpro_epi_runtime=off" option.

(3) A runtime library is not called for the prologue/epilogue of an interrupt function.
However, a function called from an interrupt function is subject to runtime library calling.

4.4 Far Jump Function

The CA850 outputs a code using the jarl instruction when a function is called.

```
jarl    _func1, lp
```

The architecture allows only a sign-extended value of up to 22 bits (22-bit displacement) to be specified as the first operand of the jarl instruction. This means that, if the branch destination is not within a 1 MB range from the branch point, branching cannot take place and the linker outputs the following error message.

```
F4161:symbol " function-name"(output section : section-name) is too far from
output section " section-name".(value : disp-value, file : main.o, input sec-
tion : .text, offset: offset-value, type : R_V850_PC22 ).
```

This can be solved easily by allocating **the branch destination within a 1 MB range from the branch point**.

However, the branch destination may not be able to be located within this range depending on target system. The "far jump" function solves this.

When the far jump function is used, a code that uses the jmp instruction is output when a function is called. As a result, execution can branch to the entire 32-bit space of the V850. Function calling using the far jump function is called "far jump calling".

4.4.1 Specifying far jump

When calling a function using the far jump function, prepare a file in which functions to be called by the far jump function are enumerated (file listing functions to be called by the far jump function), and use the compiler option "-Xfar_jump".

```
-Xfar_jump file listing functions to be called by far jump function
```

The "-Xfar_jump" option can also be used with "=" as follows.

```
-Xfar_jump=file listing functions to be called by far jump function
```

Refer to the next section for the format of the file listing the functions to be called by the far jump function.

4.4.2 File listing functions to be called by far jump function

This section explains the format of the file that enumerates the functions to be called by using the far jump function. Describe one function to which the far jump function is applied in one line. Describe a C function name with "_" (underscore) prefixed.

Sample of file listing functions to be called by far jump

```
_func_led
_func_beep
_func_motor
  :
  :
_func_switch
```

If the following description is made instead of "_function-name", all the functions are called using the far jump function.

```
{all_function}
```

If {all_function} is specified, all the functions are called by the far jump function, even if "_function-name" is specified.

The far jump function can also be applied to the following functions, as well as to user functions.

- Standard library functions
- Runtime library functions
- Prologue/epilogue runtime function of function
- System calls of real-time OS

Refer to "4.4.3 Examples of using far jump function" for examples of specification.

Note the following points when describing the file listing the functions to be called by the far jump function.

- Only ASCII characters can be used.
- Comments must not be inserted.
- Describe only one function in one line.
- A blank and tab may be inserted before and after a function name.
- Up to 1,023 characters can be described in one line. A blank or tab is also counted as one character.
- Describe a C function name with "_" (underscore) prefixed to the function name.
- The far jump function cannot be used together with the re-link function of the flash memory/external ROM.

4.4.3 Examples of using far jump function

Examples of using the far jump function are shown below.

(1) User function (same applies to standard functions)

```
[C language source file]
extern void func3(void);

void func(void)
{
    func3();
}
```

```
[file listing functions to be called by far jump]
_func3
```

[Normal calling code]	[far jump calling code]

`#@CALL_ARG`
`jarl    _func3, lp`
	`#@CALL_ARG`
	`movea   #_func3, tp, r10`
	`movea   .L18, tp, lp`
	`jmp     [r10]`
	`.L18:`

(2) Runtime function (when calling a macro)

```
[file listing functions to be called by far jump]
___mul
```

```
[Normal calling code]
.macro mul arg1, arg2

    add     -8, sp
    st.w    r6, [sp]
    st.w    r7, 4[sp]
    mov     arg1, r6
    mov     arg2, r7
    jarl    ___mul, lp





    ld.w    4[sp], r7
    mov     r6, arg2
    ld.w    [sp], r6
    add     8, sp
.endm
```

```
[far jump calling code]
.macro mul arg1, arg2
    .local  macro_ret
    add     -8, sp
    st.w    r6, [sp]
    st.w    r7, 4[sp]
    mov     arg1, r6
    mov     arg2, r7
    movea   macro_ret, tp, r31
    .option nowarning
    movea   #___mul, tp, r1
    jmp     [r1]
    .option warning
macro_ret:
    ld.w    4[sp], r7
    mov     r6, arg2
    ld.w    [sp], r6
    add     8, sp
.endm
```

(3) Runtime function (direct calling)

```
[file listing functions to be called by far jump]
___mul
```

```
[Normal calling code]
    mov     r12, r6
    mov     r13, r7
    #@CALL_ARG  r6, r7
    #@CALL_USE  r6, r7
    jarl    ___mul, lp



    mov     r6, r13
```

```
[far jump calling code]
    mov     r12, r6
    mov     r13, r7
    #@CALL_ARG  r6, r7
    #@CALL_USE  r6, r7
    movea   #___mul, tp, r14
    movea   .L13, tp, lp
    jmp     [r14]
.L13 :
    mov     r6, r13
```

The compiler automatically selects whether a runtime macro is called or a runtime function is directly called by judging the register efficiency in the process of optimization.

(4) System calls of real-time OS

```
[file listing functions to be called by far jump]
_ext_tsk
```

```
[Normal calling code]
    #@B_EPILOGUE
    #@BEGIN_NO_OPT
    add     .S4, sp
    jr      _ext_tsk        -- C NR

    #@END_NO_OPT
    #@E_EPILOGUE
```

```
[far jump calling code]
    #@B_EPILOGUE
    #@BEGIN_NO_OPT
    add     .S4, sp
    movea   #_ext_tsk, tp, r10

    jmp     [r10]           -- C NR
    #@END_NO_OPT
    #@E_EPILOGUE
```

(5) Prologue/epilogue runtime function

```
[file listing functions to be called by far jump]
___pop2900
___push2900
```

```
[Normal calling code]
    #@B_EPILOGUE
    jarl    ___pop2900, lp -- 1

    #@E_EPILOGUE
.L3 :
    jarl    ___push2900, r10



    #@E_PROLOGUE
```

```
[far jump calling code]
    #@B_EPILOGUE
    movea   #___pop2900, tp, r11
    jmp     [r11] --1
    #@E_EPILOGUE
.L3 :
    movea   #___push2900, tp, r11
    movea   .L5, tp, r10
    jmp     [r11]
.L5 :
    #@E_PROLOGUE
```

Table 4 - 7 shows the prologue/epilogue function names that can be specified by the far jump function. Before specifying a prologue/epilogue runtime function, confirm the functions used in the assembly source output after compilation.

Table 4 - 7 List of Prologue/Epilogue Runtime Functions

___pop2000,	___pop2001,	___pop2002,	___pop2003,	___pop2004,	___pop2040,
___pop2100,	___pop2101,	___pop2102,	___pop2103,	___pop2104,	___pop2140,
___pop2200,	___pop2201,	___pop2202,	___pop2203,	___pop2204,	___pop2240,
___pop2300,	___pop2301,	___pop2302,	___pop2303,	___pop2304,	___pop2340,
___pop2400,	___pop2401,	___pop2402,	___pop2403,	___pop2404,	___pop2440,
___pop2500,	___pop2501,	___pop2502,	___pop2503,	___pop2504,	___pop2540,
___pop2600,	___pop2601,	___pop2602,	___pop2603,	___pop2604,	___pop2640,
___pop2700,	___pop2701,	___pop2702,	___pop2703,	___pop2704,	___pop2740,
___pop2800,	___pop2801,	___pop2802,	___pop2803,	___pop2804,	___pop2840,
___pop2900,	___pop2901,	___pop2902,	___pop2903,	___pop2904,	___pop2940,
___poplp00,	___poplp01,	___poplp02,	___poplp03,	___poplp04,	___poplp40,
___push2000,	___push2001,	___push2002,	___push2003,	___push2004,	___push2040,
___push2100,	___push2101,	___push2102,	___push2103,	___push2104,	___push2140,
___push2200,	___push2201,	___push2202,	___push2203,	___push2204,	___push2240,
___push2300,	___push2301,	___push2302,	___push2303,	___push2304,	___push2340,
___push2400,	___push2401,	___push2402,	___push2403,	___push2404,	___push2440,
___push2500,	___push2501,	___push2502,	___push2503,	___push2504,	___push2540,
___push2600,	___push2601,	___push2602,	___push2603,	___push2604,	___push2640,
___push2700,	___push2701,	___push2702,	___push2703,	___push2704,	___push2740,
___push2800,	___push2801,	___push2802,	___push2803,	___push2804,	___push2840,
___push2900,	___push2901,	___push2902,	___push2903,	___push2904,	___push2940,
___pushlp00,	___pushlp01,	___pushlp02,	___pushlp03,	___pushlp04,	___pushlp40

Refer to "4.3 Prologue/Epilogue Processing of Function" for details of the prologue/epilogue runtime library of functions.

CHAPTER 5 STARTUP ROUTINE

This chapter explains the startup routine.

5.1 Operation of Startup Routine

A startup routine is a routine that is executed after the V850 is reset and before the main function is executed. Basically, it performs initialization as follows after the system is reset. The specific operations are shown below.

- Setting RESET handler when a reset is input
- Setting register mode of startup routine
- Securing stack area and setting stack pointer
- Securing argument area of main function
- Setting text pointer (tp)
- Setting global pointer (gp)
- Setting element pointer (ep)
- Setting mask value to mask registers r20 and r21
- Initializing peripheral I/O registers that must be initialized before execution of main function
- Initializing user target that must be initialized before execution of main function
- Clearing sbss area to 0
- Clearing bss area to 0
- Clearing sebss area to 0
- Clearing tibss.byte area to 0
- Clearing tibss.word area to 0
- Clearing sibss area to 0
- Setting CTBP value for prologue/epilogue runtime library of functions **[V850E]**
- Setting programmable peripheral I/O register values **[V850E]**
- Setting r6 and r7 as argument of main function
- Branching to main function (when real-time OS is not used)
- Branching to initialization routine of real-time OS (when real-time OS is used)

Depending on the system, some of these operations may not be necessary and can be omitted. In addition to the above, the user can describe necessary processing. The above operations must be described basically with assembler instructions. How to describe each operation is explained below. Startup routine samples are provided in the CA850.

The samples are stored in the location shown in the following table.

Table 5 - 1 Startup Routine Samples

Storage Location	File Name	Meaning
Install Folder\lib850\r22	crtN.s	Startup routine sample for V850 core and for 22-register mode
	crtE.s	Startup routine sample for V850Ex core and for 22-register mode
Install Folder\lib850\r26	crtN.s	Startup routine sample for V850 core and for 26-register mode
	crtE.s	Startup routine sample for V850Ex core and for 26-register mode
Install Folder\lib850\r32	crtN.s	Startup routine sample for V850 core and for 32-register mode
	crtE.s	Startup routine sample for V850Ex core and for 32-register mode

If the startup routine is not added to the project, the CA850 automatically links a default startup routine (object). The file to be linked is as follows.

- Project for V850 core: crtN.o
- Project for V850Ex core: crtE.o

These files result from compiling (assembling) sample startup routines "crtN.s" and "crtE.s". These objects are assembled with the assembler options "-cn" and "-cnv850e" and can be used commonly in the V850 microcontrollers.

5.1.1 Setting RESET handler when reset is input

Describe the processing to be performed when a reset (reset interrupt) is input. Execution branches to the handler address 0x0 when a reset is input in the V850. Therefore, allocate an instruction that branches to the beginning of the startup routine to address 0x0. As explained in "3.7.4 Notes on describing interrupt/exception handler", describe a reset interrupt with assembler instructions because it cannot be described in C language by specifying the #pragma interrupt directive. The description is as follows.

```
    .section "RESET", text
    jr  __start

__start:
```

Use the .section quasi directive to allocate an instruction to the handler address. If the above description is made, the "jr __start" instruction is allocated to the handler address. If the jr instruction cannot reach the destination, i.e., if "__start" is not within ±1Mbytes from address 0x0, use the jmp instruction as follows.

```
    .section "RESET", text
    mov #__start, lp
    jmp [lp]

__start:
```

In this case, one register is used. The lp (r31) register is used in the above example. Any general-purpose register whose contents can be lost at this point can be used. The lp (r31) register in which the return address from a function is stored is not used when a reset is input. Therefore, it is safe to use the lp (r31) register.

The description of the .section quasi directive does not always have to be in the startup routine. In the above example, the symbol of the startup routine is "__start". This may be another name.

5.1.2 Setting register mode of startup routine

Describe the setting of the register mode in the startup routine described with assembler instructions. However, this setting is necessary only when the 22-register mode or 26-register mode is used for the overall system. It is not necessary to describe this setting when the 32-register mode is used.

(1) 22-register mode

```
    .option reg_mode 5 5
```

(2) 26-register mode

```
    .option reg_mode 7 7
```

If this setting is not described, the linker outputs the following warning message.

```
W4608: input files have different register modes. use -rc option for more
information.
```

5.1.3 Securing stack area and setting stack pointer (sp)

Secure the stack area used by the system and set the stack pointer (sp = r3) at the beginning of this area. When a real-time OS is used, however, the stack specified here is used until execution branches to the initialization routine of the real-time OS. In other words, it is hardly used or not used at all. If a large stack area is secured, therefore, the RAM area is wasted. Check if the stack is used before execution branches to the initialization routine of the real-time OS. Interrupts must be especially noted. It seems, however, that the startup routine is mostly executed with interrupts disabled.

The stack area is secured as follows.

```
    .set STACKSIZE, 0x200
    .bss
    .lcomm __stack, STACKSIZE, 4

    mov #__stack+STACKSIZE, sp
```

This is an example of securing a 0x200-byte stack in the .bss area. The contents of the stack are allocated to a bss attribute area because they do not have an initial value. Of course, they can be allocated to the sbss area, but the size of the stack that can be allocated to the sbss area is limited because the sbss area is accessed with a single gp-relative instruction. It is recommended to allocate the stack contents to the bss area if the stack size is great, as it may be better to allocate other variables to the sbss area. Change the value written to the .set instruction to change the stack size to be secured.

The CA850 generates codes on the assumption that the sp is at a 4-byte boundary when it references the memory relatively with the stack pointer (sp). Therefore, be sure to allocate the stack pointer at a 4-byte boundary. If necessary, use the quasi directive ".align 4".

The stack has a serious effect on the operation of the system. If the stack area runs short, the stack size exceeds the secured area and the stack contents are lost, which may cause a system hang-up. Estimate the stack size to be used by functions using stack850 included with the CA850, and secure a sufficient stack size.

5.1.4 Securing argument area for main function

The ANSI C specification defines the format of the main function as follows without dummy argument:

```
int main(void) { /* ... */ }
```

or, as the following function with two dummy arguments.

```
int main(int argc, char *argv[]) { /* ... */ }
```

argc of the function having two dummy arguments is a value that is not negative and indicates the total number of dummy arguments. argv indicates an array of pointers to argument character strings. argv[argc] is null (vacant pointer). If argc is 1 or more, argv[0] to argv[argc-1] are pointers to character strings.

Secure the areas for argc and argv in the startup routine, as shown below.

```
        .data
        .size __argc, 4
        .align 4
__argc:
        .word 0
        .size __argv, 4
__argv:
        .word #.L16
.L16:
        .byte 0
        .byte 0
        .byte 0
        .byte 0
```

The above area is not necessary if the main function is defined in this format.

```
int main(void) { /* ... */ }
```

The used RAM area can be reduced by deleting the above area.

Actually, processing that sets arguments (r6 and r7) of the main function is performed immediately before the main function. If r6 and r7 are not used in the startup routine, the processing can be executed immediately after the above program. Refer to "5.1.19 Setting r6 and r7 as argument of main function" for the processing to be set.

5.1.5 Setting text pointer (tp)

The text pointer (tp) is a pointer prepared to implement referencing (PIC: Position Independent Code) independent of the position at which the text area of an application, i.e., program code is allocated when the program code is referenced. For example, if it is necessary to reference a specific location in the code during program execution, the CA850 outputs the code to be accessed in tp-relative mode. Since the code is output on the assumption that tp is correctly set, tp must be correctly set in the startup routine.

The text pointer value is determined during linking, and is in a symbol defined by a symbol directive that is described in the link directive file. For example, suppose that the symbol directive of the text pointer is described as follows.

```
__tp_TEXT @ %TP_SYMBOL {TEXT};
```

The text pointer value is the beginning of the TEXT segment, and is in "__tp_TEXT".
Describe as follows to set tp in the startup routine.

```
    .extern __tp_TEXT, 4
    mov #__tp_TEXT, tp
```

Refer to CA850 for Link Directive User's Manual for details of symbol directives and link directives.

5.1.6 Setting global pointer (gp)

External variables or data defined in an application are allocated to the memory. The global pointer (gp) is a pointer prepared to implement referencing independent of location position (PID: Position Independent Data) when the variables or data allocated to the memory are referenced. The CA850 outputs a code for the section that is to be accessed in gp-relative mode. Since the code is output on the assumption that gp is correctly set, gp must be correctly set in the startup routine.

The global pointer value is determined during linking, and is in a symbol defined by a symbol directive that is described in the link directive file. For example, suppose that the symbol directive of the global pointer is described as follows.

```
__gp_DATA @ %GP_SYMBOL {DATA};
```

The gp symbol value can be defined the beginning of "data segment" of the DATA segment as shown above, or offset from a text symbol. A gp symbol can be specified not only by specifying the start address of a data segment (such as the DATA segment), but also by using an offset value from the text symbol as its address.

Using the second method, the gp symbol value is determined by adding an offset value from tp to tp. In other words, a code that is independent of location can be generated. To copy a program code and data used by that code to the RAM area simultaneously and execute them, the value of gp can be acquired immediately if the start address of the copy destination is known. In this case, the symbol directive is described as follows.

```
__tp_TEXT @ %TP_SYMBOL {TEXT};
__gp_DATA @ %GP_SYMBOL&__tp_TEXT {DATA};
```

The global pointer value is "__tp_TEXT to which the value of __gp_DATA is added", and the value to be added, i.e., offset value, is stored in "__gp_DATA". Therefore, describe as follows to set gp in the startup routine.

```
    .extern __tp_TEXT, 4
    .extern __gp_DATA, 4
    mov #__tp_TEXT, tp
    mov #__gp_DATA, gp
    add tp, gp
```

This sets the correct value of the global pointer to gp.

Refer to CA850 for Link Directive User’s Manual for details of symbol directives and link directives.

5.1.7 Setting element pointer (ep)

Of the external variables or data defined in an application, those that are allocated to the following sections are accessed from the element pointer (ep) in relative mode.

- sedata/sebss attribute section
- sidata/sibss attribute section
- tidata.byte/tibss.byte section
- tidata.word/tibss.word section

If these sections exist, the CA850 outputs a code to access these areas in ep-relative mode. Since the code is output on the assumption that ep is correctly set, ep must be correctly set in the startup routine. The element pointer value is determined during linking, and is in a symbol defined by a symbol directive that is described in the link directive file. For example, suppose that the symbol directive of the element pointer is described as follows.

The element pointer value is the beginning of the SIDATA segment by default, and its value is in "__ep_DATA". Therefore, describe as follows to set ep in the startup routine.

```
__ep_DATA@%EP_SYMBOL;
```

The element pointer value is the beginning of the SIDATA segment by default, and its value is in "__ep_DATA".
Therefore, describe as follows to set ep in the startup routine.

```
    .extern __ep_DATA, 4
    mov #__ep_DATA, ep
```

Reference the absolute address of __ep_DATA and set that value to ep.

Refer to CA850 for Link Directive User’s Manual for details of symbol directives and link directives.

5.1.8 Setting mask value to mask registers (r20 and r21)

When using mask registers, set them in the startup routine. The mask registers are "r20" and "r21". Set these registers to the following values.

- r20 to 8-bit mask value "0xff"
- r21 to 16-bit mask value "0xffff"

Set these values as follows.

```
    .option nowarning
    mov 0xff, r20
    mov 0xffff, r21
    .option warning
```

The portion between ".option nowarning" and ".option warning" is the quasi directive that suppresses output of warning messages during assembling. If the assembler option "-m" (use of mask option) is set, codes in which mask values are set are output to r20 and r21. If the user intentionally attempts to substitute values in r20 and r21, therefore, the following warning message is output.

```
W3013: mask register r20 or r21 used as destination register.
```

Refer to "2.5 Mask Register" for details of the mask registers.

5.1.9 Initializing peripheral I/O registers that must be initialized before execution of main function

When the external RAM is initialized by the startup routine, the external memory must first be set to the peripheral I/O; otherwise the memory area cannot be accessed and initialized. In addition, initialize the peripheral I/O registers that must be set for executing the startup routine.

Register setting can be described with assembler instructions, or execution may once branch from the startup routine to a C function and register setting can be described in this function. If it is described in C, reading and substitution in the peripheral I/O can be described in a visually simple way. For example, when creating the C function "void reset(void)" and calling it from the startup routine, describe the following instruction in the startup routine.

```
    jarl    _reset, lp
```

Differences between assembler instruction description and C description are shown below using the following examples. An instruction that substitutes "1" in P0 (port 0) is described as an assembly language source and as a C language source is as follows.

Assembly language source

```
    mov     1, r10
    st.b    r10, P0
```

r10 is used in this example.

C language source

```
#pragma ioreg

    P0 = 1;
```

The external memory setting differs depending on the device. Refer to the Relevant Device's Hardware User's Manual of each device.

With a clock generation function, the "internal system clock" that is supplied to each unit built in the V850 needs to be generated. In this case, the clock needs to be multiplied by a PLL (Phase locked loop) synthesizer before use. In other words, the clock must be correctly set to the frequency used; otherwise the clock operates slower or faster than the assumed operation speed.

Regarding the default value of the PLL, usually, the multiplication value is small and the operation frequency is low. These also apply to the startup routine. If the clearing of the memory area that is explained in "5.1.11 Clearing sbss area to 0" and later sections is executed while the operating frequency is low, it takes a lot of time to complete the execution. Therefore, it is recommended that the PLL be set during the early stages of the startup routine.

```
-- Setting 5 MHz to the value multiplied by four (20 MHz) in V850ES/SG
    mov     0x80,r10
    st.b    r10,PRCMD
    st.b    r10,PCC         -- fcpu = fxx
    nop
    nop
    nop
    nop
    nop
    set1    0, PLLCTL       -- PLLON = 1
```

Aside from the above settings, set the following settings: the "system wait control register (VSWC)", the "command register (PRCMD)", and, if necessary, the "watch dog timer (WDT)". For the correct settings, refer to the Relevant Device’s Hardware User’s Manual.

5.1.10 Initializing user target that must be initialized before execution of main function

Describe the necessary initialization processing for the user target, if any, in the startup routine.

The processing can be described with assembler instructions, or execution may once branch from the startup routine to a C function and the processing can be described in this function.

5.1.11 Clearing sbss area to 0

Initialize the sbss area, one of the bss attribute areas that do not have an initial value. Since the memory contents are undefined after the V850 is reset, it is recommended to clear the sbss area to zero. This processing is not necessary if the sbss attribute section has not been created or if it is not necessary to clear the sbss area to zero.

Use symbols "__ssbss" and "__esbss" reserved for the CA850 to clear the sbss area. The meaning of each symbol is as follows.

Table 5 - 2 Symbols of sbss Area

Symbol Name	Meaning
__ssbss	Symbol indicating start of sbss area
__esbss	Symbol indicating end of sbss area

The values (addresses) of these symbols are determined during linking. The program that clears the sbss area using these symbols is as follows.

```
    .extern __ssbss, 4
    .extern __esbss, 4
    mov     #__ssbss, r13
    mov     #__esbss, r12
    cmp     r12, r13
    jnl     .L11
.L12:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L12
.L11:
```

This program clears the sbss area to zero in 4-byte units.

5.1.12 Clearing bss area to 0

Initialize the bss area, one of the bss attribute areas that do not have an initial value. Since the memory contents are undefined after the V850 is reset, it is recommended to clear the bss area to zero. This processing is not necessary if the bss attribute section has not been created or if it is not necessary to clear the bss area to zero.

Use symbols "__sbss" and "__ebss" reserved for the CA850 to clear the bss area. The meaning of each symbol is as follows.

Table 5 - 3 Symbols of bss Area

Symbol Name	Meaning
__sbss	Symbol indicating start of bss area
__ebss	Symbol indicating end of bss area

The values (addresses) of these symbols are determined during linking. The program that clears the bss area using these symbols is as follows.

```
    .extern __sbss, 4
    .extern __ebss, 4
    mov     #__sbss, r13
    mov     #__ebss, r12
    cmp     r12, r13
    jnl     .L14
.L15:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L15
.L14:
```

This program clears the bss area to zero in 4-byte units.

5.1.13 Clearing sebss area to 0

Initialize the "sebss area", one of the bss attribute areas that do not have an initial value.

Since the memory contents are undefined after the V850 is reset, it is recommended to clear the sebss area to zero. This processing is not necessary if the sebss attribute section has not been created or if it is not necessary to clear the sebss area to zero.

Use symbols "__ssebss" and "__esebss" reserved for the CA850 to clear the sebss area. The meaning of each symbol is as follows.

