

CMOS 32-BIT SINGLE CHIP MICROCOMPUTER **E0C33 Family**

***C COMPILER PACKAGE MANUAL***  
***(ver. 2)***



## ***NOTICE***

---

*No part of this material may be reproduced or duplicated in any form or by any means without the written permission of Seiko Epson. Seiko Epson reserves the right to make changes to this material without notice. Seiko Epson does not assume any liability of any kind arising out of any inaccuracies contained in this material or due to its application or use in any product or circuit and, further, there is no representation that this material is applicable to products requiring high level reliability, such as medical products. Moreover, no license to any intellectual property rights is granted by implication or otherwise, and there is no representation or warranty that anything made in accordance with this material will be free from any patent or copyright infringement of a third party. This material or portions thereof may contain technology or the subject relating to strategic products under the control of the Foreign Exchange and Foreign Trade Control Law of Japan and may require an export license from the Ministry of International Trade and Industry or other approval from another government agency.*

Windows95 and Windows NT are registered trademarks of Microsoft Corporation, U.S.A.

PC/AT and IBM are registered trademarks of International Business Machines Corporation, U.S.A.

All other product names mentioned herein are trademarks and/or registered trademarks of their respective owners.

# Introduction

This document describes the development procedure from compiling C source files to debugging and creating the mask data which is finally submitted to Seiko Epson. It also explains how to use each development tool of the E0C33 Family C Compiler Package common to all the models of the E0C33 Family.

## How to read the manual

This manual was edited particularly for those who are engaged in program development. Therefore, it assumes that the reader already possesses the following fundamental knowledge:

- Knowledge about C language (based on ANSI C) and C source creation methods
- Basic knowledge about assembler language
- Basic knowledge about the general concept of program development by a C compiler and an assembler
- Basic operating methods for Windows<sup>®</sup>95, Windows<sup>®</sup>98 or Windows NT<sup>®</sup>4.0

Please refer to manuals or general documents which describe ANSI C and Windows<sup>®</sup> for the above contents.

### Before installation

See Chapter 1. Chapter 1 describes the composition of this package, and provides a general outline of each tool.

### Installation

Install the tools following the installation procedure described in Chapter 2.

### To understand the flow of program development and the operating procedure

See the Tutorial described in Chapter 3. This will give you an overview of program development using the C compiler to the debugger and the mask data creation using the mask data checker.

### For coding

See the necessary parts in Chapter 4. Chapter 4 describes notes on creating source files and the grammar for the assembler language. Also refer to the following manuals when coding:

E0C33xxx Technical Manual

Covers device specifications, and the operation and control method of the peripheral circuits.

E0C33000 Core CPU Manual

Has the instructions and details the functions and operation of the Core CPU.

### For debugging

Chapter 16 explains details of the debugger. Sections 16.1 to 16.8 give an overview of the functions of the debugger. See Section 16.9 for details of the debug commands. Also refer to the following manuals to understand operations of the debugging tools:

E0C33 Family In-circuit Emulator (ICE33) Manual

Explains the functions and handling methods of the In-Circuit Emulator ICE33.

E0C33 Family Peripheral Circuit Board (PRC33xxx) Manual

Explains the functions and handling methods of the peripheral circuit board of the ICE33.

E0C33 Family In-circuit Debugger (ICD33) Manual

Explains the functions and handling methods of the In-Circuit Debugger ICD33.

E0C33 Family MON33 Debug Monitor Manual

Explains the functions and implementation of the Debug Monitor MON33.

### For details of each tool

Chapters 5 to 17 explain the details of each tool. Refer to it if necessary.

### Once familiar with this package

Refer to the listings of instructions and commands contained in Appendices.

## Manual Notations

This manual was prepared by following the notation rules detailed below:

### (1) Sample screens

The sample screens provided in the manual are all examples of displays under Windows<sup>®</sup>95. These displays may vary according to the system or fonts used.

### (2) Names of each part

The names or designations of the windows, menus and menu commands, buttons, dialog boxes, and keys are annotated in brackets [ ]. Examples: [Command] window, [File] menu/[Exit] command ([Exit] command in [File] menu), [Escape break] button, [q] key, etc.

### (3) Names of instructions and commands

The CPU instructions and the debugger commands that can be written in either uppercase or lowercase characters are annotated in lowercase characters in this manual, except for user-specified symbols.

### (4) Notation of numeric values

Numeric values are described as follows:

**Decimal numbers:** Not accompanied by any prefix or suffix (e. g., 123, 1000).

**Hexadecimal numbers:** Accompanied by the prefix "0x" (e. g., 0x0110, 0xffff).

**Binary numbers:** Accompanied by the prefix "0b" (e. g., 0b0001, 0b10).

However, please note that some sample displays may indicate hexadecimal or binary numbers not accompanied by any symbol.

### (5) Mouse operations

**To click:** The operation of pressing the left mouse button once, with the cursor (pointer) placed in the intended location, is expressed as "to click". The clicking operation of the right mouse button is expressed as "to right-click".

**To double-click:** Operations of pressing the left mouse button twice in a row, with the cursor (pointer) placed in the intended location, are all expressed as "to double-click".

**To drag:** The operation of clicking on a file (icon) with the left mouse button and holding it down while moving the icon to another location on the screen is expressed as "to drag".

**To select:** The operation of selecting a menu command by clicking is expressed as "to select".

### (6) Key operations

The operation of pressing a specific key is expressed as "to enter a key" or "to press a key".

A combination of keys using "+", such as [Ctrl]+[C] keys, denotes the operation of pressing the [C] key while the [Ctrl] key is held down. Sample entries through the keyboard are not indicated in [ ]. Moreover, the operation of pressing the [Enter] key in sample entries is represented by ↵.

In this manual, all the operations that can be executed with the mouse are described only as mouse operations. For operating procedures executed through the keyboard, refer to the Windows manual or help screens.

### (7) General forms of commands, startup options, and messages

Items given in [ ] are those to be selected by the user, and they will work without any key entry involved.

An annotation enclosed in < > indicates that a specific name should be placed here. For example, <file name> needs to be replaced with an actual file name.

Items enclosed in { } and separated with | indicate that you should chosen an item. For example, {A | B} needs to have either A or B selected.

## Contents

**Chapter 1 General**

1.1	Features .....	1
1.2	Tool Composition.....	1
1.2.1	Composition of Package.....	1
1.2.2	Outline of Software Tools .....	1

**Chapter 2 Installation**

2.1	Working Environment.....	3
2.2	Installation Method.....	4

**Chapter 3 Software Development Procedures**

3.1	Software Development Flow .....	6
3.2	Tutorial (Flow of Operations with Work Bench) .....	9
3.2.1	Startup of Work Bench wb33 .....	9
3.2.2	Selecting Directory and Displaying File Contents.....	11
3.2.3	Creating Make File .....	12
3.2.4	Auto-execution from Compiling to Linking.....	13
3.2.5	To Execute Tools Individually .....	13
3.2.6	Creating Parameter File for Debugger .....	14
3.2.7	Debugging .....	15
3.2.8	Creating Disassembly File.....	22
3.2.9	Creating ROM Data.....	23
3.2.10	Optimization .....	24
3.2.11	Epilogue.....	25
3.3	Debugging Environment .....	26
3.3.1	In-Circuit Emulator ICE33 .....	26
3.3.2	Debug Monitor MON33 .....	27
3.3.3	In-Circuit Debugger ICD33 .....	30
3.4	Relationship between Program Structure and Memory .....	33

**Chapter 4 Source Files**

4.1	File Format and File Name .....	38
4.2	Grammar of C Source.....	39
4.2.1	Data Type .....	39
4.2.2	Library Functions and Header Files.....	39
4.2.3	In-line Assemble.....	40
4.3	Grammar of Assembly Source.....	41
4.3.1	Statements.....	41
4.3.2	Notations of Operands .....	45
4.3.3	Extended Instructions .....	48
4.3.4	Additional Preprocessor Functions .....	49
4.4	Precautions for Creation of Sources .....	50

**Chapter 5 Work Bench**

5.1	Functions .....	51
5.2	Operations .....	52
5.2.1	Starting Up and Terminating wb33 .....	52
5.2.2	Window .....	53
5.2.3	Selecting File and Displaying Source.....	55
5.2.4	Executing Individual Tools .....	56
5.2.5	Selecting Execution Conditions.....	61

## CONTENTS

5.2.6	Make .....	62
5.2.7	Parameter File Generator .....	68
5.2.8	Specifying a General-purpose Editor .....	70
5.2.9	Entering Command Lines .....	71
5.2.10	Saving and Restoring Options .....	71
5.3	Error Messages .....	72
5.4	Precautions.....	72

### Chapter 6 C Compiler

---

6.1	Functions .....	73
6.2	Input/Output Files.....	73
6.2.1	Input File.....	73
6.2.2	Output Files .....	73
6.3	Starting Method .....	74
6.3.1	Startup Format .....	74
6.3.2	Startup Options .....	74
6.4	Messages.....	76
6.5	Compiler Output .....	77
6.5.1	Output Contents .....	77
6.5.2	Data Representation .....	78
6.5.3	Method of Using Registers .....	79
6.5.4	Function Call .....	80
6.5.5	Stack Frame .....	81
6.6	Debugging Information .....	82
6.6.1	Source Information .....	82
6.6.2	Symbol Information.....	82
6.7	Functions of gcc33 and Usage Precautions .....	87

### Chapter 7 Emulation Library

---

7.1	Overview .....	88
7.2	Floating-point Calculation Library (fp.lib).....	88
7.2.1	Function List.....	88
7.2.2	Floating-point Format.....	89
7.3	Integral Remainder Calculation Library (idiv.lib) .....	90
7.4	Floating-point Calculation Library (fpp.lib).....	90

### Chapter 8 ANSI Library

---

8.1	Overview .....	91
8.2	ANSI Library Function List.....	92
8.2.1	Input/Output Functions (io.lib).....	92
8.2.2	Utility Functions (lib.lib).....	93
8.2.3	Date and Time Functions (lib.lib).....	93
8.2.4	Mathematical Functions (math.lib).....	94
8.2.5	Character Functions (string.lib) .....	94
8.2.6	Character Type Determination/Conversion Functions (ctype.lib) .....	95
8.2.7	Variable Argument Macros (stdarg.h) .....	95
8.3	Declaring and Initializing Global Variables .....	96
8.4	Lower-level Functions .....	97
8.4.1	"read" Function .....	97
8.4.2	"write" Function.....	98
8.4.3	"_exit" Function.....	98

**Chapter 9 Preprocessor**


---

9.1	Functions .....	99
9.2	Input/Output Files.....	99
9.2.1	Input File .....	99
9.2.2	Output Files .....	99
9.3	Starting Method.....	100
9.3.1	Startup Format .....	100
9.3.2	Startup Options.....	100
9.4	Messages .....	101
9.5	Preprocessor Pseudo-Instructions.....	102
9.5.1	Include Instruction (#include) .....	102
9.5.2	Define Instruction (#define) .....	103
9.5.3	Macro Instructions (#macro ... #endm).....	105
9.5.4	Conditional Assembly Instructions (#ifdef ... #else ... #endif, #ifndef... #else ... #endif).....	107
9.6	Operators .....	109
9.7	Debugging Information .....	111
9.8	Comment Adding Function .....	112
9.9	Other Functions .....	112
9.9.1	ASCII to HEX Conversion .....	112
9.9.2	Comment Line .....	112
9.10	Process Flow.....	113
9.11	Sample Executions .....	113
9.12	Error/Warning Messages.....	115
9.12.1	Errors.....	115
9.12.2	Warning.....	116
9.13	Precautions .....	116

**Chapter 10 Instruction Extender**


---

10.1	Functions .....	117
10.2	Input/Output Files.....	117
10.2.1	Input Files.....	117
10.2.2	Output Files .....	118
10.3	Starting Method.....	118
10.3.1	Startup Format .....	118
10.3.2	Startup Options .....	118
10.4	Command File .....	120
10.5	Messages .....	121
10.6	Extended Instructions .....	122
10.6.1	Arithmetic Operation Instructions .....	122
10.6.2	Comparison Instructions.....	123
10.6.3	Logic Operation Instructions .....	124
10.6.4	Shift & Rotate Instructions .....	125
10.6.5	Data Transfer Instructions (between Stack and Register).....	126
10.6.6	Data Transfer Instructions (between Memory and Register).....	127
10.6.7	Immediate Data Load Instructions.....	132
10.6.8	Bit Operation Instructions .....	133
10.6.9	Branch Instructions.....	136
10.7	Optimize Function.....	138
10.7.1	Optimizing Relative Branch Instruction .....	138
10.7.2	Optimization by the Global Pointer.....	139
10.7.3	Optimization by Symbol Information.....	139

## CONTENTS

10.8	Other Functions .....	140
10.8.1	Comment Adding Function .....	140
10.8.2	Classification of Local Symbols .....	140
10.8.3	Syntactic Check .....	140
10.9	Sample Execution .....	141
10.10	Error/Warning Messages .....	148
10.10.1	Errors .....	148
10.10.2	Warning .....	149
10.11	Precautions.....	149

### Chapter 11 Assembler

---

11.1	Functions .....	150
11.2	Input/Output Files.....	150
11.2.1	Input File.....	150
11.2.2	Output Files .....	150
11.3	Starting Method .....	151
11.3.1	Startup Format .....	151
11.3.2	Startup Options .....	151
11.4	Messages.....	152
11.5	Relocatable Assembling and Absolute Assembling.....	153
11.5.1	Relocatable Assembling.....	153
11.5.2	Absolute Assembling.....	153
11.6	Scope .....	154
11.7	Definition of Sections.....	155
11.8	Assembler Pseudo-Instructions.....	158
11.8.1	Absolute Assembling Pseudo-Instruction (.abs).....	158
11.8.2	Section Defining Pseudo-Instructions (.code, .data).....	159
11.8.3	Area Securing Pseudo-Instructions (.comm, .lcomm) .....	160
11.8.4	Location Counter Control Pseudo-Instruction (.org).....	162
11.8.5	Symbol Defining Pseudo-Instruction (.set).....	163
11.8.6	Data Defining Pseudo-Instruction (.word, .half, .byte, .ascii, .space).....	164
11.8.7	Alignment Pseudo-Instruction (.align).....	167
11.8.8	Global Declaring Pseudo-Instruction (.global) .....	167
11.8.9	List Control Pseudo-Instructions (.list, .nolist).....	168
11.8.10	Debugging Pseudo-Instructions (.file, .endfile, .loc, .def).....	169
11.9	Assembly List File.....	170
11.10	Error/Warning Messages .....	171
11.10.1	Errors .....	171
11.10.2	Warning .....	172
11.11	Precautions.....	172

### Chapter 12 Linker

---

12.1	Functions .....	173
12.2	Input/Output Files.....	173
12.2.1	Input Files .....	173
12.2.2	Output Files .....	174
12.3	Starting Method .....	175
12.3.1	Startup Format .....	175
12.3.2	Startup Options .....	175
12.4	Messages.....	176
12.5	Linker Commands .....	177
12.5.1	Linker Command File.....	177
12.5.2	Linker Command List.....	179



12.6	Locating Sections .....	184
12.7	Virtual and Shared (U) Sections .....	187
12.8	Section Symbols .....	190
12.9	Linking Libraries .....	192
12.10	Resolving Symbols .....	193
12.11	Link Map File .....	194
12.12	Symbol File .....	195
12.13	Error/Warning Messages .....	196
12.13.1	Errors .....	196
12.13.2	Warning .....	197
12.14	Precautions .....	198
<hr/>		
<b>Chapter 13 Disassembler</b>		
13.1	Functions .....	199
13.2	Input/Output Files .....	199
13.2.1	Input Files .....	199
13.2.2	Output Files .....	199
13.3	Starting Method .....	200
13.3.1	Startup Format .....	200
13.3.2	Startup Options .....	200
13.4	Messages .....	201
13.5	Disassembling Output .....	202
13.5.1	Mix Output .....	202
13.5.2	Code Output .....	204
13.5.3	Data Output .....	205
13.6	Error/Warning Messages .....	206
13.6.1	Errors .....	206
13.6.2	Warning .....	206
13.7	Precautions .....	207
<hr/>		
<b>Chapter 14 Binary/HEX Converter</b>		
14.1	Functions .....	208
14.2	Input/Output Files .....	208
14.2.1	Input File .....	208
14.2.2	Output Files .....	208
14.3	Starting Method .....	209
14.3.1	Startup Format .....	209
14.3.2	Startup Options .....	209
14.4	Messages .....	210
14.5	Contents of HEX File .....	211
14.5.1	Motorola S3 Format .....	211
14.5.2	Absolute Address Output .....	211
14.5.3	Offset Address Output .....	211
14.6	Error/Warning Messages .....	212
14.6.1	Errors .....	212
14.6.2	Warning .....	212
14.7	Precautions .....	213
<hr/>		
<b>Chapter 15 Librarian</b>		
15.1	Functions .....	214
15.2	Input/Output Files .....	214
15.2.1	Input Files .....	214
15.2.2	Output Files .....	214

## CONTENTS

15.3	Starting Method .....	215
15.3.1	Startup Format .....	215
15.3.2	Startup Options .....	215
15.4	Messages .....	216
15.5	Library Editing Functions .....	217
15.5.1	Creating a New Library .....	217
15.5.2	Adding Modules to a Library .....	217
15.5.3	Listing Registered Modules .....	218
15.5.4	Deleting Modules from a Library .....	218
15.5.5	Restoring Object Files .....	218
15.6	Error/Warning Messages .....	219
15.6.1	Errors .....	219
15.6.2	Warnings .....	219
15.7	Precautions .....	219

## Chapter 16 Debugger

---

16.1	Features .....	220
16.2	Input/Output Files .....	220
16.2.1	Input Files .....	220
16.2.2	Output File .....	221
16.3	Starting Method .....	222
16.3.1	Startup Format .....	222
16.3.2	Startup Options .....	222
16.3.3	Startup Messages .....	223
16.3.4	Method of Termination .....	226
16.4	Windows .....	227
16.4.1	Basic Structure of Window .....	227
16.4.2	[Command] Window .....	229
16.4.3	[Source] Window .....	230
16.4.4	[Memory] Window .....	233
16.4.5	[Register] Window .....	234
16.4.6	[Trace] Window .....	235
16.4.7	[Symbol] Window .....	237
16.4.8	[Simulated I/O] Window .....	238
16.5	Tool Bar .....	239
16.5.1	Tool Bar Structure .....	239
16.5.2	[Key break] Button .....	239
16.5.3	[Load file] Button .....	239
16.5.4	[Source], [Mix] and [Unassemble] Buttons .....	239
16.5.5	[Go], [Go to], [Step], [Next], [Reset cold] and [Rest hot] Buttons .....	239
16.5.6	[Soft PC break] and [Hard PC break] Buttons .....	240
16.5.7	[Symbol watch], [Symbol add] and [Symbol delete] Buttons .....	240
16.5.8	[Display trace] and [Resume trace] Buttons .....	241
16.5.9	[Select source] Combo Box .....	241
16.6	Menu .....	242
16.6.1	Menu Structure .....	242
16.6.2	[File] Menu .....	242
16.6.3	[Edit] Menu .....	242
16.6.4	[Run] Menu .....	242
16.6.5	[Break] Menu .....	243
16.6.6	[Symbol] Menu .....	243
16.6.7	[Window] Menu .....	243
16.6.8	[Help] Menu .....	244

16.7	Method for Executing Commands.....	245
16.7.1	Entering Commands from Keyboard.....	245
16.7.2	Parameter Input Formats.....	246
16.7.3	Executing from Menu or Tool Bar.....	249
16.7.4	Executing from Command File.....	250
16.7.5	Log File.....	251
16.8	Debug Functions.....	252
16.8.1	Debugger Mode.....	252
16.8.2	Loading Files.....	256
16.8.3	Source Display and Symbolic Debugging Function.....	257
16.8.4	Displaying and Modifying Memory Data and Register.....	260
16.8.5	Executing Program.....	262
16.8.6	Break Functions.....	266
16.8.7	Trace Functions.....	271
16.8.8	Simulated I/O.....	280
16.8.9	Operation of Flash Memory.....	282
16.8.10	Other Functions.....	285
16.8.11	Big-Endian Support.....	286
16.9	Command Reference.....	287
16.9.1	Command List.....	287
16.9.2	Commands to Operate Memory.....	288
	fb (fill byte) [ICD / ICE / SIM / MON].....	288
	fh (fill half) [ICD / ICE / SIM / MON].....	289
	fw (fill word) [ICD / ICE / SIM / MON].....	290
	db (dump byte) [ICD / ICE / SIM / MON].....	291
	dh (dump half) [ICD / ICE / SIM / MON].....	293
	dw (dump word) [ICD / ICE / SIM / MON].....	295
	df (dump file) [ICD / ICE / SIM / MON].....	297
	eb (enter byte) [ICD / ICE / SIM / MON].....	298
	eh (enter half) [ICD / ICE / SIM / MON].....	299
	ew (enter word) [ICD / ICE / SIM / MON].....	300
	mv (move) [ICD / ICE / SIM / MON].....	301
	mvh (move half) [ICD / (ICE) / SIM / MON].....	302
	mvw (move word) [ICD / (ICE) / SIM / MON].....	303
	w (watch) [ICD / ICE / SIM / MON].....	304
	rm (read memory) [ICD].....	305
16.9.3	Commands to Operate on Register.....	306
	rd (register display) [ICD / ICE / SIM / MON].....	306
	rs (register set) [ICD / ICE / SIM / MON].....	307
16.9.4	Commands to Execute Program.....	308
	g (go) [ICD / ICE / SIM / MON].....	308
	s (step) [ICD / ICE / SIM / MON].....	310
	n (next) [ICD / ICE / SIM / MON].....	312
16.9.5	Commands to Reset CPU.....	313
	rstc (cold reset CPU) [ICD / ICE / SIM / MON].....	313
	rsth (hot reset CPU) [ICD / ICE / SIM / MON].....	314
16.9.6	Interrupt Command.....	315
	int (interrupt) [SIM].....	315
16.9.7	Commands to Set Breaks.....	316
	bp (break point set) [ICD / ICE / SIM / MON].....	316
	bs (break software) [ICD / ICE / SIM / MON].....	320
	bc (break clear) [ICD / ICE / SIM / MON].....	321
	bh (break hardware) [ICD / ICE / SIM / MON].....	322
	bhc (break hardware clear) [ICD / ICE / SIM / MON].....	323

bh2 (break hardware 2) [ICD / (ICE) / SIM / MON].....	324
bhc2 (break hardware 2 clear) [ICD / (ICE) / SIM / MON].....	325
bd (data break) [ICD / ICE / SIM / MON].....	326
bsq (break sequential) [ICE].....	328
bl (break list) [ICD / ICE / SIM / MON].....	331
bac (break all clear) [ICD / ICE / SIM / MON].....	332
16.9.8 Commands to Display Program.....	333
u (unassemble) [ICD / ICE / SIM / MON].....	333
sc (source code) [ICD / ICE / SIM / MON].....	335
m (mix) [ICD / ICE / SIM / MON].....	337
ss (search strings) [ICD / ICE / SIM / MON].....	339
16.9.9 Commands to Display Symbol Information.....	340
sy (symbol list) [ICD / ICE / SIM / MON].....	340
sa (symbol add) [ICD / ICE / SIM / MON].....	345
sd (symbol delete) [ICD / ICE / SIM / MON].....	348
sw (symbol watch) [ICD / ICE / SIM / MON].....	349
16.9.10 Commands to Load Files.....	352
lf (load file) [ICD / ICE / SIM / MON].....	352
lh (load hex) [ICD / ICE / SIM / MON].....	354
ld (load file) [ICD / ICE / SIM / MON].....	355
16.9.11 Commands to Operate Flash Memory.....	356
fls (flash memory set) [ICD / (ICE) / MON].....	356
fle (flash memory erase) [ICD / (ICE) / MON].....	357
lfl (load from flash memory) [ICE].....	358
sfl (save to flash memory) [ICE].....	359
efl (erase flash memory) [ICE].....	360
maf (map flash memory) [ICE].....	361
16.9.12 Trace Commands.....	362
tm (trace mode) [ICD / ICE / SIM].....	362
td (trace dump) [ICD / ICE].....	368
ts (trace search) [ICD / ICE].....	373
tf (trace file) [ICD / ICE].....	375
16.9.13 Simulated I/O.....	376
stdin (standard input) [ICD / ICE / SIM / MON].....	376
stdout (standard output) [ICD / ICE / SIM / MON].....	377
16.9.14 Other Commands.....	378
com (execute command file) [ICD / ICE / SIM / MON].....	378
cmw (execute command file with wait) [ICD / ICE / SIM / MON].....	379
log (logging) [ICD / ICE / SIM / MON].....	380
od (option data dump) [ICE].....	381
ct (change type) [ICD / ICE / SIM / MON].....	382
ext (extended instruction) [ICD / ICE / SIM / MON].....	384
ma (map information) [ICD / ICE / SIM / MON].....	386
md (mode) [ICD / ICE / SIM / MON].....	387
q (quit) [ICD / ICE / SIM / MON].....	389
? (help) [ICD / ICE / SIM / MON].....	390
ice (ice) [ICE].....	391
16.10 Parameter File.....	392
16.11 Status/Error/Warning Messages.....	397
16.11.1 Status Messages.....	397
16.11.2 Error Messages.....	397
16.11.3 Warning Messages.....	400

**Chapter 17 Other Tools**


---

17.1	Make.....	401
17.1.1	Starting Method.....	401
17.1.2	Messages.....	402
17.1.3	Make File.....	403
17.1.4	2-pass make .....	408
17.1.5	clean.....	408
17.1.6	Error/Warning Messages .....	409
17.1.7	Precautions .....	409
17.2	cwait.....	410
17.2.1	Functions .....	410
17.2.2	Method for Using cwait.....	410
17.3	ccap.....	411
17.3.1	Functions .....	411
17.3.2	Method for Using ccap .....	411

**Appendix srf33 File Structure**


---

A-1	srf33 Object File Structure .....	413
A-2	Library File Structure.....	418

**Quick Reference**

# Chapter 1 General

## 1.1 Features

---

The E0C33 Family C Compiler Package contains software development tools for compiling C source files, assembling assembly source files, linking object files, debugging executable files, making mask data and other utilities. The tools are common to all the models of the E0C33 Family.