Table 5 - 4 Symbols of sebss Area

Symbol Name	Meaning
__ssebss	Symbol indicating start of sebss area
__esebss	Symbol indicating end of sebss area

The values (addresses) of these symbols are determined during linking. The program that clears the sebss area using these symbols is as follows.

```
    .extern __ssebss, 4
    .extern __esebss, 4
    mov     #__ssebss, r13
    mov     #__esebss, r12
    cmp     r12, r13
    jnl     .L17
.L18:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L18
.L17:
```

This program clears the sebss area to zero in 4-byte units.

5.1.14 Clearing tibss.byte area to 0

Initialize the tibss.byte area, one of the bss attribute areas that do not have an initial value.

Since the memory contents are undefined after the V850 is reset, it is recommended to clear the tibss.byte area to zero. This processing is not necessary if the tibss.byte section has not been created or if it is not necessary to clear the tibss.byte area to zero.

Use symbols "__stibss.byte" and "__etibss.byte" reserved for the CA850 to clear the tibss.byte area. The meaning of each symbol is as follows.

Table 5 - 5 Symbols of tibss.byte Area

Symbol Name	Meaning
__stibss.byte	Symbol indicating start of tibss.byte area
__etibss.byte	Symbol indicating end of tibss.byte area

The values (addresses) of these symbols are determined during linking. The program that clears the tibss.byte area using these symbols is as follows.

```
    .extern __stibss.byte, 4
    .extern __etibss.byte, 4
    mov     #__stibss.byte, r13
    mov     #__etibss.byte, r12
    cmp     r12, r13
    jnl     .L20
.L21:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L21
.L20:
```

This program clears the tibss.byte area to zero in 4-byte units.

5.1.15 Clearing tibss.word area to 0

Initialize the tibss.word area, one of the bss attribute areas that do not have an initial value.

Since the memory contents are undefined after the V850 is reset, it is recommended to clear the tibss.word area to zero. This processing is not necessary if the tibss.word section has not been created or if it is not necessary to clear the tibss.word area to zero.

Use symbols "__stibss.word" and "__etibss.word" reserved for the CA850 to clear the tibss.word area. The meaning of each symbol is as follows.

Table 5 - 6 Symbols of tibss.word Area

Symbol Name	Meaning
__stibss.word	Symbol indicating start of tibss.word area
__etibss.word	Symbol indicating end of tibss.word area

The values (addresses) of these symbols are determined during linking. The program that clears the tibss.word area using these symbols is as follows.

```
    .extern __stibss.word, 4
    .extern __etibss.word, 4
    mov     #__stibss.word, r13
    mov     #__etibss.word, r12
    cmp     r12, r13
    jnl     .L23
.L24:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L24
.L23:
```

This program clears the tibss.word area to zero in 4-byte units.

5.1.16 Clearing sibss area to 0

Initialize the sibss area, one of the bss attribute areas that do not have an initial value.

Since the memory contents are undefined after the V850 is reset, it is recommended to clear the sibss area to zero. This processing is not necessary if the sibss attribute section has not been created or if it is not necessary to clear the sibss area to zero.

Use symbols "__ssibss" and "__esibss" reserved for the CA850 to clear the sibss area. The meaning of each symbol is as follows.

Table 5 - 7 Symbols of sibss Area

Symbol Name	Meaning
__ssibss	Symbol indicating start of sibss area
__esibss	Symbol indicating end of sibss area

The values (addresses) of these symbols are determined during linking. The program that clears the sibss area using these symbols is as follows.

```
    .extern __ssibss, 4
    .extern __esibss, 4
    mov     #__ssibss, r13
    mov     #__esibss, r12
    cmp     r12, r13
    jnl     .L26
.L25:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L25
.L26:
```

This program clears the sibss area to zero in 4-byte units.

5.1.17 Setting CTBP value for prologue/epilogue runtime library of functions

This setting is necessary when the V850Ex core is used and when the prologue/epilogue runtime library is used.

Since the CALLT instruction is used when the prologue/epilogue runtime library of functions is called by the V850Ex core, the value of CTBP necessary for the CALLT instruction must be set at the beginning of the function table of the prologue/epilogue runtime library of functions.

The prologue/epilogue runtime library is used in the following case.

- Compiler option "-Xpro_epi_runtime=on" is set

If a compiler option other than "-Ot" is specified for optimization, "-Xpro_epi_runtime=on" is automatically specified.

- ___PROLOG_TABLE

Describe the following code using this symbol.

```
    mov     #___PROLOG_TABLE, r12
    ldsr    r12, 20
```

CTBP is system register 20. Set a value to it using the ldsr instruction.

5.1.18 Setting BPC value of programmable peripheral I/O register

The peripheral area select control register (BPC) must be set when using a V850 microcontrollers product in which programmable peripheral I/O registers are provided and using a programmable peripheral I/O register.

For example, the peripheral area select control register of the V850E/IA1 is configured as follows.

BPC register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PA15	0	PA13	PA12	PA11	PA10	PA9	PA8	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0

Table 5 - 8 BPC Register

Bit Position	Bit Name	Meaning
15	PA15	Enables or disables use of programmable peripheral I/O area. 0: Use of programmable peripheral I/O area disabled. 1: Use of programmable peripheral I/O area enabled.
13 - 0	PA13 - PA0	Set address of programmable peripheral I/O area.

When using a programmable peripheral I/O register, a value must be set to the programmable peripheral I/O register using the compiler option "-Xbps". As a result, the CA850 outputs a code to access the programmable peripheral I/O register. However, this option does not set a value to BPC. To set a value to BPC, processing to write a value to the BPC register must be described in the startup routine.

In the case of the V850E/IA1, PA15 is set to 1, and a programmable peripheral I/O area address is set to PA13 to PA0. Set the BPC register, for example, to set the address of the programmable peripheral I/O area to 0x1234 as follows.

```
    mov     0x9234, r13
    st.h    r13, BPC
```

Because PA15 must be set to 1, set BPC to the logical sum (OR) of 0x1234 and 0x8000. The value set by the compiler option "-Xbps" is 0x1234, and the value set to BPC is 0x9234. Therefore, care must be exercised that no contradiction occurs.

Refer to the Relevant Device’s Architecture User’s Manual of each device for details of the programmable peripheral I/O registers.

5.1.19 Setting r6 and r7 as argument of main function

If the main function is defined to have two dummy arguments as follows,

```
int main(int argc, char *argv[]) { /* ... */ }
```

processing that sets a value to the arguments (r6 and r7) must be performed before execution branches to the main function. Refer to "5.1.4 Securing argument area for main function" for how to secure an area. This processing is not necessary for an application using a real-time OS because the main function is not created.

Processing to set a value to r6 and r7 is as follows.

```
    ld.w     $__argc, r6
    movea    $__argv, gp, r7
```

The argument area of the main function is allocated to the .sdata section, so describe an access code in gp-relative mode.

5.1.20 Branching to main function

Then the processing necessary for the startup routine has been completed, execute an instruction that branches to the main function. However, this processing is not necessary for an application using a real-time OS because the main function is not created. Instead, an instruction that branches to the initialization routine of the OS is necessary. Refer to "5.1.21 Branching to initialization routine of real-time OS" for details.

Describe the following code to branch to the main function.

```
    jarl    _main, lp
```

When the main function has been executed, execution returns to the 4 bytes subsequent to this branch instruction. The following instruction can also be used if it is known that execution does not return.

```
    jr      _main
```

```
    mov     #_main, lp
    jmp     [lp]
```

The entire 32-bit space can be accessed using the jmp instruction. When the "jarl_main, lp" instruction is used, execution returns after the main function is executed. It is recommended to take appropriate action to prevent deadlock from occurring when execution returns.

5.1.21 Branching to initialization routine of real-time OS

In an application using a real-time OS, execution branches to the initialization routine when the processing that must be performed by the startup routine has been completed. In an application not using a real-time OS, execution branches to the main function. Refer to "5.1.20 Branching to main function".

Branching to the initialization routine is performed differently depending on whether NEC Electronics' real-time OS RX850 or RX850 Pro is used.

In RX850

```
    .extern __urx_start
    jr      __urx_start
```

In RX850 Pro

```
    .extern _sit
    mov     #_sit, r10
    .extern __rx_start
    mov     #__rx_start, lp
    jmp     [lp]
```

Note that the start symbol of the initialization routine of each OS differs. Refer to the User's Manual of each real-time OS for details.

5.2 Example of Startup Routine

This section shows an example of the startup routine.

Figure 5 - 1 Example of Startup Routine

```
#-----------------------------------------------------------------------------
# external label declaration reserved for CA850 (1)
# for tp, gp, ep
#-----------------------------------------------------------------------------
    .extern __tp_TEXT, 4
    .extern __gp_DATA, 4
    .extern __ep_DATA, 4
#-----------------------------------------------------------------------------
# external label declaration reserved for CA850 (2)
# for initializing bss attribute section deleted if there is
# a section not used If the section to be used is not determined,
# write all sections and suppress the assemble error of the startup
# routine that occurs due to addition/deletion of sections.
#-----------------------------------------------------------------------------
    .extern __ssbss, 4
    .extern __esbss, 4
    .extern __sbss, 4
    .extern __ebss, 4
    .extern __ssebss, 4
    .extern __esebss, 4
    .extern __stibss.byte, 4
    .extern __etibss.byte, 4
    .extern __stibss.word, 4
    .extern __etibss.word, 4
    .extern __ssibss, 4
    .extern __esibss, 4
#-----------------------------------------------------------------------------
# external label declaration of symbol reserved for CA850
# Declare start address of function table as external label
# when using prologue/epilogue runtime library in V850Ex.
#-----------------------------------------------------------------------------
    .extern ___PROLOG_TABLE
#-----------------------------------------------------------------------------
# external label declaration of main function
#-----------------------------------------------------------------------------
    .extern _main
```

Figure 5 - 1 Example of Startup Routine

```
#-------------------------------------------------------------
# external label declaration of main function
# unnecessary if void main(void) type is used
#-------------------------------------------------------------
    .data
    .size __argc, 4
    .align 4
__argc:
    .word 0
    .size __argv, 4
__argv:
    .word #.L16
.L16:
    .byte 0
    .byte 0
    .byte 0
    .byte 0
#-------------------------------------------------------------
# The following is dummy data for section generation.
# This dummy data is used to clear the bss attribute section
# that appears later to zero.
#
# The start symbol and end symbol are generated if data exists
# in the corresponding section during linking.
# If the section to be used has not yet been determined, however,
# an assemble error of the startup routine occurs each time a section
# is added or deleted. To avoid this, generate the start and
# end symbols of a section by allocating dummy data to the section.
# The bss attribute section is not described because data is allocated
# by a stack generation code and dummy data does not have to be created
# in that section.
#
# If the section to be used is determined, delete this dummy data and
# the zero clear routine except the necessary part of the routine.
# This can eliminate waste and enhance the code efficiency.
#-------------------------------------------------------------
    .sbss
    .lcomm __sbss_dummy, 0, 0
    .sebss
    .lcomm __sebss_dummy, 0, 0
    .tibss.byte
    .lcomm __tibss_byte, 0, 0
    .tibss.word
    .lcomm __tibss_word, 0, 0
    .sibss
    .lcomm __sibss_dummy, 0, 0
```

Figure 5 - 1 Example of Startup Routine

```
#-------------------------------------------------------------
# securing stack
# securing 0x200 bytes in bss area
#-------------------------------------------------------------
    .set STACKSIZE, 0x200
    .bss
    .lcomm __stack, STACKSIZE, 4
#-------------------------------------------------------------
# reset handler
# Describe instructions to be allocated to the reset handler.
#-------------------------------------------------------------
    .section "RESET", text
    jr  __start
#-------------------------------------------------------------
# startup routine entity
#-------------------------------------------------------------
    .text
    .align 4
    .globl __start
    .globl __exit
    .globl __startend
__start:
#-------------------------------------------------------------
# It is assumed that __gp_DATA is set by a symbol directive
# that uses a relative value from tp.
# Therefore, gp adds the value of __gp_DATA to tp.
#-------------------------------------------------------------
    mov     #__tp_TEXT, tp
    mov     #__gp_DATA, gp
    add     tp, gp
    mov     #__stack+STACKSIZE, sp
    mov     #__ep_DATA, ep
#-------------------------------------------------------------
# mask register setting
# Delete this description to reduce the code if a mask register is
# not used. There is no problem even if it is not deleted in operation
# because it is overwritten in the program.
#-------------------------------------------------------------
    .option nowarning
    mov     0xff, r20
    mov     0xffff, r21
    .option warning
.L11:
```

Figure 5 - 1 Example of Startup Routine

```
#-----------------------------------------------------------------------------
# clearing sbss attribute section to zero
# delete this description to reduce the code if the sbss attribute
# section is not used
#-----------------------------------------------------------------------------
    .extern __ssbss, 4
    .extern __esbss, 4
    mov     #__ssbss, r13
    mov     #__esbss, r12
    cmp     r12, r13
    jnl     .L11
.L12:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L12
#-----------------------------------------------------------------------------
# clearing bss attribute section to zero
# delete this description to reduce the code if the sbss attribute
# section is not used
#-----------------------------------------------------------------------------
    .extern __sbss, 4
    .extern __ebss, 4
    mov     #__sbss, r13
    mov     #__ebss, r12
    cmp     r12, r13
    jnl     .L14
.L15:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L15
.L14:
#-----------------------------------------------------------------------------
# clearing sebss attribute section to zero
# delete this description to reduce the code if the sbss attribute
# section is not used
#-----------------------------------------------------------------------------
    .extern __ssebss, 4
    .extern __esebss, 4
    mov     #__ssebss, r13
    mov     #__esebss, r12
    cmp     r12, r13
    jnl     .L17
```

Figure 5 - 1 Example of Startup Routine

```
.L18:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L18
.L17:
#-----------------------------------------------------------------------------
# clearing tibss.byte attribute section to zero
# delete this description to reduce the code if the sbss attribute
# section is not used
#-----------------------------------------------------------------------------
    .extern __stibss.byte, 4
    .extern __etibss.byte, 4
    mov     #__stibss.byte, r13
    mov     #__etibss.byte, r12
    cmp     r12, r13
    jnl     .L20
.L21:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L21
.L20:
#-----------------------------------------------------------------------------
# clearing tibss.word attribute section to zero
# delete this description to reduce the code if the sbss attribute
# section is not used
#-----------------------------------------------------------------------------
    .extern __stibss.word, 4
    .extern __etibss.word, 4
    mov     #__stibss.word, r13
    mov     #__etibss.word, r12
    cmp     r12, r13
    jnl     .L23
.L24:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L24
.L23:
#-----------------------------------------------------------------------------
# clearing sibss attribute section to zero
# delete this description to reduce the code if the sbss attribute
# section is not used
#-----------------------------------------------------------------------------
    .extern __ssibss, 4
    .extern __esibss, 4
```

Figure 5 - 1 Example of Startup Routine

```
    mov     #__ssibss, r13
    mov     #__esibss, r12
    cmp     r12, r13
    jnl     .L26
.L25:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L25
.L26:
#-----------------------------------------------------------------------------
# setting of prologue/epilogue runtime library of functions
# The start address of the library function table is set to
# CTBP (system register #20). Delete this description when a core
# other than the V850Ex is used.
#-----------------------------------------------------------------------------
    mov     #___PROLOG_TABLE, r12
    ldsr    r12, 20
#-----------------------------------------------------------------------------
# programmable peripheral I/O register setting
# Delete this description if a V850 not having programmable
# peripheral I/O registers is used.
# Shown below is an example where the BPC register value
# (set address) is 0x1234. The logical sum of 0x1234 (address) and
# 0x8000 (use of programmable peripheral I/O) is set to BPC.
#-----------------------------------------------------------------------------
    mov     0x9234, r13
    st.h    r13, BPC
#-----------------------------------------------------------------------------
# setting argument of main function to r6 and r7
#-----------------------------------------------------------------------------
    ld.w    $__argc, r6
    movea   $__argv, gp, r7
#-----------------------------------------------------------------------------
# branching to main function
#-----------------------------------------------------------------------------
    jarl    _main, lp
#-----------------------------------------------------------------------------
# processing when main function returns
#-----------------------------------------------------------------------------
__exit:
    halt
__startend:
```

CHAPTER 6 LIBRARY FUNCTION

This chapter explains the library functions provided in the CA850.

6.1 Supplied Libraries

The CA850 provides the following libraries.

Table 6 - 1 Supplied Libraries

Library Type	Library Name	Function
Standard library	libc.a	Definition of Function with Variable Number of Arguments Management of Character String and Memory Character Type Macros and Functions Standard Input/Output Standard Utility Functions Runtime Library Prologue/epilogue runtime library of functions
Mathematical library	libm.a	Mathematical Functions
ROMization library	libr.a	ROMization copy functions

When the standard library or mathematical library is used in an application, include the related header files to use the library function. Reference these libraries using the linker option (-l). However, it is not necessary to reference the libraries if only "definition of a function with a variable number of arguments" and "character type macro/character type function" are used.

When PM+ is used, these libraries are referenced by default. Since the mathematical library internally references the standard library, the standard library is required when the mathematical library is used. The runtime library is a part of the standard library.

It is a routine that is automatically called by the CA850 when a floating-point operation or integer operation (such as 32-bit integer multiplication, division, or remainder calculation) is performed. Unlike the other library functions, therefore, the runtime library and prologue/epilogue runtime library of functions is not described in the C language source or assembly language source.

When the mask register function is used in the 32-register mode, use the standard library stored in the mask register folder (*Install Folder*\lib850\r32msk).

The linker automatically references the standard library in the above folder in the following cases.

- When 32-bit register mode is specified.
- When the mask register function is used with the compiler option "-Xmask_reg".

The ROMization library is referenced by the linker when the compiler option "-Xr" is specified. This library stores functions "_rcopy", "_rcopy1", "_rcopy2", and "_rcopy4", which are used to copy packed data.

6.1.1 Standard library

The functions contained in the standard library are listed below. These functions are described in the "libc.a" file. The prologue/epilogue runtime library of functions is explained in 6.1.5. The meaning of each element in the list is as follows.

Function name	Name of function
Outline	Functional outline of function
Header file	Header file that must be included in the C language source when this function is used. Include this file using the #include directive. "errno.h" must also be included if errno is used when an exception occurs.
ANSI	Indicates whether or not the function is stipulated by the ANSI standard. If it is stipulated, "O" is shown in this column; if not, "---" is shown.
Use of sdata	Indicates whether or not this function uses the memory area "sdata". In other words, whether or not data for which the function has an initial value is allocated to RAM is indicated. Because the section name must be ".sdata", generate the .sdata section even when this area is not used by the user application. If the .sdata section is used, "O" is shown in this column; if not, "---" is shown. If "O" is shown, data with an initial value is necessary, so the initial value must be copied to RAM before program execution. In other words, ROMization processing must be performed using the _rcopy function. Refer to CA850 for Operation User's Manual for details of this processing.
Use of sbss	Indicates whether or not this function uses the memory area "sbss". In other words, whether or not the function uses RAM as a temporary area is indicated. Because the section name must be ".sbss", generate the .sbss section even when this area is not used by the user application. If the .sbss section is used, "O" is shown in this column; if not, "---" is shown. If "O" is shown, data without an initial value is allocated, so unlike when .sdata is used, it is not necessary to perform ROMization processing.
Re-entrancy	Indicates whether or not the function is re-entrant. If it is re-entrant, "O" is shown; if not, "---" is shown. "Re-entrant" means that the function can "re-enter". A re-entrant function can be correctly executed even if an attempt is made in another process to execute that function while the function is being executed. In an application using a real-time OS, for example, this function is correctly executed even if dispatching to another task is triggered by an interrupt while a certain task is executing this function, and even if the function is executed in that task. A function that must use RAM as a temporary area may not necessarily be re-entrant.

(1) Definition of Function with Variable Number of Arguments

Table 6 - 2 Definition of Functions with Variable Number of Arguments

Function Name	Outline	Header File	ANSI	Use of sdata	Use of sbss	Re-entrancy
va_start	Initialization of variable for scanning argument list	stdarg.h	---	---	---	---
va_arg	Moving variable for scanning argument list	stdarg.h	---	---	---	---
va_end	End of scanning argument list	stdarg.h	---	---	---	---

(2) Management of Character String and Memory

Table 6 - 3 Character String Functions

Function Name	Outline	Header File	ANSI	Use of sdata	Use of sbss	Re-entrancy
bcmp	Memory comparison (char argument version of memcmp)	string.h	---	---	---	O
bcopy	Memory copy (char argument version of memcpy)	string.h	---	---	---	O
memchr	Memory search	string.h	O	---	---	O
memcmp	Memory comparison	string.h	O	---	---	O
memcpy	Memory copy	string.h	O	---	---	O
memmove	Memory move	string.h	O	---	---	O
memset	Memory set	string.h	O	---	---	O

Table 6 - 4 Memory Management Functions

Function Name	Outline	Header File	ANSI	Use of sdata	Use of sbss	Re-entrancy
index	Character string search (start position)	string.h	---	---	---	O
rindex	Character string search (end position)	string.h	---	---	---	O
strcat	Character string concatenation	string.h	O	---	---	O
strchr	Character string search (start position of specified character)	string.h	O	---	---	O
strcmp	Character string comparison	string.h	O	---	---	O
strcpy	Character string copy	string.h	O	---	---	O
strcspn	Character string search (maximum length not including specified character)	string.h	O	---	---	O
strerror	Character string conversion of error number	string.h	O	O	---	---
strlen	Length of character string	string.h	O	---	---	O
strncat	Character string concatenation (with number of characters specified)	string.h	O	---	---	O
strncmp	Character string comparison (with number of characters specified)	string.h	O	---	---	O
strncpy	Character string copy (with number of characters specified)	string.h	O	---	---	O
strpbrk	Character string search (start position)	string.h	O	---	---	O
strrchr	Character string search (end position)	string.h	O	---	---	O
strspn	Character string search (maximum length including specified character)	string.h	O	---	---	O

Table 6 - 4 Memory Management Functions

Function Name	Outline	Header File	ANSI	Use of sdata	Use of sbss	Re-entrancy
strstr	Character string search (start position of specified character)	string.h	O	---	---	O
strtok	Token division	string.h	O	---	O	---

(3) Character Type Macros and Functions

Table 6 - 5 Conversion of Character

Function Name	Outline	Header File	ANSI	Use of sdata	Use of sbss	Re-entrancy
_tolower	Conversion from uppercase to lowercase (correctly converted only if argument is in uppercase)	ctype.h	---	---	---	O
_toupper	Conversion from lowercase to uppercase (correctly converted only if argument is in lowercase)	ctype.h	---	---	---	O
toascii	Conversion from integer to ASCII character	ctype.h	---	---	---	O
tolower	Conversion from uppercase to lowercase (not converted if argument is not in uppercase)	ctype.h	O	O	---	O
toupper	Conversion from lowercase to uppercase (not converted if argument is not in lowercase)	ctype.h	O	O	---	O

Table 6 - 6 Classification of Characters

Function Name	Outline	Header File	ANSI	Use of sdata	Use of sbss	Re-entrancy
isalnum	Identification of ASCII letter or numeral	ctype.h	O	O	---	O
isalpha	Identification of ASCII letter	ctype.h	O	O	---	O
isascii	Identification of ASCII code	ctype.h	---	---	---	O
iscntrl	Identification of control character	ctype.h	O	O	---	O
isdigit	Identification of decimal number	ctype.h	O	O	---	O
isgraph	Identification of display character other than space	ctype.h	O	O	---	O
islower	Identification of lowercase character	ctype.h	O	O	---	O
isprint	Identification of display character	ctype.h	O	O	---	O
ispunct	Identification of delimiter character	ctype.h	O	O	---	O
isspace	Identification of space/tab/carriage return/line feed/vertical tab/page feed	ctype.h	O	O	---	O

Table 6 - 6 Classification of Characters

Function Name	Outline	Header File	ANSI	Use of sdata	Use of sbss	Re-entrancy
isupper	Identification of uppercase character	ctype.h	O	O	---	O
isxdigit	Identification of hexadecimal number	ctype.h	O	O	---	O

(4) Standard Input/Output

Table 6 - 7 Standard I/O Functions

Function Name	Outline	Header File	ANSI	Use of sdata	Use of sbss	Re-entrancy
perror	Error processing	stdio.h	O	O	---	Note
fread	Read from stream	stdio.h	O	O	---	---
fwrite	Write to stream	stdio.h	O	O	---	---
fgetc	Read one character from stream (same as getc)	stdio.h	O	O	---	---
fgets	Read one line from stream	stdio.h	O	O	---	---
getc	Read one character from stream (same as fgetc)	stdio.h	O	O	---	---
getchar	Read one character from standard input	stdio.h	O	O	---	---
gets	Read character string from standard input	stdio.h	O	O	---	---
ungetc	Push one character back to input stream	stdio.h	O	O	---	---
rewind	Reset file position indicator	stdio.h	O	O	---	---
fputc	Write character to stream	stdio.h	O	O	---	---
fputs	Output character string to stream	stdio.h	O	O	---	---
putc	Write character to stream	stdio.h	O	O	---	---
putchar	Write character to standard output stream	stdio.h	O	O	---	---
puts	Output character string to standard output stream	stdio.h	O	O	---	---
sprintf	Output with format	stdio.h	O	O	O	---
fprintf	Output text in specified format to stream	stdio.h	O	O	O	---
printf	Output text in specified format to standard output stream	stdio.h	O	O	O	---
vfprintf	Write text in specified format to stream	stdio.h	O	O	O	---
vprintf	Write text in specified format to standard output stream	stdio.h	O	O	O	---

Table 6 - 7 Standard I/O Functions

Function Name	Outline	Header File	ANSI	Use of sdata	Use of sbss	Re-entrancy
vsprintf	Write text in specified format to character string	stdio.h	O	O	O	---
sscanf	Input with format	stdio.h	O	O	O	---
fscanf	Read and interpret data from stream	stdio.h	O	O	O	---
scanf	Read and interpret text from standard output stream	stdio.h	O	O	O	---

Note stderr is not re-entrant.

(5) Standard Utility Functions

Table 6 - 8 Standard Utility Functions

Function Name	Outline	Header File	ANSI	Use of sdata	Use of sbss	Re-entrancy
abs	Output absolute value (int type)	stdlib.h	O	---	---	O
labs	Output absolute value (long type)	stdlib.h	O	---	---	O
bsearch	Binary search	stdlib.h	O	---	---	O
qsort	Align	stdlib.h	O	---	---	O
div	Division (int type)	stdlib.h	O	---	---	O
ldiv	Division (long type)	stdlib.h	O	---	---	O
ecvtf	Conversion of floating-point value to numeric character string (with total number of characters specified)	stdlib.h	---	O	O	---
fcvtf	Conversion of floating-point value to numeric character string (with number of digits below decimal point specified)	stdlib.h	---	O	O	---
gcvtf	Conversion of floating-point value to numeric character string (in specified format)	stdlib.h	---	O	O	---
itoa	Conversion of integer (int type) to character string	stdlib.h	---	---	---	O
ltoa	Conversion of integer (long type) to character string	stdlib.h	---	---	---	O
ultoa	Conversion of integer (unsigned long type) to character string	stdlib.h	---	---	---	O
calloc	Memory allocation (initialized to zero)	stdlib.h	O	O	O	Note 1
free	Memory release	stdlib.h	O	O	O	Note 1
malloc	Memory allocation (not initialized to zero)	stdlib.h	O	O	O	Note 1
realloc	Memory re-allocation	stdlib.h	O	O	O	Note 1

Table 6 - 8 Standard Utility Functions

Function Name	Outline	Header File	ANSI	Use of sdata	Use of sbss	Re-entrancy
rand	Pseudorandom number sequence generation	stdlib.h	O	O	---	---
srand	Setting of type of pseudorandom number sequence	stdlib.h	O	O	---	---
atoff	Conversion of character string to floating-point number	stdlib.h	O	O	---	Note 2
strtodf	Conversion of character string to floating-point number (storing pointer in last character string)	stdlib.h	O	O	O	Note 2
atoi	Conversion of character string to integer (int type)	stdlib.h	O	O	---	Note 2
atol	Conversion of character string to integer (long type)	stdlib.h	O	O	---	Note 2
strtol	Conversion of character string to integer (long type) and storing pointer in last character string	stdlib.h	O	O	O	Note 2
strtoul	Conversion of character string to integer (unsigned long type) and storing pointer in last character string	stdlib.h	O	O	O	Note 2

Notes 1 A function that can be called recursively.

2 A function is not re-entrant if errno is updated when an exception occurs.

Remark errno.h must be included if errno is used when an exception occurs.

(6) Non-Local Jump Functions

Table 6 - 9 Non-Local Jump Functions

Function Name	Outline	Header File	ANSI	Use of sdata	Use of sbss	Re-entrancy
setjmp	Set destination of non-local jump	setjmp.h	O	---	---	O
longjmp	Non-local jump	setjmp.h	O	---	---	O

6.1.2 Mathematical library

The functions contained in the mathematical library are listed below. These functions are described in the "libm.a" file. The meaning of each element in the list is as follows.

Function name	Name of function
Outline	Functional outline of function
Header file	Header file that must be included in a C language source when this function is used. Include this file using the #include directive. "errno.h" must also be included if errno is used when an exception occurs, "limits.h" if limit values of general integer type shown in "1.1.10 Quantitative limit" are used as a macro name, and "float.h" if limit values of floating-point type are used.Header file that must be included in the C language source when this function is used. Include this file using the #include directive.
ANSI	Indicates whether or not the function is stipulated by the ANSI standard. If it is stipulated, "O" is shown in this column; if not, "---" is shown.
Use of sdata	Indicates whether or not this function uses the memory area "sdata". In other words, whether or not data for which the function has an initial value is allocated to RAM is indicated. Because the section name must be ".sdata", generate the .sdata section even when this area is not used by the user application. If the .sdata section is used, "O" is shown in this column; if not, "---" is shown. If "O" is shown, data with an initial value is necessary, so the initial value must be copied to RAM before program execution. In other words, ROMization processing must be performed using the _rcopy function. Refer to CA850 for Operation User's Manual for details of this processing.
Use of sbss	Indicates whether or not this function uses the memory area "sbss". In other words, whether or not the function uses RAM as a temporary area is indicated. Because the section name must be ".sbss", generate the .sbss section even when this area is not used by the user application. If the .sbss section is used, "O" is shown in this column; if not, "---" is shown. If "O" is shown, data without an initial value is allocated, so unlike when .sdata is used, it is not necessary to perform ROMization processing.
Re-entrancy	Indicates whether or not the function is re-entrant. If it is re-entrant, "O" is shown; if not, "---" is shown. "Re-entrant" means that the function can "re-enter". A re-entrant function can be correctly executed even if an attempt is made in another process to execute that function while the function is being executed. In an application using a real-time OS, for example, this function is correctly executed even if dispatching to another task is triggered by an interrupt while a certain task is executing this function, and even if the function is executed in that task. A function that must use RAM as a temporary area may not necessarily be re-entrant.

(1) Mathematical Functions

Table 6 - 10 Mathematical Functions

Function Name	Outline	Header File	ANSI	Use of sdata	Use of sbss	Re-entrancy
j0f	Bessel function of first kind (0 order)	math.h	---	O	O	Note
j1f	Bessel function of first kind (first order)	math.h	---	O	O	Note
jnf	Bessel function of first kind (n order)	math.h	---	O	O	Note
y0f	Bessel function of second kind (0 order)	math.h	---	O	O	Note
y1f	Bessel function of second kind (first order)	math.h	---	O	O	Note
ynf	Bessel function of second kind (n order)	math.h	---	O	O	Note
erff	Error function (approximate value)	math.h	---	O	O	Note
erfcf	Error function (complementary probability)	math.h	---	O	O	Note
expf	Exponent function	math.h	O	O	O	Note
logf	Logarithmic function (natural logarithm)	math.h	O	O	O	Note
log2f	Logarithmic function (base = 2)	math.h	O	O	O	Note
log10f	Logarithmic function (base = 10)	math.h	O	O	O	Note
powf	Power function	math.h	O	O	O	Note
cbrtf	Cubic root function	math.h	---	---	---	O
sqrtf	Square root function	math.h	O	O	O	Note
ceilf	ceiling function	math.h	O	---	---	O
fabsf	Absolute value function	math.h	O	---	---	O
floorf	floor function	math.h	O	---	---	O
fmodf	Remainder function	math.h	O	O	O	Note
frexpf	Divide floating-point number into mantissa and power	math.h	O	O	O	Note
ldexpf	Convert floating-point number to power	math.h	O	O	O	Note
modff	Divide floating-point number into integer and decimal	math.h	O	---	---	O
gammaf	Logarithmic gamma function	math.h	---	O	O	Note
hypotf	Euclidean distance function	math.h	---	O	O	Note
matherr	Error processing function	math.h	---	---	---	O
acoshf	Inverse hyperbolic cosine	math.h	---	O	O	Note
asinhf	Inverse hyperbolic sine	math.h	---	O	O	Note

Table 6 - 10 Mathematical Functions

Function Name	Outline	Header File	ANSI	Use of sdata	Use of sbss	Re-entrancy
atanhf	Inverse hyperbolic tangent	math.h	---	O	O	Note
coshf	Hyperbolic cosine	math.h	O	O	O	Note
sinhf	Hyperbolic sine	math.h	O	O	O	Note
tanhf	Hyperbolic tangent	math.h	O	O	O	Note
acosf	Inverse cosine	math.h	O	O	O	Note
asinf	Inverse sine	math.h	O	O	O	Note
atanf	Inverse tangent	math.h	O	O	O	Note
atan2f	Inverse tangent (y/x)	math.h	O	O	O	Note
cosf	Cosine	math.h	O	O	O	Note
sinf	Sine	math.h	O	O	O	Note
tanf	Tangent	math.h	O	O	O	Note

Note These functions are not re-entrant only when errno is updated when an exception occurs and when matherr is called.

Remark errno.h" must also be included if errno is used when an exception occurs, "limits.h" if limit values of general integer type shown in "1.1.10 Quantitative limit" are used as a macro name, and "float.h" if limit values of floating-point type are used.

6.1.3 Runtime library

The functions contained in the runtime library are listed below. These functions are described in the "libc.a" file. The functions of the runtime library are automatically called by the CA850 when a floating-point operation or integer operation (32-bit integer multiplication/division or remainder calculation) is executed in a C-source program, so they are not described in the C language source or assembly-language source, like the prologue/epilogue runtime library of functions. The meaning of each element in the list is as follows.

Function name	Name of function
Outline	Functional outline of function
Use of sdata	Indicates whether or not this function uses the memory area "sdata". In other words, whether or not data for which the function has an initial value is allocated to RAM is indicated. Because the section name must be ".sdata", generate the .sdata section even when this area is not used by the user application. If the .sdata section is used, "O" is shown in this column; if not, "---" is shown. If "O" is shown, data with an initial value is necessary, so the initial value must be copied to RAM before program execution. In other words, ROMization processing must be performed using the _rcopy function. Refer to CA850 for Operation User’s Manual for details of this processing.
Use of sbss	Indicates whether or not this function uses the memory area "sbss". In other words, whether or not the function uses RAM as a temporary area is indicated. Because the section name must be ".sbss", generate the .sbss section even when this area is not used by the user application. If the .sbss section is used, "O" is shown in this column; if not, "---" is shown. If "O" is shown, data without an initial value is allocated, so unlike when .sdata is used, it is not necessary to perform ROMization processing.
Re-entrancy	Indicates whether or not the function is re-entrant. If it is re-entrant, "O" is shown; if not, "---" is shown. "Re-entrant" means that the function can "re-enter". A re-entrant function can be correctly executed even if an attempt is made in another process to execute that function while the function is being executed. In an application using a real-time OS, for example, this function is correctly executed even if dispatching to another task is triggered by an interrupt while a certain task is executing this function, and even if the function is executed in that task. A function that must use RAM as a temporary area may not necessarily be re-entrant.

(1) Runtime Library

Table 6 - 11 Runtime Library

Function Name	Outline	Use of sdata	Use of sbss	Re-entrancy
___mul	Multiplication of signed 32-bit integer	---	---	O
___mulu	Multiplication of unsigned 32-bit integer	---	---	O
___div	Division of signed 32-bit integer	---	---	O
___divu	Division of unsigned 32-bit integer	---	---	O
___mod	Remainder of signed 32-bit integer	---	---	O
___modu	Remainder of unsigned 32-bit integer **[V850]**	---	---	O
___addf.s	Addition of single-precision floating-point	O	---	Note
___subf.s	Subtraction of single-precision floating-point	O	---	Note
___mulf.s	Multiplication of single-precision floating-point	O	---	Note
___divf.s	Division of single-precision floating-point	O	---	Note
___cmpf.s	Comparison of single-precision floating-point and change of flag	O	---	Note
___cvt.ws	Conversion from integer to single-precision floating-point number	---	---	O
___trnc.sw	Conversion from single-precision floating-point number to integer	---	---	O

Note These functions are not re-entrant only when errno is updated when an exception occurs and when matherr is called.

Remark "errno.h" must also be included if errno is used when an exception occurs, "limits.h" if limit values of general integer type shown in "1.1.10 Quantitative limit" are used as a macro name, and "float.h" if limit values of floating-point type are used.

6.1.4 ROMization library

The functions contained in the ROMization library are listed below. These functions are described in the "libr.a" file. These functions are routines that copy data and program codes with initial values to RAM. Refer to Operation User's Manual for the ROMization procedure.

Function name	Name of function
Outline	Functional outline of function

- A ROMization function itself does not use the sdata and sbss areas but writes data to the sdata area.
- A ROMization function is usually called only once before the main program is executed, so it is not re-entrant.
- When a load module is downloaded to the in-circuit emulator (ICE), the data with initial values and placed in the data or sdata area is set as soon as the load module has been downloaded. Therefore, debugging can be performed without calling the _rcopy function. If a ROMization load module is created and executed on the actual machine, however, the initial values are not set and the operation is not performed as expected unless data with an initial value is copied using the _rcopy function. The reason for the trouble is that an initial value is not set by this _rcopy function in most of the cases. If a routine that clears RAM to zero is executed during initialization, call the _rcopy function before that routine; otherwise the initial values will also be cleared to zero.

(1) ROMization copy functions

Table 6 - 12 ROMization Copy Functions

Function Name	Outline
_rcopy	Copies packed data to RAM, 1 byte at a time (same as _rcopy1).
_rcopy1	Copies packed data to RAM, 1 byte at a time (same as _rcopy).
_rcopy2	Copies packed data to RAM, 2 bytes at a time.
_rcopy4	Copies packed data to RAM, 4 bytes at a time.

Caution _rcopy and _rcopy1 perform the same operation. These functions are provided to maintain compatibility with the previous version.

When a program code is copied to the internal instruction RAM of a V850 device that has an internal instruction RAM (such as the V850E/ME2), it must be copied in 4-byte units because of the hardware specifications. In this case, the program code is copied using the "_rcopy4" function. Any function could be used were it not for hardware restrictions. When a program code is copied in 2-byte or 4-byte units, the area that must be copied may be exceeded. If the size of a packed data area is not a multiple of 4, therefore, an area other than the packed data area is also copied at the same time. Take this into consideration.

6.1.5 Prologue/epilogue runtime library of functions

The functions contained in the prologue/epilogue runtime library are listed below. These functions are described in the "libc.a" file. These functions are routines that are automatically called by the CA850 for prologue/epilogue processing of functions, so they are not described in the C language source or assembly-language source, like the runtime library.

The V850Ex core uses the CALLT instruction to call the prologue/epilogue runtime library of functions. The code efficiency can be enhanced by calling these functions from the table of the CALLT instruction.

Calling the prologue/epilogue runtime library of functions is valid when:

- an optimization option other than "-Ot" (execution speed priority optimization) is specified.
- the compiler option "-Xpro_epi_runtime=on" is specified.

The following functions are used for prologue and epilogue processing of functions.

Table 6 - 13 List of Prologue Runtime Library Functions

Functional Outline	Function Name
Prologue processing of functions	`___push2000, ___push2001, ___push2002, ___push2003, ___push2004, ___push2040,` `___push2100, ___push2101, ___push2102, ___push2103, ___push2104, ___push2140,` `___push2200, ___push2201, ___push2202, ___push2203, ___push2204, ___push2240,` `___push2300, ___push2301, ___push2302, ___push2303, ___push2304, ___push2340,` `___push2400, ___push2401, ___push2402, ___push2403, ___push2404, ___push2440,` `___push2500, ___push2501, ___push2502, ___push2503, ___push2504, ___push2540,` `___push2600, ___push2601, ___push2602, ___push2603, ___push2604, ___push2640` `___push2700, ___push2701, ___push2702, ___push2703, ___push2704, ___push2740` `___push2800, ___push2801, ___push2802, ___push2803, ___push2804, ___push2840,` `___push2900, ___push2901, ___push2902, ___push2903, ___push2904, ___push2940,` `___pushlp00, ___pushlp01, ___pushlp02, ___pushlp03, ___pushlp04, ___pushlp40`

Table 6 - 14 List of Prologue Runtime Library Functions **[V850E]**

Functional Outline	Function Name

Prologue processing of functions	`___Epush250, ___Epush251, ___Epush252, ___Epush253, ___Epush254,`
	`___Epush260, ___Epush261, ___Epush262, ___Epush263, ___Epush264,`
	`___Epush270, ___Epush271, ___Epush272, ___Epush273, ___Epush274,`
	`___Epush280, ___Epush281, ___Epush282, ___Epush283, ___Epush284,`
	`___Epush290, ___Epush291, ___Epush292, ___Epush293, ___Epush294,`
	`___Epushlp0, ___Epushlp1, ___Epushlp2, ___Epushlp3, ___Epushlp4`

Table 6 - 15 List of Epilogue Runtime Library Functions

Functional Outline	Function Name

Epilogue processing of functions	`___pop2000, ___pop2001, ___pop2002, ___pop2003, ___pop2004, ___pop2040,`
	`___pop2100, ___pop2101, ___pop2102, ___pop2103, ___pop2104, ___pop2140,`
	`___pop2200, ___pop2201, ___pop2202, ___pop2203, ___pop2204, ___pop2240,`
	`___pop2300, ___pop2301, ___pop2302, ___pop2303, ___pop2304, ___pop2340,`
	`___pop2400, ___pop2401, ___pop2402, ___pop2403, ___pop2404, ___pop2440,`
	`___pop2500, ___pop2501, ___pop2502, ___pop2503, ___pop2504, ___pop2540,`
	`___pop2600, ___pop2601, ___pop2602, ___pop2603, ___pop2604, ___pop2640`
	`___pop2700, ___pop2701, ___pop2702, ___pop2703, ___pop2704, ___pop2740`
	`___pop2800, ___pop2801, ___pop2802, ___pop2803, ___pop2804, ___pop2840,`
	`___pop2900, ___pop2901, ___pop2902, ___pop2903, ___pop2904, ___pop2940,`
	`___poplp00, ___poplp01, ___poplp02, ___poplp03, ___poplp04, ___poplp40`

Table 6 - 16 List of Epilogue Runtime Library Functions **[V850E]**

Functional Outline	Function Name
Epilogue processing of functions	`___Epop250`, `___Epop251`, `___Epop252`, `___Epop253`, `___Epop254`, `___Epop260`, `___Epop261`, `___Epop262`, `___Epop263`, `___Epop264`, `___Epop270`, `___Epop271`, `___Epop272`, `___Epop273`, `___Epop274`, `___Epop280`, `___Epop281`, `___Epop282`, `___Epop283`, `___Epop284`, `___Epop290`, `___Epop291`, `___Epop292`, `___Epop293`, `___Epop294`, `___Epoplp0`, `___Epoplp1`, `___Epoplp2`, `___Epoplp3`, `___Epoplp4`

6.2 Header Files

The header files required for using the libraries of the CA850 are listed below. The macro definitions and function declarations are described in each file.

Table 6 - 17 Header Files

File Name	Outline
ctype.h	Header file for character conversion and classification
errno.h	Header file for reporting error condition
float.h	Header file for floating-point representation and floating-point operation
limits.h	Header file for quantitative limiting of integers
math.h	Header file for mathematical calculation
setjmp.h	Header file for non-local jump
stdarg.h	Header file for supporting functions having variable number of arguments
stddef.h	Header file for common definitions
stdio.h	Header file for standard I/O
stdlib.h	Header file for standard utilities
string.h	Header file for memory manipulation and character string manipulation

6.3 Object Names Linked

When a load module is generated by compiling and linking C language sources using libraries, objects stored in the libraries can be selected and linked as necessary. The names of the objects to be linked can be confirmed in a link map file that indicates the result of linking. The names of the objects that are linked are almost the same as the library function names. Routines commonly used for each function are combined and the object names are different from the library function names. The following objects combine the routines commonly used, and are automatically linked as necessary.

- com1f.o
- com1xf.o
- com2f.o
- com3f.o
- com4f.o
- com5f.o
- com6f.o
- com7f.o

6.4 Explanation of Format

About the library function that CA850 supports, the following format is used for the explanation.

Classification of library function

[Overview]

Outlines the feature of each function.

[Syntax]

Indicates the specification format of each function.

[Description]

Details of features of each function.

[Return value]

Indicates the return value of each function.

[Cautions]

Explains the supplementary points to be noted on each function.

[Example]

Indicates a simple example of each function.

Because the runtime library is written as an assembler, the following items may be added.

[Preprocessing]

Indicates the necessary preprocessing.

[Argument setting register]

Indicates the name of the register used for argument setting.

[Flag]

Indicates the flags that are affected.

6.5 Definition of Function with Variable Number of Arguments

This section explains the macros that define functions with a variable number of arguments in a portable form. The declarations and definitions of these macros are described in the "stdarg.h" file.

Table 6 - 18 Definition of Function with Variable Number of Arguments

Classification	Function Name	Outline
STDARG	va_start	Initializes variable for scanning argument list
	va_arg	Moves variable for scanning argument list
	va_end	End of scanning argument list

STDARG

[Overview]

Defines a function with a variable number of arguments.

va_start, va_arg, va_end

[Syntax]

```
#include <stdarg.h>

void   va_start(va_list ap, last-named-argument)
type   va_arg(va_list ap, type)
void   va_end(va_list ap)
```

[Description]

To define function `func` having a variable number of arguments in a portable form, the following format is used.

```
#include<stdarg.h>

void func(arg-declarations ...)
{
    va_list ap;
    type    argN;

    va_start(ap, last-named-argument);
    argN = va_arg(ap, type);

    va_end(ap);
}
```

arg-declarationsis an argument list with the last-named-argument declared at the end. "..." that follows indicates a list of the variable number of arguments.

va_listis the type of the variable (ap in the above example) used to scan the argument list.

va_start(va_list ap, last-named-argument)

This function initializes variable ap so that it indicates the beginning (argument next to last-named-argument) of the list of the variable number of arguments.

va_arg(va_list ap, type)

This function returns the argument indicated by variable ap, and advances variable ap to indicate the next argument. For the type of va_arg, specify the type converted when the argument is passed to the function.

With the CA850, specify the int type for an argument of char and short types, and specify the unsigned int type for an argument of unsigned char and unsigned short types.

Although a different type can be specified for each argument, stipulate "which type of argument is passed" according to the conventions between the called function and calling function.

Also stipulate "how many functions are actually passed" according to the conventions between the called function and calling function.

va_end(va_list ap)

This function indicates the end of scanning the list. By enclosing va_arg ... between va_start and va_end, scanning the list can be repeated.

[Example]

```
#include <stdarg.h>

void abc(int first, int second, ...)
{
    va_list ap;
    int     i;
    char    c, *fmt;

    va_start(ap, second);
    i = va_arg(ap, int);
    c = va_arg(ap, int); /* char type is converted into int type. */
    fmt = va_arg(ap, char *);

    va_end(ap);
}
```

6.6 Management of Character String and Memory

This section explains the character string processing features and memory area management features.

The declarations and definitions related to these functions are described in the "string.h" file.