Its principal features are as follows:

### Powerful optimizing function

The C Compiler is designed to suit to the E0C33 architecture, it makes it possible to deliver minimized codes. The high-optimize ability does not lose most of the debugging information, and it enables C source level debugging.

Furthermore, the Instruction Extender also provides the optimizing function using the map/symbol information after linking.

### Useful extended instructions are provided

The extended instructions allow the programmer to describe assembly source simply without the need of knowing the data size. The immediate data extension using the "ext" instruction and some useful functions that need multiple basic instructions are described with an extended instruction.

### C and assembly source level debugger with a simulator function

The debugger supports C source level debugging and assembly source level debugging. By using the ICE33, ICD33 or MON33, the program can be debugged even when the target board is operating. It also provides a simulator function that allows debugging on a personal computer without using the ICE33.

### Integrated working environment, by Work Bench

The Work Bench supports Windows GUI and allows a series of tools to be executed through its windows. All the basic operations can be executed by the mouse alone.

## 1.2 Tool Composition

---

### 1.2.1 Composition of Package

The E0C33 Family C Compiler Package contains the elements listed below. Please check to make sure that all elements are supplied.

- |   |                                  |
|---|----------------------------------|
| 1) Tool disks (CD-ROM)                                  | One                              |
| 2) E0C33 Family C Compiler Package Manual (this manual) | One each in English and Japanese |
| 3) Warranty card  | One each in English and Japanese |

### 1.2.2 Outline of Software Tools

The following shows the outlines of the principle tools included in the package:

#### (1) C Compiler (gcc33.exe)

This tool is made based on GNU C Compiler designed by Free Software Foundation, Inc. and is compatible with ANSI C.

The gcc33 compile C source files to the assembly source files for the E0C33 Family. It has a powerful optimizing ability that can generate minimized assembly codes. The gcc33 consists of three files: gcc33.exe, cpp.exe and cc1.exe.

#### (2) Preprocessor (pp33.exe)

The Preprocessor pp33 starts the processing procedure of assembly source files when developing programs in assembler language. The pp33 expands the range of program-creating functions, such as for macro statements that makes it possible to use a group of multiple statements as if they were one single statement and include statements that insert other files, and thus creates assembly source files to be entered into the Instruction Extender ext33.

**(3) Instruction Extender (ext33.exe)**

The Instruction Extender ext33 optimizes the assembly source files by decreasing the immediate extension instructions (ext) of the E0C33000 instruction set. The extended instructions that enable program description without the need of knowing immediate data extension are provided by the ext33. The ext33 also supports a 2-pass make that optimizes source codes using the map/symbol information after linking.

**(4) Assembler (as33.exe)**

The Assembler as33 assembles assembly source files output by the ext33 and converts the mnemonics of the source files into object codes (machine language) of the E0C33000. The results are output in an object file that can be linked or added to a library.

**(5) Linker (lk33.exe)**

The linker defines the memory locations of object codes created by the as33, and creates executable object codes. This tool puts together multiple objects and library files into one file.

**(6) Disassembler (dis63.exe)**

The Disassembler dis33 disassembles the srf33 object file output by the lk33, and creates a file that can be referred to with mnemonic codes and source codes. This function is effective when viewing the correspondence between source codes and absolute addresses after linking.

**(7) Binary/HEX Converter (hex33.exe)**

The Mask Data Checker converts the srf33 object file output by the lk33 into a Motorola S3 format HEX file for writing to the ROM. HEX data for the external ROM can be written to ROMs using a ROM writer. HEX data for the internal ROM becomes the mask data.

**(8) Debugger (db33.exe)**

The Debugger db33 serves to perform debugging by controlling the hardware tool (ICE33 or ICD33) or the debug monitor (MON33). It also comes with a simulator function that allows debugging on a personal computer. Commands that are used frequently, such as break and step, are registered on the tool bar, minimizing the necessary keyboard operations. Moreover, it supports C and assembly source level debugging, and various data can be displayed in multi windows, with a resultant increased efficiency in the debugging tasks.

**(9) Librarian (lib33.exe)**

The Librarian lib33 edits libraries. The lib33 can register object modules created by the as33 to libraries, delete object modules in libraries and restore library modules to the original object files.

**(10) Make (make.exe)**

The Make automatically executes from compile to link according to the command lines described in the make file. The make file can be created by the wb33.

**(11) Work Bench (wb33.exe)**

This software enables the tools mentioned above to be started up from one single window. The selection of files, major startup options, and the startup of each tool can be executed by mouse operations alone. The wb33 establishes an efficient working environment for development tasks.

This package contains sample programs and several utility programs. For details on those programs, please refer to "readme.txt" (English) or "readmeja.txt" (Japanese) on the disk.

## Chapter 2 Installation

This chapter describes the required working environments for the tools supplied in the E0C33 Family C Compiler Package and their installation methods.

### 2.1 Working Environment

---

To use the E0C33 Family C Compiler Package, the following conditions are necessary:

#### Personal computer

An IBM PC/AT or a compatible machine which is equipped with a CPU equal to or better than a Pentium 90 MHz, and 32MB or more of memory is recommended.

To use the optional In-Circuit Emulator ICE33 or In-Circuit Debugger ICD33, the personal computer also requires a serial port (with a D-sub 9 pin) and a parallel port (D-sub 25 pin). When using the Debug Monitor MON33 with the DMT33MON board, only a serial port (with a D-sub 9 pin) is required.

#### Display

A display unit capable of displaying 800 × 600 dots or more is necessary.

#### Hard drive

The hard drive must have at least 10MB of empty space to install the E0C33 Family C Compiler Package.

#### CD-ROM drive

Since the installation is done from a CD-ROM, a CD-ROM drive is required.

#### Mouse

A mouse is necessary to operate the tools.

#### Debugging tool

To debug the program and the target system, the optional In-Circuit Emulator (ICE33), In-Circuit Debugger (ICD33), or Debug Monitor (MON33 and DMT33MON) is needed in addition to this software package.

#### System software

The E0C33 Family C Compiler Package supports Microsoft® Windows®95, Windows NT®4.0 or higher version (English or Japanese version).

#### Other

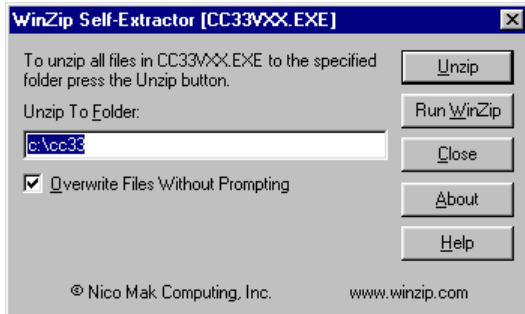
Please go through the precautions and restrictions given in "readme.txt" (English) or "readmeja.txt" (Japanese) on the disk.



## 2.2 Installation Method

All the tools in the E0C33 Family C Compiler Package are supplied on one CD-ROM. Execute the self-extract file "cc33vXX.exe" on the CD-ROM to install the files. ("XX" in the file name represents the version number, for example, "cc33v20.exe" is the file name of ver. 2.0.)

When "cc33vXX.exe" is started up by double-clicking the file icon, the following dialog box appears.



Enter a path/folder name in the text box then click [Unzip]. The specified folder will be created and all the files will be copied to the folder.

When the specified folder already exists on the specified path, the folder will be overwritten without prompting if [Overwrite Files Without Prompting] is checked.

The following lists the configuration of directories and files after copying.

RootDIR-	readme.txt	Information of tools (English)
(C:\CC33\)	readmeja.txt	Information of tools (Japanese)
	GNU_COPYRIGHT	GNU copyright
	wb33.exe, ccap.exe	Work Bench and accompany tool
	make.exe, cwait.exe	make and accompany tool
	gcc33.exe, cpp.exe, cc1.exe	C Compiler
	pp33.exe	Preprocessor
	ext33.exe	Instruction Extender
	as33.exe	Assembler
	lk33.exe	Linker
	lib33.exe	Librarian
	db33.exe	Debugger
	dis33.exe	Disassembler
	hex33.exe	Binary/HEX Converter
	vb40032.dll, olepro32.dll, msvcr40.dll	dll files for Work Bench
	lib\ - io.lib, lib.lib, math.lib, ctype.lib, string.lib, idiv.lib, fp.lib	
	include\ - stdio.h, stdlib.h, time.h, math.h, errno.h, float.h, limits.h, ctype.h, string.h, stdarg.h	
	sample\	
	utility\	

Refer to the "readme.txt" (English), "readmeja.txt" (Japanese) or "\*\_man.txt" (English) for the contents of the "sample" and "utility" directories.

### Precautions on setting the OS

- Set the display property as "Small fonts" used by the "Display" in the control panel.
- When using a drive on the network as the tool and/or work drive, be sure to assign a drive name to it. The network name cannot be used.
- Do not use the COM and LPT ports for the debugging tool (ICE33, ICD33 or MON33) in other drivers and applications. Furthermore, make sure that the port has been enabled when using a note PC as some can disable COM ports.
- If the debugger db33 or work bench wb33 have a problem on the GUI that causes an abnormal display, decrease the function level of the graphics or use a low-level standard display driver which has been supplied in the Windows package.

### To delete tools

The files are all installed in the specified directory (default is "C:\CC33\"). To delete all the tools, delete the directory (folder).

### GNU copyright

The C Compiler gcc33 in this package is made based on the GNU C Compiler designed by Free Software Foundation, Inc. Please read the "GNU\_COPYRIGHT" text file for the license before using.

# Chapter 3 Software Development Procedures

This chapter explains the flow through the basic operating methods of Work Bench wb33, from compiling program to debugging and creating mask data. The sample programs discussed in this chapter are installed in the "sample\tst\" and "sample\dm33005\" directories. It is possible to practice the operations by following the manual.

## 3.1 Software Development Flow

Figure 3.1.1 shows the flow of software development work.

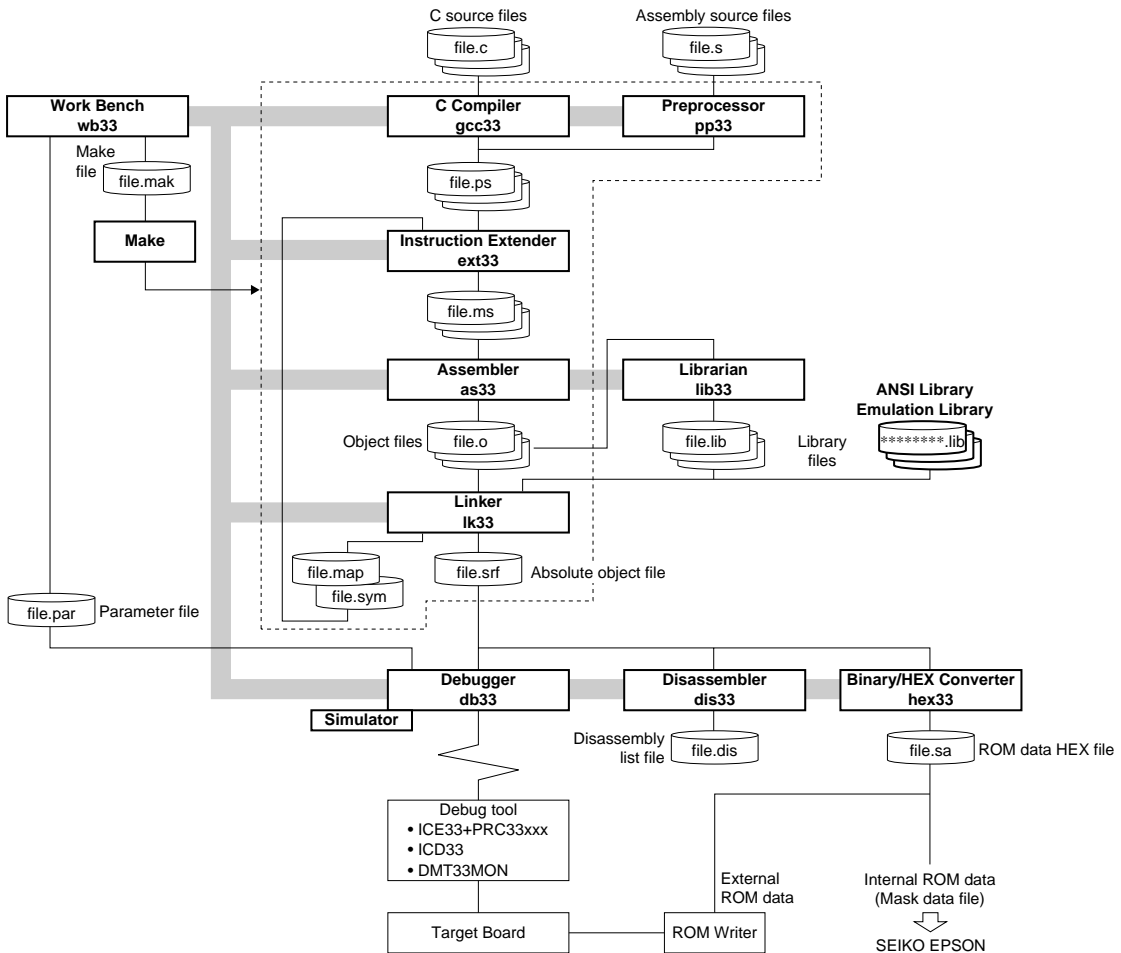


Fig. 3.1.1 Software development flow

As shown in the figure, the tools of this package support for all the software processing after creating source files.

The development flow is detailed below.

### (1) Creating a source program

Create source files using a general-purpose editor. A program can be created in several separate modules (source files).

### (2) Creating a make file

Create a make file for automatic processing from compiling or preprocessing to linking. A basic make file can be created easily on the Work Bench.

### (3) Executing make

Execute the make using the make file created to generate an srf33 object file that can be debugged.

The make sequentially executes the necessary processes from among the ones below.

#### Compiling (in case of C sources)

The C source files are compiled by the C Compiler gcc33. The gcc33 delivers the assembly source files (.ps) to be entered in the Instruction Extender ext33.

#### Preprocessing by the Preprocessor (in case of assembly sources)

The source files that are created in assembler language are first processed by the Preprocessor pp33.

The pp33 expands the preprocessor instructions into mnemonics that can be assembled with the Assembler and delivers assembly source files (.ps) to be entered in the Instruction Extender ext33.

#### Optimization by the Instruction Extender

The Instruction Extender ext33 expands the extended instructions described in the source file (.ps) into mnemonics that can be assembled with the Assembler and delivers assembly source files (.ms) to be entered in the Assembler as33.

Furthermore, the ext33 optimizes the assembly source by decreasing unnecessary immediate extension instructions (ext).

The absolute addresses of symbols cannot be defined until the linking has finished when developing the program with multiple modules. The ext33 supports a 2-pass make that optimizes the codes using the symbol/map files created when linking. When a 2-pass make is specified, the make executes the ext33 and the following process again after the first linking has finished.

#### Assembling

The source files that are delivered from the Instruction Extender ext33 are assembled by the Assembler as33. The as33 converts the source codes into machine codes and delivers the object file that can be linked with other modules be registered to libraries.

When a multi-module software program (multiple source files) is developed, all the source files are subjected to the above processing.

#### Linking

One or more object files are produced by the assembling. The Linker lk33 bundles those multiple files into one to create an executable object file mapped on the ROM. The lk33 delivers object files in srf33 format, which contains necessary information for debugging, along with other information.

### (4) Debugging

The srf33 object file that is delivered from the linker should be debugged by the Debugger db33. Using the ICE33, ICD33 or Debug Monitor allows the programmer to perform debugging, including that for the hardware operation. The db33 also provides a simulator mode in which the operations of the E0C33000 Core CPU and memory models can be simulated on a personal computer.

**(5) Disassembling**

The Disassembler dis33 disassembles a linked object file for the purpose of verifying the correspondence between source codes and absolute addresses, or for dumping data from the data area. It is not an indispensable tool for program development, but it is suggested to use it as a utility tool.

**(6) Creating ROM data/mask data**

To make the target ROM and/or the mask data, create the external ROM data HEX file and/or the internal ROM data HEX file from the srf33 object file delivered by the Linker using the Binary/HEX converter hex33. Finally submit the mask data (internal ROM data) to Seiko Epson.

The tools above can be executed in the Work Bench. Each tool can also be executed individually without using the make.

Besides these tools, the Librarian lib33 is provided. The lib33 can make and edit libraries with the general-purpose modules (object files delivered from the Assembler). It will be effective for developing applications using the E0C33 Family in the future.

## 3.2 Tutorial (Flow of Operations with Work Bench)

The tools described in the preceding section can readily be started up from the Work Bench wb33, which comprises part of this package. In this section, the flow of operations with the Work Bench wb33 will be learned by a tutorial. For details on each tool, refer to the corresponding chapter.

### Files to be used

The explanation in this section presupposes that the files listed below exist in the "sample\tst\" directory.

**main.c** ...C source file

**boot.s** ...Assembly source file

The following description covers basic operation procedures from compiling/preprocessing to linking for two sample source files (main.c and boot.s) using the make, and basic debugging procedures. Then explains the process necessary for masking the ROM.

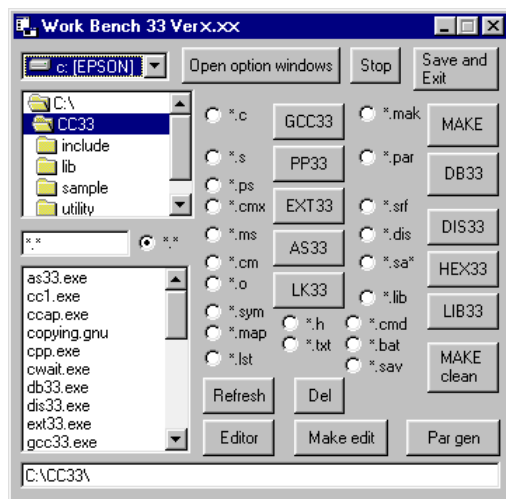
### 3.2.1 Startup of Work Bench wb33



wb33.exe

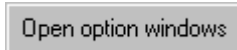
Start up the Work Bench wb33 by double clicking the "wb33.exe" icon located in the "cc33" folder.

The execution window opens as below.

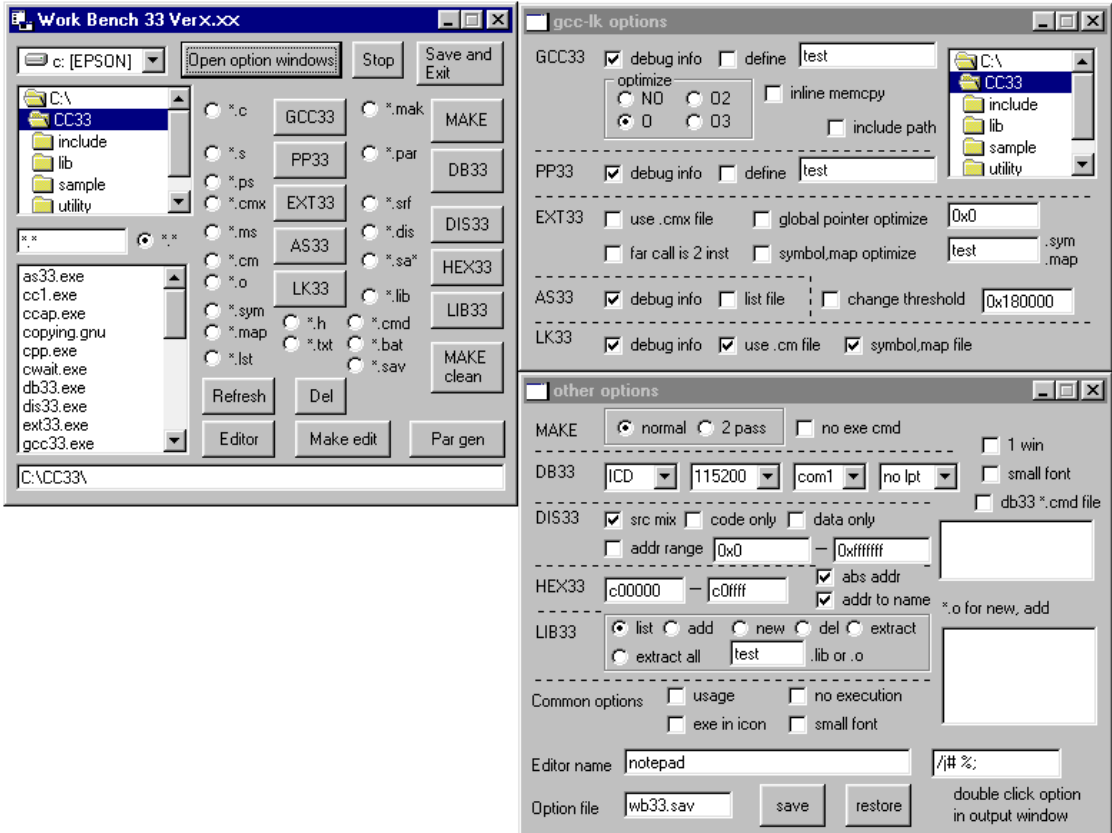


The execution window has the list boxes for choosing files and the buttons for starting up the tools.

**Step 1)** Click [Open option window]. Two option windows open.



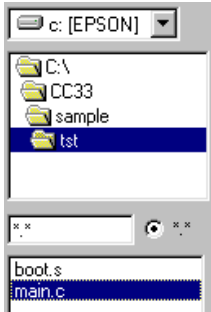
[Open option window] button



In the [gcc-lk options] window, the start-up options from the C Compiler to the Linker can be specified. Other tool options and the options common to all tools can be specified in the [Other options] window.

The check boxes designed to specify an option are initially selected and specified during startup of the Work Bench wb33, and the one usually specified displays a check in it. The explanation below assumes the initial settings, unless otherwise specified. For details, refer to Chapter 5 "Work Bench" and the chapters corresponding to the respective tools.

## 3.2.2 Selecting Directory and Displaying File Contents



There is a file selection part in the execution window. When the Work Bench (wb33) starts up, it shows the drive name and the directory in which the tools are installed. First, display the files to be used in tutorial.

**Step 2)** Select the "sample\tst\" directory in the directory list box.

Since the initial setting checked the [\*.\*] radio button, the names of all files in the "tst\" directory appear in the file list box.

It is possible to change the file type to be displayed by selecting the radio button on the left of each tool button. The radio buttons show the file types that can be input to the corresponding tool.

**Step 3)** Click [\*.\*].

The file list box shows the main.c only.

### To display contents of source file

The Work Bench wb33 has a text file display function.

**Step 4)** Double-click the source file name (main.c) in the file list box.



The output window opens and displays the contents of the main.c.

Notes:

- Only text files can be displayed in this window, and they are limited to a maximum size of 32KB. If codes other than ASCII characters are contained in the files, they may appear as gibberish.

- A character string can be copied or corrected inside the window, but changes cannot be saved. This facility should be used only as a display function.

### To open an editor

The Work Bench wb33 can open an editor for displaying the selected text file.



**Step 5)** Select "main.c", then click [Editor] in the execution window.

[Editor] button



The notepad of Windows opens and displays the contents of the main.c.

This function allows editing source files instantly.

The notepad is selected as the editor by the initial setting. It is possible to change it to the editor always in use by entering the start-up command (full-path name) of the editor to the [Editor name] text box in the [other options] window.

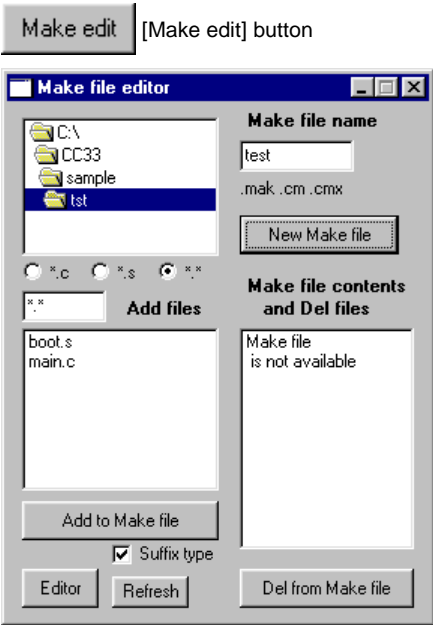


### 3.2.3 Creating Make File

The make file contains a processing procedure from compiling to linking and the procedure is automatically executed by the make tool. The make can judge whether the files are updated or not, and executes the process only when the necessary file has been modified or there is no target file.

The following operation creates the make file for processing the sample source files (main.c and boot.s).

#### To create a make file



**Step 6)** Click [Make edit].

The [Make file editor] window appears.

**Step 7)** Select the main.c and boot.s in the file list box, then click [New Make file].

To select two files, first click the boot.s, then hold down the [Ctrl] key and click the main.c.

The make generator creates the following three files:

- test.mak    make file
- test.cm    Command file for Linker
- test.cmx    Command file for Instruction Extender

These files are created in text format, so they can be displayed in the output window or with an editor.

The make editor uses the name that is entered in the [Make file name] text box as the make file name (default is test). Modify the name in the text box if another name is to be used. This name also applies to the object file that will be created by linking and other files.

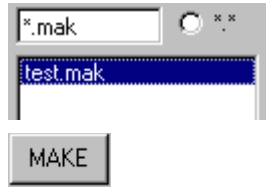
The make editor creates a make file with basic contents, therefore use it as a template and customize the contents if necessary. See Section 17.1, "Make" for details of the contents of make file.

Use the close button to terminate the Make file editor.

### 3.2.4 Auto-execution from Compiling to Linking

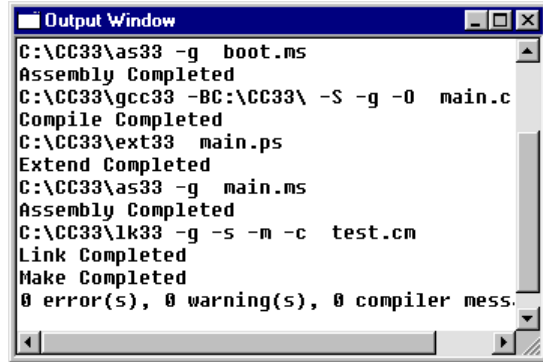
Execute the Make using the test.mak created in the previous section.

#### To execute the Make



[MAKE] button

**Step 8)** Select the "test.mak" in the file list box on the execution window, then click [MAKE].



The Make sequentially executes preprocessing and assembling the boot.s, compiling and assembling the main.c and linking the object files. As a result, the output files of the executed tools appear in the "sample\tst\" directory.

boot.ps: Output file of the Preprocessor pp33

main.ps: Output file of the C Compiler gcc33

boot.ms, main.ms: Output file of the Instruction

Extender ext33

boot.o, main.o: Output file of the Assembler as33

test.srf, test.sym: Output file of the Linker lk33

### 3.2.5 To Execute Tools Individually

The tools can be executed individually. For example, to execute the Compiler only,

**Step 9)** Display the C source file (main.c) by selecting the [\*.\*] radio button (if necessary).

**Step 10)** Select the main.c in the file list box, then click [GCC33].

When correcting syntax errors in source files, the Compiler can only be executed in this method.

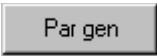
Other tools can also be executed individually with a similar operation.

### 3.2.6 Creating Parameter File for Debugger

It is necessary to create a parameter file for the Debugger before starting to debug.

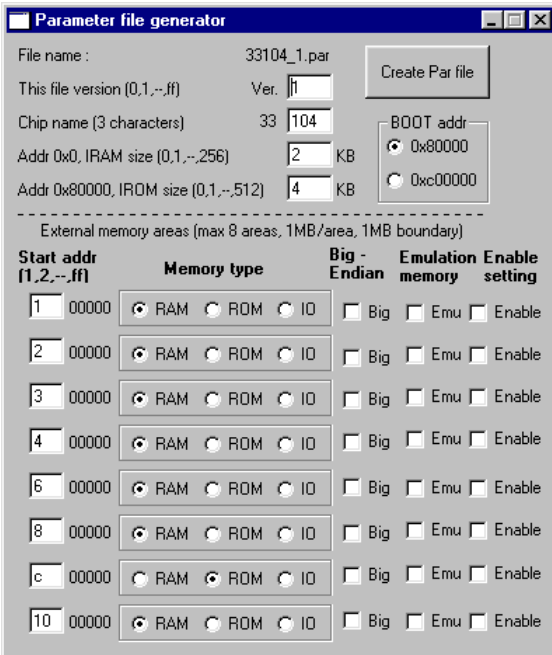
The Debugger db33 supports all the models of the E0C33 Family. However, since each model comes with its own memory configuration and different PRC board, information concerning the available memory range and PRC board is necessary for each specific mode. The parameter is used to set the information to the debugger.

#### To create a parameter file



[Par gen] button

**Step 11)** Click [Par gen].



The [Parameter file generator] window appears.

This tutorial uses the default settings for creating a parameter file. In the actual development, memory map information should be specified in the [Parameter file generator] window.

**Step 12)** Click [Create Par file].

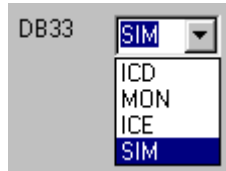
The parameter file 33104\_1.par is created.

See Section 16.10, "Parameter File" for the contents of the parameter file and specifying the parameters.

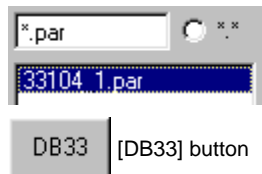
### 3.2.7 Debugging

Now that an object file is created by the Make in an executable format (srf33), debugging of the program can be performed. Although more sophisticated debugging could be done using the ICE33 or ICD33, this section explains how to start up the Debugger db33 in the simulator mode, in which debugging can be executed on a personal computer alone. This will enable practice and understanding of the fundamental operations of the package.

#### To start up the Debugger db33



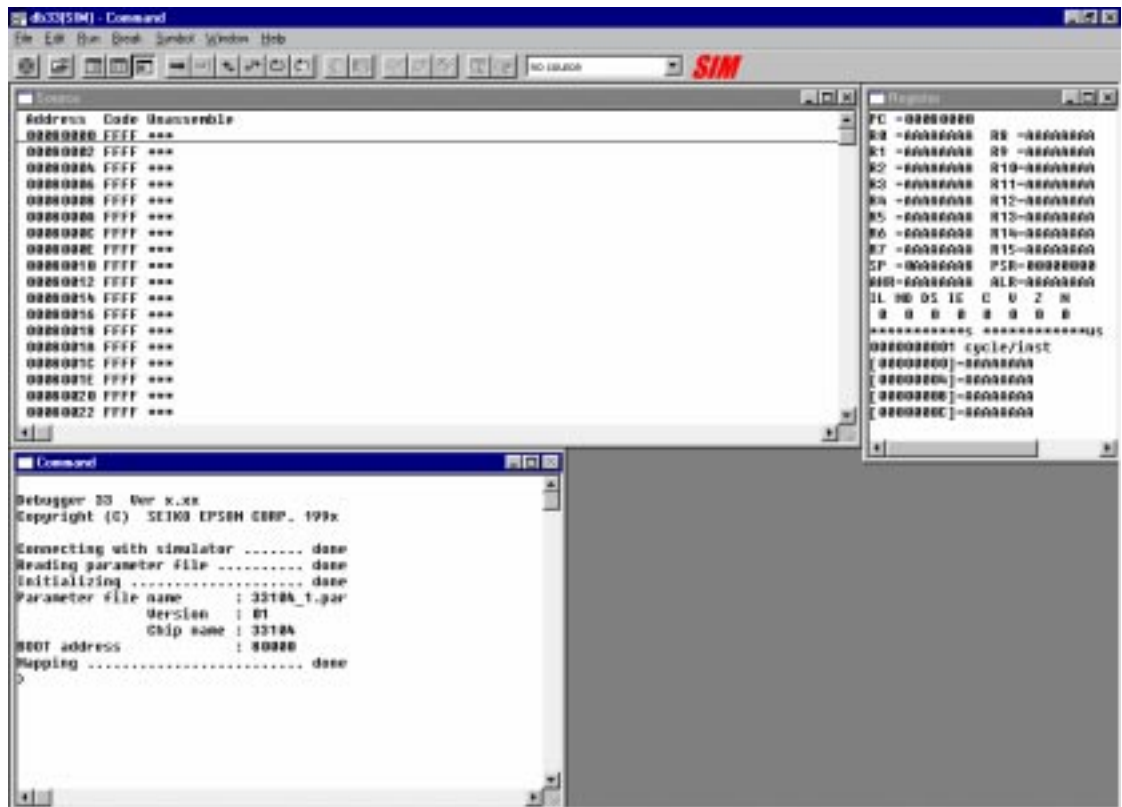
**Step 13)** Select [SYM] at the DB33 option field in the [other options] window. (Simulator mode specified)



**Step 14)** Select the parameter file (33104\_1.par) in the file list box, then click [DB33].

To select the file name easily, select the [.par] radio button.

The window below opens when the Debugger db33 starts up.



First, the file to be debugged should be read.

**To read a file**



[Load file] button

**Step 15)** Click [Load file]. A dialog box for file selection opens.

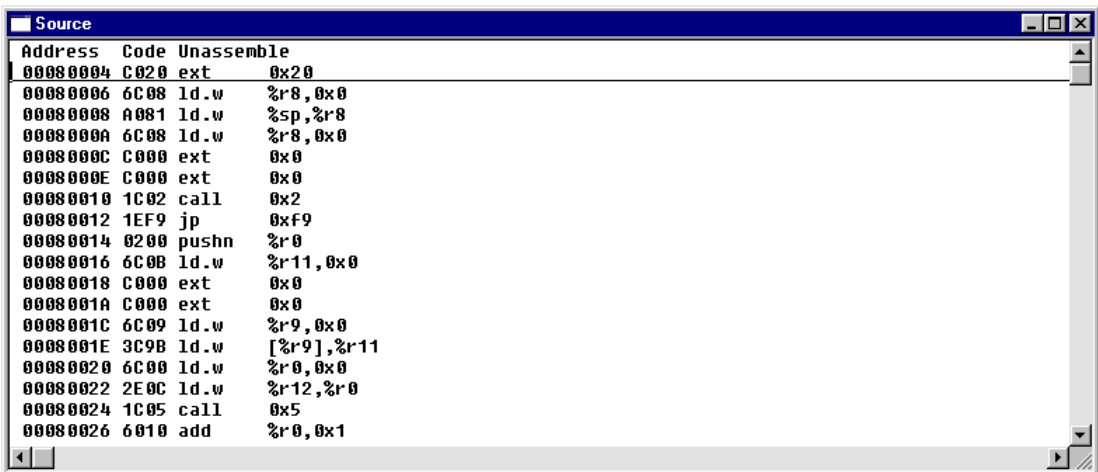


**Step 16)** Select the test.srf in the file list box of the dialog box, then click [OK]. The object file test.srf is read.



[Reset cold] button

**Step 17)** Click [Reset cold]. (The PC is set to the program start address.)



The [Source] window shows the disassembled object codes. This display can be changed for a display of the source or for a mixed display (display of both the disassembled contents and the source).

## To display a source



**Step 18)** Click [Source] on the tool bar. The [Source] window display changes.

[Source] button

```

Source - boot.s
Address  Line  SourceCode
00001   ; boot.s 1997.2.13
00002   ; boot program
00003
00004 #define SP_INI 0x0000 ; sp is in end of 2KB internal RAM
00005 #define GP_INI 0x0000 ; global pointer %r8 is 0x0
00006
00007         .code
00008         .word BOOT           ; BOOT VECTOR
00009 BOOT:
00080004 00010         xld.w   %r8,SP_INI
00080008 00011         ld.w    %sp,%r8       ; set SP
0008000A 00012         ld.w    %r8,GP_INI     ; set global pointer
0008000C 00013         xcall  main           ; goto main
00080012 00014         xjp    BOOT           ; infinity loop
  
```

The [Source] window displays the contents of the source file (boot.s) which contains the code at the current PC address. Another source (e.g. main.c) can be displayed by selecting it from the combo box on the tool bar if the object file can refer to the source file.

## To display a mix



**Step 19)** Click [Mix] on the tool bar. The [Source] window display changes.

[Mix] button

```

Source
Address  Code  Unassemble          Line  SourceCode
00080004 C020  ext                0x20  00010         xld.w   %r8,SP_INI
00080006 6C08  ld.w               %r8,0x0
00080008 A081  ld.w               %sp,%r8  00011         ld.w    %sp,%r8       ; set SP
0008000A 6C08  ld.w               %r8,0x0  00012         ld.w    %r8,GP_INI     ; set global po
0008000C C000  ext                0x0     00013         xcall  main           ; goto main
0008000E C000  ext                0x0
00080010 1C02  call               0x2
00080012 1EF9  jp                 0xF9  00014         xjp    BOOT           ; infinity loop
                                --- main.c ---
00001 /* tst_main.c 1997.2.13 */
00002 /* C main program */
00003
00004 int i;
00005
00006 main()
00007 {
00008     int j;
00009
00080014 0200  pushn              %r0
  
```

The [Source] window displays the results of disassembling and the contents of the source file. This display clearly shows the correspondence between the source and the mnemonic. The underlined line denotes the instruction (address) to be executed next.

The program can now be executed on the file that was just read.

**To execute a program**



[Go] button

**Step 20)** Click [Go] on the tool bar.

This program infinitely repeats increments by using the variable i (address 0x0000000–0x0000003) in the RAM area as a counter. In the ICE mode, it can be seen that the on-the-fly function updates the contents of the [Register] window in real time. In the simulator mode, the contents of the [Register] window are not displayed until the program is broken.

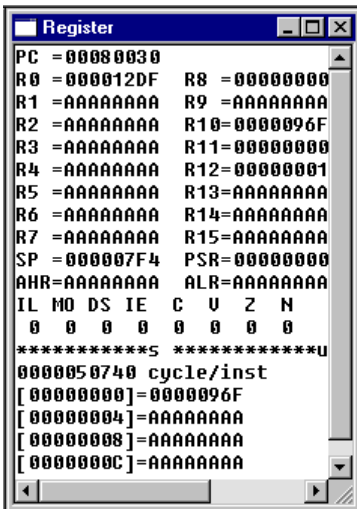
Such a perpetual loop should be halted with a forcible break.

**To break forcibly**



[Key break] button

**Step 21)** Click on the [Key break] on the tool bar.



This illustration shows that the program had a break at address 0x80030 (PC), and that it had executed 50740 cycles by that time. The counter set from address 0x0 has reached 0x96F. (The addresses [0000001] to [000000C] are for monitoring the data memory. Here, the initial settings of addresses 0x0, 0x4, 0x8 and 0xC are shown. The memory that appears to the right of "[0000000]=" holds address 0 on the right end and address 3 on the left end.)

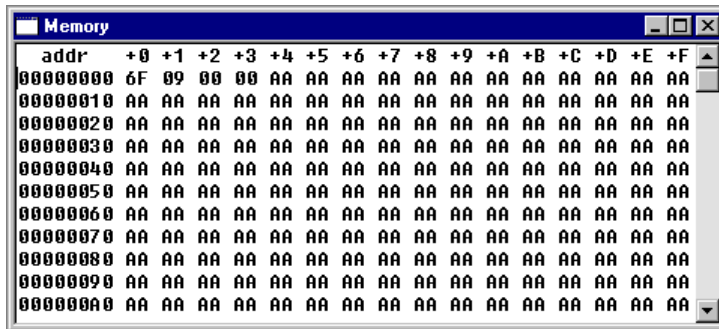
The contents of the data memory at addresses other than the monitoring addresses can also be checked in the [Memory] window.

**To open the [Memory] window**



[Window] menu

**Step 22)** Select the [Memory] command from the [Window] menu.



The [Memory] window opens and displays the contents of the memory. Display the top of the memory using the vertical scroll bar. ("AA" at addresses other than 0–3 denotes the initial setting in the RAM area.)

Data of the entire memory area may be verified by scrolling the screen vertically.

Thus far the contents of the variable `i` have been checked by the address, but it is not practical in C source level debugging. Contents of variables can also be displayed by specifying symbol names. The information can be displayed in the [Command] window and the [Symbol] window. The following explanation uses the [Symbol] window.

### To open the [Symbol] window



[Window] menu

**Step 23)** Select the [Symbol] command from the [Window] menu.



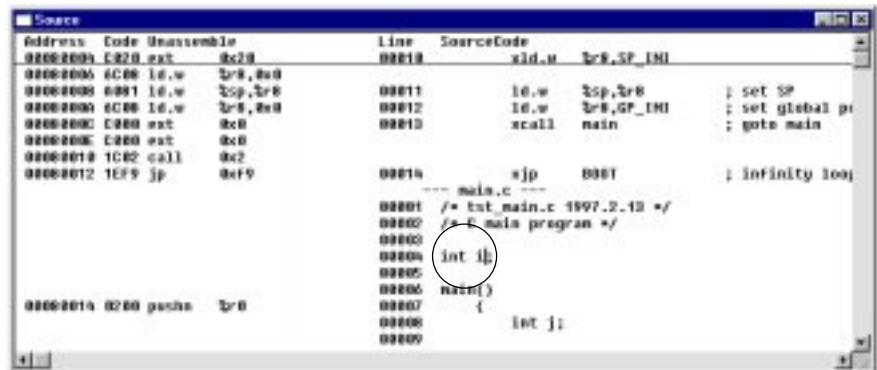
To display contents of a variable, it is necessary to register the symbol to the [Symbol] window.

### To add the symbol to be monitored to the [Symbol] window

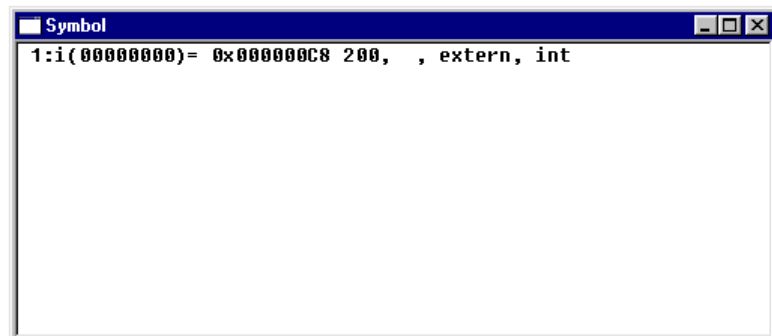


[Symbol add] button

**Step 24)** Place the cursor at the symbol name (variable `i`) displayed in the [Source] window, then click [Symbol add] on the tool bar.



The [Symbol] window displays the information of the variable `i`.



Execute the program again (Steps 20 and 21). The content of the variable `i` will be updated after breaking.



Since the working of the program cannot be observed very well during the operation described above, a break will be placed at an appropriate point.

### To set a break point



[Soft PC break]  
button

**Step 25)** Place the cursor on the line at address 0x00080030 (i++), then click [Soft PC break] on the tool bar.

A "!" mark appears at the beginning of the line at address 0x00080030, indicating that the break point has been placed here. (Another click of [Soft PC break] in this condition will clear the setting of the break.)

Once a break point has been set, execute the program once again.

**Step 26)** Click [Go] on the tool bar.

The line at address 0x00080030 is displayed with an underline, indicating the program has broken. Repeating Step 26 thereafter will demonstrate that the variable i increases by increments.

This method allows checking, to see whether the intended motion is being implemented or not. If any problem is detected in the motion, the functioning will have to be looked at more closely.

The Step and Next operations are two ways of proceeding through the program.

### To execute the Step operation



[Step] button

**Step 27)** Click [Step] on the tool bar.

The program executes the instruction underlined in the [Source] window, and the underline moves on to the instruction to be executed next. Each step is executed successively as Step 27 is repeated. If the program is error-free, the register changes its display correctly according to each step executed.

In the Step operation, all the instructions are executed on a step-by-step basis.

The Next operation is basically identical to the Step operation, except that a function, subroutine or software interrupt routine is skipped (executed as one step). This Next operation comes in handy, since a subroutine in which debugging was already completed does not need to be executed step by step.

### To execute the Next step



[Next] button

**Step 28)** Click [Next] on the tool bar.

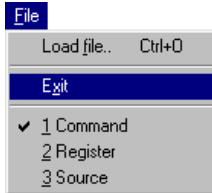
Repeat Step 28 to see the difference between the Step and Next executions in the [Source] window.

Note that a skip was made inside the function sub( ), but the variable i is updated, and the function was executed continuously.

In the preceding paragraphs, the fundamental operations of Debugger db33 have been discussed. A more sophisticated debugging may be implemented by keying in commands in the [Command] window from the keyboard. See Chapter 16, "Debugger" for more information.

The following instructions explain how to quit the Debugger db33.

### To quit the Debugger



**Step 29)** Select the [Exit] command from the [File] menu.

The window closes, and the Work Bench window returns.

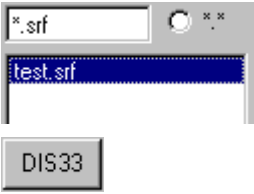
[File] menu (db33)

Besides the simulator mode used in the tutorial, the Debugger db33 supports three other debugging modes: the ICE mode that uses the In-Circuit Emulator ICE33, the Debug Monitor mode that uses the DMT33MON with the target board in which the debug monitor has been implemented, and the ICD mode that uses the In-Circuit Debugger ICD33 with the target board. Refer to Section 3.3 for the debugging method in each mode.

### 3.2.8 Creating Disassembly File

The Disassembler dis33 disassembles the srf33 object file delivered from the Linker and creates a list file that contains the C sources or assembly sources corresponding to the disassembled codes. This list shows the correspondence between the sources and object codes.

#### To create a disassembly file

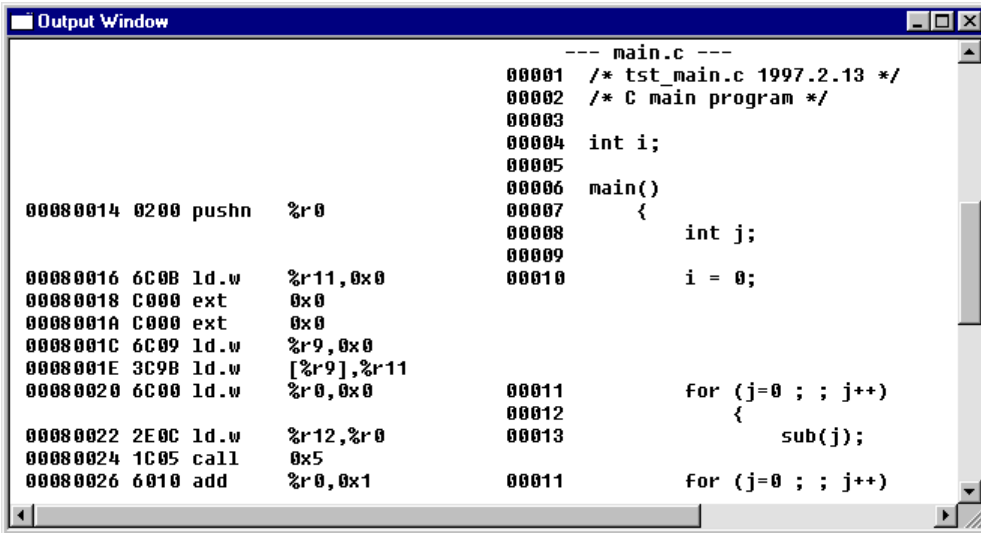


**Step 30)** Select the test.srf from the file list box in the Work Bench, then click [DIS33].

To select the file name easily, select the [\*srf] radio button.