Table 6 - 19 Functions for Character String/Memory Management

Classification	Function Name	Outline
STRING	index	Character string search (first position)
	rindex	Character string search (last position)
	strcat	Character string concatenation
	strchr	Character string search (first position of specifiedcharacter)
	strcmp	Character string comparison
	strcpy	Character string copy
	strcspn	Character string search (maximum length not including specified character)
	strerror	Character string conversion of error number
	strlen	Length of character string
	strncat	Character string concatenation (number of characters specification)
	strncmp	Character string comparison (number of characters specification)
	strncpy	Character string copy (number of characters specification)
	strpbrk	Character string search (first position)
	strrchr	Character string search (last position)
	strspn	Character string search (maximum length including specified character)
	strstr	Character string search (first position of specified character string)
	strtok	Token division
MEMORY	bcmp	Memory comparison (char argument)
	bcopy	Memory copy (char argument)
	memchr	Memory search
	memcmp	Memory comparison (void argument)
	memcpy	Memory copy (void argument)
	memmove	Memory move
	memset	Memory set

STRING

[Overview]

Manipulates a character string.

index, rindex, strcat, strchr, strcmp, strcpy, strcspn, strerror, strlen, strncat, strncmp, strncpy, strpbrk, strrchr, strspn, strstr, strtok

[Syntax]

```
#include <string.h>

char   *index(const char *s, int c)
char   *rindex(const char *s, int c)
char   *strcat(char *dst, const char *src)
char   *strchr(const char *s, int c)
int    strcmp(const char *s1, const char *s2)
char   *strcpy(char *dst, const char *src)
size_t strcspn(const char *s1, const char *s2)
char   *strerror(int errnum)
size_t strlen(const char *s)
char   *strncat(char *dst, const char *src, size_t length)
int    strncmp(const char *s1, const char *s2, size_t length)
char   *strncpy(char *dst, const char *src, size_t length)
char   *strpbrk(const char *s1, const char *s2)
char   *strrchr(const char *s, int c)
size_t strspn(const char *s1, const char *s2)
char   *strstr(const char *s1, const char *s2)
char   *strtok(char *s, const char *delimiters)
```

[Description]

index(const char *s, int c)

This function is a function having the same feature as strchr.

rindex(const char *s, int c)

This function a function having the same feature as strrchr.

strcat(char *dst, const char *src)

This function concatenates the duplication of the character string indicated by src to the end of the character string indicated by dst, including the null character (\0). The first character of src overwrites the null character (\0) at the end of dst.

strchr(const char *s, int c)

This function obtains the position at which a character the same as c converted into char type appears in the character string indicated by s. The null character (\0) indicating termination is regarded as part of this character string.

strcmp(const char *s1, const char *s2)

This function compares the character string indicated by s1 with the character string indicated by s2.

strcpy(char *dst, const char *src)

This function copies the character string indicated by src to the array indicated by dst.

strcspn(const char *s1, const char *s2)

This function obtains the length of the maximum and first portion consisting of characters missing from the character string indicated by s2 (except the null character (\0) at the end) in the character string indicated by s1.

strerror(int errnum)

This function converts error number errnum into a character string according to the correspondence relationship of the processing system definition. The value of errnum is usually the duplication of global variable errno. Do not change the specified array of the application program.

strlen(const char *s)

This function obtains the length of the character string indicated by s.

strncat(char *dst, const char *src, size_t length)

This function concatenates up to length characters (including the null character (\0) of src) to the end of the character string indicated by dst, starting from the beginning of the character string indicated by src. The null character (\0) at the end of dst is written over the first character of src. The null character indicating termination (\0) is always added to this result.

strncmp(const char *s1, const char *s2, size_t length)

This function compares up to length characters of the array indicated by s1 with characters of the array indicated by s2.

strncpy(char *dst, const char *src, size_t length)

This function copies up to length characters (including the null character (\0)) from the array indicated by src to the array indicated by dst. If the array indicate by src is shorter than length characters, null characters (\0) are appended to the duplication in the array indicated by dst, until all length characters are written.

strpbrk(const char *s1, const char *s2)

This function obtains the position in the character string indicated by s1 at which any of the characters in the character string indicated by s2 (except the null character (\0)) appears first.

strrchr(const char *s, int c)

This function obtains the position at which c converted into char type appears last in the character string indicated by s. The null character (\0) indicating termination is regarded as part of this character string.

strspn(const char *s1, const char *s2)

This function obtains the maximum and first length of the portion consisting of only the characters (except the null character (\0)) in the character string indicated by s2, in the character string indicated by s1.

strstr(const char *s1, const char *s2)

This function obtains the position of the portion (except the null character (\0)) that first coincides with the character string indicated by s2, in the character string indicated by s1.

strtok(char *s, const char *delimiters)

This function divides the character string indicated by s into strings of tokens by delimiting the character string with a character in the character string indicated by delimiters. If this function is called first, s is used as the first argument. Then, calling with the null pointer as the first argument continues. The delimiting character string indicated by delimiters can differ on each call.

On the first call, the character string indicated by s is searched for the first character not included in the delimiting character string indicated by delimiters. If such a character is not found, a token does not exist in the character string indicated by s, and strtok returns the null pointer. If a character is found, that character is the beginning of the first token. After that, strtok searches from the position of that character for a character included in the delimiting character string at that time. If such a character is not found, the token is expanded to the end of the character string indicated by s, and the subsequent search returns the null pointer. If a character is found, the subsequent character is overwritten by the null character (\0) indicating the termination of the token. strtok saves the pointer indicating the subsequent character. If the null pointer is used as the value of the first argument, a code that is not re-entrant is returned.

This can be avoided by preserving the address of the last delimiting character in the application program, and passing s as an argument that is not vacant, by using this address.

[Return value]

strcat	Returns the value of dst.
strchr	Returns a pointer indicating the character that has been found. If c does not appear in this character string, the null pointer is returned.
strcmp	Returns an integer greater than, equal to, or less than 0, depending on whether the character string indicated by s1 is greater than, equal to, or less than the character string indicated by s2.
strcpy	Returns the value of dst.
strcspn	Returns the length of the portion that has been found.
strerror	Returns a pointer to the converted character string.
strlen	Returns the number of characters existing before the null character (\0) indicating termination.
strncat	Returns the value of dst.
strncmp	Returns an integer greater than, equal to, or less than 0, depending on whether the character string indicated by s1 is greater than, equal to, or less than the character string indicated by s2.
strncpy	Returns the value of dst.
strpbrk	Returns the pointer indicating this character. If any of the characters from s2 does not appear in s1, the null pointer is returned.
strrchr	Returns a pointer indicating c that has been found. If c does not appear in this character string, the null pointer is returned.
strspn	Returns the length of the portion that has been found.
strstr	Returns the pointer indicating the character string that has been found. If character string s2 is not found, the null pointer is returned. If s2 indicates a character string with a length of 0, s1 is returned.
strtok	Returns a pointer to a token. If a token does not exist, the null pointer is returned.

[Cautions]

Because the null character (\0) is always appended when strncat is used, if copying is limited by the number of length arguments, the number of characters appended to dst is n + 1.

[Example]

```
#include <string.h>

void func(char *str, const char *src)
{
    strcpy(str, src);   /* Copies character string indicated by src to */
                        /*  array indicated by str. */
        :
}
```

MEMORY

[Overview]

Manipulates the memory contents.

bcmp, bcopy, memchr, memcmp, memcpy, memmove, memset

[Syntax]

```
#include <string.h>

int     bcmp(const char *s1, const char *s2, size_t n)
void    bcopy(const char *in, char *out, size_t n)
void    *memchr(const void *s, int c, size_t length)
int     memcmp(const void *s1, const void *s2, size_t n)
void    *memcpy(void *out, const void *in, size_t n)
void    *memmove(void *dst, void *src, size_t length)
void    *memset(const void *s, int c, size_t length)
```

[Description]

bcmp(const char *s1, const char *s2, size_t n)

This function is a function having the same feature as memcmp.

bcopy(const char *in, char *out, size_t n)

This function is a function having the same feature as memcpy.

memchr(const void *s, int c, size_t length)

This function obtains the position at which character c (converted into char type) appears first in the first length number of characters in an area indicated by s.

memcmp(const void *s1, const void *s2, size_t n)

This function compares the first n characters of an object indicated by s1 with the object indicated by s2.

memcpy(void *out, const void *in, size_t n)

This function copies n bytes from an object indicated by in to an object indicated by out.

memmove(void *dst, void *src, size_t length)

This function moves the length number of characters from a memory area indicated by src to a memory area indicated by dst. Even if the copy source and copy destination areas overlap, the characters are correctly copied to the memory area indicated by dst.

memset(const void *s, int c, size_t length)

This function copies the value of c (converted into unsigned char type) to the first length character of an object indicated by s.

[Return value]

memchr	If c is found, a pointer indicating this character is returned. If c is not found, the null pointer is returned.
bcmp, memcmp	An integer greater than, equal to, or less than 0 is returned, depending on whether the object indicated by s1 is greater than, equal to, or less than the object indicated by s2.
bcopy, memcpy	Returns the value of out. The operation is undefined if the copy source and copy destination areas overlap.
memmove	Returns the value of dst at the copy destination.
memset	Returns the value of s.

[Example]

```
#include <string.h>

int func(const void *s1, const void *s2)
{
    int i;

    i = memcmp(s1, s2, 5);  /* Compares the first five characters of */
                            /* the character string indicated by s1 with */
                            /* the first five characters of the character*/
                            /* string indicated by s2 */

    return i;
}
```

6.7 Character Type Macros and Functions

This section explains the macros that classify characters into several categories (such as letters, numerals, control characters, and blanks) and macros that perform simple mapping of characters.

These macros can also be used as subroutines.

These macros are defined by "ctype.h".

Table 6 - 20 Character Type Macros

Classification	Function Name	Outline
CONV	_tolower	Conversion from uppercase characters to lowercase characters
	_toupper	Conversion from lowercase characters to uppercase characters
	toascii	Conversion from integer to ASCII character
	tolower	Conversion from uppercase characters to lowercase characters
	toupper	Conversion from lowercase characters to uppercase characters
CTYPE	isalnum	Whether ASCII letter or numeral
	isalpha	Whether ASCII letter
	isascii	Whether ASCII code
	iscntrl	Whether control character
	isdigit	Whether decimal number
	isgraph	Whether display character (other than space)
	islower	Whether lowercase character
	isprint	Whether display character
	ispunct	Whether delimiter
	isspace	Whether space, tab, line feed, carriage return, vertical tab, or form feed
	isupper	Whether uppercase character
	isxdigit	Whether hexadecimal character

CONV

[Overview]

Converts characters.

_tolower, _toupper, toascii, tolower, toupper

[Syntax]

```
#include <ctype.h>

int    _tolower(int c)
int    _toupper(int c)
int    toascii(int c)
int    tolower(int c)
int    toupper(int c)
```

[Description]

_tolower(int c)

This function is a macro that performs the same operation as tolower if the argument is of uppercase characters. Because the argument is not checked, the correct conversion is performed only if the argument is of uppercase characters. If otherwise, the operation will be undefined. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef _tolower".

_toupper(int c)

This function is a macro that performs the same operation as toupper if the argument is of lowercase characters. Because the argument is not checked, the correct conversion is performed only if the argument is of lowercase characters. If otherwise, the operation will be undefined. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef _toupper".

toascii(int c)

This function is a macro that forcibly converts an integer into an ASCII character (0 to 127) by clearing bit 8 and higher of the argument to 0. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef toascii".

tolower(int c)

This function is a macro that converts uppercase characters into the corresponding lowercase characters and leaves the other characters unchanged. This macro is defined only when c is an integer in the range of EOF to 255. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef tolower".

toupper(int c)

This function is a macro that converts lowercase characters into the corresponding uppercase characters and leaves the other characters unchanged. This macro is defined only when c is an integer in the range of EOF.

[Return value]

toascii	Returns an integer in the range of 0 to 127.
_tolower, tolower	If isupper is true with respect to c, returns a character that makes islower true in response; otherwise, returns c. Also with _tolower, operation can be inconsistent when specifying illegal values for c.
_toupper, toupper	If islower is true with respect to c, returns a character that makes islower true in response; otherwise, returns c. Also with _toupper, operation can be inconsistent when specifying illegal values for c.

[Example]

```
#include <ctype.h>

int     chc = 'a';
int     ret = func(chc);

int func(int c)
{
    int i;

    i = toupper(c);     /* Converts lowercase character ’a’ of c into */
                        /* uppercase character ’A’. */

    return i;
}
```

CTYPE

[Overview]

Classifies characters

isalnum, isalpha, isascii, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit

[Syntax]

```
#include <ctype.h>

int    isalnum(int c)
int    isalpha(int c)
int    isascii(int c)
int    iscntrl(int c)
int    isdigit(int c)
int    isgraph(int c)
int    islower(int c)
int    isprint(int c)
int    ispunct(int c)
int    isspace(int c)
int    isupper(int c)
int    isxdigit(int c)
```

[Description]

isalnum(int c)

This function is a macro that checks whether a given character is an ASCII alphabetic character or numeral. This macro is defined for all integer values. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef isalnum".

isalpha(int c)

This function is a macro that checks whether a given character is an ASCII alphabetic character. This macro id defined only when c is made true by isascii or when c is EOF. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef isalpha".

isascii(int c)

This function is a macro that checks whether a given character is an ASCII code (0x00 to 0x7f). This macro is defined for all integer values. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef isascii".

iscntrl(int c)

This function is a macro that checks whether a given character is a control character (0x00 to 0x1F or 0x7F). This macro is defined only when c is made true by isascii or when c is EOF. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef iscntrl".

isdigit(int c)

This function is a macro that checks whether a given character is a decimal number. This macro is defined only when c is made true by isascii or when c is EOF. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef isdigit".

isgraph(int c)

This function is a macro that checks whether a given character is a display character[Note] (0x20 to 0x7E) other than space (0x20). This macro is defined only when c is made true by isascii or when c is EOF. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef isgraph".

Note printing character

islower(int c)

This function is a macro that checks whether a given character is a lowercase character (a to z). This macro is defined only when c is made true by isascii or when c is EOF. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef islower".

isprint(int c)

This function is a macro that checks whether a given character is a display character (0x20 to 0x7F). This macro is defined only when c is made true by isascii or when c is EOF. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef isprint".

ispunct(int c)

This function is a macro that checks whether a given character is a printable delimiter (isgraph(c) && !isalnum(c)). This macro is defined only when c is made true by isascii or when c is EOF. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef ispunct".

isspace(int c)

This function is a macro that checks whether a given character is a space, tap, line feed, carriage return, vertical tab, or form feed (0x09 to 0x0D, or 0x20). This macro is defined only when c is made true by isascii or when c is EOF. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef isspace".

isupper(int c)

This function is a macro that checks whether a given character is an uppercase character (A to Z). This macro is defined only when c is made true by isascii or when c is EOF. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef isupper".

isxdigit(int c)

This function is a macro that checks whether a given character is a hexadecimal number (0 to 9, a to f, or A to F). This macro is defined only when c is made true by isascii or when c is EOF. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef isxdigit".

[Return value]

These macros return a value other than 0 if the value of argument c matches the respective description (i.e., if the result is true). If the result is false, 0 is returned.

[Example]

```
#include <ctype.h>

void func(void)
{
    int     i, j = 0;
    char    s[50];


    for(i = 50; i <= 99; i++) {      /* Stores characters that can be */
                                     /* displayed in codes 50 through 99 to s. */
        if(isprint(i)) {
            s[j] = i;
            j++;
        }
    }
}
```

6.8 Standard Input/Output

This section explains the functions that generate and scan character strings in accordance with the specification of a formatted character string. Definitions and declarations related to these functions are described in the "stdio.h" file.

Table 6 - 21 Standard Input/Output

Classification	Function Name	Outline
ERROR	perror	Error processing
FILEIO	fread[Note]	Read from stream
	fwrite[Note]	Write to stream
GETS	fgetc[Note]	Read one character from stream
	fgets[Note]	Read one line from stream
	getc[Note]	Read one character from stream
	getchar[Note]	Read one character from standard input
	gets[Note]	Read string from standard input
	ungetc[Note]	Push one character back into input stream
	rewind[Note]	Reset file position indicator
PUTS	fputc[Note]	Write character to stream
	fputs[Note]	Output string to stream
	putc[Note]	Write character to stream
	putchar[Note]	Write character to standard output stream
	puts[Note]	Output string to standard output stream
SPRINTF	sprintf	Formatted output
	vsprintf	Write format-specified text to string
PRINTF	fprintf[Note]	Output format-specified text to stream
	printf[Note]	Output format-specified text to standard output stream
	vfprintf[Note]	Write format-specified text to stream
	vprintf[Note]	Write format-specified text to standard output stream
SSCANF	sscanf	Formatted input
SCANF	fscanf[Note]	Read and interpret data from stream
	scanf[Note]	Read and interpret text from standard output stream

Note These functions are not supported by the NEC Electronics integrated debugger or the system simulator.

ERROR

[Overview]

Error processing

perror

[Syntax]

```
#include <stdio.h>

void   perror(const char *s)
```

[Description]

perror(const char *s)

This function outputs to stderr the error message that corresponds to global variable errno.

The message that is output is as follows.

When s is not NULL	fprintf(stderr, "%s:%s\n", s, s_fix);
When s is NULL	fprintf(stderr, "%s\n", sfix);

s_fix is as follows.

When errno is EDOM	"EDOM error"
When errno is ERANGE	"ERANGE error"
When errno is 0	"no error"
Otherwise	"error xxx"(xxx is abs (errno)%1000)

[Example]

```
#include <stdio.h>

void func1(double x)
{
    double d;

    errno = 0;
    d = exp(x);
    if(errno) perror("func1");   /* If a calculation exception is generated */
                                 /* by exp perror is called */
}
```

FILEIO

[Overview]

Direct input/output

fread, fwrite

Caution These functions are not supported by the NEC Electronics integrated debugger or the system simulator.

[Syntax]

```
#include <stdio.h>

size_t fread(void *ptr, size_t, size, size_t nmemb, FILE *stream)
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)
```

[Description]

fread(void *ptr, size_t, size, size_t nmemb, FILE *stream)

This function inputs nmemb elements of size size from the input stream pointed to by stream and stores them in ptr. Only the standard input/output stdin can be specified for stream.

fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)

This function outputs nmemb elements of size size from the array pointed to by ptr to the output stream pointed to by stream. Only the standard input/output stdout or stderr can be specified for stream

[Return value]

fread	The number of elements that were input (nmemb) is returned.
fwrite	The number of elements that were output (nmemb) is returned.

Error return does not occur for either function.

[Example]

```
#include <stdio.h>

void func(void)
{
    struct {
        int     c;
        double  d;
    } buf[10];

    fread(buf, sizeof(buf[0]), sizeof(buf)/sizeof(buf[0]), stdin);
}
```

GETS

[Overview]

Character or string input

fgetc, fgets, getc, getchar, gets, ungetc, rewind

Caution These functions are not supported by the NEC Electronics integrated debugger or the system simulator.

[Syntax]

```
#include <stdio.h>

int    fgetc(FILE *stream)
char   *fgets(char *s, int n, FILE *stream)
int    getc(FILE *stream)
int    getchar()
char   *gets(char *s)
int    ungetc(int c, FILE *stream)
void   rewind(FILE *stream)
```

[Description]

fgetc(FILE *stream)

This function inputs one character from the input stream pointed to by stream. Only the standard input/output stdin can be specified for stream.

fgets(char *s, int n, FILE *stream)

This function inputs at most n-1 characters from the input stream pointed to by stream and stores them in s. Character input is also ended by the detection of a new-line character. In this case, the new-line character is also stored in s. The end-of-string null character is stored at the end in s. Only the standard input/output stdin can be specified for stream.

getc(FILE *stream)

This function inputs one character from the input stream pointed to by stream. The getc function is completely equivalent to fgetc.

getchar()

This function inputs one character from the standard input/output stdin.

gets(char *s)

This function inputs characters from the standard input/output stdin until a new-line character is detected and stores them in s. The new-line character that was input is discarded, and an end-of-string null character is stored at the end in s.

ungetc(int c, FILE *stream)

This function pushes the character c back into the input stream pointed to by stream. However, if c is EOF, no pushback is performed. The character c that was pushed back will be input as the first character during the next character input. Only one character can be pushed back by ungetc. If ungetc is executed continuously, only the last ungetc will have an effect. Only the standard input/output stdin can be specified for stream.

rewind(FILE *stream)

This function clears the error indicator of the input stream pointed to by stream, and positions the file position indicator at the beginning of the file. However, only the standard input/output stdin can be specified for stream. Therefore, rewind only has the effect of discarding the character that was pushed back by ungetc.

[Return value]

fgetc, getc, getchar	The input character is returned.
fgets, gets	s is returned.
ungetc	The character c is returned.

Error return does not occur for any of these functions.

[Example]

```
#include <stdio.h>

void func(void)
{
    int c;

    c = fgetc(stdin);
}
```

PUTS

[Overview]

Character or string output

fputc, fputs, putc, putchar, puts

Caution These functions are not supported by the NEC Electronics integrated debugger or the system simulator.

[Syntax]

```
#include <stdio.h>

int    fputc(int c, FILE *stream)
int    fputs(const char *s, FILE *stream)
int    putc(int c, FILE *stream)
int    putchar(int c)
int    puts(const char *s)
```

[Description]

fputc(int c, FILE *stream)

This functionoutputs the character c to the output stream pointed to by stream. Only the standard input/ output stdout or stderr can be specified for stream.

fputs(const char *s, FILE *stream)

This function outputs the string s to the output stream pointed to by stream. The end-of-string null character is not output. Only the standard input/output stdout or stderr can be specified for stream.

putc(int c, FILE *stream)

This function outputs the character c to the output stream pointed to by stream. The putc function is completely equivalent to fputc.

putchar(int c)

This function outputs the character c to the standard input/output stdout.

puts(const char *s)

This function outputs the string s to the standard input/output stdout. The end-of-string null character is not output, but a new-line character is output in its place.

[Return value]

fputc, putc, putchar	The character c is returned.
fputs, puts	0 is returned.

Error return does not occur for any of these functions.

[Example]

```
#include <stdio.h>

void func(void)
{
    fputc('a', stdout);
}
```

SPRINTF

[Overview]

Formatted output

sprintf, vsprintf

[Syntax]

```
#include <stdio.h>

int    sprintf(char *s, const char *format [, arg, ...])
int    vsprintf(char *s, const char *format, va_list arg)
```

[Description]

sprintf(char *s, const char *format [, arg, ...])

This function applies the format specified by the string pointed to by format to the respective arg arguments, and writes out the formatted data that was output as a result to the array pointed to by s.

If there are not sufficient arguments for the format, the operation is undefined. If the end of the formatted string is reached, control returns. If there are more arguments that those required by the format, the excess arguments are ignored. If the area of s overlaps one of the arguments, the operation is undefined.

The argument format specifies "the output to which the subsequent argument is to be converted". The null character (\0) is appended at the end of written characters (the null character (\0) is not counted in a return value).

The format consists of the following two types of directives:

Ordinary characters	Characters that are copied directly without conversion (other than "%").
Conversion specifications	Specifications that fetch zero or more arguments and assign a specification.

Each conversion specification begins with character "%" (to insert "%" in the output, specify "%%" in the format string). The following appear after the "%":

%[flag][field-width][precision][size][type-specification-character]

The meaning of each conversion specification is explained below

(1) flag

Zero or more flags, which qualify the meaning of the conversion specification, are placed in any order.

The flag characters and their meanings are as follows:

-	The result of the conversion will be left-justified in the field, with the right side filled with blanks (if this flag is not specified, the result of the conversion is right-justified).
+	The result of a signed conversion will start with a + or - sign (if this flag is not specified, the result of the conversion starts with a sign only when a negative value has been converted).
Space	If the first character of a signed conversion is not a sign and a signed conversion is not generated a character, a space (" ") will be appended to the beginning of result of the conversion. If both the space flag and + flag appear, the space flag is ignored.
#	The result is to be converted to an alternate format. For o conversion, the precision is increased so that the first digit of the conversion result is 0. For x or X conversion, 0x or 0X is appended to the beginning of a non-zero conversion result. For e, f, g, E, or G conversion, a decimal point "." is added to the conversion result even if no digits follow the decimal point[Note]. For g or G conversion, trailing zeros will not be removed from the conversion result. The operation is undefined for conversions other than the above.
0	For d, e, f, g, i, o, u, x, E, G, or X conversion, zeros are added following the specification of the sign or base to fill the field width. If both the 0 flag and - flag are specified, the 0 flag is ignored. For d, i, o, u, x, or X conversion, when the precision is specified, the zero (0) flag is ignored. Note that 0 is interpreted as a flag and not as the beginning of the field width. The operation is undefined for conversion other than the above.

Note Normally, a decimal point appears only when a digit follows it.

(2) field width

This is an optional minimum field width. If the converted value is smaller than this field width, the left side is filled with spaces (if the left justification flag explained above is assigned, the right side will be filled with spaces). This field width takes the form of "*" or a decimal integer. If "*" is specified, an int type argument is used as the field width. A negative field width is not supported. If an attempt is made to specify a negative field width, it is interpreted as a minus (-) flag appended to the beginning of a positive field width.

(3) precision

For d, i, o, u, x, or X conversion, the value assigned for the precision is the minimum number of digits to appear. For e, f, or E conversion, it is the number of digits to appear after the decimal point. For g or G conversion, it is the maximum number of significant digits. The precision takes the form of "*" or "." followed by a decimal integer. If "*" is specified, an int type argument is used as the precision. If a negative precision is specified, it is treated as if the precision were omitted. If only "." is specified, the precision is assumed to be 0. If the precision appears together with a conversion specification other than the above, the operation is undefined.

(4) size

This is an arbitrary optional size character h, l, or L, which changes the default method for interpreting the data type of the corresponding argument.

When h is specified, a following d, i, o, u, x, or X type specification is forcibly applied to a short or unsigned short argument.

When l is specified, a following d, i, o, u, x, or X type specification is forcibly applied to a long or unsigned long argument. l is also causes a following n type specification to be forcibly applied to a pointer to long argument. If another type specification character is used together with h or l, the operation is undefined.

When L is specified, a following e, E, f, g, or G type specification is forcibly applied to a long double argument. If another type specification character is used together with L, the operation is undefined.

(5) type specification character

These are characters that specify the type of conversion that is to be applied.

The characters that specify conversion types and their meanings are as follows.