The list file test.dis is created. Display the contents by double-clicking the file name. See how the C source was converted into mnemonic.

[DIS33] button



### 3.2.9 Creating ROM Data

Hex files are used for making the external ROM to be mounted on the target board and the mask data for the internal ROM. The Binary/HEX Converter hex33 converts the specified address range of the srf33 object file (delivered from the Linker) into a Motorola S3 format HEX file.

#### To create a HEX file



**Step 31)** Select the test.srf in the file list box, then click [HEX33].

To select the file name easily, select the [\* .srf] radio button.

The Binary/HEX Converter hex33 delivers the HEX file for the external ROM with the name test.sa\_c00000\_c0ffff.

The Binary/HEX Converter was executed using the default option settings of the Work Bench, so the HEX file contains 64KB data from address 0xc00000 to address 0xc0ffff. In the actual development, the address range must be specified according to the memory configuration of the model. It can be specified at the HEX33 option selection part in the [other options] window.

#### Creating submission mask data

When the program development for a mask ROM model has finished, the mask data for the internal ROM should be submitted to Seiko Epson. Mask data can also be created using the Binary/HEX Converter. In this case, make sure that the internal ROM address range is specified correctly and the [abs addr] check box is selected to create absolute address data.

The following setting is an example for creating 4KB of mask data within the address 0x80000 to address 0x80fff range .



The created mask data file should be submitted after renaming to one specified by Seiko Epson.

Example: c3264010.sa0 (mask data file for the E0C33264)

#### Notes on creating mask data

To prevent file copy errors, bugs in the tools, and other problems, perform a final operation check by reading the HEX files (.sa) in Motorola S3 format by the lh command. Do not use the srf33 file.

### 3.2.10 Optimization

The development procedure have been reviewed. As the final step, this section explains optimization of the code, one of the features of this package.

The gcc33 options select part of the [gcc-lk options] window has an [optimize] field with radio buttons that allow specifying the optimization level. Since the effects of code optimization cannot be confirmed with a sample program, the following shows other methods.

One method is to use a global pointer.

A global pointer is the start address of a global variable area, and a general-purpose register R8 is used exclusively for accessing this area. This helps to reduce the number of instructions necessary to access global variables. Initialization of the R8 register in the assembly source of the sample program is the processing performed to set up this global pointer.

This function is an option to the Instruction Extender ext33, and is deselected by default for the Work Bench. Therefore, make in the tutorial was not optimized by using a global pointer.

When make is performed after selecting [global pointer optimize] which is an option to the ext33, the difference in output code can be verified. The following shows the difference in the sample program where global variable i (address being mapped to location 0) is accessed.

#### When not using a global pointer

```
00080032 C000 ext    0x0                00022                i++;
00080034 C000 ext    0x0
00080036 6C09 ld.w   %r9, 0x0
00080038 309A ld.w   %r10, [%r9]
0008003A 601A add    %r10, 0x1
0008003C C000 ext    0x0
0008003E C000 ext    0x0
00080040 6C09 ld.w   %r9, 0x0
00080042 3C9A ld.w   [%r9], %r10
```

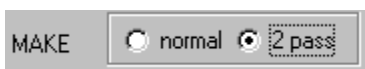
#### When using a global pointer

```
00080030 C000 ext    0x0                00022                i++;
00080032 C000 ext    0x0
00080034 308A ld.w   %r10, [%r8]
00080036 601A add    %r10, 0x1
00080038 C000 ext    0x0
0008003A C000 ext    0x0
0008003C 3C8A ld.w   [%r8], %r10
```

The above example shows that use of a global pointer made it possible to eliminate two instructions.

Another method of optimization is 2-pass make. In 2-pass make, the program modules are linked, then processed again by the Instruction Extender based on the absolute address information of the symbols determined by linkage processing. This helps to delete the unnecessary "ext" instructions used for referencing the jump address labels and symbols in external modules.

#### To perform 2-pass make



**Step 32)** Check the [2 pass] radio button in the make options select part of the [other options] window, then use the [MAKE] button to execute make.

The following shows a part of the sample file that has been optimized by 2-pass make.

**For 1-pass make**

```
00080030 C000 ext      0x0                00022          i++;
00080032 C000 ext      0x0
00080034 308A ld.w     %r10, [%r8]
00080036 601A add      %r10, 0x1
00080038 C000 ext      0x0
0008003A C000 ext      0x0
0008003C 3C8A ld.w     [%r8], %r10
```

**For 2-pass make**

```
0008002C 308A ld.w     %r10, [%r8]          00022          i++;
0008002E 601A add      %r10, 0x1
00080030 3C8A ld.w     [%r8], %r10
```

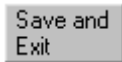
The above example shows part of a disassembly list that is created from the object file "test.srf" that was created by executing 1-pass and 2-pass make by reading it in with the Disassembler dis33. This can also be verified in the debugger window by setting mixed display mode.

In the above example, you will notice that since variable i is found to be located at address 0, the "ext" instruction used for access is deleted by 2-pass make. Specifically, this processing is performed by the ext33.

### 3.2.11 Epilogue

This tutorial explained the basic operations of the C Compiler along with the flow of the development procedure. For more information about each tool, refer to the chapters in this manual in which they are detailed.

#### To terminate the Work Bench



**Step 33)** Click [Save and Exit].

[Save and Exit] button

The wb33 terminates after saving the option setting information to the wb33.sav file.

From the next time, the wb33 will be able to start up with the current option settings by dragging wb33.sav on the wb33 exe icon. To perform this drag and drop operation, the shortcut of wb33.exe should be created on the desktop.

**Note:** The tools including the make can be invoked on the DOS prompt by entering the command or using a batch file.

If the target system has ICE33, ICD33 or DMT33MON, refer next to the operating procedure for each tool described in Section 3.3, "Debugging Environment".

### 3.3 Debugging Environment

Besides debugging in simulator mode, as shown in the previous section, db33 allows debugging programs including target system operation using the Debug Monitor (MON33), In-Circuit Emulator (ICE33) or In-Circuit Debugger (ICD33).

This section explains the outline of each debugging system and how to start debugging. Refer to each tool manual for details.

Note: Make sure that all the equipment is off before connecting or disconnecting the system.

#### 3.3.1 In-Circuit Emulator ICE33

The ICE33 is the in-circuit emulator for the E0C33 Family Model 1 microcomputers, and provides the most advanced debugging environment. The on-chip peripheral functions are implemented with the PRC33001 board. This system allows the use of almost all the db33 functions. It also allows up to 8 MB of external memory emulation using the optional memory card.

#### System configuration

Figure 3.3.1.1 shows the debugging system configuration using the ICE33.

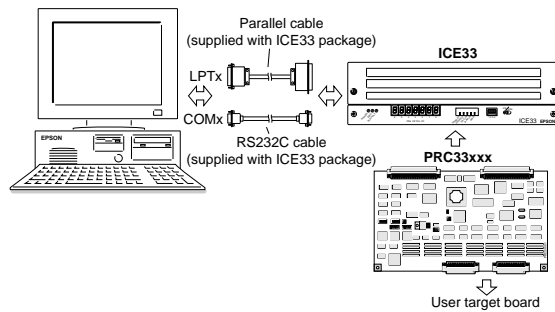


Fig. 3.3.1.1 Debugging system configuration using ICE33

#### Starting up and checking operation

Start up the Debugger in the ICE mode (select [ICE] in wb33).



Before starting up the Debugger, check the following:

- 1) Is the RS232C cable in use one of the specified types?
- 2) Is the ICE33 connected to the COM1 port on the personal computer side?  
(When using another COM port, changes have to be done on the Work Bench wb33 side.)
- 3) Is the ICE/RUN switch of the ICE33 set to ICE?
- 4) Are the ICE33 DIP switches 1, 3 and 4 set in the OPEN position and switch 2 set in the ON position (115200 bps, self-diagnostic deactivated)?
- 5) Is the PRC board correctly mounted on the ICE33?
- 6) Is the ICE33 switched on (Power LED lit)?

If the above settings are not executed correctly, "time-out" errors will result, and the Debugger db33 will fail to start up normally.

After the Debugger is started in the ICE mode, the operations should be done like the simulator mode (refer to Section 3.2.7). When the program is executed by the [Go] button, the contents of the PC, flags and monitoring data in the [Register] window are real-time updated.

## Precautions

- (1) The ICE33 emulation memory is configured according to the contents of the debugger parameter file. Therefore, the parameter file should be created correctly according to the memory configuration of the target system.
- (2) All the ICE33 functions can be used even if the ICE33 is only connected to the COM port using the RS232C cable. The parallel cable should be used to connect the ICE33 to the LPT port when high-speed file downloading is required. The following shows the typical downloading speed of the different ports (the values may vary according to the PC used and operating conditions).
 

Serial transfer:	Downloading to RAM	9KB/S
	Downloading to Flash memory	8KB/S
Parallel transfer:	Downloading to RAM	50KB/S
	Downloading to Flash memory	30KB/S
- (3) The ICE33 is shipped with the firmware Ver. 1. It can be used with the Debugger db33 in this package. However, the firmware Ver. 1 does not support writing to the flash memory on the target board (fls, fle), hardware PC break 2 (bh2, bh2) and memory copy in half word and word units (mvh, mvw). If these functions are required, update the ICE firmware using the program located in the "cc33\utility\ice33v20\" directory.
- (4) Refer to the "E0C33 Family In-Circuit Emulator (ICE33) Manual" for more information on the ICE33.

### 3.3.2 Debug Monitor MON33

The Debug Monitor MON33 is a middleware designed for E0C33 Family single-chip microcomputers. It provides program-debugging functions on the user target board or DMT33xxx boards. By connecting the board in which MON33 has been implemented to the personal computer via the DMT33MON board, the program can be debugged using the Debugger db33. This section explains how to debug the program using the DMT33004/DMT33005 board in which MON33 has been implemented as a development tool.

#### System configuration and connection

Figure 3.3.2.1 shows the debugging system configuration using the DMT33004/DMT33005 board.

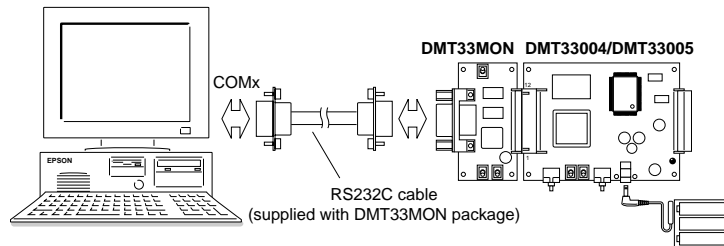


Fig. 3.3.2.1 Debugging system using DMT33004/DMT33005 board

#### Starting up and checking operation

The following sample programs are provided to check the system operation:

For DMT33004 board: `"\cc33\sample\dmt33004\led.srf"`

`"\cc33\sample\dmt33004\led2.srf"`

For DMT33005 board: `"\cc33\sample\dmt33005\led.srf"`

`"\cc33\sample\dmt33005\led2.srf"`

These programs blink the LED on the DMT board. "led.srf" and "led2.srf" are created to be able to debug in the RAM (0x600000~) and in the Flash memory (0x200000~), respectively.

For the contents of the program, refer to the source file (led.s) in the directory.

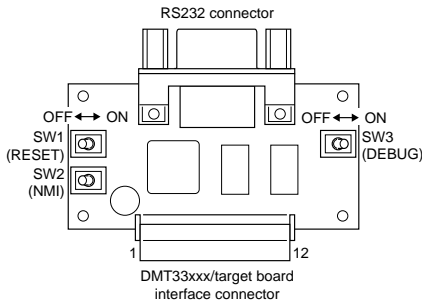
It is not necessary to execute Make when modification of the source is not needed since the executable object files ("led.srf," "led2.srf") are provided. When the source is modified, execute Make using the make file provided in the directory.



(1) Starting up the Debug Monitor

The boot routine mapped from address 0xC00000 on the DMT33004/33005 starts the debug monitor when the K63 input port is set to "0" (the [DEBUG] switch of the DMT33MON is set to ON).

Start up the debug monitor following the procedure below after connecting the target system to the personal computer.



- 1) Turn SW3[DEBUG] of the DMT33MON on.
- 2) Turn the power of the DMT33004/33005 on.
- 3) Reset the DMT33004/33005  
(DMT33MON SW1 [RESET] ON→OFF).
- 4) Turn the personal computer on and start up Windows.
- 5) Start up the debugger db33  
(start-up method is described later).

Fig. 3.3.2.2 DMT33MON board layout

Note: When the power of the DMT33004/33005 is turned on while the SW3 [DEBUG] of the DMT33MON is off, the debug monitor does not start up. The DMT33004/33005 sets TTBR at the beginning of the Flash memory (0x200000~), so the program sequence branches to the boot address. In this case, turn the SW3 [DEBUG] on and reset the DMT33004/33005 with the SW1 [RESET] to start up the debug monitor.

(2) Debugging in the RAM

The sample program for debugging in the RAM (0x600000~) of the DMT33004/33005 is "led.srf". When starting up the debugger, specify the debug command file "led.cmd" with the -c option. "led.cmd" sets the trap table address to the start address of the RAM and loads "led.srf" to the RAM.

Operating procedure for starting up the Debugger from the DOS prompt is as follows:

- 1) Start up the debug monitor as described above.
- 2) Set "\cc33\sample\dmt33004\" (or "\cc33\sample\dmt33005\") as the current directory.
- 3) Set a path to db33.exe.
- 4) Start up the debugger with the following command at the DOS prompt.

```
C:\cc33\sample\dmt33004\>db33 -mon -b 115200 -p 33104_m.par -c led.cmd
```

The debugger starts in debug monitor mode and is ready to debug "led.srf". For example, the LED on the DMT33004/33005 board will start blinking by executing the g command.

The debug monitor does not support forced break functions such as key break.

To suspend the program execution, "led.cmd" contains a command that sets a breakpoint at the label located in the NMI routine of "led.srf". When the SW2 of the DMT33MON is turned on, a NMI is generated and it suspends the program execution.

(3) Debugging in the Flash memory

The sample program for debugging in the Flash memory (0x200000~) of the DMT33004/33005 is "led2.srf". To write the sample program to the Flash memory, first load the Flash erase/write routine "am29f800.srf". Then initialize the Flash memory functions using the fls and fle commands and load the sample program into the Flash memory using the lf command. Refer to the sample debug command file "led2.cmd" for executing procedure.

When starting up the debugger, specify the debug command file "led2.cmd" with the -c option. "led2.cmd" contains debug commands for loading the Flash erase/write routine, setting the trap table address and loading "led2.srf" to the Flash memory.

Operating procedure for starting up the Debugger from the DOS prompt is as follows:

- 1) Start up the debug monitor as described above.
- 2) Set "\\cc33\\sample\\dmt33004\" (or "\\cc33\\sample\\dmt33005\") as the current directory.
- 3) Set a path to db33.exe.
- 4) Start up the debugger with the following command at the DOS prompt.

```
C:\cc33\sample\dmt33004\>db33 -mon -b 115200 -p 33104_m.par -c led2.cmd
```

The debugger starts in debug monitor mode and is ready to debug "led2.srf". For example, the LED on the DMT33004/33005 board will start blinking by executing the g command.

The debug monitor does not support forced break functions such as key break.

When the SW2 of the DMT33MON is turned on, a NMI is generated and it suspends the program execution forcibly.

After writing the program to the Flash memory, it can be executed by the DMT33004/33005 alone.

- 1) Terminate the Debugger.
  - 2) Turn the system power off and then disconnect the RS232C cable.
  - 3) Turn SW3 [DEBUG] of the DMT33MON off and then turn the DMT33004/33005 on.
- The "led2.srf" program will be executed in the Flash memory and the LED will start blinking.

#### (4) Executing from wb33

- 1) Start up the debug monitor as described above.
- 2) Start up wb33 and then select the parameter file "\\cc33\\sample\\dmt33004\\33104\_m.par" (or "\\cc33\\sample\\dmt33005\\33104\_m.par") on the execution window.
- 3) Select DB33 options on the [Other options] windows.  
MON, 115200 bps, command file "led.cmd" or "led2.cmd"



- 4) Start up the Debugger using the [DB33] button.
- This procedure starts debugging the same as in items (2) and (3) above.

## Precautions

When debugging the user program on the DMT33004/33005 board, observe the precautions described below.

- (1) The debug monitor on the DMT33004 has been implemented by linking with the "mon33ch1.lib". Therefore, the built-in serial interface Ch.1 cannot be used from the user program.  
The debug monitor on the DMT33005 has been implemented by linking with the "mon33ch0.lib". Therefore, the built-in serial interface Ch.0 cannot be used from the user program.
- (2) Forced break functions cannot be used in the Debug Monitor. A forced break function can be realized by setting a hardware PC break point at a label position in the NMI or key input interrupt routine of the target program.  
Furthermore, other debugging functions are also restricted. Refer to Chapter 16, "Debugger", for the functions and commands that are supported by the Debug Monitor.
- (3) The downloading speed is approx. 8KB/S for RAM and approx. 7KB/S for Flash memory. However, it varies according to the PC used and operating conditions.
- (4) The program to be debugged should be created so that it can be loaded and executed in the free area of the RAM or the Flash memory on the DMT33004/33005. The program load address must be specified when linking since it cannot be specified by the Debugger.  
The MON33 uses 0 to 0x2F of the internal RAM and 0x6FF640 to 0x6FFFFFF in the external SRAM. Be aware that the MON33 will not be able to work if the area above is rewritten. Furthermore, this precaution applies when rewriting the memory using a memory operation command.

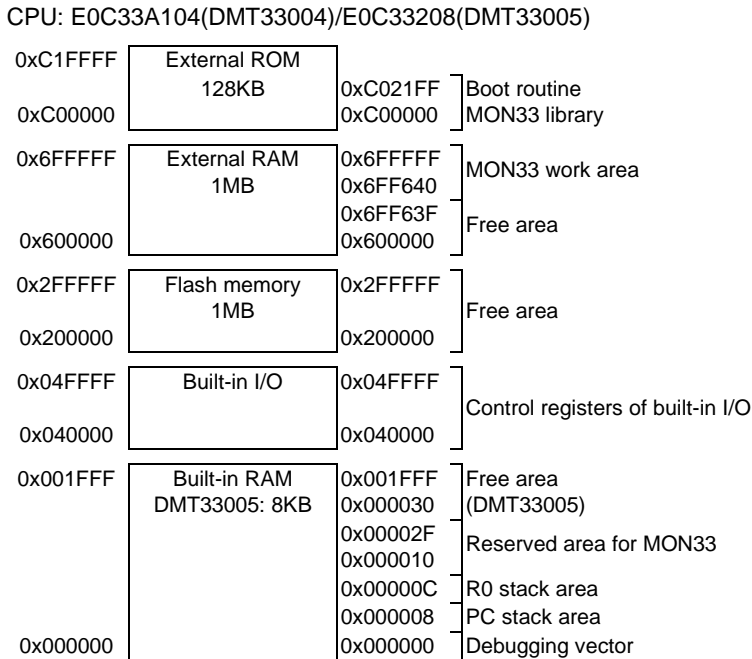


Fig. 3.3.2.3 DMT33004/33005 memory map

(5) Refer to the "E0C33 Family MON33 Debug Monitor Manual" for more information on the Debug Monitor.

### 3.3.3 In-Circuit Debugger ICD33

The In-Circuit Debugger ICD33 is a development tool that controls the E0C33 on-chip debugging function according to the command sent from the Debugger db33. It provides a trace function as well as the debugging function the same as the Debug Monitor. This section explains how to debug the program using the ICD33 with the DMT33005 board as a development tool.

#### System configuration and connection

Figure 3.3.3.1 shows the debugging system configuration using the ICD33 and the DMT33005 board.

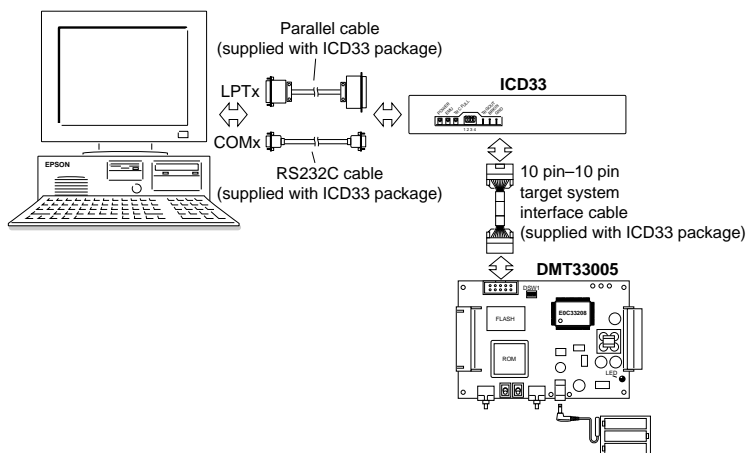


Fig. 3.3.3.1 Debugging system using ICD33 and DMT33005

Note: The ICD33 cannot be used with the Debug Monitor. Do not connect DMT33MON to the DMT33005 board. To use the DMT33005 board with DMT33MON, be sure to turn the [DEBUG] switch (SW3) of the DMT33MON off.

## Starting up and checking operation

The following sample programs are provided to check the system operation:

```
"\cc33\sample\dmt33005\led.srf"
```

```
"\cc33\sample\dmt33005\led2.srf"
```

These programs blink the LED on the DMT33005 board. "led.srf" and "led2.srf" are created to be able to debug in the RAM (0x600000~) and in the Flash memory (0x200000~), respectively.

For the contents of the program, refer to the source file (led.s) in the directory.

It is not necessary to execute Make when modification of the source is not needed since the executable object files ("led.srf," "led2.srf") are provided. When the source is modified, execute Make using the make file provided in the directory.

### (1) Starting up the system

Start up the system following the procedure below after connecting the ICD33 and DMT33005 to the personal computer.

- 1) Set all the DIP switches of the ICD33 to OPEN (upper position).
- 2) Turn the DMT33005 on.
- 3) Turn the ICD33 on.
- 4) Turn the personal computer on and start up Windows.
- 5) Start up the debugger db33 (start-up method is described later).

### (2) Debugging in the RAM

The sample program for debugging in the RAM (0x600000~) of the DMT33005 is "led.srf". When starting up the debugger, specify the debug command file "led.cmd" with the -c option. "led.cmd" sets the trap table address to the start address of the RAM and loads "led.srf" to the RAM.

Operating procedure for starting up the Debugger from the DOS prompt is as follows:

- 1) Start up the system as described above.
- 2) Set "\cc33\sample\dmt33005\" as the current directory.
- 3) Set a path to db33.exe.
- 4) Start up the debugger with the following command at the DOS prompt.

```
C:\cc33\sample\dmt33005\>db33 -icd -b 115200 -p 33104_m.par -c led.cmd
```

The debugger starts in ICD mode and is ready to debug "led.srf". For example, the LED on the DMT33005 board will start blinking by executing the g command.

The ICD33 supports the key break function. The program execution can be suspended using the [Key break] button of the Debugger. Also the trace function is available. Refer to Chapter 16, "Debugger", for tracing.

### (3) Debugging in the Flash memory

The sample program for debugging in the Flash memory (0x200000~) of the DMT33005 is "led2.srf".

To write the sample program to the Flash memory, first load the Flash erase/write routine "am29f800.srf". Then initialize the Flash memory functions using the fls and fle commands and load the sample program into the Flash memory using the lf command. Refer to the sample debug command file "led2.cmd" for executing procedure.

When starting up the debugger, specify the debug command file "led2.cmd" with the -c option. "led2.cmd" contains debug commands for loading the Flash erase/write routine, setting the trap table address and loading "led2.srf" to the Flash memory.

Operating procedure for starting up the Debugger from the DOS prompt is as follows:

- 1) Start up the system as described above.
- 2) Set "\cc33\sample\dmt33005\" as the current directory.
- 3) Set a path to db33.exe.
- 4) Start up the debugger with the following command at the DOS prompt.

```
C:\cc33\sample\dmt33005\>db33 -icd -b 115200 -p 33104_m.par -c led2.cmd
```

(4) Executing from wb33

- 1) Start up the system as described above.
- 2) Start up wb33 and then select the parameter file "\cc33\sample\dmt33005\33104\_m.par" on the execution window.
- 3) Select DB33 options on the [Other options] windows.  
ICD, 115200 bps, command file "led.cmd" or "led2.cmd"



- 4) Start up the Debugger using the [DB33] button.  
This procedure starts debugging the same as in items (2) and (3) above.

**Precautions**

When debugging the program using the ICD33 and DMT33005 board, observe the precautions described below.

- (1) The program to be debugged should be created so that it can be loaded and executed in the free area of the RAM or the Flash memory on the DMT33005. The program load address must be specified when linking since it cannot be specified by the Debugger. See Figure 3.3.2.3 for the DMT33005 memory map.
- (2) The debugging functions are restricted compared to the ICE33. Refer to Chapter 16, "Debugger", for the functions and commands that are supported by the ICD33.
- (3) All the ICD33 functions can be used even if the ICD33 is only connected to the COM port using the RS232C cable. The parallel cable should be used to connect the ICD33 to the LPT port when high-speed file downloading is required. The following shows the typical downloading speed of the different ports (the values may vary according to the PC used and operating conditions).
 

Serial transfer:	Downloading to RAM	8KB/S
	Downloading to Flash memory	7KB/S
Parallel transfer:	Downloading to RAM	30KB/S
	Downloading to Flash memory	20KB/S
- (4) Refer to the "E0C33 Family In-Circuit Debugger (ICD33) Manual " for more information on the ICD33.

## 3.4 Relationship between Program Structure and Memory

This section briefly explains the concept of section management applied to the creation and linkage of source files. Although it is not specifically have to been concerned about sections in the C source, the assembly source requires that sections be explicitly be defined so that they can be created and linked.

In addition to programs to control the CPU and peripheral circuits, the source file contains descriptions of data such as font data which are always fixed and do not require initialization, symbols for the variables placed in RAM, and I/O memory control registers. Data and symbols that take on different attributes like these finally need to be relocated into the corresponding physical memory locations by the Linker. For example, programs are relocated into a program ROM area, fixed data are relocated into a data ROM area. For this reason, the object code is designed to be classified into sections by attribute.

The following three types of sections exist:

- |                        |   |
|------------------------|---|
| <b>1. CODE section</b> | Block for programs and fixed data that have initial values                    |
| <b>2. DATA section</b> | Block for data that have initial values and can be accessed for read or write |
| <b>3. BSS section</b>  | Block that is mapped into RAM   |

### For assembly source

For the assembly source, use the following assembler pseudo-instructions to specify a section:

<b>.code</b> pseudo-instruction	Beginning of a CODE section
<b>.data</b> pseudo-instruction	Beginning of a DATA section
<b>.comm/.lcomm</b> pseudo-instructions	Symbol definition to a BSS section and area allocation

The following shows the method of specification (see Chapter 11 for details):

- Before describing the program and fixed data to be written to the ROM, declare the beginning of a CODE section by using the `.code` pseudo-instruction. The source code following this declaration is assembled as the object of a CODE section. If no section is defined, the Assembler assumes a CODE section from the beginning of the file.
- Before setting RAM data that have initial values, declare the beginning of a DATA section by using the `.data` pseudo-instruction. The source code following this declaration is assembled as the object of a DATA section. However, the initial values in the DATA section have to be copied to the RAM by program.
- If the program requires to secure a variable or work area in the RAM and reference its address with a symbol, allocate this area and define the symbol by using the `.lcomm` pseudo-instruction. The Assembler allocates a specified area in the BSS section. This area is mapped in the RAM or I/O area, with no object code created there. Symbol information enabling multiple modules to reference this area is created as a BSS section.

For relocatable assembly sources (including one that is created by compiling a C source), sections of the same attribute are located together as one continuous section. Consequently, the assembled module becomes an object that has one CODE section, one DATA section, and one BSS section. (Even undeclared sections are created as those that do not have any actual data.)

For an assembly source where absolute addresses are specified, sections of the same type cannot be put together into one section. In this case, therefore, as many sections as specified separately in the source are created.

### For C source

For C sources, there is no need to specify sections in the source because sections are declared by the C Compiler. After the source is compiled, all instructions are located in the CODE section. Data is located in each corresponding section according to its attribute as follows:

Variables without an initial value (e.g., <code>int i;</code> ):	BSS section
Variables with an initial value (e.g., <code>int i=1;</code> ):	DATA section
Constants (e.g., <code>const int i=1;</code> ):	CODE section

The following shows the section definition of the sample program used in the tutorial as a simple example. Since the assembly source program "boot.s" consists of only a program code, only the .code pseudo-instruction is used.

```
<boot.s>
: boot.s 1997.2.13
: boot program

#define SP_INI 0x0800      ; sp is in end of 2KB internal RAM
#define GP_INI 0x0000      ; global pointer %r8 is 0x0

        .code
        .word BOOT        ; BOOT VECTOR
BOOT:
        xld.w   %r8, SP_INI
        ld.w    %sp, %r8   ; set SP
        ld.w    %r8, GP_INI ; set global pointer
        xcall   main       ; goto main
        xjp    BOOT        ; infinity loop
```

(Program explanation)

Boot processing is performed to initialize the stack and global pointers, and call the main function.

Do not use this program in actual applications because the actual applications require setting up the trap processing vector, etc.

```
<main.c>
/* main.c 1997.2.13 */
/* C main program */

int i;

main()
{
    int j;

    i = 0;
    for (j=0 ; ; j++)
    {
        sub(j);
    }
}

sub(k)
{
    int k;
    {
        if (k & 0x1)
        {
            i++;
        }
    }
}
}
```

(Program explanation)

The main() function is cleared global variable i to 0 because it is used as a counter. Then an endless loop is created by local variable j, and the sub() function is called repeatedly by using j as the argument. The sub() function increases global variable i by one every two calls.

Taking a look at the C Compiler output "main.ps" you will find that the CODE and the BSS sections are defined by the Compiler.

<main.ps>

```

.file      "main.c"

; GNU C 2.7.2 [AL 1.1, MM 40] RISC NEWS-OS compiled by CC
; Cc1 defaults:
; -mmemcpy
; Cc1 arguments (-G value = 0, Cpu = 3000, ISA = 1):
; -quiet -dumpbase -g -0 -o

gcc2_compiled.:
__gnu_compiled_c:
    .code
    .align      1
    .def        main,      val      main,      scl      2,      type      0x24,      endef
    .global     main
    :
    :
    .comm       i 4
    .endfile

```

The symbol of global variable *i* is defined, and a 4-byte area is located in the BSS section by the ".comm" pseudo-instruction. Since variables *j* and *k* are local variables, they are allocated to general-purpose registers and stacks. No BSS section is used for these variables.

When these modules are linked by make in the tutorial, separate CODE sections are combined into one section.

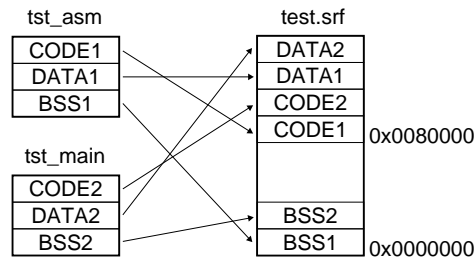


Fig. 3.4.1 Section allocation after linkage

During a linking, each file and each section can be address-specified so that they correspond to the actual memory configuration. For details, refer to Section 12.5, "Linker Commands", and Section 12.6, "Locating Sections".



## CHAPTER 3: SOFTWARE DEVELOPMENT PROCEDURES

Take a look at the link map file generated by the lk33 and the disassembly list created from the linked object file "test.srf" by the dis33. They will show how the sections are allocated after linkage.

### Link map file <test.map>

#### Code Section mapping

Address	Vaddress	Size	File	ID	Attr
00080000	-----	00000014	boot.o	0	REL
00080014	-----	00000034	main.o	0	REL

#### Data Section mapping

Address	Vaddress	Size	File	ID	Attr
00080048	-----	00000000	boot.o	1	REL
00080048	-----	00000000	main.o	1	REL

#### Bss Section mapping

Address	Vaddress	Size	File	ID	Attr
00000000	-----	00000000	boot.o	2	REL
00000000	-----	00000004	main.o	2	REL

The link map file shows the relationship between the sections in each file (File) and the located addresses (Address).

### Disassembly list file <test.dis>

\*\*\*\* Disassemble code and source code \*\*\*\*

Addr	Code	Unassemble	Line	Source
00080000	0004	***		
00080002	0008	***		
				--- boot.s ---
			00001	: boot.s 1997.2.13
			00002	: boot program
			00003	
			00004	#define SP_INI 0x0800 : sp is in end of 2KB internal RAM
			00005	#define GP_INI 0x0000 : global pointer %r8 is 0x
			00006	
			00007	.code
			00008	.word BOOT : BOOT VECTOR
			00009	BOOT:
00080004	C020	ext 0x20	00010	xld.w %r8, SP_INI
00080006	6C08	ld.w %r8, 0x0		
00080008	A081	ld.w %sp, %r8	00011	ld.w %sp, %r8 : set SP
0008000A	6C08	ld.w %r8, 0x0	00012	ld.w %r8, GP_INI : set global pointer
0008000C	C000	ext 0x0	00013	xcall main : goto main
0008000E	C000	ext 0x0		
00080010	1C02	call 0x2		
00080012	1EF9	jp 0xf9	00014	xjp BOOT : infinity loop
				--- main.c ---
			00001	/* tst_main.c 1997.2.13 */
			00002	/* C main program */
			00003	
			00004	int i;
			00005	
			00006	main()
00080014	0200	pushn %r0	00007	{
			00008	int j;
			00009	
00080016	6C0B	ld.w %r11, 0x0	00010	i = 0;
00080018	C000	ext 0x0		
0008001A	C000	ext 0x0		
0008001C	6C09	ld.w %r9, 0x0		
0008001E	3C9B	ld.w [%r9], %r11		
00080020	6C00	ld.w %r0, 0x0	00011	for (j=0 ; ; j++)
			00012	{
00080022	2E0C	ld.w %r12, %r0	00013	sub(j);
00080024	1C05	call 0x5		
00080026	6010	add %r0, 0x1	00011	for (j=0 ; ; j++)
00080028	1EFD	jp 0xfd		
			00014	}
0008002A	0240	popn %r0	00015	}
0008002C	0640	ret		
			00016	
			00017	sub(k)
			00018	int k;
			00019	{
0008002E	701C	and %r12, 0x1	00020	if (k & 0x1)
00080030	180A	jreq 0xa		

```

00080032 C000 ext 0x0          00021      {
00080034 C000 ext 0x0          00022      i++;
00080036 6C09 ld.w  %r9, 0x0
00080038 309A ld.w  %r10, [%r9]
0008003A 601A add  %r10, 0x1
0008003C C000 ext 0x0
0008003E C000 ext 0x0
00080040 6C09 ld.w  %r9, 0x0
00080042 3C9A ld.w  [%r9], %r10

00080044 0640 ret          00023      }
00080046 0000 nop          00024      }

```

The above is just a quick review of the sections. For more information, refer to the chapters where the Assembler and Linker are discussed.

## Chapter 4 Source Files

This chapter explains the rules and grammar involved with the creation of source files.

### 4.1 File Format and File Name

---

Source files should be created on a general-use word processor or editor.

**File format** Save data in a standard text file.

**File name** C source file <file name>.c

Assembly source file <file name>.s

Specify the <file name> with not more than 32 alphanumeric characters shown as follows:  
a-z, A-Z, 0-9 and \_

This rule applies to file names for all the E0C33 tools.

Make sure the extension of the C source file is ".c" (small letter can only be used). If any other extension is used, the file cannot be input to the C Compiler gcc33.

**Directory name** Only alphanumeric characters can be used for directory names just as for file names. Do not use spaces or other symbols. Up to 64 characters can be used for a path name including directory and file names.

**Number of lines and number of characters**

The following shows the number of lines and the number of characters per line that can be accepted in one C source file and one assembly source file.

Number of lines Max. 30,000 lines

Number of characters Max. 100 characters per line

**Tab setting** Place a tab stop every 8 characters. Mixed processing by the Disassembler dis33 or source display/mixed display with the Debugger db33 of a source set at a tab interval other than of 8 characters will result in a displaced output of the source part.

**EOF** Make sure that each statement starts on a new line and that EOF is entered after line feed (so that EOF will stand independent at the file end).

## 4.2 Grammar of C Source

The C Compiler gcc33 included in this package is the GNU C Compiler (ver. 2.7.2) under ANSI C standards. Since everything except the asm function of this compiler conforms to standard specifications, make sure C sources are created according to ANSI C standards. If you want information about the syntax, please refer to ANSI C textbooks generally available on the market.

### 4.2.1 Data Type

The gcc33 supports all data types under ANSI C. The size of each data type (in bytes) and the effective range of values that can be expressed are listed in Table 4.2.1.1.

Table 4.2.1.1 Data type and size

Data type	Size	Effective range of a number
char	1	-128 to 127
unsigned char	1	0 to 255
short	2	-32768 to 32767
unsigned short	2	0 to 65535
int	4	-2147483648 to 2147483647
unsigned int	4	0 to 4294967295
long	4	-2147483648 to 2147483647
unsigned long	4	0 to 4294967295
pointer	4	0 to 4294967295
float	4	1.175e-38 to 3.403e+38 (normalized number)
double	8	2.225e-308 to 1.798e+308 (normalized number)

The float and double types conform to the IEEE standard format.

### 4.2.2 Library Functions and Header Files

This package contains an ANSI standard library and an emulation library for calculating floating-point numbers and the remainders of divided integral numbers.

The header files in the "include" directory contain library function declarations and macro definitions. When using a library function, include the header file that contains its declaration by using the "#include" instruction.

The table below shows the relationship between the types of library files and the header files.

Table 4.2.2.1 List of library files and functions

#### ANSI standard library

File name	Functions/macros	Corresponding header file
io.lib	tmpfile*, tmpnam*, remove*, fopen*, freopen, fclose*, setbuf*, setvbuf*, fflush*, clearerr*, feof*, ferror*, perror, fseek*, fgetpos*, fsetpos*, ftell*, rewind*, getchar, fgetc, getc, gets, fgets, fscanf, scanf, sscanf, fread, putchar, fputc, putc, puts, fputs, ungetc, fprintf, printf, sprintf, vfprintf, vprintf, vsprintf, fwrite	stdio.h
	abort, exit, atexit*, getenv*, system*, malloc, calloc, realloc, free, atoi, atol, atof, strtol, strtoul, strtod, abs, labs, div, ldiv, rand, srand, bsearch, qsort	stdlib.h
	time, difftime*, clock*, mktime, localtime*, gmtime, asctime*, ctime*	time.h
math.lib	acos, asin, atan, atan2, ceil, cos, cosh, exp, fabs, floor, fmod, frexp, ldexp, log, log10, modf, pow, sin, sinh, sqrt, tan, tanh	math.h, errno.h, float.h, limits.h
string.lib	memchr, memmove, strchr, strcspn, strncat, strpbrk, strstr, memcmp, memset, strcmp, strerror, strncmp, strrchr, strtok, memcpy, strcat, strcpy, strlen, strncpy, strspn	string.h
ctype.lib	isalnum, iscntrl, isgraph, isprint, isspace, isxdigit, toupper, isalpha, isdigit, islower, ispunct, isupper, tolower	ctype.h
-	va_start, va_arg, va_end	stdarg.h

The functions marked with an asterisk (\*) are dummy functions.

#### Emulation library

File name	Functions
fp.lib	adddf3, subdf3, muldf3, divdf3, negdf2, addds3, subds3, mulds3, divds3, negds2, fixunsdfsi, fixdfsi, floatsidf, fixunssfsi, fixfsi, floatsisf, truncdfsf2, extendsfdf2, fcmpd, fcmps
idiv.lib	divsi3, udivsi3, modsi3, umodsi3

For details about the functions included in the libraries, refer to Chapter 7, "Emulation Library", and Chapter 8, "ANSI Library".

When using a library function, be sure to specify the library file that contains the function used by using a linker command when linking. The linker extracts only the necessary object modules from the specified library file as it links them.

### 4.2.3 In-line Assemble

The gcc33 supports in-line assembly, so the asm statement can be used. As a result, the word "asm" is reserved for system use.

Format: `asm("<character string>");`

Example 1: `/* HALT mode */  
asm("halt");`

Example 2: `/* Trap Table*/  
asm(".word BOOT  
.space 8  
.word ZERO_DIV  
.space 4  
.word ADDR_ERR  
.word NMI  
.space 32  
.word INT1  
.word INT2");`

Note: Up to 100 characters can be included in one line.

Example 3: `#define SP_INI 0x0800  
#define GP_INI 0x0000  
BOOT() {  
asm("x|d.w %r8, SP_INI");  
asm("|d.w %sp, %r8"); /* set SP */  
asm("|d.w %r8, GP_INI"); /* set global pointer */  
:  
}`

For details on how to write an assembly source, refer to Section 4.3, "Grammar of Assembly Source". Note that although the extended instructions that can be processed by the Instruction Extender ext33 and assembler pseudo-instructions (not including those used for absolute assembly) can be used in the assembly source, the functions provided by the Preprocessor pp33 cannot be used in the assembly source.

## 4.3 Grammar of Assembly Source

### 4.3.1 Statements

Each individual instruction or definition of an assembly source is called a statement. The basic composition of a statement is as follows:

#### Syntax pattern

1	Mnemonic	(Operand)	(;Comment)
2	Assembler pseudo-instruction	(Parameter)	(;Comment)
3	Label:		(;Comment)
4	;Comment		
5	Extended instruction	Operand	(;Comment)
6	Preprocessor pseudo-instruction	(Parameter)	(;Comment)

Example: <Statement>	<Syntax Pattern>
; boot.s 1997.2.13	.. 4
; boot program	.. 4
#define SP_INI 0x0800 ; sp is in end of 2KB internal RAM	.. 6
#define GP_INI 0x0000 ; global pointer %r8 is 0x0	.. 6
.code	.. 2
.word BOOT ; BOOT VECTOR	.. 2
BOOT:	.. 3
xld.w %r8, SP_INI	.. 5
ld.w %sp, %r8 ; set SP	.. 1
ld.w %r8, GP_INI ; set global pointer	.. 1
xcall main ; goto main	.. 5
xjp BOOT ; infinity loop	.. 5
:	:
:	:
:	:

The example given above is an ordinary source description method. For increased visibility, the elements composing each statement are aligned with tabs and spaces.

\* 5 is the function of the Instruction Extender ext33, 6 is the function of the Preprocessor pp33, and not statements that can be processed by the Assembler as33.

#### Restrictions

- Only one statement can be described in one line. A description containing more than two instructions in one line or a mixture of label and instruction will result in an error. However, comments may be described in the same line with an instruction or label.

```
Examples: ;OK
          BOOT:
                ld.w    %r0,%r1
          ;Error
          BOOT: ld.w    %r0,%r1
```

- One statement cannot be described in more than one line. A statement not complete in one line will result in an error.

```
Examples: ;OK
                .byte    0x0,0x1,0x2,0x3
                .byte    0xa,0xb,0xc,0xd
          ;Error
                .byte    0x0,0x1,0x2,0x3,
                0xa,0xb,0xc,0xd
```

- The maximum describable number of characters in one line is 100 (ASCII characters).
- The usable characters are limited to ASCII characters (alphanumeric symbols), except for use in comments. Also, the usable symbols have certain limitations (details below). Comments can be described using other characters than ASCII characters.

## (1) Instructions (Mnemonics and Operands)

An instruction to the CPU is generally composed of [mnemonic] + [operand]. Some instructions do not contain an operand.

### General notation forms of instructions

General forms: <Mnemonic>  
 <Mnemonic> tab or space <Operand>  
 <Mnemonic> tab or space <Operand 1>, <Operand 2>

Examples: nop  
 call SUB1  
 ld.w %r0,0x4

There is no restriction as to where the description of a mnemonic may begin in a line. A tab or space preceding a mnemonic is ignored. Generally, mnemonics are justified left by tab setting.

An instruction containing an operand needs to be broken with one or more tabs or spaces between the mnemonic and the operand. If there are plural operands, the operands are separated from each other with one comma (.). Space between operands is ignored.

The elements of operands will be described further below.

### Types of mnemonics

The following 61 types of mnemonics can be used in the E0C33 Family:

adc add and bclr bnot brk bset btst call cmp div0s div0u div1 div2s div3s ext  
 halt int jp jreq/jreq.d jrge/jrge.d jrgt/jrgt.d jrle/jrle.d jrle/jrle.d jrne/jrne.d jruge/jruge.d  
 jrugt/jrugt.d jrule/jrule.d rult/jrult.d ld.b ld.h ld.ub ld.uh ld.w mac mirror mlt.h mlt.w  
 mltu.h mltu.w nop not or popn pushn ret/ret.d retd reti rl rr sbc scan0 scan1 sla  
 sll slp sra srl sub swap xor

Refer to the "E0C33000 Core CPU Manual" for details of each instruction.

### Restrictions on characters

Mnemonics can be written in uppercase (A–Z) characters, lowercase (a–z) characters, or both. For example, "ld.w", "LD.W", and "Ld.w" are all accepted as "ld.w" instructions.

For purposes of discrimination from symbols, this manual uses lowercase characters.

More will be said about operands later.

## (2) Assembler Pseudo-Instructions

The following 20 types of pseudo-instructions are available for the Assembler as33:

Pseudo-Instruction	Function
.abs	Specifies absolute assembling.
.org <address>	Sets an absolute address in a code (*).
.code	Declares the CODE section
.data	Declares the DATA section
.word <data 1>[,<data 2>...,<data n>]	Defines word data in the CODE/DATA section.
.half <data 1>[,<data 2>...,<data n>]	Defines half word data in the CODE/DATA section.
.byte <data 1>[,<data 2>...,<data n>]	Defines byte data in the CODE/DATA section.
.ascii <string>	Defines an ASCII character string in the CODE/DATA section.
.space <length>	Defines a blank area (0x0) in the CODE/DATA section.
.align <value>	Moves to an address boundary.
.comm <symbol>,<length>	Secures a global area in the BSS section.
.lcom <symbol>,<length>	Secures a local area in the BSS section.
.global <symbol>	Declares a global symbol.
.set <symbol>,<address>	Defines an absolute address for a symbol (*).
.list	Controls assembly list output.
.nolist	Controls assembly list output.
.file <strings>	Debugging information.
.endfile	Debugging information.
.loc <value>	Debugging information.
.def <symbol>, ... .endef	Debugging information.

(\*: Dedicated absolute assembling)

Each instruction begins with a period (.).

Examples:     .data  
              .align     2  
              .word     1,2,3,4

For details on the notation of each pseudo-instruction and its functionality, refer to Section 11.8 "Assembler Pseudo-Instructions".

## (3) Labels

A label is an identifier designed to refer to an arbitrary address in the program. You can refer to a branch destination of a program or an address in the CODE/DATA section by using a symbol defined as a label.

### Definition of a label

A symbol described in the following format is regarded as a label.

<Symbol>:

Preceding spaces and tabs are ignored. It is a general practice to describe from the top of a line.

A defined symbol denotes the address of a described location.

An actual address value will be determined in the linking process.

### Restrictions

- A label occupies one line of a source program. An instruction described in the same line will result in an error. However, comments may be described in the same line with a label.
- The maximum number of characters of a label is 32 (not including colons).



- Only the following characters can be used:

A-Z a-z \_ 0-9

A label cannot begin with a numeral. Uppercase and lowercase are discriminated.

Examples:     ;OK                             ;Error  
               FOO:                            llabel:  
               \_Abcd:                         0\_ABC:  
               L1:

**(4) Comments**

Comments are used to describe the meaning of a series of routines or each statement. Comments cannot comprise part of coding.