%	Output the character "%". No argument is converted. The conversion specification is "%%".
c	Convert an int type argument to unsigned char type and output the characters of the conversion result.
d	Convert an int type argument to a signed decimal number.
e, E	Convert a double type argument to [-]d.dddde±dd format, which has one digit before the decimal point (not 0 if the argument is not 0) and the number of digits after the decimal point is equal to the precision. The E conversion specification generates a number in which the exponent part starts with "E" instead of "e".
f	Convert a double type argument to decimal notation of the form [-]dddd.dddd.
g, G	Convert a double type argument to e (E for a G conversion specification) or f format, with the number of digits in the mantissa specified for the precision. Trailing zeros of the conversion result are excluded from the fractional part. The decimal point appears only when it is followed by a digit.
i	Perform the same conversion as d.
n	Store the number of characters that were output in the same object. A pointer to int type is used as the argument.
p	Output a pointer in an implementation-defined format. The CA850 handles a pointer as unsigned long (this is the same as the lu specification).
o, u, x, X	Convert an unsigned int type argument to octal notation (o), unsigned decimal notation (u), or unsigned hexadecimal notation (x or X) with dddd format. For x conversion, the letters abcdef are used. For X conversion, the letters ABCDEF are used.
s	The argument must be a pointer pointing to a character type array. Characters from this array are output up until the null character (\0) indicating termination (the null character (\0) itself is not included). If the precision is specified, no more than the specified number of characters will be output. If the precision is not specified or if the precision is greater than the size of this array, make sure that this array includes the null character (\0).

vsprintf(char *s, const char *format, va_list arg)

This function applies the format specified by the string pointed to by format to the argument string pointed to by arg, and outputs the formatted data that was output as a result to the array pointed to be s. The vsprintf function is equivalent to sprintf with the list of a variable number of real arguments replaced by arg. arg must be initialized by the va_start macro before the vsprintf function is called.

[Return value]

The number of characters that were output (excluding the null character (\0)) is returned.

Error return does not occur.

[Example]

```
#include <stdio.h>

void func(int val)
{
    char    s[20];

    sprintf(s,"%-10.51x\n", val);
                /* Specifies left-justification, field width 10, precision 5,*/
                /*  size long, and hexadecimal notation for the value of val, */
                /* and outputs the result with an appended new-line character to */
                /* the array pointed to by s. */

}
```

PRINTF

[Overview]

Formatted output

fprintf, printf, vfprintf, vprintf

Caution These functions are not supported by the NEC Electronics integrated debugger or the system simulator.

[Syntax]

```
#include <stdio.h>

int     fprintf(FILE *stream, const char *format[, arg, ...])
int     printf(const char *format,[, arg, ...])
int     vfprintf(FILE *stream, const char *format, va_list arg)
int     vprintf(const char *format, va_list arg)
```

[Description]

Stdin (standard input) and stdout (standard error) are specified for the argument streams in each of the fprintf, printf, vfprintf, and vprintf functions. 1 memory addresses such as an I/O address is allocated for the I/O destination of stream. To use these streams in combination with a debugger, the initial values of the stream structure defined in stdio.h must be set. Be sure to set the initial values prior to calling the function.

Definition of stream structure in stdio.h

```
typedef struct {
    int     mode;       /* with error descriptions */
    unsigend handle;
    int     unget_c;
}FILE;
typedef int fpos_t;

#pragma section sdata begin
extern FILE __struct_stdin;
extern FILE __struct_stdout;
extern FILE __struct_stderr;
#pragma section sdata end
#define stdin(&__struct_stdin)
#define stdout(&__struct_stdout)
#define stderr(&__struct_stderr)
```

The first structure member, mode, indicates the I/O status and is internally defined as ACCSD_OUT/ ADDSD_IN. The third member, unget_c, indicates the pushed-back character (stdin only) setting and is internally defined as -1. When the definition is -1, it indicates that there is no pushed-back character. The second member, handle, indicates the I/O address. Set the value according to the debugger to be used.

Example of I/O address setting

```
__struct_stdout.handle = 0xfffff000;
__struct_stderr.handle = 0x00fff000;
__struct_stdin.handle = 0xfffff002;
extern FILE __struct_stdout;
extern FILE __struct_stderr;
#pragma section sdata end
#define stdin(&__struct_stdin)
#define stdout(&__struct_stdout)
#define stderr(&__struct_stderr)
```

fprintf(FILE *stream, const char *format[, arg, ...])

This function applies the format specified by the string pointed to by format to the respective arg arguments, and outputs the formatted data that was output as a result to stream. Only the standard input/output stdout or stderr can be specified for stream. The method of specifying format is the same as described for the sprintf function. However, fprintf differs from sprintf in that no null character (\0) is output at the end.

printf(const char *format,[, arg, ...])

This function applies the format specified by the string pointed to by format to the respective arg arguments, and outputs the formatted data that was output as a result to the standard input/output stdout. The method of specifying format is the same as described for the sprintf function. However, printf differs from sprintf in that no null character (\0) is output at the end.

vfprintf(FILE *stream, const char *format, va_list arg)

This function applies the format specified by the string pointed to by format to argument string pointed to by arg, and outputs the formatted data that was output as a result to stream. Only the standard input/output stdout or stderr can be specified for stream. The method of specifying format is the same as described for the sprintf function. The vfprintf function is equivalent to fprintf with the list of a variable number of real arguments replaced by arg. arg must be initialized by the va_start macro before the vfprintf function is called.

vprintf(const char *format, va_list arg)

This function applies the format specified by the string pointed to by format to the argument string pointed to by arg, and outputs the formatted data that was output as a result to the standard input/output stdout. The method of specifying format is the same as described for the sprintf function. The vprintf function is equivalent to printf with the list of a variable number of real arguments replaced by arg. arg must be initialized by the va_start macro before the vprintf function is called.

[Return value]

The number of characters that were output is returned.

[Example]

```
#include <stdio.h>

void func(int val)
{
    fprintf(stdout, "%-10.5x\n", val);
}

/* Example using vfprintf in a general error reporting routine */
void error(char *function_name, char *format, ...)
{
    va_list arg;

    va_start(arg, format);

    /* Output function name for which error occurred */
    fprintf(stderr, "ERROR in %s:", function_name);

    /* Output remaining messages */
    vfprintf(stderr, format, arg);

    va_end(arg);
}
```

SSCANF

[Overview]

Formatted input

sscanff

[Syntax]

```
#include <stdio.h>

int    sscanf(const char *s, const char *format[, arg, ...])
```

[Description]

sscanf(const char *s, const char *format[, arg, ...])

This function reads the input to be converted according to the format specified by the character string pointed to by format from the array pointed to by s and treats the arg arguments that follow format as pointers that point to objects for storing the converted input.

An input string that can be recognized and "the conversion that is to be performed for assignment" are specified for format. If sufficient arguments do not exist for format, the operation is undefined. If format is used up even when arguments remain, the remaining arguments are ignored.

The format consists of the following three types of directives:

One or more Space characters	Space ( ), tab (\t), or new-line (\n). If a space character is found in the string when sscanf is executed, all consecutive space characters are read until the next non-space character appears (the space characters are not stored).
Ordinary characters	All ASCII characters other than "%". If an ordinary character is found in the string when sscanf is executed, that character is read but not stored. sscanf reads a string from the input field, converts it into a value of a specific type, and stores it at the position specified by the argument, according to the conversion specification. If an explicit match does not occur according to the conversion specification, no subsequent space character is read.
Conversion specification	Fetches 0 or more arguments and directs the conversion.

Each conversion specification starts with "%". The following appear after the "%":

%[assignment-suppression-character][field-width][size][type-specification-character]

Each conversion specification is explained below.

(1) Assignment suppression character

The assignment suppression character "*" suppresses the interpretation and assignment of the input field.

(2) field width

This is a non-zero decimal integer that defines the maximum field width.

It specifies the maximum number of characters that are read before the input field is converted. If the input field is smaller than this field width, sscanf reads all the characters in the field and then proceeds to the next field and its conversion specification.

If a space character or a character that cannot be converted is found before the number of characters equivalent to the field width is read, the characters up to the white space or the character that cannot be converted are read and stored. Then, sscanf proceeds to the next conversion specification.

(3) size

This is an arbitrary optional size character h, l, or L, which changes the default method for interpreting the data type of the corresponding argument.

When h is specified, a following d, i, n, o, u, or x type specification is forcibly converted to short int type and stored as short type. Nothing is done for c, e, f, n, p, s, D, I, O, U, or X.

When l is specified, a following d, i, n, o, u, or x type specification is forcibly converted to long int type and stored as long type. An e, f, or g type specification is forcibly converted to double type and stored as double type. Nothing is done for c, n, p, s, D, I, O, U, and X.

When L is specified, a following c, i, o, u, or x type specification is forcibly converted to long double type and stored as long double type. Nothing is done for other type specifications.

In cases other than the above, the operation is undefined.

(4) type specification character

These are characters that specify the type of conversion that is to be applied.

The characters that specify conversion types and their meanings are as follows.

%	Match the character "%". No conversion or assignment is performed. The conversion specification is "%%".
c	Scan one character. The corresponding argument should be "char *arg".
d	Read a decimal integer into the corresponding argument. The corresponding argument should be "int *arg".
e, f, g	Read a floating-point number into the corresponding argument. The corresponding argument should be "float *arg".
i	Read a decimal, octal, or hexadecimal integer into the corresponding argument. The corresponding argument should be "int *arg".
n	Store the number of characters that were read in the corresponding argument. The corresponding argument should be "int *arg".
o	Read an octal integer into the corresponding argument. The corresponding argument must be "int *arg".
p	Store the pointer that was scanned. This is an implementation definition. The ca processes %p and %U in exactly the same manner. The corresponding argument should be "void **arg".
s	Read a string into a given array. The corresponding argument should be "char arg[ ]".
u	Read an unsigned decimal integer into the corresponding argument. The corresponding argument should be "unsigned int *arg".

x, X	Read a hexadecimal integer into the corresponding argument. The corresponding argument should be "int *arg".
D	Read a decimal integer into the corresponding argument. The corresponding argument should be "long *arg".
E, F, G	Read a floating-point number into the corresponding argument. The corresponding argument should be "double *arg".
I	Read a decimal, octal, or hexadecimal integer into the corresponding argument. The corresponding argument should be "long *arg".
O	Read an octal integer into the corresponding argument. The corresponding argument should be "long *arg".
U	Read an unsigned decimal integer into the corresponding argument. The corresponding argument should be "unsigned long *arg".
[ ]	Read a non-empty string into the memory area starting with argument arg. This area must be large enough to accommodate the string and the null character (\0) that is automatically appended to indicate the end of the string. The corresponding argument should be "char *arg". The character pattern enclosed by [ ] can be used in place of the type specification character s. The character pattern is a character set that defines the search set of the characters constituting the input field of sscanf. If the first character within [ ] is "^", the search set is complemented, and all ASCII characters other than the characters within [ ] are included. In addition, a range specification feature that can be used as a shortcut is also available. For example, %[0-9] matches all decimal numbers. In this set, "-" cannot be specified as the first or last character. The character preceding "-" must be less in lexical sequence than the succeeding character. Examples %[abcd] Matches character strings that include only a, b, c, and d. %[^abcd] Matches character strings that include any characters other than a, b, c, and d. %[A-DW-Z] Matches character strings that include A, B, C, D, W, X, Y, and Z. %[z-a] Matches z, -, and a (this is not considered a range specification).

Make sure that a floating-point number (type specification characters e, f, g, E, F, and G) corresponds to thefollowing general format.

[+ | -]ddddd[.]ddd[E | e[+ | -]ddd]

However, the portions enclosed by [ ] in the above format are arbitrarily selected, and ddd indicates a decimal,octal, or hexadecimal digit.

[Return value]

The number of input fields for which scanning, conversion, and storage were executed normally is returned. The return value does not include scanned fields that were not stored.

If an attempt is made to read to the end of the file, the return value is EOF.

If no field was stored, the return value is 0.

[Cautions]

- sscanf may stop scanning a specific field before the normal end-of-field character is reached or may stop completely.
- sscanf stops scanning and storing a field and moves to the next field under the following conditions.

(a) The substitution suppression character (*) appears after "%" in the format specification, and the input field at that point has been scanned but not stored.

(b) A field width (positive decimal integer) specification character was read.

(c) The character to be read next cannot be converted according to the conversion specification (for example, if Z is read when the specification is a decimal number).

(d) The next character in the input field does not appear in the search set (or appears in the complement search set).

If sscanf stops scanning the input field at that point because of any of the above reasons, it is assumed that the next character has not yet been read, and this character is used as the first character of the next field or the first character for the read operation to be executed after the input.

- sscanf ends under the following conditions:

(a) The next character in the input field does not match the corresponding ordinary character in the string to be converted.

(b) The next character in the input field is EOF.

(c) The string to be converted ends.

- If a list of characters that is not part of the conversion specification is included in the string to be converted, make sure that the same list of characters does not appear in the input. sscanf scans matching characters but does not store them. If there was a mismatch, the first character that does not match remains in the input as if it were not read.

[Example]

```
#include <stdio.h>

void func(void)
{
    int         i, n;
    float       x;
    const char  *s;
    char        name[10];

    s = "23 11.1e-1 NAME";

    n = sscanf(s,"%d%f%s", &i, &x, name);  /* Stores 23 in i, 1.110000 in x, */
                                           /* and "NAME" in name. */
                                           /* The return value n is 3. */
}
```

SCANF

[Overview]

Formatted input

fscanf, scanf

Caution These functions are not supported by the NEC Electronics integrated debugger or the system simulator.

[Syntax]

```
#include <stdio.h>

int    fscanf(FILE *stream, const char *format[, arg, ...])
int    scanf(const char *format[, arg, ...])
```

[Description]

fscanf(FILE *stream, const char *format[, arg, ...])

reads the input to be converted according to the format specified by the character string pointed to by format from stream and treats the arg arguments that follow format as objects for storing the converted input. Only the standard input/output stdin can be specified for stream. The method of specifying format is the same as described for the sscanf function.

scanf(const char *format[, arg, ...])

reads the input to be converted according to the format specified by the character string pointed to by format from the standard input/output stdin and treats the arg arguments that follow format as objects for storing the converted input. The method of specifying format is the same as described for the sscanf function.

[Return value]

The return value is similar to the one described for sscanf. See the section about sscanf.

[Example]

```
#include <stdio.h>

void func(void)
{
    int     i, n;
    double  x;
    char    name[10];

    n = scanf("%d%lf%s", &i, &x, name);     /* Perform formatted input of input */
                                            /* from stdin using the format */
                                            /* "23 11.1e-1 NAME" */
}
```

6.9 Standard Utility Functions

This section explains the utility functions useful for various programs. Definitions and declarations related to these functions are described in the "stdlib.h" file

Table 6 - 22 Standard Utility Functions

Classification	Function Name	Outline
ABS	abs	Absolute value (int type)
	labs	Absolute value (long type)
BSEARCH	bsearch	Binary search
	qsort	Sorting
DIV	div	Division (int type)
	ldiv	Division (long type)
ECVTF	ecvtf	Conversion of floating-point value to numeric character string (total number of characters specified)
	fcvtf	Conversion of floating-point value to numeric character string (number below decimal point specified)
	gcvtf	Conversion of floating-point value to numeric character string (format specified)
ITOA	itoa	Conversion of integer (int type) to character string
	ltoa	Conversion of integer (long type) to character string
	ultoa	Conversion of integer (unsigned long type) to character string
MALLOC	calloc	Dynamic memory allocation
	free	Dynamic memory release
	malloc	Dynamic memory allocation
	realloc	Dynamic memory reallocation
RAND	rand	Pseudo random number generation
	srand	Setting of pseudo random number seed
STRTODF	atoff	Conversion of character string to floating point
	strtodf	Conversion of character string to floating point (stores pointer in last character string)
STRTOL	atoi	Absolute value (int type)
	atol	Absolute value (long type)
	strtol	Binary search
	strtoul	Sorting

ABS

[Overview]

Integer absolute value

abs, labs

[Syntax]

```
#include <stdlib.h>

int     abs(int j)
long    labs(long j)
```

[Description]

abs(int j)

This function obtains the absolute value of j (size of j), | j |. If j is a negative number, the result is the reversal of j. If j is not negative, the result is j.

labs(long j)

This function is the same as abs, but uses long type instead of int type, and the return value is also of long type.

[Return value]

Returns the absolute value of j (size of j), | j |.

[Example]

```
#include <stdlib.h>

void func(int i)
{
    int val;

    val = -15;
    i = abs(val);    /* Returns absolute value of val, 15, to 1. */
}
```

BSEARCH

[Overview]

Binary search

bsearch, qsort

[Syntax]

```
#include <stdlib.h>

void bsearch(const void *key, const void *base, size_t nmemb, size_t size,
int(*compar)(const void *, const void*))
void qsort(void *base, size_t nmemb, size_t size, int(*compar)(const void*, const
void*))
```

[Description]

bsearch(const void *key, const void *base, size_t nmemb, size_t size, int(*compar)(const void *, const void*))

This function searches an element that coincides with key from an array starting with base by means of binary search. nmemb is the number of elements of the array. size is the size of each element. The array must be arranged in the ascending order in respect to the compare function indicated by compar (last argument). Define the compare function indicated by compar to have two arguments. If the first argument is less than the second, a negative integer must be returned as the result. If the two arguments coincide, zero must be returned. If the first is greater than the second, a positive integer must be returned.

qsort(void *base, size_t nmemb, size_t size, int(*compar)(const void*, const void*))

This function sorts the array pointed to by base into ascending order in relation to the comparison function pointed to by compar. nmemb is the number of array elements, and size is the size of each element. The comparison function pointed to by compar is the same as the one described for bsearch.

[Return value]

A pointer to the element in the array that coincides with key is returned. If there are two or more elements that coincide with key, the one that has been found first is indicated.

[Example]

```
#include <stdlib.h>
#include <string.h>

int     compar(char **x, char **y);

void func(void)
{
    static char *base[] = {"a", "b", "c", "d", "e", "f"};
    char        *key = "c";     /* Search key is "c". */
    char        **ret;

    ret =(char **)bsearch((char *)&key,(char *)base, 6,
    sizeof(char *), compar);    /* Pointer to "c" is stored in ret. */
}

int compar(char **x, char **y)
{
    return(strcmp(*x, *y));     /* Returns positive, zero, or negative /*
                                /* integer as result of comparing arguments. */
}
```

DIV

[Overview]

Division

div, ldiv

[Syntax]

```
#include <stdlib.h>

div_t  div(int n, int d)
ldiv_t ldiv(long n, long d)
```

[Description]

div(int n, int d)

This function calculates the quotient and remainder resulting from dividing numerator n by denominator d, and stores these two integers as the members of the following structure div_t.

```
typedef struct {
    int quot;
    int rem;
} div_t
```

quot the quotient, and rem is the remainder. If d is not zero, and if "r = div(n, d);", n is a value equal to "r.rem + d * r.quot".

If d is zero, the resultant quot member has a sign the same as n and has the maximum size that can be expressed. The rem member is 0.

ldiv(long n, long d)

This function is used to divide a value of long type, not a value of int type. The result is stored as the member of the following structure ldiv_t.

```
typedef struct {
    long    quot;
    long    rem;
} ldiv_t
```

[Return value]

The structure storing the result of the division is returned.

[Example]

```
#include <stdlib.h>

void func(void)
{
    div_t r;

    r = div(110, 3);      /* 36 is stored in r.quot, and 2 is stored in r.rem. */
}
```

ECVTF

[Overview]

Conversion from floating point number to character string

ecvtf, fcvtf, gcvtf

[Syntax]

```
#include <stdlib.h>

char    *ecvtf(float val, int chars, int *decpt, int *sgn)
char    *fcvtf(float val, int decimals, int *decpt, int *sgn)
char    *gcvtf(float val, int prec, char *buf)
```

[Description]

ecvtf(float val, int chars, int *decpt, int *sgn)

This function generates a character string indicating a numeric value val of float type in number (terminated with the null character (\0)). The second argument chars specifies the total number of characters to be written (because only numbers are written, this argument specifies the valid number of numerals in the converted character string). The digits of the integer of val are always included.

fcvtf(float val, int decimals, int *decpt, int *sgn)

This function is the same as ecvt, except the interpretation of the second argument. The second argument decimals specify the number of characters to be written after the decimal point.

ecvtf and fcvtf only write a number to an output character string. Therefore, record the position of the decimal point to *decpt and the sign of the numeric value to *sgn. After the number has been formatted, the number of digits at the left of the decimal point is stored in *decpt. If the numeric value is positive, 0 is stored in *sgn; if it is negative, 1 is stored.

gcvtf(float val, int prec, char *buf)

This function converts a numeric value into a character string, and stores it to buffer buf. gcvtf uses the same rule as the format "%.prec" (sign is appended to the negative number only) of sprintf, and selects an exponent format or normal decimal point format according to the valid number of digits (specified by prec).

[Return value]

ecvtf, fcvtf	Returns a pointer indicating a new character string including the character string representation of val.
gcvtf	Returns a pointer (same as argument buf) to the formatted character string representation of val.

[Example]

```
#include <stdlib.h>

void func(void)
{
    float   val;
    int     dec, sgn;

    val = 111.11;
    ecvtf(val, 12, &dec, &sgn);
        /* Converts value 111.11 of val to character string of 12 characters. */
        /* dec records number of digits, 3, at left of decimal point,  */
        /* and sgn records sign(0 because numeric value is positive). */
}
```

ITOA

[Overview]

Conversion from integer to character string

itoa, ltoa, ultoa

[Syntax]

```
#include <stdlib.h>

char   *itoa(int value, char *string, int radix)
char   *ltoa(long int value, char *string, int radix)
char   *ultoa(unsigned long int value, char *string, int radix)
```

[Description]

itoa(int value, char *string, int radix)

This function converts an int type numeric value to a character string for a radix-based number and stores it in the array indicated by string. The terminating null character (\0) always is added at the end of the character string. Numeric values from 2 to 36 can be specified for radix. If radix is 10, value is handled as a signed numeric value, and when value < 0, the "-" character is appended at the beginning of the character string. Otherwise, value is handled as an unsigned numeric value. If radix > 10, the lowercase letters a to z are assigned for 10 to 35.

ltoa(long int value, char *string, int radix)

This function converts a long int type numeric value to a character string for a radix-based number and stores it in the array indicated by string. Except for the type of value, this is the same as itoa.

ultoa(unsigned long int value, char *string, int radix)

This function converts an unsigned long int type numeric value to a character string for a radix-based number and stores it in the array indicated by string. Except for the type of value, this is the same as itoa

[Return value]

itoa, ltoa, ultoa	string is returned.

[Example]

```
#include <stdlib.h>

void func(void)
{
    char    buf[128];

    itoa(12345, buf, 16); 3/* Converts 12345 to a hexadecimal character string */
}
```

MALLOC

[Overview]

Memory allocation and management

calloc, free, malloc, realloc

[Syntax]

```
#include <stdlib.h>

void   *calloc(size_t nmemb, size_t size)
void   free(void *ptr)
void   *malloc(size_t size)
void   *realloc(void *ptr, size_t size)
```

[Description]

The memory area management functions automatically allocate memory area as necessary from the heap memory area. Also, since the compiler does not automatically allocate this area, when calloc, malloc, or realloc is used, the heap memory area must be allocated. The area allocation should be performed first by an application.

Heap memory setup example

```
#define SIZEOF_HEAP 0x1000
int      __sysheap[SIZEOF_HEAP>>2];
size_t __sizeof_sysheap = SIZEOF_HEAP;
```

Remarks 1 The symbol "sysheap" (three underscores "_") of the variable " _sysheap" (two under-scores "_") points to the starting address of heap memory. This value should be a word integer value.

2 The required heap memory size (bytes) should be set for the variable "_sizeof_sysheap" (two leading underscores). If assembly language is used for coding, this value should be set for the symbol " _sizeof_sysheap" (three leading underscores).

calloc(size_t nmemb, size_t size)

This function allocates an area for an array of nmemb elements. The allocated area is initialized to zeros.

free(void *ptr)

This function releases the area pointed to by ptr so that this area is subsequently available for allocation. The area that was acquired by calloc, malloc, or realloc must be specified for ptr.

malloc(size_t size)

This function allocates an area having a size indicated by size. The area is not initialized.

realloc(void *ptr, size_t size)

This function changes the size of the area pointed to by ptr to the size indicated by size. The contents of the area are unchanged up to the smaller of the previous size and the specified size. If the area is expanded, the contents of the area greater than the previous size are not initialized. When ptr is a null pointer, the operation is the same as that of malloc (size). Otherwise, the area that was acquired by calloc, malloc, or realloc must be specified for ptr.

[Return value]

calloc, malloc, realloc	When area allocation succeeds, a pointer to that area is returned. When the area could not be allocated, a null pointer is returned.

[Example]

```
#include <stdlib.h>

typedef struct {
    double  d[3];
    int     i[2];
} s_data;

int func(void)
{
    sdata *buf;
    int     i;

    /* Allocate an area for 40 s_data*/
    if((buf = calloc(40, sizeof(s_data))) == NULL) return(1);

    for(i = 0; i<40; i++)
    {

    }

    /* Release the area */
    free(buf);

    return(0);
}
```

[Cautions]

The memory area to be acquired or released by the calloc, free, malloc, and realloc functions is called the heap area. The area for allocating this heap area and its size must be set in advance. When the heap area is set via a C language source program, it is written as [Heap memory setup example], however, when specified using the assembler, the following program should be added to the startup module.