**Definition of comment**

A character string beginning with a semicolon (;) and ending with a line feed is interpreted as a comment.

Not only ASCII characters, but also other non-ASCII characters can be used to describe a comment.

It can be described with a label or instruction in one line.

Examples:       ; This line is a comment line.  
                   LABEL:                             ;This is the comment for LABEL.  
                           ld                         ;a,%b     ;This is the comment for the instruction on the left.

**Restrictions**

- A comment is limited in length to 100 characters, including: a semicolon (;); spaces before, after and inside the comment; and a return/line feed code.
- When a comment extends to several lines, each line must begin with a semicolon.

Examples:

   ;These are  
       comment lines.     The second line will not be regarded as a comment. An error will result.  
  
    ;These are  
    ;   comment lines.     Both lines will be regarded as comments.

**(5) Blank Lines**

This assembler also allows a blank line containing only a return/line feed code. It need not be made into a comment line using a semicolon; for example, when used as a break in a series of routines.

## 4.3.2 Notations of Operands

This section explains the notations for the register names, symbols, and constants that are used in the operands of instructions.

### (1) Register Names

The names of the internal registers of the E0C33000 Core CPU all contain a percentage symbol (%). Register names may be written in either uppercase or lowercase letters.

<u>General-purpose register (%rd, %rs, %rb)</u>	<u>Notation</u>
General-purpose register R0–R15	%r0–%r15 or %R0–%R15
<u>Special register (%sd, %ss)</u>	<u>Notation</u>
Processor status register PSR	%psr or %PSR
Stack pointer SP	%sp or %SP
Arithmetic operation low register ALR	%alr or %ALR
Arithmetic operation high register AHR	%ahr or %AHR

Register names placed in brackets ([ ]) for indirect addressing must include the % symbol.

Examples: [%r8] [%r1]+ [%sp+imm6]

Note: A register name not containing % will be regarded as a symbol.

Conversely, all notations beginning with % will be regarded as registers, and will give rise to an error if it is not a register name.

### (2) Numerical Notations

The assembler supports three kinds of numerical notations: decimal, hexadecimal and binary.

#### Decimal notations of values

Notations represented with 0–9 only will be regarded as decimal numbers. To specify a negative value, put a minus sign (-) before the value.

Examples: 1 255 -3

Characters other than 0–9 and the sign (-) cannot be used.

#### Hexadecimal notations of values

To specify a hexadecimal number, place "0x" before the value.

Examples: 0x1a 0xff00

"0x" cannot be followed by characters other than 0–9, a–f, and A–F.

Note: Only the lowercase "x" can be used. "0X" will result in an error.

#### Binary notations of values

To specify a binary number, place "0b" before the value.

Examples: 0b1001 0b01001100

"0b" cannot be followed by characters other than 0 or 1.

Note: Only a lowercase "b" can be used. "0B" will result in an error.

**Specified ranges of values**

The size (specified range) of immediate data varies with each instruction.  
 The specifiable ranges of different immediate data are given below.

Table 4.3.2.1 Types of immediate data and their specifiable ranges

Symbol	Type	Decimal	Hexadecimal	Binary
imm2	2-bit immediate data	0–3	0x0–0x3	0b0–0b11
imm3	3-bit immediate data	0–7	0x0–0x7	0b0–0b111
imm4	4-bit immediate data	0–15	0x0–0xf	0b0–0b1111
imm6	6-bit immediate data	0–63	0x0–0x3f	0b0–0b111111
sign6	Signed 6-bit immediate data	-32–31	0x0–0x3f	0b0–0b111111
imm8	8-bit immediate data	0–255	0x0–0xff	0b0–0b11111111
sign8	Signed 8-bit immediate data	-128–127	0x0–0xff	0b0–0b11111111
imm10	10-bit immediate data	0–1023	0x0–0x3ff	0b0–0b1111111111
imm13	13-bit immediate data	0–8191	0x0–0x1fff	0b0–0b11111111111111
imm32	32-bit immediate data	0–4294967295	0x0–0xffffffff	0b0– 0b11111111111111111111111111111111
sign32	Signed 32-bit immediate data	-2147483648– 2147483647	0x0–0xffffffff	0b0– 0b11111111111111111111111111111111

**(3) Symbols**

In specifying an address with immediate data, you can use a symbol defined in the source files.

Note: The symbols discussed here represent addresses that can be processed by the assembler. Symbols representing defined names and other character strings will be covered in the chapter relating to the Preprocessor pp33.

**Definition of symbols**

Usable symbols are defined as 32-bit values by any of the following methods:

1. It is described as a label (in CODE or DATA section)  
 Example: LABEL1: LABEL1 is a symbol that indicates the address of a described location in CODE or DATA section.
2. It is defined with a .comm or .lcomm pseudo-instruction (in BSS section)  
 Example: .comm BUF1 4 BUF1 is a symbol that indicates the address of a described location in BSS section.
3. It is defined with a .set pseudo-instruction (symbol definition dedicated absolute assembly)  
 Example: .set ADDR1 0xff00 ADDR1 is a symbol that represents absolute address 0x0000ff00.

**Restrictions on characters**

- The maximum number of symbol characters is 32. If this number is exceeded, an error will result.
- The characters that can be used are limited to the following:  
 A–Z a–z \_ 0–9  
 Note that a symbol cannot begin with a numeral. Uppercase and lowercase characters are discriminated.

**Local and global symbols**

Defined symbols are normally local symbols that can only be referenced in the file where they are defined. Therefore, you can define symbols with the same name in multiple files. To reference a symbol defined in some other file, you must declare it to be global in the file where the symbol is defined by using the .global pseudo-instruction.

\* The symbols defined by the .comm pseudo-instruction are handled as symbols declared to be global. Declaration by the .global pseudo-instruction is unnecessary.

**Extended notation of symbols**

When referencing an address with a symbol, you normally write the name of that symbol in the operand where an address is specified.

```
Examples: call LABEL          ...LABEL = sign8
           ld.w %rd, LABEL     ...LABEL = sign6
```

The Assembler also accepts the referencing of an address with a specified displacement as shown below.

```
LABEL + imm32 LABEL + sign32
```

```
Example: call LABEL+0x10
```

**Symbol mask**

The basic instructions in the EOC33000 instruction set are characterized by the fact that the immediate size that can be specified in the operand of each instruction is limited. Consequently, an assembler error results when a symbol whose value exceeds the size is used. When using the basic instructions, the high-order bits must be written separately in the ext instruction. A symbol mask is used for this purpose.

Specifically, a symbol mask is used to get the values from a symbol value that are written separately in the ext instruction and the basic instruction, and is entered immediately after the symbol.

When using extended instructions, the Instruction Extender ext33 attaches the necessary symbol mask as it expands the instruction. Therefore, you do not specifically need to be concerned about the ext instruction or symbol mask.

**Types of symbol masks**

The following 8 types of symbol masks can be used:

Symbol mask	Function
@rh or @RH	Acquires the 10 high-order bits of a relative address.
@rm or @RM	Acquires the 13 mid-order bits of a relative address.
@rl or @RL	Acquires the 8 low-order bits of a relative address.
@h or @H	Acquires the 13 high-order bits of an absolute address.
@m or @M	Acquires the 13 mid-order bits of an absolute address.
@l or @L	Acquires the 6 low-order bits of an absolute address.
@ah or @AH	Acquires the 13 high-order bits of a relative address.
@al or @AL	Acquires the 13 low-order bits of a relative address.

Examples:

```
ext LABEL@rh
ext LABEL@rm
call LABEL@rl Functions as "call LABEL".

ext LABEL@h
ext LABEL@m
ld.w %rd, LABEL@l Functions as "ld.w %rd, LABEL".

ext LABEL@ah
ext LABEL@al
ld.w %rd, [%rb] Functions as "ld.w %rd, [%rb+LABEL]".
```

- Notes:
- The symbol masks are effective only on the defined symbols. If a symbol mask is applied to a numeric value, an error will result.
  - If a symbol mask is omitted, the lower bits effective for that instruction will be used. However, if the bit value does not fall within the instruction range, an error or warning will be issued.

### 4.3.3 Extended Instructions

The Instruction Extender ext33 provides extended instructions for creating assembly source files. An extended instruction is such that the contents which normally are written in multiple instructions including the ext instruction can be written in one instruction. Extended instructions are expanded into the smallest possible basic instructions by the Instruction Extender.

#### Types of extended instructions

```
xadd xsub xcmp xand xoor xxor xnot xsll xsrl xsla xsra xrl xrr
xld.b xld.ub xld.h xld.uh xld.w xbset xbcrl xbtst xbnor
xjp xjreq xjrne xjrgt xjrge xjrle xjrle xjrult xjrult xjrult xjrule xcall xjp.d xjreq.d
xjrne.d xjrgt.d xjrge.d xjrle.d xjrle.d xjrult.d xjrult.d xjrult.d xjrult.d xjrule.d xcall.d
```

An extended instruction is derived from one of the basic instructions by adding the prefix "x". ("xoor" for the or instruction.)

#### Method for using extended instructions

The value or symbol for the expanded immediate size can be written directly in the operand.

```
Examples: xcall LABEL ;ext LABEL@rh
           ;ext LABEL@rm
           ;call LABEL@rl

           xld.w %r1, sign32 ;ext sign32@h
           ;ext sign32@m
           ;ld.w sign@l
```

In addition to the immediate expansion function of the basic instructions, a special operand specification like the one shown below is accepted for some instructions.

```
Examples: xadd %r0, %r1, 0x1 ; R0 ← R1 + 1
          xsub %sp, %sp, %r1 ; SP ← SP + R1
          xld.w %r0, [symbol + 0x10] ; R0 ← [symbol + 0x10]
          xjp LABEL + 5 ; Jumps to address LABEL + 5.
          xrl %r0, 15 ; Rotates the R0 content left by 15 bits.
```

For details about the extended instructions that include operands, refer to Section 10.6, "Extended Instructions".

**Note:** Extended instructions must be processed by the Instruction Extender ext33. They cannot be input directly into the Assembler as33 (this results in an error).

### 4.3.4 Additional Preprocessor Functions

The Preprocessor pp33 offers additional functions for the creation of assembly source files. This section will deal only with the notations for these functions. For details on each one of the functions, refer to Chapter 9, "Preprocessor". The preprocessor processes the notations of the said functions into mnemonic statements that can be assembled, thereby delivering assembly source files.

Note: The statements dealt with in this section need to be processed by the preprocessor, and cannot be entered directly into the Assembler as33. (Direct entry into the assembler will result an error.)

#### Preprocessor pseudo-instructions

The following five types of pseudo-instructions are provided for the Preprocessor pp33.

#include	Insertion of file
#define	Definition of character strings and numbers
#macro—#endm	Definition of macros
#ifdef(ifndef)—#else—#endif	Conditional assembly

All of these pseudo-instructions begin with a sharp (#).

```
Examples: #include "define.h"
          #define NULL 0
          #macro ADDM $1, $2
            xld.w %r0, [$1]
            xld.w %r1, [$2]
            add %r0, %r1
            xld.w [$1], %r0
          #endm
          #ifdef TYPE1
            ld.w %r0, 0
          #else
            ld.w %r0, -1
          #endif
```

For details on the notation of each pseudo-instruction and function, refer to Section 9.5 "Preprocessor Pseudo-Instructions".

#### Operators

To specify a value in the source, an expression using the following operators can be used:

		Examples
+	Addition, Plus sign	+0xff, 1+2
-	Subtraction, Minus sign	-1+2, 0xffff-0b111
*	Multiplication	0xf*5
/	Division	0x123/0x56
%%	Residue	0x123%%0x56
>>	Shifting to right	1>>2
<<	Shifting to left	0x113<<3
&	Logical product	0b1101&0b111
	Logical sum	0x123 0xff
^	Exclusive OR	12^35
~	Logical denial	~0x1234
^H	Acquires bit field (31:19)	0x1234^H
^M	Acquires bit field (18:6)	0x1234^M
^L	Acquires bit field (5:0)	0x1234^L
^AH	Acquires bit field (25:13)	0x1234^AH
^AL	Acquires bit field (12:0)	0x1234^AL
(, )	Parentheses	1+(1+2*5)

In the numeric parts of an expression, you can use a symbol whose value is defined by the preprocessor pseudo-instruction #define.

## 4.4 *Precautions for Creation of Sources*

---

- (1) Place a tab stop every 8 characters. Mixed processing by the Disassembler dis33 or source display/mixed display with the Debugger db33 of a source set at a tab interval other than 8 characters will result in displaced output of the source part.
- (2) When compiling/assembling a C source or assembly source that includes debugging information, do not include other source files (by using #include). It may cause a debugger operation error. This does not apply to ordinary header files that do not contain sources.
- (3) When describing an assembly source in absolute format, do not define two or more CODE, DATA or BSS sections. Actually, a source file can contain two or more of the same type of sections, note, however the program may not work correctly if the sections are not described in ascending order or because of other problems. Therefore, the absolute source in which the same section is separately defined cannot be guaranteed to work.
- (4) When using C and assembler modules in a program, pay attention to the interface between the C functions and assembler routines, such as arguments, size of return values and the parameter passing conventions.

## Chapter 5 Work Bench

This chapter describes the functions and operating method of the Work Bench wb33.

### 5.1 Functions

---

The Work Bench wb33 (hereafter called "wb33") provides an integrated operating environment ranging from the C Compiler or the Preprocessor to the Debugger. Its functions and features are summarized below:

- The software tools required for E0C33 Family program development can be started up from one window via the same method of operation.
- The basic make file and debugger parameter files can be created simply without using an editor.
- Almost all operations can be performed using only the mouse. Furthermore, the standard startup options of each tool can be selected simply by clicking on check boxes.
- When selecting a source file, you can display its contents on the screen (up to 32KB). What's more, the selected source file can be opened by a specified editor, allowing you to efficiently edit the source for correction.
- The wb33 also allows command lines including DOS command execution to be input from the keyboard.



## 5.2 Operations

### 5.2.1 Starting Up and Terminating wb33

#### To start up wb33

Choose "Work Bench 33" from the [Program] menu to start up the wb33.

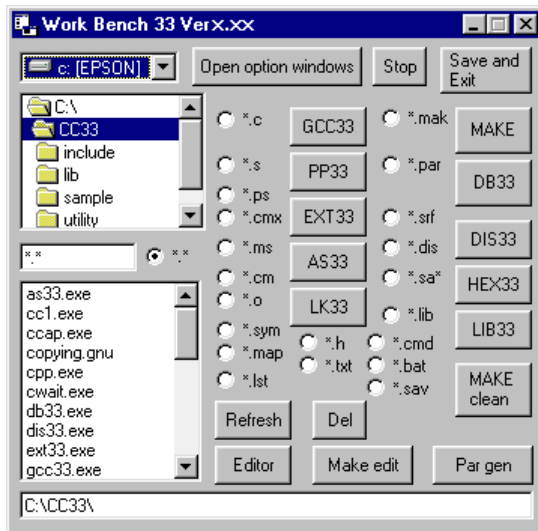


wb33.exe

To start up the wb33, double click the "wb33.exe" icon.

Also the wb33 starts up by dragging an option file (.sav) on the "wb33.exe" icon.

When the wb33 starts up, the execution window shown below appears.



#### wb33 startup command

The following shows the wb33 startup command:

##### Startup command

```
>wb33 [<option file name>],
```

##### Startup option

<option file name>: Specify an option file in which the settings in the option windows are recorded editor with full path.

Example: C:\CC33>wb33 c:\cc33\sample\tst\wb33.sav,

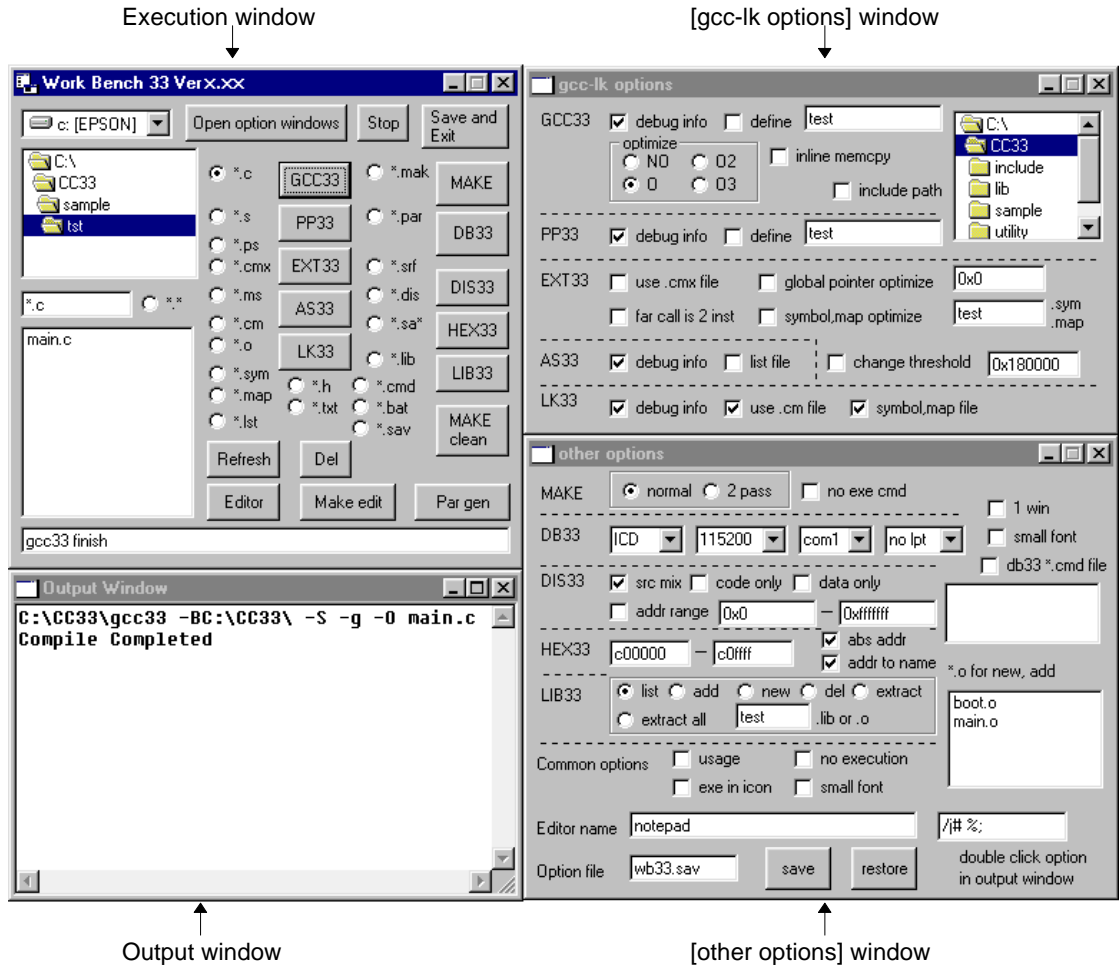
If this option is omitted, the option windows will be set to the default status.

#### To terminate wb33

Click on the [Save and Exit] button in the execution window.

## 5.2.2 Window

The diagram below shows the structure of the wb33 window.



### Execution window

Immediately after the wb33 is started up, only the execution window is open.

The following can be performed in the execution window:

- Choose the file you want to input into a software tool. (Refer to Section 5.2.3.)
- Execute a software tool. (Refer to Section 5.2.4.)
- Open the option windows. (Refer to the section below.)
- Create a make file. (Refer to Section 5.2.6.)
- Create a parameter file. (Refer to Section 5.2.7.)
- Display the source file and open the source file by using an editor. (Refer to Sections 5.2.3 and 5.2.8.)
- Input a command line to execute a DOS command, etc. (Refer to Section 5.2.9.)

## Option windows

When you click on the [Open option window] button in the execution window, two option windows are opened simultaneously.

### [gcc-lk options] window

This window allows you to select or specify the startup options of the following software tools (see Section 5.2.4.):

- C Compiler gcc33
- Preprocessor pp33
- Instruction Extender ext33
- Assembler as33
- Linker lk33

These are options of the tools that are executed by make.

### [other options] window

This window allows you to select or specify the startup options of the following software tools (see Section 5.2.4.):

- make
- Debugger db33
- Disassembler dis33
- Binary/HEX Converter hex33
- Librarian lib33

In addition to the above, you can choose the wb33 options and the options common to all tools (see Section 5.2.5), as well as save and restore the settings of selected options (see Section 5.2.10).

## Output window

The output window is used to display the source or display the execution results of each software tool. It opens up in the following two cases:

- When a software tool is executed
- When you double-click on the source file (text file) in the file list box of the execution window (see Section 5.2.3)

## Precautions to be taken when operating in wb33 windows

- The maximize buttons of the execution window and two option windows do not work, and the scroll bar is not displayed when the window size is reduced. In this case, try using the default size as much as possible. The output window can be maximized and returned to its original size without a problem.
- Minimization to a task bar button is supported in all windows. Each window except the execution window can be minimized and returned to its original size independently. When the execution window is minimized, all other windows are minimized simultaneously. The same applies when the execution window is restored to its original size.
- The wb33 can be terminated by clicking on the [Close] button of the execution window. The [Close] button in the option windows closes only the window to which it is attached. Note, however, that if the option windows are opened by the [Open option window] button again, all options selected before the windows were closed are restored to their initial settings.
- The list and the text boxes in each window except the output window can only be used for displaying or entering ASCII characters. For this reason, kanji and other unsupported characters are erratically represented. Although the output window can display kanji, even in this window, kanji may be erratically represented if the source file contains control characters, etc.

## 5.2.3 Selecting File and Displaying Source



The execution window has a list box for selecting a drive, directory or file. Use this list box to select the file you want to be input to a software tool.

### Selecting directory

Immediately after the wb33 is started up, the drive and directory where the wb33 and other tools are installed are selected.

Create a work directory for program development purposes, and elect that directory.

**Note:** Make sure that all wb33 processing is performed in the same directory. Furthermore, do not change the current directory by using the CD command while the wb33 is open.

### Types of files

Immediately after the wb33 is started up, the [\*.\*] radio button is selected and all file names in the selected directory are displayed in the file list box.

Each software tool has a radio button in front of the execution button for selecting the type of file to be input for the tool. This radio button facilitates the selection of a file.

**Note:** Although you can choose a file while all file types are being displayed before executing a software tool, care must be taken because the tool will start up even if you have selected a type of file that is not acceptable for the tool. Furthermore, the types of input files available for the Instruction Extender and Linker change depending on which options are selected.

### Updating the file list box



[Refresh] button

The file list box is updated when you choose a directory or execute a software tool; but it is not updated when you copy a file or create a file in some other application.

The file list box can be updated by clicking on the [Refresh] button in the execution window. Use this button whenever you want to update the list box. However, the directory list box is not updated.

### Deleting files

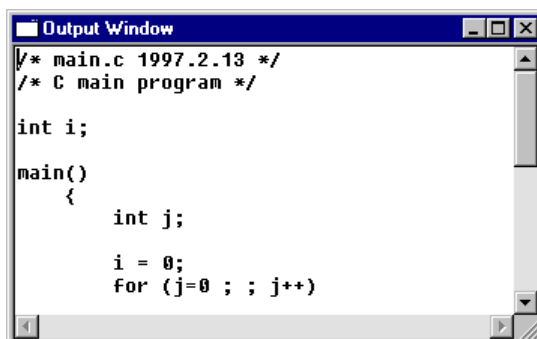


[Del] button

The [Del] button deletes the files selected in the file list box.

### Displaying the source file

By double-clicking on a source file name (text file) in the file list box, you can display the contents of that file in the output window. If the output window is closed, it will be opened when you double-click on a file name.

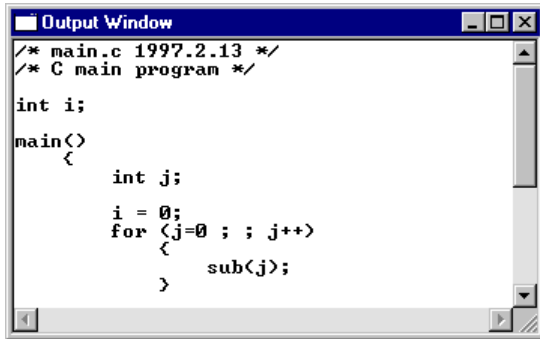


Only text files can be displayed in the output window, and the display size is limited to 32KB from the beginning of the file.

The output window is used for display-only. Although characters can be input or copied, and pasted in this window, no data can be saved.

If you want to display more than 32KB of text or edit the contents of a file, use an editor (see Section 5.2.8).

- \* When you choose the [smaller font] check box of the [other options] window, the font size in the output window is reduced, allowing for a greater amount of information to be displayed at a time.



The 10-point Terminal font is used for the reduced display. If this font is not installed in your system, the effect of [smaller font] cannot be guaranteed. The default font is 14-point FixedSys.

### 5.2.4 Executing Individual Tools

Each software tool can be executed using the buttons in the execution window.

#### To execute a software tool

1. Choose the startup options of the tool you want to be executed in the execution window.
2. Choose the file you want to be input to a software tool using the file list box and click on the tool's execution button.  
Multiple input files can be selected. In this case, the software tool is executed repeatedly as many times as the number of files selected.

#### About [Stop] button



Once you execute a software tool using the execution button, processing cannot be stopped in the wb33 until the tool (including make) finishes processing. However, if the tool is executed after selecting multiple files, processing can be halted by using the [Stop] button. Since a software tool processes one file at a time no matter how many files are selected, execution of the tool is halted when it finishes processing the file that was being processed when you clicked on the [Stop] button.

The following outlines the files input and output by each software tool and the startup options of each tool that can be selected by the wb33. (For make, refer to Section 5.2.6.) Explained below is the function of each option when selected.

For more information, refer to the chapters where each tool is detailed.

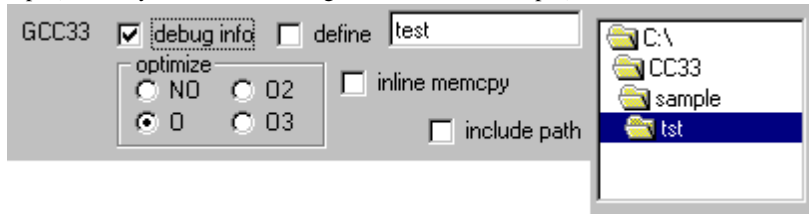
#### C Compiler gcc33

Execution button:

Input file: \*.c (C source file; lower-case is only allowed for ".c", ".C" cannot be used.)

Output file: \*.ps (assembly source file bearing the same name as input)

Options:



**debug info:** Selection of the -g option (turned on by default)

The information required for the C source level debug is generated in the output file. Normally, you should choose this option.

**define**: Selection of the -D option (turned off by default)

This option defines the macro name used in a conditional compilation. Input the definitions in the text box to the right of this option. When defining multiple macro names, separate each with a comma (.).

**NO**,  **O**,  **O2**,  **O3**: Selection of a -O, -O2 or -O3 option (O by default)

This option selects the optimization level.

**inline memcpy**: Selection of the -mno-memcpy option (turned off by default)

This option expands the strcpy or memcpy function in-line.

**include path**: Selection of the -I option (turned off by default)

The directory selected in the directory list box to the right of this option is set in one of the directories where the include file is searched.

### Preprocessor pp33

Execution button:

PP33

Input file: \*.s (assembly source file)

Output file: \*.ps (assembly source file bearing the same name as the input)

Options:

PP33  debug info  define test

**debug info**: Selection of the -g option (turned on by default)

The information required for debugging at the assembly source level is generated in the output file. Normally, you should choose this option.

**define**: Selection of the -d option (turned off by default)

This option defines the define name used in a conditional assembly. Input the definitions in the text box to the right of this option. When defining multiple define names, separate each with a comma (.).

### Instruction Extender ext33

Execution button:

EXT33

Input files: \*.ps (assembly source file)

\*.cmx (command file, specification of option required)

Output file: \*.ms (assembly source file bearing the same name as the input)

Options:

EXT33  use .cmx file  global pointer optimize 0x0  
 far call is 2 inst  symbol, map optimize test .sym  
 .map  
 change threshold 0x180000

**use .cmx file**: Selection of the -c option (turned off by default)

This option inputs a command file (.cmx) and executes it. When this option is selected, be sure to choose a .cmx file from the file list box of the execution window.

**global pointer optimize**: Selection of the -gp option (turned off by default)

This option performs optimization by a global pointer. When this option is selected, input the address of the global pointer in the text box to the right of this option.

**far call is 2 inst**: Selection of the -near option (turned off by default)

This option generates two instructions (one ext + branch instruction) for a jump to a nonexistent label in the file being processed. If this option is not selected, three instructions (two ext + branch instruction) are generated for the jump.

**symbol, map optimize**: Selection of the -lk option (turned off by default)

This option performs optimization using the symbol and link map files output by the linker. The source files that have been linked can be optimized. When this option is selected, input a common name for the symbol and map files in the text box to the right of this option.

**change threshold:** Selection of the -j option (turned off by default)

This option specifies the threshold value to be used for a branch instruction over a relatively long distance. When this option is specified, input a threshold value in the text box to the right of this option. If this option is not specified, a threshold value of 0x180000 is assumed.

### Assembler as33

Execution button:

Input file: \*.ms (assembly source file)

Output file: \*.o (object file bearing the same name as the input)

Options:   debug info  list file

**debug info:** Selection of the -g option (turned on by default)

The information required for debugging is generated in the output file. Normally, you should choose this option.

**list file:** Selection of the -l option (turned off by default)

This option generates an assembly list file.

### Linker lk33

Execution button:

Input files: \*.cm (command file, specification of option required)

\*.o (object file)

Output file: \*.srf (object file in srf33 format)

Options:   debug info  use .cm file  symbol,map file

**debug info:** Selection of the -g option (turned on by default)

The information required for debugging is generated in the output file. Normally, you should choose this option.

**use .cm file:** Selection of the -c option (turned on by default)

This option links modules according to the commands written in a command file. When this option is selected, be sure to choose a .cm file from the file list box of the execution window. Normally, you should specify this option.

**symbol,map file:** Selection of the -s and -m options (turned on by default)

This option generates a symbol and a link map file. These files are used during optimization by the Instruction Extender.

### Debugger db33

Execution button:

Input file: \*.par (parameter file)

Options:       1 win  
 small font  
 db33 \*.cmd file

**Debugger mode:**

Selection of the -sim, -icd or -mon option

(ICD mode by default)

This option selects a debugger operating mode.

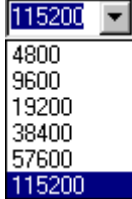
**1 win:** Selection of the -w option (turned off by default)

This option opens only the [Command] window when the Debugger starts up. If this option is not selected, the [Command], [Source] and [Register] windows are opened.

**small font:** Selection of the -sf option (turned off by default)

This option changes the font used in the debugger window to 10-point Terminal. The default font is 14-point FixedSys.

**Communication rate:** Selection of the -b option (115200 bps by default)



This option selects the rate of communication with the ICE33, ICD33 or MON33 (DMT33MON).

When using the ICE33 or ICD33, make sure that the DIP switch on the ICE33/ICD33 has been set correctly.

**Serial port:**



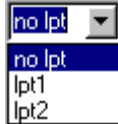
Selection of the -comX option (com1 by default)

This option selects the personal computer's serial port.

**db33 \*.cmd file:** Selection of the -c option (turned off by default)

This option executes a specified debug command file when the Debugger starts up. When this option is selected, choose a debug command file from the file list box located below the option select button. This list box displays the debug command file names in the directory currently selected in the execution window.

**Parallel port:**



Selection of the -lptX option (No by default)

This option selects the personal computer's parallel port.

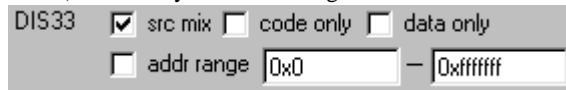
## Disassembler dis33

Execution button:

Input file: \*.srf (object file in srf33 the format)

Output file: \*.dis (disassembly list file bearing the same name as the input)

Options:



**src mix:** Selection of the -m option (turned on by default)

This option outputs disassembled lines to a disassembly list file with the source corresponding to it. If a data section is encountered, its dump is output.

**code only:** Selection of the -c option (turned off by default)

This option outputs a disassembly list of only code sections.

**data only:** Selection of the -d option (turned off by default)

This option outputs only a dump of data sections.


**Note:** These three options can be selected simultaneously, however, choose one option at a time. If multiple options are selected simultaneously, one or all specifications may be invalidated depending on the combination of selected options. (Refer to Section 13.3.2.)

**addr range:** Selection of the -a option (turned off by default)

This option specifies an address range for disassembling. When using this function, enter the start and end addresses in the text box.




### Binary/HEX Converter hex33

Execution button: 

Input file: \*.srf (object file in srf33 format)

Output file: \*.sa\* (file in Motorola S3 format bearing the same name as the input)

Options: 
 HEX33   
 -    
 abs addr  
 addr to name

**Address range:** Selection of an address range (0xc00000 to 0xc0ffff by default)

This option specifies an address range in the input file representing the extent to which the data is converted. The start and end addresses specified here must reside on the 32-byte boundaries.

**abs addr:** Selection of the -z option (turned on by default)-

This option generates an output file that contains absolute address information. Unless this option is selected, the output address is an offset address from the address at which conversion is started.

**addr to name:** Selection of the -x option (turned on by default)

This option adds information on a specified address range after the extension ".sa" of the output file.

Example: test.sa\_c00000\_c0fff

If this option is not selected, the extension will consist of only ".sa".

### Librarian lib33

Execution button: 

Input files: \*.lib (library file) Selected in the execution window

\*.o (relocatable object file) Selected in the option window

Output files: \*.lib (library file)

\*.o (relocatable object file)

Options: 
LIB33   
 list   
 add   
 new   
 del   
 extract  
 extract all   
 .lib or .o   
  

\*.o for new, add  
 boot.o  
 main.o

**Olist:** Selection of the -l option (turned on by default)

This option lists the object files registered in the library file in the output window (default) in the order in which they are registered.

**Oadd:** Selection of the -a option (turned off by default)

This option adds the object files (can be multiple files) that are selected in the [\*o for new, add] list box to a specified library file.

**Onew:** Deselecting all options (turned off by default)

This option creates a new library file. Input the library file name in the text box located below the option button. No extension is required. Choose the object files you want to register from the [\*o for new, add] list box (multiple files selectable).

If you specify a library name present in the current directory, the object files are added to the existing file in the same way as when the [add] option is selected.

**Odel:** Selection of the -d option (turned off by default)

This option deletes a specified object from the library file. Input the object name you want deleted in the text box provided below the option button. No extension is required.

**Oextract:** Selection of the -x option (turned off by default)

This option restores a specified object of the library file in the current directory as a file. Input the object name you want restored in the text box provided below the option button. No extension is required.

**Oextract all:** Selection of the -x option (turned off by default)

This option restores all the objects of the library file in the current directory as a file.

## 5.2.5 Selecting Execution Conditions

This section describes the options associated with the execution of a software tool and the display of execution results.

The options described below are the functions of the wb33, and not the startup options of any software tool. Use the [other options] window to select these options.

Common options	<input type="checkbox"/> usage	<input type="checkbox"/> no execution
	<input type="checkbox"/> exe in icon	<input type="checkbox"/> small font

**[usage] check box** (turned off by default)

When you click on the execution button of a software tool after checking this check box, the startup commands and the startup options of the tool are listed in the output window (default). The tool itself is not executed even when an input file or option is selected.

Example: usage display of gcc33

```

C:\CC33\gcc33
GNU C Compiler for 33 Ver 2.7.2 (Rev 0.42)
Usage:
  gcc33 -S [options] filename
Options:
  -g : generate debug information
  -O : optimize output code
  -E : preprocess source files and output the results to stdout
  -I<directory>\ : specifies the directory which cc1.exe and cpp.exe exit
  -I<directory> : specifies the directory which include files exist
  -D<macro=defn> : define macro "macro" as "defn"
  -D<macro> : define macro "macro" as '1'
  -merr : produce log file (gcc33.err)
  -fno-builtin : does not produce some C library functions inline
  -ms-memcpy : produce string variable initializer statement inline
Output:
  Extended assembler source files(.ps) for ext33
Example:
  gcc33 -S -Bc:\usr\local\bin\ -O -g test.c

```

**[no execution] check box** (turned off by default)

When you click on the execution button of a software tool after checking this check box, the startup commands of the tool including the input files or options selected in the window are displayed in the text box of the execution window. The tool itself is not executed.

This is effective when you want an option that cannot be specified in the option window to be added to a command line in the text box before executing a software tool. (Refer to Section 5.2.9.)

Example: Execute as33 after selecting [no execution]

```

C:\CC33\as33 -g tst_asm.ms

```

If the [usage] check box is turned on simultaneously with this option, a command line for displaying usage is displayed in the text box, and usage display is not performed.

**[exe in icon] check box** (turned off by default)

While a software tool is being executed, the MS-DOS window is normally open, displaying the tool's output messages. If this check box is turned on, the tool is minimized to a task bar when it is executed.

## 5.2.6 Make

The wb33 has a function that allows it to create a basic make file.

### To create a make file

1. Using the [gcc-lk options] window, set up the options for a range of software tools from the C Compiler to the Linker.
2. Click the [Make edit] button. The [Make file editor] window will appear.



3. Input the make file in the [Make file name] text box. No extension is required.
4. Choose all source files you want used from the file list box. All source files must be prepared in the same directory. Note that the files whose extensions are not ".c" or ".s" will be ignored even if they are selected.
5. When suffix definition is not used, deselect the [Suffix type] check box. The sample make file shown below was created with suffix definition.
6. Click on the [New Make file] button.

After the above is completed, a make file (.mak) will be created in the current directory. Two command files (.cmx, .cm) – one for the Instruction Extender and one for the Linker – will be created simultaneously.

### make file (.mak)

This file contains a description of execution procedures using the specified options and the source file for a range of software tools from the C Compiler to the Linker. For details about the make file, refer to Section 17.1, "Make".

Example: make file used in the tutorial (test.mak)

```
# make file made by wb33

# macro definitions for tools & dir

TOOL_DIR = C:\CC33
GCC33 = $(TOOL_DIR)\gcc33
PP33 = $(TOOL_DIR)\pp33
EXT33 = $(TOOL_DIR)\ext33
AS33 = $(TOOL_DIR)\as33
LK33 = $(TOOL_DIR)\lk33
LIB33 = $(TOOL_DIR)\lib33
MAKE = $(TOOL_DIR)\make
SRC_DIR =

# macro definitions for tool flags

GCC33_FLAG = -B$(TOOL_DIR)\ -S -g -O
PP33_FLAG = -g
EXT33_FLAG =
AS33_FLAG = -g
LK33_FLAG = -g -s -m -c
EXT33_CMX_FLAG = -lk test -c

# suffix & rule definitions

.SUFFIXES : .c .s .ps .ms .o .srf

.c.ms :
```

```

$(GCC33) $(GCC33_FLAG) $(SRC_DIR)$*.c
$(EXT33) $(EXT33_FLAG) $*.ps

.s.ms :
$(PP33) $(PP33_FLAG) $(SRC_DIR)$*.s
$(EXT33) $(EXT33_FLAG) $*.ps

.ms.o :
$(AS33) $(AS33_FLAG) $*.ms

# dependency list start
test.srf : test.cm \
    boot.o \
    main.o \
    $(LK33) $(LK33_FLAG) test.cm

## boot.s
boot.ms : $(SRC_DIR)boot.s
boot.o : boot.ms

## main.c
main.ms : $(SRC_DIR)main.c
main.o : main.ms

# dependency list end

# optimization by 2 pass make
opt:
$(MAKE) -f test.mak
$(TOOL_DIR)\cwait 2
$(EXT33) $(EXT33_CMX_FLAG) test.cmx
$(MAKE) -f test.mak

# clean files except source
clean:
del *.srf
del *.o
del *.ms
del *.ps
del *.map
del *.sym

```

### Command file for Instruction Extender (.cmx)

This file contains a list of the file names to be input to the Instruction Extender (the selected source files with their extensions changed to ".ps"). When executing 2-pass make, the Instruction Extender inputs the files written in this file in the second pass to optimize processing.

Example: command file used in the tutorial (test.cmx)

```

;Files start
boot.ps
main.ps
;Files end

```

**Command file for Linker (.cm)**

This file contains a description of the linker commands that control the link operation. When executing make, the linker uses the commands written in this file while executing a link operation.

Example: command file used in the tutorial (test.cm)

```

;Map set
;-code 0x0080000          ; set relative code section start address
;-data 0x0081000         ; set relative data section start address
;-bss 0x0000000         ; set relative bss section start address

;-code 0x0080100 [test2.o test3.o] ; set code sections to absolute address
;-data 0x0081100 [test2.o test3.o] ; set data sections to absolute address
;-bss 0x0000200 [test2.o test3.o] ; set bss sections to absolute address

;Library path
-I C:\CC33\lib

;Executable file
-o test.srf

;Object files start
boot.o
main.o
;Object files end

;Library files
;io.lib
;lib.lib
math.lib
string.lib
ctype.lib
fp.lib
idiv.lib

```

[Make gen] uses the commands that specify location addresses as comments when creating the linker command file. Customize this file according to the memory configuration of your development system before using it.

For details about the linker commands, refer to Section 12.5, "Linker Commands".

**To edit the make file**

To add/delete source files to/from the existing make file, open the [Make file editor] window with the following procedure:

1. Select the make file to be edited from the file list box on the execution window.
2. Click the [Make edit] button. The [Make file editor] window appears and the [Make file contents and Del files] list box shows the source files defined in the selected make file.



To add new source files to the make file, select the source files from the [Add files] list box and then click the [Add to Make file] button. The source files to be added must be prepared in the same directory of the already defined source files.

To delete source files from the make file, select the source files to be deleted from the [Make file contents and Del files] list box and then click the [Del from Make file] button.

When the files are added or deleted using the [Add to Make file] or [Del from Make file] button, the ".cm" and ".cmx" files with the same name will be automatically modified as well as the make file.

The [Editor] button on the [Make file editor] window has the same function as one on the execution window. It opens the text file selected in the [Add files] list box with the editor.

The [Refresh] button updates the file list in the [Add files] list box.

## Precautions on editing the make file

- The Make file editor defines the SRC\_DIR macro, that represents the source file directory, in the make file as follows:
  - The SRC\_DIR becomes blank when the source files are selected from the current directory (directory selected in the execution window).
  - The absolute path to the source files are defined when the source files are selected from another directory.

Therefore, the make file must be created again or the SRC\_DIR must be modified if the source file is moved to another directory after the make file is created. Furthermore, it is necessary to modify the SRC\_DIR when the source files are located in two or more directories.

- The Make file editor adds/deletes files using the comments and the character pattern in the make file as shown below. Do not modify the comments and patterns as the make file cannot edited with the Make file editor correctly

### Addition to the make file

Original make file

```
# dependency list start

test.srf : test.cm \
    boot.o \
    main.o \
    ..... *1
    $(LK33) $(LK33_FLAG) test.cm

## boot.s
boot.ms : $(SRC_DIR)boot.s
boot.o : boot.ms

## main.c
main.ms : $(SRC_DIR)main.c
main.o : main.ms

# dependency list end
```

Example: Make file after "sub.s" is added

```
# dependency list start

test.srf : test.cm \
    boot.o \
    main.o \
    sub.o \
    ..... *1
    $(LK33) $(LK33_FLAG) test.cm

## boot.s
boot.ms : $(SRC_DIR)boot.s
boot.o : boot.ms

## main.c
main.ms : $(SRC_DIR)main.c
main.o : main.ms

## sub.s
sub.ms : $(SRC_DIR)sub.s
sub.o : sub.ms

# dependency list end
```

File names and the dependency list are added between the comments "# dependency list start" and "# dependency list end". Pay attention when modifying this part.

The object file name is inserted above the line indicated with \*1.

The \*1 line includes a space character, so do not delete this line.

The object file name is inserted in the following format:

```
^^^<file>.o\          (^ denotes a space.)
```

The dependency list is inserted above the "# dependency list end" line.

Do not delete the "# dependency list end" line.

The dependency list is inserted in the following format:

```
##_<file>.s/c
<file>.ms^:^(SRC_DIR)<file>.s/c
<file>.o^:<file>.ms
(blank line)
```

**Deletion from the make file**

```
Original make file
# dependency list start

test.srf : test.cm \
    boot.o \
    main.o \
        ..... *1
    $(LK33) $(LK33_FLAG) test.cm

## boot.s
boot.ms : $(SRC_DIR)boot.s
boot.o : boot.ms

## main.c
main.ms : $(SRC_DIR)main.c
main.o : main.ms

# dependency list end
```

```
Example: Make file after "boot.s" is deleted
# dependency list start

test.srf : test.cm \
    main.o \
        ..... *1
    $(LK33) $(LK33_FLAG) test.cm

## main.c
main.ms : $(SRC_DIR)main.c
main.o : main.ms

# dependency list end
```

For the object file name, the Make file editor deletes the line that contains the specified file name with the "`^file>.o^`" format (^ denotes a space). Therefore, do not modify this format including the number of spaces.

For the dependency list, the Make file editor deletes the range from the source file name line that begins with `###` to the last blank line. Do not modify or delete the lines that begin with `###` or the blank lines.

**Addition to and deletion from the linker command file**

```
;Object files start
boot.o
main.o
;Object files end
```

When the source file configuration in the make file is modified (files are added/deleted) using the Make file editor, the object file configuration in the linker command file is also modified. The Make file editor modifies the file name list between the comments `";Object files start"` and `";Object files end"`, so do not modify or delete these comments.

When source files are added to the make file, the corresponding object file names are inserted above the `";Object files end"` line in the linker command file.

When source files are deleted from the make file, the lines that contain the corresponding object file name are deleted.

**Addition to and deletion from the instruction extender command file**

```
;Files start
boot.ps
main.ps
;Files end
```

When the source file configuration in the make file is modified (files are added/deleted) using the Make file editor, the file configuration in the instruction extender command file is also modified. The Make file editor modifies the file name list between the comments `";Files start"` and `";Files end"`, so do not modify or delete these comments.

When source files are added to the make file, the corresponding file names are inserted above the `";Files end"` line in the instruction extender command file.

When source files are deleted from the make file, the lines that contain the corresponding file name are deleted.

## To execute make

1. Before executing make, customize the make file and the linker command file if necessary.
2. Choose options for the make tool from the [other options] window.
3. Choose a make file from the file list box of the execution window and click on the [MAKE] button.

By following only the above operation, you can get an object file in srf33 format after linkage. For details about the functions and the operation of make, refer to Section 17.1, "Make".

## Options for make



### normal: 1-pass make (turned on by default)

This option executes make without specifying an argument for the target name.

Since this is to make the first target in the make file, 1-pass processing is performed until a .srf file is created or updated.

### 2 pass: 2-pass make (turned off by default)

This option executes make by using the target name "opt" as an argument.

The commands written in the make file are executed from "opt:" in the file. As a result, after processing up to linkage in the first pass is completed, optimization by the Instruction Extender is performed based on the linked symbol information. The file generated by this optimization process is assembled and linked one more time.

### no exe cmd: Selection of the -n option (turned off by default)

This option executes make after specifying the -n option of make.

The commands executed by make are only displayed: no command is actually executed. This option may be used to verify whether there is any file that has been modified after the previous execution of make.

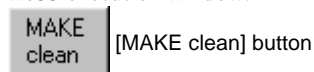
## MAKE clean

The make file created by the wb33 contains a description of the commands to delete intermediate and object files other than the sources. These commands are defined with the target name "clean".

The following lists the commands included in a make file:

```
# clean files except source
clean:
    del *.srf
    del *.o
    del *.ms
    del *.ps
    del *.map
    del *.sym
```

These commands can be executed by clicking the [MAKE clean] button after selecting the make file in the wb33 execution window.



All files in the current directory that have extensions ".srf", ".o", ".ms", ".ps", ".map" and ".sym" are deleted.



## 5.2.7 Parameter File Generator

The parameter file required for debugging can be created simply by the wb33.

A parameter file is used to set in the Debugger db33 the memory configuration of the microcomputer to be developed.

### [Parameter file generator] window

When you click on the [Par gen] button of the execution window, a [Parameter file generator] window opens up.