If the heap area is specified by the C language source program and assembler simultaneously, an error occurs, so specify via one or the other.

```
#-----------------------------------------------------------
#    system heap
#-----------------------------------------------------------
    .set    HEAPSIZE, 0x1000
    .globl  __sysheap
    .bss
    .lcomm  __sysheap, HEAPSIZE, 4
    .data
    .globl  __sizeof_sysheap
__sizeof_sysheap:
    .word HEAPSIZE

# In this example, a heap area of 0x1000 bytes is allocated in the .bss area.
```

RAND

[Overview]

Pseudo random number generation

rand, srand

[Syntax]

```
#include <stdlib.h>

int    rand()
void   srand(unsigned int seed)
```

[Description]

rand()

This function returns a random number that is greater than or equal to zero and less than or equal to RAND_MAX.

srand(unsigned int seed)

This function assigns seed as the new pseudo random number sequence seed to be used by the rand call that follows. If srand is called using the same seed value, the same numbers in the same order will appear for the random numbers that are obtained by rand. If rand is executed without executing srand, the results will be the same as when srand(1) was first executed.

[Return value]

rand	Random numbers are returned.

[Example]

```
#include <stdlib.h>

void func(void)
{
    if(rand()& 0xf)<4) func1();    /* Execute func1 with a probability of 25% */
}
```

STRTODF

[Overview]

Conversion from character string to floating-point number

atoff, strtodf

[Syntax]

```
#include <stdlib.h>

float  atoff(const char *str)
float  strtodf(const char *str, char **p)
```

[Description]

atoff(const char *str)

This function converts the first portion of the character string indicated by str into a float type representation. atoff is the same as the following.

```
strtodf(str, NULL);
```

strtodf(const char *str, char **p)

This function converts the first part of the character string indicated by str into a long type representation. The part of the character string to be converted is in the following format and is at the beginning of str with the maximum length, starting with a normal character that is not a space.

```
[+|-]digits[.][digits][(e|E)[+|-]digits]
```

If str is vacant or consists of space characters only, if the first normal character is other than "+", "-", ".", or a numeral, the partial character string does not include a character. If the partial character string is vacant, conversion is not executed, and the value of str is stored in the area indicated by ptr. If the partial character string is not vacant, it is converted, and a pointer to the last character string (including the null character (\0) indicating at least the end of str) is stored in the area indicated by ptr. This function is not re-entrant.

[Return value]

If the partial character string has been converted, the resultant value is returned. If the character string could not be converted, 0 is returned. If an overflow occurs (the value is not in the range in which it can be expressed), HUGE_VAL or -HUGE_VAL is returned, and ERANGE is set to global variable errno. If an underflow occurs, 0 is returned, and macro ERANGE is set to global variable errno.

[Example]

```
#include <stdlib.h>
#include <stdio.h>

void func(float ret)
{
    char    *p, *str, s[30];

    str = "+5.32a4e";
    ret = strtodf(str, &p);      /* 5.320000 is returned to ret, and pointer */
                                 /* to "a" is stored in area of p. */

    sprintf(s, "%lf\t%c", ret, *p);   /* "5.320000 a" is stored in array */
                                      /* indicated by s. */
}
```

STRTOL

[Overview]

Conversion from character string to integer

atoi, atol, strtol, strtoul

[Syntax]

```
#include <stdlib.h>

int     atoi(const char *str)
long    atol(const char *str)
long    strtol(const char *str, char **ptr, int base)
unsigned long strtoul(const char *str, char **ptr, int base)
```

[Description]

atoi(const char *str)

This function converts the first part of the character string indicated by str into an int type representation. atoi is the same as the following.

```
(int) strtol(str, NULL, 10);
```

atol(const char *str)

This function converts the first part of the character string indicated by str into a long int type representation. atol is the same as the following.

```
strtol(str, NULL, 10);
```

strtol(const char *str, char **ptr, int base)

This function converts the first part of the character string indicated by str into a long type representation. strol first divides the input characters into the following three parts: the "first blank", "a string represented by the base number determined by the value of base and is subject to conversion into an integer", and "the last one or more character string that is not recognized (including the null character (\0))". Then strtol converts the string into an integer, and returns the result.

(1) Specify 0 or 2 to 36 as argument base.

(a) If base is 0, the expected format of the character string subject to conversion is of integer format having an optional + or - sign and "0x", indicating a hexadecimal number, prefixed.

(b) If the value of base is 2 to 36, the expected format of the character string is of character string or numeric string type having an optional + or - sign prefixed and expressing an integer whose base is specified by base. Characters "a" (or "A") through "z" (or "Z") are assumed to have a value of 10 to 35. Only characters whose value is less than that of base can be used.

(c) If the value of base is 16, "0x" is prefixed (suffixed to the sign if a sign exists) to the string of characters and numerals (this can be omitted).

(2) The string subject to conversion is defined as the longest partial string at the beginning of the input character string that starts with the first character other than blank and has an expected format.

(a) If the input character string is vacant, if it consists of blank only, or if the first character that is not blank is not a sign or a character or numeral that is permitted, the subject string is vacant.

(b) If the string subject to conversion has an expected format and if the value of base is 0, the base number is judged from the input character string. The character string led by 0x is regarded as a hexadecimal value, and the character string to which 0 is prefixed but x is not is regarded as an octal number. All the other character strings are regarded as decimal numbers.

(c) If the value of base is 2 to 36, it is used as the base number for conversion as mentioned above.

(d) If the string subject to conversion starts with a - sign, the sign of the value resulting from conversion is reversed.

(3) The pointer that indicates the first character string

(a) This is stored in the object indicated by ptr, if ptr is not a null pointer.

(b) If the string subject conversion is vacant, or if it does not have an expected format, conversion is not executed.

(c) The value of str is stored in the object indicated by ptr if ptr is not a null pointer.

This function is not re-entrant.

strtoul(const char *str, char **ptr, int base)

This function is the same as strtol except that the type of the return value is of unsigned long type.

[Return value]

atoi, atol	Returns the converted value if the partial character string could be converted. If it could not, 0 is returned.
strtol	Returns the converted value if the partial character string could be converted. If it could not, 0 is returned. If an overflow occurs (because the converted value is too great), LONG_MAX or LONG_MIN is returned, and macro ERANGE is set to global variable errno.
strtoul	Returns the converted value if the partial character string could be converted. If it could not, 0 is returned. If an overflow occurs, ULONG_MAX is returned, and macro ERANGE is set to global variable errno.

[Example]

```
#include <stdlib.h>

void func(long ret)
{
    char *p;

    ret = strtol("10", &p, 0);     /* 10 is returned to ret. */

    ret = strtol("0x10", &p, 0);   /* 16 is returned to ret. */

    ret = strtol("10x", &p, 2);    /* 2 is returned to ret, and pointer to "x" */
                                   /* is returned to area of p. */

    ret = strtol("2ax3", &p, 16);  /* 42 is returned to ret, and pointer to "x" */
                                   /* is returned to area of p. */
         :
}
```

6.10 Non-Local Jump Functions

This section describes the non-local jump functions.

Declarations and definitions concerning these functions are described in the setjmp.h file

Table 6 - 23 Non-Local Jump Functions

Classification	Function Name	Outline
SETJMP	setjmp	Sets the destination of the non-local jump
	longjmp	Non-local jump

SETJMP

[Overview]

Non-local jumps

setjmp, longjmp

[Syntax]

```
#include <setjmp.h>

int    setjmp(jmp_buf env)
void   longjmp(jmp_buf env, int val)
```

[Description]

setjmp(jmp_buf env)

This function sets env as the destination for a non-local jump. In addition, the environment in which setjmp was run is saved to env.

longjmp(jmp_buf env, int val)

This function performs a non-local jump to the place immediately after setjmp using env saved by setjmp.val as a return value for setjmp.

[Return value]

setjmp	0 is returned if returning from setjmp. The second argument val in longjmp is returned if a non-local jump is performed by longjmp. However, 1 is returned if val is 0.

[Example]

```
#include <setjmp.h>

#define ERR_XXX1    1
#define ERR_XXX2    2

jmp_buf jmp_env;

void func(void)
{
    for(;;) {
        switch(setjmp(jmp_env))
        {
        case ERR_XXX1 :
            /* Termination of error XXX1 */
            break;
        case ERR_XXX2 :
            /* Termination of error XXX2 */
            break;
        case 0 :                /* No non-local jumps */
        default :
            break;
        }
    }
}

void func1(void)
{
    longjmp(jmp_env, ERR_XXX1);   /* Non-local jumps are performed upon */
                                  /* generation of error XXX1 */

    longjmp(jmp_env, ERR_XXX2);   /* Non-local jumps are performed upon */
                                  /* generation of error XXX2 */
}
```

6.11 Mathematical Functions

This section explains the mathematical functions.

Definitions and declarations related to these functions are described in the "math.h" file.

Mathematical library libm.a internally references standard library libc.a. When referencing libm.a by starting ld850 alone, therefore, libc.a must also be referenced. When referencing two or more archive files by starting ld850 alone, undefined symbols are searched in the sequence in which reference is specified; therefore, specify the libc.a reference "-lc" at the end of the sequence.

If ld850 is started from the compiler, however, the libc.a file is automatically referenced

Table 6 - 24 Mathematical Functions

Classification	Function Name	Outline
BESSEL	j0f	Bessel function of first kind (0 degree)
	j1f	Bessel function of first kind (first degree)
	jnf	Bessel function of first kind (n degree)
	y0f	Bessel function of second kind (0 degree)
	y1f	Bessel function of second kind (first degree)
	ynf	Bessel function of second kind (n degree)
ERFF	erff	Error function (approximate value)
	erfcf	Error function (complementary probability)
EXPF	expf	Exponential function
	logf	Logarithmic function (natural logarithm)
	log2f	Logarithmic function (base 2)
	log10f	Logarithmic function (base 10)
	powf	Power function
	cbrtf	Cubic root function
	sqrtf	Square root function
FLOORF	ceilf	ceiling function
	fabsf	Absolute value function
	floorf	floor function
	fmodf	Remainder function
FREXPF	frexpf	Divides floating-point number into mantissa and power
	ldexpf	Converts floating-point number to power
	modff	Divides floating-point number into integer and decimal
GAMMAF	gammaf	Logarithmic gamma function
HYPOTF	hypotf	Euclidean distance function
MATHERR	matherr	Error processing function

Table 6 - 24 Mathematical Functions

Classification	Function Name	Outline
SINHF	acoshf	Inverse hyperbolic function, cosine
	asinhf	Inverse hyperbolic function, sine
	atanhf	Inverse hyperbolic function, tangent
	coshf	Hyperbolic function, cosine
	sinhf	Hyperbolic function, sine
	tanhf	Hyperbolic function, tangent
TRIG	acosf	Inverse cosine
	asinf	Inverse sign
	atanf	Inverse tangent
	atan2f	Inverse tangent (y/x)
	cosf	Cosine
	sinf	Sign
	tanf	Tangent

BESSEL

[Overview]

Bessel function

j0f, j1f, jnf, y0f, y1f, ynf

[Syntax]

```
#include <math.h>

float  jnf(int n, float x)
float  j0f(float x)
float  j1f(float x)
float  ynf(int n, float x)
float  y0f(float x)
float  y1f(float x)
```

[Description]

A Bessel function is a function that is the solution to the following differential equation.

$$x^2\frac{d^2y}{dx^2} + \frac{dy}{dx} + (x^2 - p^2)y = 0$$

jnf(int n, float x)

This function calculates the Bessel function of the first kind of the n degree.

j0f(float x)

This function calculates the Bessel functions of the first kind of the 0 degrees.

j1f(float x)

This function calculates the Bessel functions of the first kind of the first degrees.

ynf(int n, float x)

This function calculates the Bessel function of the second kind of the n degree.

y0f(float x)

This function calculates the Bessel functions of the second kind of the 0 degrees.

y1f(float x)

This function calculates the Bessel functions of the second kind of the first degrees.

[Return value]

jnf	Returns the Bessel function of the first kind of the n degree.
j0f	Returns the Bessel function of the first kind of the 0 degree.
j1f	Returns the Bessel function of the first kind of the first degree.
ynf	Returns the Bessel function of the second kind of the n degree.
y0f	Returns the Bessel function of the second kind of the 0 degree.
y1f	Returns the Bessel function of the second kind of the first degree.

[Example]

```
#include <math.h>

float func(void)
{
    float   ret, x;

    ret = j1f(x);   /* Calculates Bessel function of first kind and */
                    /* first decree in response to value of x, */
                    /* and returns function to ret. */
         :
    return(ret);
}
```

ERFF

[Overview]

Error function

erff, erfcf

[Syntax]

```
#include <math.h>

float  erff(float x)
float  erfcf(float x)
```

[Description]

erff(float x)

This function calculates the approximate value (numeric value between 0 and 1) of the "error function" that estimates the probability for which the observed value is in a range of standard deviation x. The expression that defines the error function is as follows.

$$\frac{2}{\sqrt{\pi}} \times \int_0^x e^{-t^2} dt$$

erfcf(float x)

This function calculates complementary probability through "1.0-erff(x)". This function is provided to prevent the accuracy from dropping if erff(x) is called by x with a large value and the result is subtracted from 1.0.

[Return value]

erff	Returns the approximate value (numeric value between 0 and 1) of the "error function".
erfcf	Returns the complementary probability.

[Example]

```
#include <math.h>

float func(void)
{
    float   ret, x;
    ret = erff(x);    /* Calculates approximate value of error function in */
                      /* response to value of x and returns it to ret. */
         :
    return(ret);
}
```

EXPF

[Overview]

Exponent/logarithm/power/cubic root/square root function

expf, logf, log2f, log10f, powf, cbrtf, sqrtf

[Syntax]

```
#include <math.h>

float  expf(float x)
float  logf(float x)
float  log2f(float x)
float  log10f(float x)
float  powf(float x, float y)
float  cbrtf(float x)
float  sqrtf(float x)
```

[Description]

expf(float x)

This function calculates the xth power of e (e is the base of a natural logarithm and is about 2.71828).

logf(float x)

This function calculates the natural logarithm of x, i.e., logarithm with base e.

log2f(float x)

This function calculates the logarithm of x with base 2. This is realized by "log(x)/log(2)".

log10f(float x)

This function calculates the logarithm of x with base 10. This is realized by "log(x)/log(10)".

powf(float x, float y)

This function calculates the yth power of x.

cbrtf(float x)

This function calculates the cubic root of x.

sqrtf(float x)

This function calculates the square root of x.

[Return value]

expf	Returns the xth power of e. expf returns an unnormalized value if an underflow occurs (if x is a negative number that cannot express the result), and sets macro ERANGE to global variable errno. If an overflow occurs (if x is too great a number), HUGE_VAL (maximum double type numerics that can be expressed) is returned, and macro ERANGE is set to global variable errno.
logf	Returns the natural logarithm of x. logf returns a non-numeric value and sets macro EDOM to global variable errno if x is negative. If x is zero, it returns -∞ (0xff800000) and sets macro ERANGE to global variable errno.
log2f	Returns the logarithm of x with base 2. log2f returns a non-numeric value and sets macro EDOM to global variable errno if x is negative. If x is zero, it returns -∞ and sets macro ERANGE to global variable errno.
log10f	Returns the logarithm of x with base 10. log10f returns a non-numeric value and sets macro EDOM to global variable errno if x is negative. If x is zero, it returns -∞ and sets macro ERANGE to global variable errno.
powf	Returns the yth power of x. powf returns a negative solution only if x < 0 and y is an odd integer. If x < 0 and y is a non-integer or if x = y = 0, powf returns a non-numeric value and sets the macro EDOM for the global variable errno. If x = 0 and y < 0 or if an overflow occurs, powf returns ±HUGE_VAL and sets the macro ERANGE for errno. If the solution vanished approaching zero, powf returns ±0 and sets the macro ERANGE for errno. If the solution is a non-normalized number, powf sets the macro ERANGE for errno.
cbrtf	Returns the cubic root of x.
sqrtf	Returns the positive square root of x. sqrtf returns a non-numeric value and sets macro EDOM to global variable errno if x is a negative real number.

The error processing of these functions can be changed by using the matherr function.

[Example]

```
#include <math.h>

float func(void)
{
    float   ret, x, y;

    ret = powf(x, y);        /* Returns yth power of x to ret. */
        :
    return(ret);
}
```

FLOORF

[Overview]

ceiling/absolute value/floor/remainder function

ceilf, fabsf, floorf, fmodf

[Syntax]

```
#include <math.h>

float  ceilf(float x)
float  fabsf(float x)
float  floorf(float x)
float  fmodf(float x, float y)
```

[Description]

ceilf(float x)

This function calculates the minimum integer value greater than x.

fabsf(float x)

This function fabsf (float x) calculates the absolute value (size) of x by directly manipulating the bit representation of x.

floorf(float x)

This function calculates the maximum integer value less than x.

fmodf(float x, float y)

This function calculates a floating-point value that is the remainder resulting from dividing x by y. In other words, it calculates the value "x - i * y" for the minimum integer i that has a sign the same as x and is less than y, if y is not zero

[Return value]

ceilf	Returns the minimum integer greater than x.
fabsf	Returns the absolute value (size) of x.
floorf	Returns the maximum integer value less than x.
fmodf	Returns a floating-point value that is the remainder resulting from dividing x by y. fmodf(x, 0) returns x.

The error processing of these functions can be changed by using the matherr function.

[Example]

```
#include <math.h>

float func(void)
{
    float   ret, x, y;

    ret = fmodf(x, y);     /* Returns remainder resulting from dividing x by y to ret. */
        :
    return(ret);
}
```

FREXPF

[Overview]

Manipulation of each part of floating-point number

frexpf, ldexpf, modff

[Syntax]

```
#include <math.h>

float  frexpf(float val, int *exp)
float  ldexpf(float val, int exp)
float  modff(float val, float *ipart)
```

[Description]

All numbers other than zero can be expressed as m x 2^p.

frexpf(float val, int *exp)

This function expresses val of float type as mantissa m and the pth power of 2. The resulting mantissa m is 0.5 <= |x| < 1.0, unless val is zero. p is stored in *exp. m and p are calculated so that val = m x 2^p.

ldexpf(float val, int exp)

This function calculates val x 2^{exp}.

modff(float val, float *ipart)

This function divides val of float type into integer and decimal parts, and stores the integer part in *ipart. Rounding is not performed. It is guaranteed that the sum of the integer part and decimal part accurately coincides with val.

For example, where realpart = modff (val, &intpart), "realpart + intpart" coincides with val.

[Return value]

frexpf	Returns mantissa m. frexpf sets 0 to *exp and returns 0 if val is 0. Although the value of val can be changed by using the matherr function, the setting of *exp cannot be changed.
ldexpf	Returns the value calculated by val x 2^{exp}. IIf an underflow or overflow occurs as a result of executing ldexpf, macro ERANGE is set to global variable errno. If an underflow occurs, ldexpf returns an unnormalized value. If an overflow occurs, it returns ∞ (+∞ = 0x7f800000, -∞ = 0xff800000) with the same sign as HUGE_VAL. This error processing can be changed by using the matherr function.
modff	Returns a decimal part. The sign of the result is the same as the sign of val.

[Example]

```
#include <math.h>

float func(void)
{
    float   ret, x;
    int     exp;

    x = 5.28;

    ret = frexpf(x, &exp);    /* Resultant mantissa 0.66 is returned to ret, */
                              /* and 3 is stored in exp */
         :
    return(ret);
}
```

GAMMAF

[Overview]

Logarithmic gamma function

gammaf

[Syntax]

```
#include <math.h>

float  gammaf(float x)
```

[Description]

gammaf(float x)

This function calculates In(Γ(x)), i.e., the natural logarithm of the gamma function of x. The gamma function (expf (gammaf(x)) is a generalized factorial, and has a relational expression of Γ(N) ≡ N x Γ(N - 1). Therefore, the result of the gamma function itself increases very rapidly. Consequently, gammaf is defined as "In(Γ(x))", instead of simply "Γ(x)", to expand the valid range of the result that can be expressed.

[Return value]

The natural logarithm of the gamma function of x is returned.

If x is 0 or an overflow occurs, HUGE_VAL is returned, and macro ERANGE is set to global variable errno.

This error processing can be changed by using the matherr function.

[Example]

```
#include <math.h>

float func(float x)
{
    float   ret;

    ret = gammaf(x);    /* Returns natural logarithm of gamma function of x to ret. */
         :
    return(ret);
}
```

HYPOTF

[Overview]

Euclidean distance function

hypotf

[Syntax]

```
#include <math.h>

float  hypotf(float x, float y)
```

[Description]

hypotf(float x, float y)

This function calculates a Euclidean distance

$\sqrt{x^2 + y^2}$ between the origin (0, 0) and a point indicated by Cartesian coordinates (x, y).

[Return value]

Returns a Euclidean distance

$\sqrt{x^2 + y^2}$ between the origin (0, 0) and a point indicated by Cartesian coordinates (x, y).

If an overflow occurs, HUGE_VAL is returned, and macro ERANGE is set to global variable errno.

This error processing can be changed by using the matherr function.

[Example]

```
#include <math.h>

float func(float x)
{
    float   ret, y;

    ret = hypotf(x, y);  /* Returns Euclidean distance between origin (0, 0) */
                         /* and coordinates (x, y) to ret. */
         :
    return(ret);
}
```

MATHERR

[Overview]

Error processing function

matherr

[Syntax]

```
#include <math.h>

int    matherr(struct exception *e)
```

[Description]

matherr(struct exception *e)

This is a function that is called if an error occurs in a mathematical library function. By preparing a function named matherr via a user subroutine, therefore, error processing can be customized. Customized matherr must return 0 if resolution of an error has failed, and a value other than 0 if the error has been resolved. If matherr returns a value other than 0, the value of global variable errno is not changed.

Error processing can be customized by using the information passed by pointer *e to structure exception. Structure exception is defined as follows in "math.h".

```
#if !defined(__cplusplus)
#define __exception exception
#endif
struct exception{
    int     type;
    char    *name;
    double  arg1, arg2, retval;
};
```

The meaning of each member is as follows:

type	Type of mathematical function error that has occurred. The type of the macro encoding error is also defined in "math.h".
name	Pointer indicating a character string that holds the name of the mathematical library function in which an error has occurred, and ends with a space character.
arg1, arg2	Arguments responsible for the error.
retval	Error return value that is returned by the calling function.

The types of mathematical library function errors that may occur are as follows.

DOMAIN	The argument is not in the range of the definition area of the function. Example: logf(-1)
OVERFLOW	Overflow Example: expf(1000)
UNDERFLOW	Underflow, solutions to non-normalized number. Solution < 1.1755e-38 and non 0 and precision is lower than the normal value.
Z_DIVISION	Zero division.

Calling matherr when an operation exception occurs and updating global variable errno with a standard function are not re-entrant.

[Return value]

By changing the value of e ->retval, the result of the function called from the customized matherr can be changed. This also applies to the function on the calling side.

The matherr returns a value other than 0 if the error has been resolved, and 0 if the error could not be resolved. If matherr returns 0, set an appropriate value to global variable errono on the calling side.

[Example]

```
#include <math.h>
#include <stdio.h>

float func(void)
{
    float   ret;
    ret = logf(-0.1);           /* 3 is returned to ret. */
        :
    return(ret);
}

int matherr(struct exception *e)
{
    char    s[30];

    switch(e->type) {
        case DOMAIN:
            sprintf(s, "%s DOMAIN error %e\n", e->name, e->arg1);
            e->retval = 3;      /* Changes error return value to 3. */
            break;
        default:
            sprintf(s, "%s other error %e\n", e->name, e->arg1);
    }

    return(1);
}
```

SINHF

[Overview]

Hyperbolic functions

acoshf, asinhf, atanhf, coshf, sinhf, tanhf

[Syntax]

```
#include <math.h>

float  acoshf(float x)
float  asinhf(float x)
float  atanhf(float x)
float  coshf(float x)
float  sinhf(float x)
float  tanhf(float x)
```

[Description]

acoshf(float x)

This function calculates the inverse hyperbolic cosine of x (where x is a numeric value of 1 or greater). The definition expression is as follows.

$$\ln(x + \sqrt{x^2 - 1})$$

asinhf(float x)

This function calculates the inverse hyperbolic sine of x. The definition expression is as follows.

$$\mathrm{sign}(x) \times \ln(|x| + \sqrt{1 + x^2})$$

atanhf(float x)

This function calculates the inverse hyperbolic tangent of x.

coshf(float x)

This function calculates the hyperbolic cosine of x. Specify the angle in radian. The definition expression is as follows.

$$\frac{(e^x + e^{-x})}{2}$$

sinhf(float x)

This function calculates the hyperbolic sine of x. Specify the angle in radian. The definition expression is as follows.

$$\frac{(e^{x} - e^{-x})}{2}$$

tanhf(float x)

This function calculates the hyperbolic tangent of x. Specify the angle in radian. The definition expression is as follows.

```
sinh(x) / cosh(x)
```

[Return value]

acoshf	Returns the inverse hyperbolic cosine of x (x is a numeric number of 1 or greater). acoshf returns a non-numeric value if x is less than 1. Macro EDOM is set to global variable errno.
asinhf	Returns the inverse hyperbolic sine of x.
atanhf	Returns the inverse hyperbolic tangent of x. atanhf returns a non-numeric value and sets macro EDOM to global variable errno if the absolute value of x is greater than 1.
coshf	Returns the hyperbolic cosine of x. coshf returns HUGE_VAL and sets macro ERANGE to global variable errno if an overflow occurs.
sinhf	Returns the hyperbolic sine of x. sinhf returns HUGE_VAL and sets macro ERANGE to global variable errno if an overflow occurs.
tanhf	Returns the hyperbolic tangent of x.

The error processing of these functions can be changed by using the matherr function.

[Example]

```
#include <math.h>

float func(float x)
{
    float   ret;
    ret = acoshf(x);    /* Returns value of inverse hyperbolic cosine of x to ret. */
        :
    return(ret);
}
```

TRIG

[Overview]

Trigonometric functions

acosf, asinf, atanf, atan2f, cosf, sinf, tanf

[Syntax]

```
#include <math.h>

float  acosf(float x)
float  asinf(float x)
float  atanf(float x)
float  atan2f(float y, float x)
float  cosf(float x)
float  sinf(float x)
float  tanf(float x)
```

[Description]

acosf(float x)

This function calculates the inverse cosine (arcosine) of x. Specify x as, -1<= x <= 1.

asinf(float x)

This function calculates the inverse sine (arcsine) of x. Specify x as, -1<= x <= 1.

atanf(float x)

This function calculates the inverse tangent (arctangent) of x.

atan2f(float y, float x)

This function calculates the inverse tangent of y/x. atan2f calculates the correct result even if the angle is in the vicinity of $\pi/2$ or - $\pi/2$(if x is close to 0).

cosf(float x)

This function calculates the cosine of x. Specify the angle in radian.

sinf(float x)

This function calculates the sine of x. Specify the angle in radian.

tanf(float x)

This function calculates the cosine of x. Specify the angle in radian.

[Return value]

acosf	Returns the inverse cosine (arccosine) of x. The returned value is in radian and in a range of 0 to π. If x is not between -1 and 1, a non-numeric value is returned, and macro EDOM is set to global variable errno.
asinf	Returns the inverse sine (arcsine) of x. The returned value is in radian and in a range of -π/2 to π/2. If x is not between -1 and 1, a non-numeric value is returned, and macro EDOM is set to global variable errno.
atanf	Returns the inverse tangent (arctangent) of x. The returned value is in radian and in a range of -π/2 to π/2.
atan2f	Returns the inverse tangent (arctangent) of y/x. The returned value is in radian and in a range of -π to π. atan2f returns a non-numeric value and sets macro EDOM to global variable errno if both x and y are 0.0. If the solution vanished approaching zero, atan2f returns ±0 and sets macro ERANGE to global variable errno. If the solution is a non-normalized number, atan2f sets macro ERANGE to global variable errno.
cosf	Returns the cosine of x.
sinf	Returns the sine of x.
tanf	Returns the tangent of x.

The error processing of these functions can be changed by using the matherr function.

[Example]

```
#include <math.h>

float func(float x)
{
    float   ret;

    ret = atanf(x);     /* Returns value of arctangent of x to ret. */
        :
    return(ret);
}
```

6.12 Runtime Library

This section explains the runtime library.

The microcontroller architecture of the V850 microcontrollers does not have instructions for multiplying or dividing and performing floating-point operations on 32-bit data. Therefore, to satisfy the language specifications of the ANSI standards, the CA850 performs multiplication, division, residue calculations, and all floating-point operations on 32-bit data by calling the runtime library contained in the libc.a file. The runtime library can also be called when creating a new assembler for the V850 microcontrollers.

However, with the V850Ex, the compiler does not use the runtime library for multiplying, dividing, and residue calculating 32-bit data. It uses the runtime library for floating-point operations.

The runtime library is a routine automatically used when the compiler executes compiling. This library is included in the libc.a file along with the standard library. The header file does not need to be included.

When using the runtime library for an application program, libc.a must be referenced by ld850 when an executable object file is created.

Figure 6 - 1 Image of Using Runtime Library

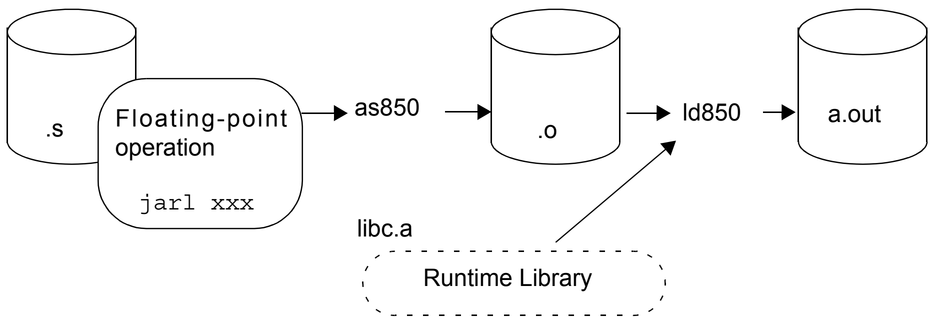


Table 6 - 25 Runtime Library

Classification	Function Name	Outline
ADDF.S	___addf.s	Addition of single-precision floating-point
CMPF.S	___cmpf.s	Comparison of single-precision floating-point and change of flag
CVT.WS	___cvt.ws	Conversion from integer to single-precision floating-point number
DIV	___div	Division of signed 32-bit integer
	___divu	Division of unsigned 32-bit integer
DIVF.S	___divf.s	Division of single-precision floating-point
MOD	___mod	Remainder of signed 32-bit integer
	___modu	Remainder of unsigned 32-bit integer
MUL	___mul	Multiplication of signed 32-bit integer
	___mulu	Multiplication of unsigned 32-bit integer
MULF.S	___mulf.s	Multiplication of single-precision floating-point
SUBF.S	___subf.s	Subtraction of single-precision floating-point
TRNC.SW	___trnc.sw	Conversion from single-precision floating-point number to integer

[Cautions]

(1) The runtime library is originally used by code generation part (cgen) and is not assumed to be used alone.Therefore, preprocessing to call the runtime library is necessary when it is used for an assembly-language source program.

(2) The runtime library cannot be used with a C language source program.

(3) The default processing of the compiler does not use the runtime library's ___mul/___mulu functions for multiplication and ___div/___divu functions for division to process integer data of 16 bits or shorter. Instead, the mulh and divh instructions are used. If the -Xe option is specified with the compiler, the runtime library is used to process integer data of 16 bits or shorter.
In this case, if the runtime library is used, multiplication/division processing strictly conforming to the ANSI standards is executed, but the execution speed is slower than when using the mulh and divh instructions.

ADDF.S

[Overview]

Addition of single-precision floating-point

___addf.s

[Syntax]

```
jarl   ___addf.s, lp
```

[Description]

___addf.s

This function adds single-precision floating-points.

This function is used by the ca850 as the entity of arithmetic operator "+" of the C language for a single-precision floating-point number. It is not used for addition of integers.

[Preprocessing]

When using this function for an assembler, general-purpose registers r6 and r7 must be saved, and values must be substituted into r6 and r7 as arguments.

[Argument setting register]

r6, r7

[Return value]

The result of the addition is set to r6.

[Example]

In the case of "value of reg2 + value of reg1" (reg2 is other than r6 and r7)

```
    add     -8, sp
    st.w    r6,[sp]         -- Saves r6 and r7.
    st.w    r7, 4[sp]
    mov     reg1, r6        -- Substitutes value as argument.
    mov     reg2, r7
    jarl    ___addf.s,lp    -- Calls function.
    mov     r6, reg2        -- Stores result of addition in reg2.
    ld.w    4[sp], r7       -- Restores r6 and r7.
    ld.w    [sp], r6
    add     8, sp
```

CMPF.S

[Overview]

Comparison of single-precision floating-point and change of flag

___cmpf.s

[Syntax]

```
jarl   ___cmpf.s, lp
```

[Description]

___cmpf.s

This function compares single-precision floating-point numbers, and changes flags S and Z according to the result of the comparison. Changes to the flags are then reflected in the passed PSW, and the PSW is changed.

[Preprocessing]

When using this function for an assembler, general-purpose registers r6 through r8 must be saved, and values to be compared as arguments must be substituted into r6 and r7. Moreover, the value of the PSW must be passed to r8.

This function changes the flags depending on the result of "r7 - r6".

[Argument setting register]

r6, r7, r8

[Return value]

The contents of the PSW are changed depending on the result of the comparison, and the value of the PSW is set to r6.

[Flag]

CY	1 if the result of comparison is negative; otherwise, 0
OV	0
S	1 if the result of comparison is negative; otherwise, 0
Z	1 if the result of comparison is zero; otherwise, 0

[Example]

In the case of "comparing value of reg2 and value of reg1" (reg2 is other than r6 through r8)

```
    add      -12, sp
    st.w     r6, [sp]         -- Saves r6 through r8.
    st.w     r7, 4[sp]
    st.w     r8, 8[sp]
    mov      reg1, r6         -- Substitutes value as argument.
    mov      reg2, r7
    stsr     5, r8            -- Passes value of PSW to r8.
    jarl     ___cmpf.s,lp     -- Calls function
    mov      r6, reg2         -- Stores changed PSW value in reg2.
    ld.w     8[sp], r8        -- Restores r6 through r8.
    ld.w     4[sp], r7
    ld.w     [sp], r6
    add      12, sp
```

CVT.WS

[Overview]

Conversion from integer to single-precision floating-point number

___cvt.ws

[Syntax]

```
jarl   ___cvt.ws, lp
```

[Description]

___cvt.ws

This function converts an integer to a single-precision floating-point number.

[Preprocessing]

When using this function for an assembler instruction, general-purpose register r6 must be saved and a value to be converted as an argument must be saved to r6.

[Argument setting register]

r6

[Return value]

The converted value is set to r6.

[Example]

In the case of "converting value of reg1 and storing the result of conversion in reg2" (reg2 is other than r6)

```
    add      -4, sp
    st.w     r6, [sp]         -- Saves r6.
    mov      reg1, r6         -- Substitutes function as argument.
    jarl     ___cvt.ws,lp     -- Calls function
    mov      r6, reg2         -- Stores value resulting from conversion in reg2.
    ld.w     [sp], r6         -- Restores r6.
    add      4, sp
```

DIV

[Overview]

Division of 32-bit integer

___div, ___divu

[Syntax]

```
jarl   ___div, lp
jarl   ___divu, lp
```

[Description]

___div

This function executes division of signed 32-bit integers.

___divu

This function executes division of unsigned 32-bit integers.

These functions are used by the CA850 as the entities of the arithmetic operator "/" of the C language.

[Preprocessing]

When these functions are used for an assembler instruction, general-purpose registers r6 and r7 must be saved, and values must be substituted into r6 and r7 as arguments. These functions execute division, assuming that "r7/r6".

[Argument setting register]

r6, r7

[Return value]

The lower 32 bits of the result of division are set to r6. The remainder is ignored.

[Example]

In the case of "value of reg2/value of reg1" (reg2 is other than r6 and r7

```
    add     -8, sp
    st.w    r6, [sp]         -- Saves r6 and r7.
    st.w    r7, 4[sp]
    mov     reg1, r6         -- Substitutes value as argument.
    mov     reg2, r7
    jarl    ___div, lp       -- Calls function.
    mov     r6, reg2         -- Stores result of division in reg2.
    ld.w    4[sp], r7        -- Restores r6 and r7.
    ld.w    [sp], r6
    add     8, sp
```

DIVF.S

[Overview]

Division of single-precision floating-point

___divf.s

[Syntax]

```
jarl   ___divf.s, lp
```

[Description]

___divf.s

This function executes division of single-precision floating-points.

This function is used by the CA850 as the entity of the arithmetic operator "/" of the C language in single-precision floating-point numbers. It is not used for division of integers.

[Preprocessing]

When this function is used for an assembler instruction, general-purpose registers r6 and r7 must be saved, and values must be substituted into r6 and r7 as arguments. This function executes division, assuming that "r7/r6".

[Argument setting register]

r6, r7

[Return value]

The result of division is set to r6.

[Example]

In the case of "value of reg2/value of reg1" (reg2 is other than r6 and r7

```
    add     -8, sp
    st.w    r6, [sp]         --Saves r6 and r7.
    st.w    r7, 4[sp]
    mov     reg1, r6         -- Substitutes value as argument.
    mov     reg2, r7
    jarl    ___divf.s,lp     -- Calls function.
    mov     r6, reg2         --Stores result of division in reg2.
    ld.w    4[sp], r7        -- Restores r6 and r7.
    ld.w    [sp], r6
    add     8, sp
```

MOD

[Overview]

Remainder of 32-bit integer

___mod, ___modu

[Syntax]

```
jarl   ___mod, lp
jarl   ___modu, lp
```

[Description]

___mod

This function calculates the remainder resulting from division of signed 32-bit integers.

___modu

This function calculated the remainder resulting from division of unsigned 32-bit integers.

These functions are used by the CA850 as the entities of the arithmetic operator "%" of the C language.

[Preprocessing]

When these functions are used for an assembler instruction, general-purpose registers r6 and r7 must be saved, and values must be substituted into r6 and r7 as arguments. These functions execute division, assuming that "r7%r6".

[Argument setting register]

r6, r7

[Return value]

The remainder resulting from division is set to r6.

[Example]

In the case of "value of reg2%value of reg1" (reg2 is other than r6 and r7)

```
    add     -8, sp
    st.w    r6, [sp]        -- Saves r6 and r7.
    st.w    r7, 4[sp]
    mov     reg1, r6        -- Substitutes value as argument.
    mov     reg2, r7
    jarl    ___mod, lp      -- Calls function.
    mov     r6, reg2        -- Stores remainder resulting from division in reg2.
    ld.w    4[sp], r7       -- Restores r6 and r7.
    ld.w    [sp], r6
    add     8, sp
```

MUL

[Overview]

Multiplication of 32-bit integer

___mul, ___mulu

[Syntax]

```
jarl   ___mul, lp
jarl   ___mulu, lp
```

[Description]

___mul

This function executes multiplication of signed 32-bit integers.

___mulu

This function executes multiplication of unsigned 32-bit integers.

These functions are used by the CA850 as the entities of the arithmetic operator "*" of the C language.

[Preprocessing]

When these functions are used for an assembler struction, general-purpose registers r6 and r7 must be saved, and values must be substituted into r6 and r7 as arguments.

[Argument setting register]

r6, r7

[Return value]

The lower 32 bits of the result of multiplication are set to r6. The higher 32 bits are invalid.

[Example]

In the case of "value of reg1 * value of reg2" (reg2 is other than r6 and r7)

```
    add     -8, sp
    st.w    r6, [sp]        -- Saves r6 and r7.
    st.w    r7, 4[sp]
    mov     reg1, r6        -- Substitutes value as argument.
    mov     reg2, r7
    jarl    ___mul, lp      -- Calls function.
    mov     r6, reg2        -- Stores result of multiplication in reg2.
    ld.w    4[sp], r7       -- Restores r6 and r7.
    ld.w    [sp], r6
    add     8, sp
```

MULF.S

[Overview]

Multiplication of single-precision floating-point

___mulf.s

[Syntax]

```
jarl   ___mulf.s, lp
```

[Description]

___mulf.s

This function executes multiplication of single-precision floating-points.

This function is used by the CA850 as the entity of the arithmetic operator "*" of the C language in single-precision floating-point numbers. It is not used for multiplication of integers.

[Preprocessing]

When this function is used for an assembler instruction, general-purpose registers r6 and r7 must be saved, and values must be substituted into r6 and r7 as arguments.

[Argument setting register]

r6, r7

[Return value]

The result of multiplication is set to r6.

[Example]

In the case of "value of reg2 * value of reg1" (reg2 is other than r6 and r7)

```
    add     -8, sp
    st.w    r6, [sp]            -- Saves r6 and r7.
    st.w    r7, 4[sp]
    mov     reg1, r6            -- Substitutes value as argument.
    mov     reg2, r7
    jarl    ___mulf.s   ,lp     -- Calls function.
    mov     r6, reg2            -- Stores result of multiplication in reg2.
    ld.w    4[sp], r7           -- Restores r6 and r7.
    ld.w    [sp], r6
    add     8, sp
```

SUBF.S

[Overview]

Subtraction of single-precision floating-point

___subf.s

[Syntax]

```
jarl   ___subf.s, lp
```

[Description]

___subf.s

This function executes subtraction of single-precision floating-points.

This function is used by the CA850 as the entity of the arithmetic operator "-" of the C language is single-precision floating-point numbers. It is not used for subtraction of integers.

[Preprocessing]

When this function is used for an assembler instruction, general-purpose registers r6 and r7 must be saved, and values must be substituted into r6 and r7 as arguments. This function executes subtraction, assuming that "r7 - r6".

[Argument setting register]

r6, r7

[Return value]

The result of subtraction is set to r6.

[Example]

In the case of "value of reg2 - value of reg1" (reg2 is other than r6 and r7)

```
    add      -8, sp
    st.w     r6, [sp]            -- Saves r6 and r7.
    st.w     r7, 4[sp]
    mov      reg1, r6            -- Substitutes value as argument.
    mov      reg2, r7
    jarl     ___subf.s   ,lp     -- Calls function.
    mov      r6, reg2            -- Stores result of subtraction in reg2.
    ld.w     4[sp], r7           -- Restores r6 and r7.
    ld.w     [sp], r6
    add      8, sp
```

TRNC.SW

[Overview]

Conversion from single-precision floating-point number to integer

___trnc.sw

[Syntax]

```
jarl   ___trnc.sw, lp
```

[Description]

___trnc.sw

This function executes conversion from a single-precision floating-point number to an integer.

The result of conversion is rounded toward 0.

[Preprocessing]

When this function is used for an assembler instruction, general-purpose register r6 must be saved, and a value must be substituted into r6 as an argument.

[Argument setting register]

r6

[Return value]

The value resulting from conversion is set to r6.

[Example]

To "convert value of reg1 and store result in reg2" (reg2 is other than r6)

```
    add     -4, sp
    st.w    r6, [sp]            -- Saves r6.
    mov     reg1, r6            -- Substitutes value as argument.
    jarl    ___trnc.sw  ,lp     -- Calls function.
    mov     r6, reg2            -- Stores result of conversion in reg2.
    ld.w    [sp], r6            -- Restores r6.
    add     4, sp
```

CHAPTER 7 FOR EFFICIENT USE

This chapter explains the programming method and how to use the expansion functions for more efficient use of the CA850.

7.1 volatile Qualifier

When a variable is declared with the volatile qualifier, the variable is not optimized and is not assigned to registers. Therefore, keep volatile declaration at the minimum required level.

If it is clear that the value of a variable with volatile declared is not changed externally in a specific section, the variable can be optimized by assigning the unchanged value to a variable for which volatile not declared and referencing it, which may increase the execution speed.

7.2 Declaration of Function Without Return Value

If a function without a return value is not declared as void type, an unwanted return processing code is generated.

Be sure to declare functions without return values as void type

7.3 Pointers and Optimization

The CA850 executes analysis of pointers when the -Og/-O/-Os/-Ot options are specified and optimization is executed. If the optimization level is lower than that, pointer analysis is not executed. Consequently, if indirect memory access using a pointer exists, processing is performed on the assumption that all the variables are accessed by this indirect memory access, and optimization and register allocation cannot be executed efficiently. Even for size priority optimization or execution speed priority optimization, the same phenomenon may occur when this indirect memory access using a global pointer or a pointer argument exists. Use the global pointer as locally as possible.

For example, optimization is not executed in the case of Example 1 below, but optimization is executed efficiently when a local variable is used as shown in Example 2.

Example 1

```
int*    sp;
int     s1, s2, s3;

void func(void)
{
    int a = 0;
    int b = 1;

    *sp = s1 / s2 * 100;

    if(s1 == 0){
        s3 = *sp + a ;
    }
    else {
        s3 = *sp - b;
    }
}
```

Example 2

```
int*    sp;
int     s1, s2, s3;

void func(void)
{
    int a = 0;
    int b = 1;
    register int tmp = s1 / s2 * 100;

    if(s1 == 0) {
        s3 = tmp + a; /* a is replaced by 0 */
    }
    else {
        s3 = tmp - b; /* b is replaced by 1 */
    }
    *sp = tmp;
}
```

7.4 Assembler Code and Optimization

If descriptions of assembler directives (refer to "3.4 Describing Assembler Instruction") are included, processing is performed on the assumption that the values of all the variables are used and changed in that code. Therefore, optimization is not performed beyond the assembler code.

To avoid a drop in the processing efficiency, use functions including assembler code as little as possible. If the execution speed priority optimization is specified, and if a function including assembler code that defines a label is used, the same label will be defined at the parts of function definition and inline expansion. In this case, a label multiple definition error will occur

7.5 Registers

7.5.1 Register specifier

When the debug priority optimization (-Od) option is specified, a variable declared with the register specifier is assigned to a register variable register, taking precedence over a variable without the register specifier. Even a variable declared with the register specifier, however, is not assigned to a register if the variable is not referenced many times. Therefore, the code efficiency will not deteriorate.

Even if a variable is defined with the register specifier and referenced by the debugger, the expected value may not be obtained. This is due to the number of variables declared with the register specifier or optimization by the CA850. If the number of register variables is less than the number of variables declared with the register specifier, the variables are not assigned to registers.

The number of register variables registers in each mode is as follows.

- 22-register mode: 5 (r25 - r29)
- 26-register mode: 7 (r23 - r29)
- 32-register mode: 10 (r20 - r29)

For example, even if six or more variables are declared with the register specifier in the 22-register mode, all the variables are not stored in registers.

The debug priority optimization (-Od) option gives priority to the variables declared with the register specifier and assigns these variables to register variable registers. However, a variable for which the register specifier is specified is not assigned to register if it is referenced only a few times. If an option other than the -Od option is specified, variables that are relatively frequently referenced are assigned to registers.

Because the variables are assigned to different places in this way, the symbol information may be affected and therefore the variables cannot be referenced from the debugger. If there is no problem in the result of an operation that uses variables that cannot be referenced, it is considered that the operation is performed correctly.

7.5.2 Static variables and external variables

When the debug priority optimization (-Od) option or default optimization (-Ob) option is specified, a static variable or external variable is not assigned to a register when registers are allocated. When these variables are frequently used in a function and when the values of these variables are not changed by a function call or asm declaration in that function, the registers are used more frequently and the speed is expected to increase if the value of that static variable or external variable is substituted at the beginning of the function to an automatic variable declared with the register specifier and if the value is returned to the variable at the end of the function. By specifying the following options, even static variables and external variables can be assigned to registers if they are relatively frequently referenced.

- Default optimization (-Ob) option
- Standard optimization (-Og) option
- Level 1 advanced optimization (-O) option
- Level 2 advanced optimization (object size) [-Os] option
- Level 2 advanced optimization (execution speed) [-Ot] option

7.5.3 Argument of function in K&R format

If an argument with a type smaller in size that int type exists in a function definition in the K&R format, the argument is not allocated to a register, even if object size priority optimization or execution speed priority optimization is executed, unless a dummy argument is declared with the register specifier. To allocate this argument to a register, make a declaration with the register specifier. To describe a function definition in the K&R format, avoid using char, signed char, unsigned char, short, signed short, and unsigned short as the type of the argument.

7.5.4 Optimum number of local variables to be assigned

Keep the number of local variables (auto variables) to within 10; or preferably to six or seven. Local variables are assigned to registers[Note]. The CA850 allows a total of 20 registers, 10 work registers and 10 register variable registers, to be used for variables (in the 32-bit register mode). It is recommended to use many local variables if processing in one function takes time. If processing does not take much time, use only the 10 work registers whenever possible.

The register variable registers require overhead when they are saved or restored. The CA850 automatically judges whether register variables are to be used or not. Therefore, the efficiency can be enhanced if six to seven registers are used for local variables and the other three to four registers are used for work by the CA850.

Note Non-volatile variables that do not use addresses are subject to assignment. Therefore, the local variables that use addresses are secured in the stack area.

7.5.5 Optimum number of arguments to be used for function

Four argument registers, r6 to r9, are available. If the number of arguments is five or more, the stack is used for the fifth and subsequent arguments. Therefore, keep the number of arguments to within four whenever possible. If five or more arguments must be used, pass the arguments using the pointer of a structure, in order to enhance the efficiency.

7.5.6 Other

If a path exists in which a variable may be referenced before a value is set (Example 3), an unwanted transfer code from memory to register may be generated. Use a variable after setting a value (Example 4).

Example 3

```
int     s;

void func(int x)
{
    int y;
    int i;

    for(i = x; i < 10; i++) {
        if(i == 3) {
            y = 10;
        }
    }
    s = y * y * x;
}
```

Example 4

```
int     s;

void func(int x)
{
    int y = 0;
    int i;

    for(i = x; i < 10; i++) {
        if(i == 3) {
            y = 10;
        }
    }
    s = y * y * x;
}
```

7.6 Stack Size

The compiler allocates one variable to one stack area. Two or more variables cannot be allocated to the same area. By selecting and using variables for specific purposes[Note], the stack size can be reduced.

Note In this case, however, the program may become difficult to read.

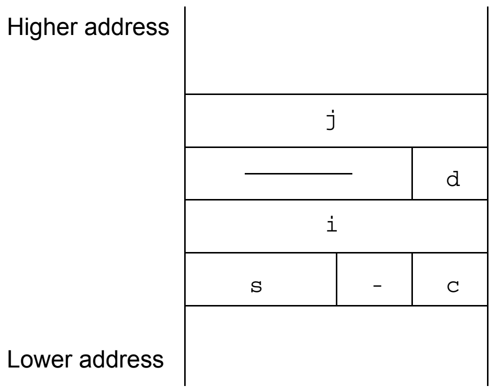
7.7 Aligning Data

Declare data definitions collectively starting from the longest data.

With the V850 microcontrollers, word data such as int type must be aligned at a word boundary, and halfword data such as short type must be aligned at a halfword boundary.

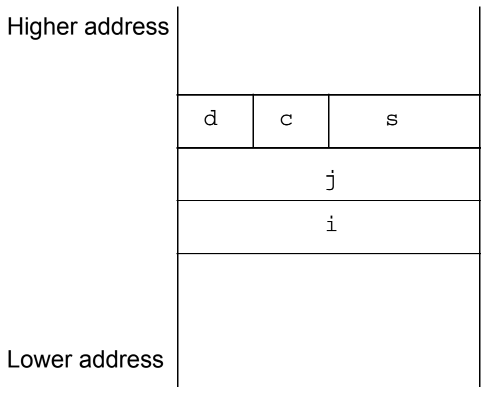
Consequently, a padding area is generated to enable alignment for the following source.

```
char     c = 'a';
short    s = 0;
int      i = 1;
char     d = 'b';
int      j = 2;
```



To prevent the generation of a padding area like this, declare data starting from the longest data.

```
int      i = 1;
int      j = 2;
short    s = 0;
char     c = 'a';
char     d = 'b';
```



7.8 Data Type

The V850 microcontrollers sign-extends byte data or halfword data to word length depending on the value of the most significant bit, when the byte data or halfword data is loaded from memory to a register. Consequently, a code that masks the higher bits may be generated[Note] as a result of operating data of unsigned char or unsigned short type. Use word data as much as possible. When using byte data or halfword data, use a signed type.

Note This mask code is not generated by an operation in which data is already stored in registers.

Caution When the V850Ex is used as the target device with the CA850, mask codes are not created because the architecture of the V850Ex has unsigned load instructions and type conversion instructions.

In the case of a register variable, a shift instruction is generated to extend the sign because an operation of signed byte data or signed halfword data integer-expands the operand[Note].

When storing the result of the operation in a register variable, a shift instruction is generated in the case of signed byte data or signed halfword data, or a code that masks the higher bits is generated in the case of unsigned byte data or unsigned halfword data. To prevent generation of this code, use word data (int, long, unsigned int, or unsigned long type data) as much as possible when using a register variable.

Note "Integer-expansion" converts values into int type if all the values of the original type can be expressed by int type; otherwise, the values are converted into unsigned int type.

Using mask register function

With a program in which word data cannot be used and therefore mask codes are generated, the code size can be reduced by using the mask register function (refer to "2.5 Mask Register").

Examples of instruction generation in the case of byte data, halfword data, and word data are shown below.

Example (Written in C)

```
int     i, j, k;
unsigned short s, t, u;
unsigned char c, d, e;

void f(void)
{
    register int    ri, rj, rk;
    register short rs, rt, ru;
    register unsigned char ruc, rud, rue;

    c = d + e;
    s = t + u;
    i = j + k;

    rs = rt + ru;
    ruc = rud + rue;
    ri = rj + rk;
}
```

(Output instructions)

```
# Byte data:
    ld.b    $_d, r10
    andi    0xff, r10, r10 -- Mask code
    ld.b    $_e, r11
    andi    0xff, r11, r11 -- Mask code
    add     r11, r10
    st.b    r10, $_c

# Halfword data:
    ld.h    $_t, r12
    andi    0xffff, r12, r12 -- Mask code
    ld.h    $_u, r13
    andi    0xffff, r13, r13 -- Mask code
    add     r13, r12
    st.h    r12, $_s

# Word data:
    ld.w    $_j, r14
    ld.w    $_k, r15
    add     r15, r14
    st.w    r14, $_i

# Signed halfword data (register variable):
    mov     r25,r16
    shl     16, r16 -- Shift instruction(integer-expansion)
    sar     16, r16 -- Shift instruction(integer-expansion)
    mov     r24, r17
    shl     16, r17 -- Shift instruction(integer-expansion)
    sar     16, r17 -- Shift instruction(integer-expansion)
    add     r17, r16
    shl     16, r16 -- Shift instruction(sign-expansion of operation result)
    sar     16, r16 -- Shift instruction(sign-expansion of operation result)

# Unsigned byte data (register variable):
    mov     r22, r18
    add     r21, r18
    addi    0xff, r18, r18 -- Mask code
    mov     r18, r23

# Word data (register variable):
    mov     r28, r19
    add     r27, r19
    mov     r19, r29
    st.w    r14, $_i
```

APPENDIX A EXPANDED FUNCTIONS OF CC78K*x*

This appendix explains the expanded functions of the CC78K*x*.

A.1 #pragma Directive

The following #pragma directive compatible with the CC78Kx can be specified in the CA850.

The **[78K-compatible]** mark indicates as follows:

[78K-compatible]	Invalid unless -cc78K option is specified
	Uppercase and lowercase characters of keywords following #pragma are not distinguished.

(1) Specifying device type

[78K-compatible]

```
#pragma pc(device-name)
```

Specify to reference a device file that defines information dependent on the device to be used.

This directive functions in the same manner as the "#pragma cpu device-name" specification and the device specification option (-cpu) of the CA850.

(2) Validating peripheral I/O register name

[78K-compatible]

```
#pragma sfr
```

A peripheral I/O register of the device is accessed using the specified peripheral I/O register name.

This directive functions in the same manner as the #pragma ioreg directive of the CA850.

(3) Disabling interrupts

[78K-compatible]

```
#pragma di
```

The function DI() is treated as the embedded function __DI().

[78K-compatible]

```
#pragma ei
```

The function EI() is treated as the embedded function __EI().

(4) Specifying CPU stop function

[78K-compatible]

```
#pragma halt
```

The function HALT() is treated as the embedded function __halt().

(5) Specifying no-operation function

[78K-compatible]

```
#pragma nop
```

The function NOP() is treated as the function __nop().

(6) #pragma directives of CC78K*x*

The following directives are not compatible with the 78K.

These directives are treated as the #pragma directive in the CA850.

(a) Specifying interrupt/exception handler

[78K-compatible]

```
#pragma interrupt interrupt-request-name function-name [stack selection] ...
#pragma vect interrupt-request-name function-name [stack selection] ...
```

"#pragma interrupt" and "#pragma vect" of the CC78Kx are treated as "#pragma interrupt interrupt-request-name function-name [allocation-method]" in the CA850.

The following message is output if description is made after "[stack selection]" and if that description cannot be.

```
W2150: unexected character(s) following directive 'directive'
```

(b) Specifying section

[78K-compatible]

```
#pragma section ...
```

This directive is treated as "#pragma section section-type ["section-name"] [begin | end]" in the CA850.

The following message is output if it is not recognized by the CA850.

```
W2162: unrecognized pragma directive '#pragma directive', ignored
```

(c) Specification related to memory manipulation

[78K-compatible]

```
#pragma inline
```

The CC78Kx expands memcpy, memset, memchr, and memcmp inline, but the CA850 attempts to expand the specified function inline, so the following message is output.

```
W2162: unrecognized pragma directive '#pragma inline', ignored
```

(d) Specifying module name

[78K-compatible]

```
#pragma name module-name
```

The CA850 outputs the following message.

```
W2162: unrecognized pragma directive '#pragma name', ignored
```

(e) Specifying data insertion function

[78K-compatible]

```
#pragma opc
```

Corresponding embedded function

```
__OPC();
```

The CA850 outputs the following message and stops compiling.

```
W2162: unrecognized pragma directive '#pragma opc', ignored
E2752: cannot call opc function
```

(f) Specifying byte address insertion/generation function

[78K-compatible]

```
#pragma addraccess
```

Corresponding embedded function

```
FP_SEG();   FP_OFF();   MK_FP();
```

The CA850 outputs the following message and stops compiling.

```
W2162: unrecognized pragma directive '#pragma addraccess', ignored
E2752: cannot call addraccess function
```

(g) Specifying function directly referencing register

[78K-compatible]

```
#pragma realregister
```

Corresponding embedded function

```
__absa();    __ashra();   __clr1cy(); __coma();   __deca();   __geta();
__getax();   __getcy();   __inca();   __nega();   __not1cy(); __rola();
__rolca();   __rora();    __rorca();  __set1cy(); __seta();   __setax();
__setcy();   __shla();    __shra();
```

The CA850 outputs the following message and stops compiling.

```
W2162: unrecognized pragma directive '#pragma realregister', ignored
E2752: cannot call realregister function
```

(h) Specifying function directly calling self-writing subroutine of firmware

[78K-compatible]

```
#pragma hromcall
```

Corresponding embedded function

```
__FlashAreaBlankCheck();__FlashAreaErase(); __FlashAreaIVerify();
__FlashAreaPreWrite();   __FlashAreaWriteBack(); __FlashBlockBlankCheck();
__FlashBlockErase();     __FlashBlockIVerify(); __FlashBlockPreWrite();
__FlashBlockWriteBack();__FlashByteRead();      __FlashByteWrite();
__FlashEnv();            __FlashGetInfo();      __FlashSetEnv();
__FlashWordWrite();      __hromcall();          __hromcalla();
__setsp();
```

The CA850 outputs the following message and stops compiling.

```
W2162: unrecognized pragma directive '#pragma hromcall', ignored
E2752: cannot call hromcall function
```

A.2 Assembler Control Instructions

[78K-compatible]

```
#asm
    assembler instructions
#endasm
```

This instruction is treated as "#pragma asm" - "#pragma endasm" in the CA850.

The following message is output for each instruction.

```
W2166: recognized pragma directive '#pragma asm'
W2166: recognized pragma directive '#pragma endasm'
```

A.3 Specifying Interrupt/Exception Handler

An interrupt/exception handler is specified in a C-source program by the following #pragma directive and qualifier (refer to "3.7 Interrupt/Exception Processing Handler").

[78K-compatible]

```
#pragma interrupt interrupt-request-name function-nameNote [allocation method]

__interrupt_brk function-definition, or function-declaration
```

The function qualifier __interrupt_brk is treated as specification of the __interrupt function in the CA850.

Note C description

A.4 Expanded Functions Not Supported

The CA850 outputs a message if an expanded specification of the CC78Kx that is not supported is specified.

[78K-compatible]

```
__banked1    __banked2    __banked3    __banked4    __banked5
__banked6    __banked7    __banked8    __banked9    __banked10
__banked11   __banked12   __banked13   __banked14   __banked15
callf        __callf      callt        __callt      noauto
norec        __pascal     sreg         __sreg       __sreg1
__temp
```

The CA850 outputs the following message.

```
W2761: unrecognized specifier 'specifier', ignored
```

APPENDIX B CAUTIONS

This chapter explains the points to be noted when using the CA850.

(1) Delimiting Folder Path

Both "\" and "/" are regarded as the delimiters of a folder.

(2) Option Specification Sequence

The CA850 has the following restriction concerning the sequence of an option specified when the driver is started on the command line:

The actual sequence in which an argument passed to a specific module using the -W option and an argument of an option recognized by the driver are passed during the module startup is not guaranteed$^{\text{Note}}$.

Note When ld850 is started from the CA850, -lm -lc is passed to ld850 as the default assumption even if the -W option is not specified. If ld850 is started from the CA850, startup module crtN.o/crtE.o is passed to ld850 as the default assumption.

Example

```
> ca850 -cpu 3201 file.o -Wl,-D,dfile.dir -m
```

The ld850 passed as follows on starting.

```
ld850 \Install Folder\lib850\r32\crtN.o -o a.out file.o -lm -lc -D dfile.dir -
m
```

However, it is assumed that ld850 has already been placed in *Install Folder*\bin.

Caution When starting the ld850 directly, allocate "-lc" after "-lm" because the mathematical library references the standard library (refer to "6.11 Mathematical Functions").

(3) Mixing with K&R Format in Function Declaration/Definition

If the K&R format and ANSI standard format exist together in the declaration and definition of a function, an error may occur on compilation by the CA850 as a result of argument expansion processing in the K&R format.

For example, a function is declared according to the ANSI standard in the example below, but the function is defined in the K&R format. Consequently, the types of the arguments do not match, and the CA850 outputs a "function redeclaration" error.

Example of error

```
void func(int a, int b, float c);
    /* Declared in ANSI standard format. */
    /* Third argument is declared to be of float type. */
    :

void func(a, b, c)
int     a, b;
float   c;
{
    /* Defined in K&R format. */
    /* Third argument is the expanded default of K&R
       and so becomes double type.*/
    :
}
```

In the above example, compilation is performed normally if the K&R format is uniformly used by specifying "void func();" for the function declaration, or if the ANSI standard format is used by specifying "void func(int a, int b, float c)" for the function definition.

Note, however, that use of the ANSI standard format is recommended in the CA850.

(4) Output of Other Than Position-Independent Codes

Basically, the CA850 outputs codes not dependent on positions (position-independent codes). However, it outputs the following codes in response to the "initialization statement with an initial value other than a numeric value for a pointer type variable other than an automatic variable".

Example

```
/* Described in C */
char    *ptr = "test\n";
```

```
# Output codes
    .size   LL20, 6
LL20 :
    .str    "test\n\0"
    .align  4
    .globl  _ptr, 4
_ptr :
    .word   #LL20
    -- Absolute address reference of label
```

When the -Xd option is specified, the CA850 outputs the following warning message and continues compiling if an initialization statement with an initial value other than a numeric value for a pointer type variable other than an automatic variable appears.

```
W2231:Initialization of non-auto pointer using non-number initializer is not
position independent.
```

(5) Count of Derivative Type Qualification for Type Configuration

The CA850 outputs the following error message and continues compiling if derivative type qualificationNote is performed 17 times or more for the type configuration

```
E2260: compiler limit : complicated type modifiers [16]
```

However, compiling may be stopped depending on the number of times the error has occurred.

Note *(pointer), [ ] (array), and function declarator included in a declarator.

(6) Length of Identifier and Valid Number of Characters

The CA850 outputs the following error message and continues compiling if an external identifier of 1023 characters or more, or an internal identifier of 1024 characters or more is described.

```
E2117: compiler limit:too long identifier 'symbol' [1022 / 1023]
```

However, compiling may be stopped depending on the number of times the error has occurred.

The valid number of characters for an identifier name is 1022 from the beginning of the identifier in the case of an external identifier and 1023 from the beginning in the case of an internal identifier.

(7) Number of Times of Block Nesting

The CA850 outputs the following message if a pair of "{" and "}" are nested 128 times or more.

```
F2020: compiler limit : scope level too deep [127]
```

(8) Number of case Labels in switch Statement

The CA850 outputs the following error message and stops compiling if 1026 or more case labels are described in one switch statement

```
F2410: compiler limit : too many case labels [1025]
```

Depending on the number of nesting switch statements, however, the above message is output and compiling is stopped even if the number of case labels is less than 1025.

(9) Floating-Point Operation Exception in Operation of Constant Expression

The CA850 outputs the following error message and continues compiling if a floating-point operation exception occurs during the operation of a constant expression.

```
E2519: exception has occurred at compile time.
```

However, compiling may be stopped depending on the number of errors that have occurred.

Moreover, depending on the type of exception, inexact, underflow, overflow, division-by-0, or others is output for exception.

(10) Merging Vast/Large-Quantity File

The CA850 merges intermediate language files according to the optimization level.

At this time, the pre-optimizer (popt850) performs processing on memory to speed up the compiling processing. To merge a vast or large-quantity intermediate language file, therefore, the following error message may be output because the memory runs short, and the compiler may be abnormally terminated.

```
F7009: out of memory
```

In this case, re-compile on the command line by specifying an option that allows the pre-optimizer to perform processing to reduce the memory consumption (-Wp, -D).

(11) Optimization of Vast File

If object size priority optimization or execution speed priority optimization is executed, the CA850 analyzes the data flow in function units inside the global optimization module (opt850) for global optimization.

Because this optimization requires a large amount of the memory, if a source file including a vast function is to be optimized, the CA850 may output the following error message and be abnormally terminated.

```
F5104: out of memory
```

If execution speed priority optimization is performed, inline expansion of a function may result in a function with a vast size. In such a case, lower the optimization level and execute compilation again.

(12) Library File Search by Specifying Option

The CA850 does not display a message even if a specified library file has not been found as a result of a library file search[Note] initiated by an option (-L or -l). However, if the library file name has been directly specified on the command line or in the command file, a message is displayed.

Note If the -L option is not specified, the standard folder (folder \lib850 to which CA850 has been installed, and each register mode folder below that folder) is searched.

Example

```
> ca850 -cpu 3201 a.c usr.a
```

```
F4002: can not open input file "usr.a".
```

(13) volatile qualifier

When a variable is declared with the volatile qualifier, the variable is not optimized and optimization for assigning the variable to a register is no longer performed. When a variable with volatile specified is manipulated, a code that always reads the value of the variable from memory and writes the value to memory after the variable is manipulated is output. The access width of the variable with volatile specified is not changed.

A variable for which volatile is not specified is assigned to a register as a result of optimization and the code that loads the variable from the memory may be deleted. When the same value is assigned to variables for which volatile is not specified, the instruction may be deleted as a result of optimization because it is interpreted as a redundant instruction. The volatile qualifier must be specified especially for variables that access a peripheral I/O register, variables whose value is changed by interrupt servicing, or variables whose value is changed by an external source. When a peripheral I/O register is accessed using the #pragma ioreg directive, however, the CA850 internally outputs a code for which volatile is specified. Therefore, volatile declaration is not necessary.

The following problem may occur if volatile is not specified where it should.

- The correct calculation result cannot be obtained.
- Execution cannot exit from a loop if the variable is used in a for loop.

If it is clear that the value of a variable with volatile specified is changed in a specific section, the variable can be optimized by assigning the unchanged value to a variable for which volatile not specified and referencing it, which may increase the execution speed.

[Example of source and output code if volatile is not specified]

If volatile is not specified for "variable a", "variable b", and "variable c", these variables are assigned to registers and optimized. Even if an interrupt occurs in the meantime and the variable value is changed by the interrupt, for example, the changed value is not reflected.

```
int a;
int b;
int c;

void func(void)
{
    if (a <= 0) {
        b++;
    } else {
        c++;
    }
    b++;
    c++;

}
```

```
_func:
    #@B_PROLOGUE
    #@E_PROLOGUE
    ld.w $_a, r12
    cmp r0, r12
    jgt .L2
    ld.w $_b, r11
    ld.w $_c, r10
    add 1, r11
    jbr .L3
.L2:
    ld.w $_c, r10
    ld.w $_b, r11
    add 1, r10
.L3:
    addi 1, r11, r13
    st.w r13, $_b
    addi 1, r10, r14
    st.w r14, $_c
    #@B_EPILOGUE
    jmp [lp]
    #@E_EPILOGUE
```

[Example of source and output code if volatile is specified]

If volatile is specified for "variable a", "variable b", and "variable c", a code that always reads the values of these variables from memory and writes them to memory after the variables are manipulated is output. Even if an interrupt occurs in the meantime and the values of the variables are changed by the interrupt, for example, the result in which the change is reflected can be obtained. (In this case, interrupts may have to be disabled while the variables are manipulated, depending on the timing of the interrupt.)

When volatile is specified, the code size increases compared with when volatile is not specified because the memory has to be read and written.

```
volatile int a;
volatile int b;
volatile int c;
void func(void)

{
    if (a <= 0) {
        b++;
    } else {
        c++;
    }
    b++;
    c++;

}
```

```
_func:
    #@B_PROLOGUE
    #@E_PROLOGUE
    .option volatile
    ld.w $_a, r10
    .option novolatile
    cmp r0, r10
    jgt .L2
    .option volatile
    ld.w $_b, r11
    .option novolatile
    add 1, r11
    .option volatile
    st.w r11, $_b
    .option novolatile
    jbr .L3
.L2:
    .option volatile
    ld.w $_c, r12
    .option novolatile
    add 1, r12
    .option volatile
    st.w r12, $_c
    .option novolatile
.L3:
    .option volatile
    ld.w $_b, r13
    .option novolatile
    add 1, r13
    .option volatile
    st.w r13, $_b
    .option novolatile
    .option volatile
    ld.w $_c, r14
    .option novolatile
    add 1, r14
    .option volatile
    st.w r14, $_c
    .option novolatile
    #@B_EPILOGUE
    jmp [lp]
    #@E_EPILOGUE
```

(14) Extra Brackets in Function Declaration

If extra brackets "( )" are described in the function declaration, ANSI-C prescribes their handling as shown below, but the CA850 outputs an error.

Example

```
typedef int Int;
void f1((Int));
```

[Prescription in ANSI-C]

In a parameter declaration, a single typedef name in parentheses is taken to be an abstract declarator that specifies a function with a single parameter, not as redundant parentheses around the identifier for a declarator.

The above example is therefore interpreted according to ANSI-C.

```
void f(int (*)(int));
```

If the code includes extra brackets, delete the unnecessary brackets as shown below.

Example

```
typedef int Int;
void f1(Int);
```

APPENDIX C INDEX

For further information, please contact:

NEC Electronics Corporation
1753, Shimonumabe, Nakahara-ku,
Kawasaki, Kanagawa 211-8668,
Japan
Tel: 044-435-5111
http://www.necel.com/

[America]

NEC Electronics America, Inc.
2880 Scott Blvd.
Santa Clara, CA 95050-2554, U.S.A.
Tel: 408-588-6000
800-366-9782
http://www.am.necel.com/

[Europe]

NEC Electronics (Europe) GmbH
Arcadiastrasse 10
40472 Düsseldorf, Germany
Tel: 0211-65030
http://www.eu.necel.com/

Hanover Office
Podbielskistrasse 166 B
30177 Hannover
Tel: 0 511 33 40 2-0

Munich Office
Werner-Eckert-Strasse 9
81829 München
Tel: 0 89 92 10 03-0

Stuttgart Office
Industriestrasse 3
70565 Stuttgart
Tel: 0 711 99 01 0-0

United Kingdom Branch
Cygnus House, Sunrise Parkway
Linford Wood, Milton Keynes
MK14 6NP, U.K.
Tel: 01908-691-133

Succursale Française
9, rue Paul Dautier, B.P. 52
78142 Velizy-Villacoublay Cédex
France
Tel: 01-3067-5800

Sucursal en España
Juan Esplandiu, 15
28007 Madrid, Spain
Tel: 091-504-2787

Tyskland Filial
Täby Centrum
Entrance S (7th floor)
18322 Täby, Sweden
Tel: 08 638 72 00

Filiale Italiana
Via Fabio Filzi, 25/A
20124 Milano, Italy
Tel: 02-667541

Branch The Netherlands
Steijgerweg 6
5616 HS Eindhoven
The Netherlands
Tel: 040 265 40 10

[Asia & Oceania]

NEC Electronics (China) Co., Ltd
7th Floor, Quantum Plaza, No. 27 ZhiChunLu Haidian
District, Beijing 100083, P.R.China
Tel: 010-8235-1155
http://www.cn.necel.com/

NEC Electronics Shanghai Ltd.
Room 2511-2512, Bank of China Tower,
200 Yincheng Road Central,
Pudong New Area, Shanghai P.R. China P.C:200120
Tel: 021-5888-5400
http://www.cn.necel.com/

NEC Electronics Hong Kong Ltd.
Unit 1601-1613, 16/F., Tower 2, Grand Century Place,
193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: 2886-9318
http://www.hk.necel.com/

NEC Electronics Taiwan Ltd.
7F, No. 363 Fu Shing North Road
Taipei, Taiwan, R. O. C.
Tel: 02-8175-9600
http://www.tw.necel.com/

NEC Electronics Singapore Pte. Ltd.
238A Thomson Road,
#12-08 Novena Square,
Singapore 307684
Tel: 6253-8311
http://www.sg.necel.com/

NEC Electronics Korea Ltd.
11F., Samik Lavied'or Bldg., 720-2,
Yeoksam-Dong, Kangnam-Ku,
Seoul, 135-080, Korea
Tel: 02-558-3737
http://www.kr.necel.com/

G07.1A