Par gen [Par gen] button

Start addr (1,2,-,ff)	Memory type	Big - Endian	Emulation memory	Enable setting
1 00000	<input checked="" type="radio"/> RAM <input type="radio"/> ROM <input type="radio"/> IO	<input type="checkbox"/> Big	<input type="checkbox"/> Emu	<input type="checkbox"/> Enable
2 00000	<input checked="" type="radio"/> RAM <input type="radio"/> ROM <input type="radio"/> IO	<input type="checkbox"/> Big	<input type="checkbox"/> Emu	<input type="checkbox"/> Enable
3 00000	<input checked="" type="radio"/> RAM <input type="radio"/> ROM <input type="radio"/> IO	<input type="checkbox"/> Big	<input type="checkbox"/> Emu	<input type="checkbox"/> Enable
4 00000	<input checked="" type="radio"/> RAM <input type="radio"/> ROM <input type="radio"/> IO	<input type="checkbox"/> Big	<input type="checkbox"/> Emu	<input type="checkbox"/> Enable
6 00000	<input checked="" type="radio"/> RAM <input type="radio"/> ROM <input type="radio"/> IO	<input type="checkbox"/> Big	<input type="checkbox"/> Emu	<input type="checkbox"/> Enable
8 00000	<input checked="" type="radio"/> RAM <input type="radio"/> ROM <input type="radio"/> IO	<input type="checkbox"/> Big	<input type="checkbox"/> Emu	<input type="checkbox"/> Enable
c 00000	<input type="radio"/> RAM <input checked="" type="radio"/> ROM <input type="radio"/> IO	<input type="checkbox"/> Big	<input type="checkbox"/> Emu	<input type="checkbox"/> Enable
10 00000	<input checked="" type="radio"/> RAM <input type="radio"/> ROM <input type="radio"/> IO	<input type="checkbox"/> Big	<input type="checkbox"/> Emu	<input type="checkbox"/> Enable

Use this window to specify the contents described below. Then, when you click on the [Create Par file] button, a parameter file is created in the current directory.

#### Parameter file name

Input a file version (v) in the text box for [This file version (0, 1,-, ff)] and the three low-order digits (xxx) of a microcomputer type name in the text box for [Chip name (3 characters)]. When this is specified, the parameter file is created in the name "33xxx\_v.par".

#### Specification of internal memory capacity

Input the internal RAM and the internal ROM capacities respectively in the text boxes for [Internal RAM size (0, 1,-, 256)] and [Internal ROM size (0, 1,-, 512)] in units of KB.

#### Selection of boot address

Choose the radio button [0x80000] when the system boots from the internal ROM or the radio button [0xc00000] when booting from the external ROM.

#### Selection of external memory area

When using an external memory, input the start address of the area to be used in the text box (in 1KB units) and choose a memory type (RAM, ROM, I/O) using the radio button. Accessing to the area will be done in little-endian format. It can be changed to big-endian format by choosing [Big]. When you check the [Enable] box, the area is made available for use as specified. Up to eight areas can be specified.

The ICE33 in-circuit emulator can contain emulation memory for up to 8MB of external memory. This helps debug a program without having to install memory in the target board. When allocating the specified external memory area in the emulation memory, check the [Emu] box. When using a device on the target board, do not check on the [Emu] box.

## Parameter file

The following shows an example of a created parameter file.

Example:

```
CHIP      33104      ; chip name (33XXX)
IROM     1000       ; internal ROM is 80000 to 80FFF
FOPT     0000       ; f option size
PRC VER  00 ff      ; allow any PRC board
PRC STATUS ***** ; allow any PRC board status
MCU      0x80000    ; 0x80000 internal boot address
VER      1          ; this file version

; Emulation memory allocation (max 8 areas, 1MB/area, 1MB boundary)
; Map allocation (max 31 areas, 256bytes boundary)

RAM      0          7FF      ; internal RAM area 2KB
IO       40000     4ffff     ; internal IO area 64KB

; Stack area except internal RAM area (max 8 areas, 256bytes boundary)

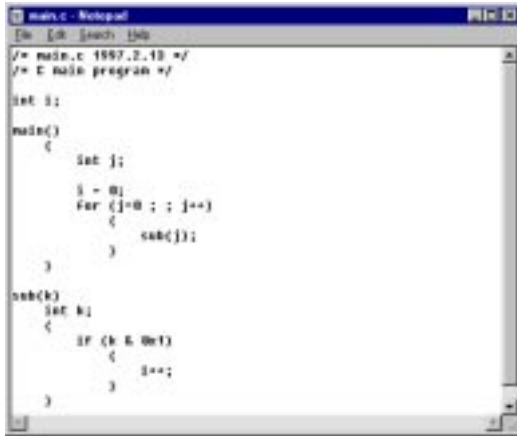
END
```

Since the file is created as a text file, it can be customized by using an editor. For details about the contents of this file, refer to Section 16.10, "Parameter File".

### 5.2.8 Specifying a General-purpose Editor

The wb33 allows you to display the source file or the execution result of a software tool by using a specified editor. In particular, a function that allows the source (text) file selected by the wb33 to be opened directly by an editor should prove effective in correcting the source.

#### To open the source file with an editor



1. Choose the text file you want to be opened from the file list box of the execution window. Multiple text files including a make file or command file other than the source file can be selected.
2. Click on the [Editor] button of the execution window.



The editor ("notepad" by default) will start up, bringing up the selected file(s) on the screen.

The [Editor] button in the [Make file editor] window has the same function.

#### To use an editor other than "notepad"

The default editor opened by the wb33 is "notepad". This editor can be changed by modifying the [Editor name] text box in the [other options] window.



Input the editor's startup command in full path here.

Since this setting returns to the default setting when the [other options] window is closed, save it using the [save options] button. (Refer to Section 5.2.10.)

#### Tag jump

The text box on the right of the [Editor name] box is used to set an editor command for tag jump from an error message in the output window to the corresponding source line in the editor window.

Notepad (default editor) does not support this function. This function is effective when an editor that supports a startup command for specifying a file name and line number is used.

The default setting "/j# %;" is the command for Hidemaru a Japanese editor.

When a file name and line number, a part of a message displayed in the output window, is double-clicked, the set command is sent to the editor after replacing # with the line number and % with the file name.

Example: boot.ms(6): Error: Invalid instruction. - define near boot.s(4)

When boot.ms(6) is double-clicked, # is substituted with 6, % with boot.ms, and then the command is sent to the editor. The editor will open boot.ms and show line 6 if the command is supported.

In the example above, boot.s(6) may also be used for tag jump.

main.c:4: warning: data definition has no type or storage class

In this message,main.c:4 may be used for tag jump.

## 5.2.9 Entering Command Lines

The execution window has a text box to display or edit the startup commands of each software tool. The startup options that are not supported by the check boxes or buttons of the option windows can be added (and executed) from this text box.

Furthermore, the DOS commands such as COPY can be executed from this text box.

### To execute a command line after editing it

1. Choose the options and the input file for the software tool you want to execute.
2. Turn on the [no execution] check box of the [other options] window and click on the execution button of the software tool.

The startup command including the selected contents will be displayed in the text box.



3. Input additional necessary options and hit the [Enter] key.

The edited command line will be executed.

**Note:** Do not start up a software tool from the execution button. If you click on the execution button, the tool will be started up with the previous command line before you edited.

Furthermore, if a software tool is started up from a command line, the execution result cannot be displayed in the output window or editor. No matter whether the -e option is specified, the messages normally output at end of execution (contents of an error file) are not displayed. Also, specification of the [exe in icon] check box is ignored. The file list box is not updated either.

### To execute a DOS command

Input a DOS command in the text box by adding ">" at the beginning of the name and hit the [Enter] key. The system executes this DOS command.



**Note:** DIR cannot be used to display the results in the output window or editor. Since CD and some other DOS commands affect the operation of the wb33, be sure to use only those commands that copy or rename a file.

## 5.2.10 Saving and Restoring Options

The contents selected in the option windows can be saved to a file using the [save] button of the [other options] window and the saved contents can be restored using the [restore] button. Since the setup contents are reset to the default settings when the option window is opened, use this method to save settings of frequently used options and editors.



The default file name of the settings saved is "wb33.sav". When saving multiple settings, use a different file name for each one. No message is output to confirm whether the file can be overwritten.

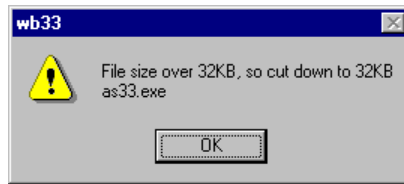
The file is created in the text format.

The option settings are also saved to the option file when the wb33 is terminated by the [Save and Exit] button. Furthermore, by dragging the created option file on the "wb33.exe" icon to start up the wb33, the saved option settings are restored. This can also be done by specifying the option file in the command line when starting up the wb33 from the DOS prompt.

## 5.3 Error Messages

When an error occurs in the wb33, a dialog box for displaying a message pops up. After checking the message, click on the [OK] button to close the dialog box.

Example:



The table below lists the error messages generated by the wb33. For the error messages output when executing a software tool, refer to the chapters in which each software tool is discussed.

Table 5.3.1 Error messages of wb33

Message	Content
Can not execute command xxxxxx	A tool button or command line (xxxxxx) cannot be executed. <ul style="list-style-type: none"> <li>• The command line is invalid.</li> <li>• The tool or necessary file cannot be located.</li> </ul>
Can not open file xxxxxx	The file (xxxxxx) cannot be opened. <ul style="list-style-type: none"> <li>• When executing [Make edt]</li> <li>• During source display</li> <li>• When saving or restoring options</li> </ul>
Write error xxxxxx	Data cannot be written to the file (xxxxxx). <ul style="list-style-type: none"> <li>• When executing [Make edit]</li> <li>• When saving options</li> </ul>
Read error xxxxxx	The file (xxxxxx) cannot be loaded. <ul style="list-style-type: none"> <li>• During source display</li> <li>• When restoring options</li> </ul>
R/W error xxxxx or xxxxxx	The file cannot be read or data cannot be written to the file. <ul style="list-style-type: none"> <li>• When executing [Make file editor]</li> </ul>
Can not delete file xxxxxx	The file cannot be deleted. <ul style="list-style-type: none"> <li>• When executing [Del]</li> </ul>
File size over 32KB, so cut down to 32KB xxxxxx	A file (xxxxxx) exceeding 32KB in size is selected. <ul style="list-style-type: none"> <li>• During source display</li> </ul> Only the first 32KB part of the file can be displayed.

## 5.4 Precautions

Make sure a series of processing in the wb33 all are performed in the same directory.

Also, be careful not to change the current directory using the CD command while the wb33 is open.

## Chapter 6 C Compiler

This chapter explains how to use the C Compiler gcc33, and provides details on interfacing with the assembly source.

For information about the standard functions of the C Compiler and the syntax of the C source programs, refer to the ANSI C literature generally available on the market.

### 6.1 Functions

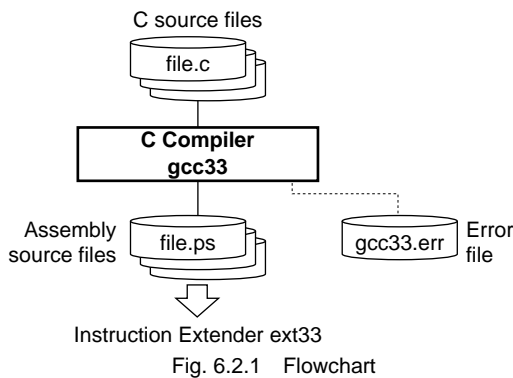
The C Compiler gcc33 (hereafter called "gcc33") compiles C source files to generate an assembly source file that includes E0C33000 instruction set mnemonics, the Instruction Extender's extended instructions, and assembler pseudo-instructions. The C Compiler gcc33 in this package is an ANSI standard C compiler. Since special syntax is not supported, the programs developed for other types of microcomputers can be transplanted easily to the E0C33 Family.

Furthermore, since this C Compiler has a powerful optimizing capability that allows it to generate a very compact code, it is best suited to developing embedded applications.

This C Compiler consists of three files: gcc33.exe, cpp.exe, and ccl.exe.

The gcc33 is based on the C Compiler of Free Software Foundation, Inc. Details about the license of this compiler are written in the text file "GNU\_COPYRIGHT", therefore, be sure to read this file before using the compiler.

### 6.2 Input/Output Files



#### 6.2.1 Input File

##### C source file

File format:	Text file
File name:	<file name>.c
Description:	File in which the C source program is described.

#### 6.2.2 Output Files

##### Assembly source file

File format:	Text file
File name:	<file name>.ps (The <file name> is the same as that of the input file.)
Output destination:	Current directory
Description:	An assembly source file to be input to the Instruction Extender ext33. The file cannot be input to the Assembler as33 directly since it includes the extended instructions.

##### Error file

File format:	Text file
File name:	gcc33.err
Output destination:	Current directory
Description:	File that is output when the startup option (-merr) is specified, and describes the contents which the C Compiler gcc33 delivers through the Standard Output (stdout), such as an error message. When the -merr option is specified, messages do not appear on the screen. It is different from other tools.

## 6.3 Starting Method

---

### 6.3.1 Startup Format

#### General form of command line

**gcc33 ^ [<Startup option>] ^ [<file name>]**

^ denotes a space.

[ ] indicates the possibility to omit.

<file name>: Specify C source file name(s) including the extension (.c).

#### Operations on work bench

Select startup options and source file(s), then click the [GCC33] button.

Multiple source files can be specified in a command line. All files can be processed at the same time. Although the wb33 also allows multiple files to be selected, it executes the gcc33 as many times as the number of files selected.

### 6.3.2 Startup Options

The gcc33 comes provided with the following 11 types of startup options:

#### -S

Function: Output of assembly code

Specification on wb33: None (always specified)

Explanation:

- This switch is used to output an assembly source file.
- This option must always be specified. If the gcc33 is started up without this option, it only displays Usage, and does not compile the source file.

#### -B<path name>

Function: Compiler's path specification

Specification on wb33: None (unnecessary)

Explanation:

- Specify the directory where the compiler proper cc1.exe and the C preprocessor cpp.exe exist.
- For <path name>, input a relative or an absolute path immediately following -B, then enter a back slash (\) at the end of the name.
- If the directory where the compiler proper and the C preprocessor exist is registered in environment variable GCC\_EXEC\_PREFIX or PATH, the -B switch is unnecessary. The priority is the -B switch, GCC\_EXEC\_PREFIX, and PATH, in that order. GCC\_EXEC\_PREFIX must be registered in the same format of relative or absolute path and a \ as required for the -B switch.
- If the -B switch and GCC\_EXEC\_PREFIX are nonexistent, the directory specified by PATH or the current directory is assumed.

#### -E

Function: Execution of C preprocessor only

Specification on wb33: None

Explanation:

- Only the C preprocessor is executed in the specified C source file, and the results are output to the standard output device.

#### -I<path name>

Function: Specification of a directory that contains the include files

Specification on wb33: Check [include path] and choose a directory from the list box.

Explanation:

- Specify the directory that contains the files included in the C source.
- Input <path name> immediately after -I.
- Multiple directories can be specified. In this case, input as many instances of -I<path name> as necessary. The include files are searched in the order they appear in the command line.

- If the directory is registered in environment variable `C_INCLUDE_PATH`, the `-I` switch is unnecessary.
- File search is performed in order of priorities, i.e., current directory, `-I` switch, and `C_INCLUDE_PATH` in that order.

**-D<macro name>[=<replacement character>]**

Function: Definition of a macro name

Specification on wb33: Check [define] and input a macro name in the text box.

- Explanation:
- Define a macro name. This option functions in the same way as `#define`. If there is `=<replacement character>` specified, define its value in the macro. If not specified, the value of the macro is set to 1.
  - Input `<macro name>[=<replacement character>]` immediately after `-D`.
  - Multiple macro names can be specified. In this case, input as many instances of `-D<macro name>[=<replacement character>]` as necessary. For the wb33, separate each instance of `<macro name>[=<replacement character>]` with a comma (,) as you input them.

**-O, -O2, -O3**

Function: Specification of optimization

Specification on wb33: Check one of [NO], [O], [O2] or [O3].

- Explanation:
- Specify one of the four switches to perform optimized processing. When generating code, the compiler optimizes it by placing emphasis on code efficiency and speed (mainly code efficiency).
  - If no switch is specified or [NO] is selected for the wb33, code optimization is not performed.
  - The greater the value of `-O`, the higher the code efficiency. However, there is a greater possibility of causing a problem, such as absence of some debugging information in the output. If optimization cannot be executed normally, reduce the value of optimization. Normally, `-O` should be specified.
  - When an optimization is specified, the compiler reuses the value loaded from the memory to the register to reduce memory read/write operations. So, sometimes the memory may not be accessed. To avoid this situation, take measures as shown below.
    - Declare variables with "volatile". Example) `volatile char IO_port1;`
    - Do not specify the optimization.
    - Use `-fvolatile`. Pointers are accessed as volatile objects.
    - Use `-fvolatile-global`. External variables are all accessed as volatile objects.

**-g**

Function: Addition of debugging information

Specification on wb33: Check [debug info].

- Explanation:
- Creates an output file containing debugging information.
  - Always specify this option when you perform the C source level debugging.
  - Refer to Section 6.6 for debugging information.

**-mno-memcopy**

Function: Inline expansion of `strcpy` and `memcpy` function calls

Specification on wb33: Check [inline memcpy].

- Explanation:
- The `strcpy` and `memcpy` function calls are expanded in-line.

**-merr**

Function: Output of error files

Specification on wb33: Non

- Explanation:
- Delivers in a file (`gcc33.err`) the contents that are output by the `gcc33` via the Standard Output (`stdout`), such as error messages.
  - When this option is specified, messages do not appear on the screen.

When entering options in the command line, you need to place one or more spaces before and after the option.

Example: `c:\cc33\gcc33 -S -Bc:\user\local\bin\ -O -g test.c`



## 6.4 Messages

---

The gcc33 delivers its messages through the Standard Output (stdout).

If the gcc33 is started up by using the wb33's [GCC33] button, the message is output to "wb33.err". When execution is completed, a message is displayed in the output window (default).

### End message

The gcc33 outputs only the following end message when it ends normally.

```
Compile Completed
```

### Usage output

If no file name was specified or an option was not specified correctly, the gcc33 ends after delivering the following message concerning the usage:

```
C:\CC33\gcc33
```

```
GNU C Compiler for E0C33 Ver 2.7.2 (Rev x.xx)
```

```
Usage:
```

```
gcc33 -S [options] filename
```

```
Options:
```

```
-g : generate debug information
```

```
-O : optimize output code
```

```
-E : preprocess source files and output the results to stdout
```

```
-B<directory>\ : specifies the directory which cc1.exe and cpp.exe exit
```

```
-I<directory> : specifies the directory which include files exist
```

```
-D<macro=defn> : define macro "macro" as "defn"
```

```
-D<macro> : define macro "macro" as '1'
```

```
-merr : produce log file (gcc33.err)
```

```
-mno-memcpy : produce string variable initializer statement inline
```

```
Output:
```

```
Extended assembler source files(.ps) for ext33
```

```
Example:
```

```
gcc33 -S -Bc:\usr\local\bin\ -O -g test.c
```

### When error/warning occurs

If an error or a warning is produced, an error/warning message will appear before the end message shows up.

In the case of an error, the gcc33 ends without creating an output file.

In the case of a warning, the gcc33 ends after creating an output file. However, the output file cannot be guaranteed to work properly.

## 6.5 Compiler Output

This section explains the assembly sources output by the gcc33 and the registers used by the gcc33.

### 6.5.1 Output Contents

After compiling C sources, the gcc33 outputs the following contents:

- E0C33000 instruction set mnemonics
- Extended instruction mnemonics
- Assembler pseudo-instructions

All but the basic instructions are output using extended instructions. Therefore, be sure to use the Instruction Extender ext33 to process the assembly source files output by the gcc33. These files cannot be assembled directly by the Assembler as33. Nor can the assembly source files output be put through the Preprocessor pp33.

Since the system control and MAC instructions cannot be expressed in the C source, use in-line assemble by asm or an assembly source file to process them.

Example: `asm("mac %r12")`

Assembler pseudo-instructions are output for section and data definitions. For details about the assembler pseudo-instructions, refer to Section 11.8, "Assembler Pseudo-instructions".

The following describes the sections where instructions and data are set.

#### Instructions

All instructions are located in the CODE section.

#### Global and static variables without initial values

These variables are located in the BSS section.

Example: `int i; .comm i 4`

#### Global and static variables with initial values

These variables are located in the DATA section.

Example: `int i=1; .global i`

```

                                .data
                                .align 2
                                i2:
                                .word 1

```

#### Constants

Constants are located in the CODE section.

Example: `const int i=1 .global i`

```

                                .code
                                .align 2
                                i2:
                                .word 1

```

For all symbols including function names and labels, symbol information by assembler pseudo-instruction `.def` is inserted (when the `-g` option is specified). For details about the symbol information, refer to Section 6.6, "Debugging Information".

Labels are output in the following format:

```

__Limm31   Jump address label
__LCimm31  Character string constant label
__Lbimm31  Beginning of block position label
__Leimm31  End of block position label
           (imm31 takes on a decimal number in the range of 0 to 2,147,483,647.)

```

## 6.5.2 Data Representation

The gcc33 supports all data types under ANSI C. Table 6.5.2.1 below lists the size of each type (in bytes) and the effective range of numeric values that can be expressed in each type.

Table 6.5.2.1 Data type and size

Data type	Size	Effective range of a number
char	1	-128 to 127
unsigned char	1	0 to 255
short	2	-32768 to 32767
unsigned short	2	0 to 65535
int	4	-2147483648 to 2147483647
unsigned int	4	0 to 4294967295
long	4	-2147483648 to 2147483647
unsigned long	4	0 to 4294967295
pointer	4	0 to 4294967295
float	4	1.175e-38 to 3.403e+38 (normalized number)
double	8	2.225e-308 to 1.798e+308 (normalized number)

The float and double types conform to IEEE standard formats.

### Store positions in memory

The positions in the memory where data is stored depend on the type. Regardless of whether it is global or local, data is located in the memory in as many bytes as are determined by the size beginning with an address that can be divided by the size.

The double type is aligned at 4-byte boundary addresses, so that the 4 low-order bytes of data (mantissa part (31–0)) are stored in 4 bytes of low-order locations of memory, and the 4 high-order bytes of data (sign, exponent, and mantissa part (51–32)) are stored in 4 bytes of high-order memory locations.

### Structure data

Structure data is located in the memory beginning with 4-byte boundaries (addresses divided by 4) in the same way as stated above for the double type. Members are located in the memory according to the size of each data type in the order they are defined.

The following shows an example of how structure is defined, and where it is located.

Example: struct Sample {

```

char    cData;
short   hData;
char    cArray[3];
int     iData;
double  dData;

```

};

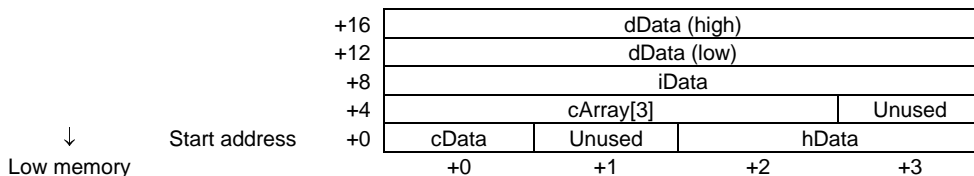


Fig. 6.5.2.1 Sample locations of structure data in the memory

As shown in the diagram above, some unused areas may remain in the memory depending on the data type of a member.

### 6.5.3 Method of Using Registers

The following shows how the gcc33 uses general-purpose registers.

Table 6.5.3.1 Method of using general-purpose registers by gcc33

Register	Method of use
R0	Registers that need have to their values saved when calling a function
R1	
R2	
R3	
R4	Scratch registers
R5	
R6	
R7	
R8	Global pointer (unused; used by ext33)
R9	Scratch register for expanding extended instruction (unused; used by ext33)
R10	Register for storing returned values (8/16/32-bit data, 32 low-order bits of double-type data)
R11	Register for storing returned values (32 high-order bits of double-type data)
R12	Register for passing argument (1st word)
R13	Register for passing argument (2nd word)
R14	Register for passing argument (3rd word)
R15	Register for passing argument (4th word)

#### Registers for saving values when calling a function (R0 to R3)

These registers are used to store the calculation results of expressions and local variables. These register values after returning from a function must be the same as those when the function was called. Therefore, the called function has to save and restore the register values if it modifies the register contents.

#### Scratch registers (R4 to R7)

These registers are used to store the temporary calculation results of expressions and local variables. These registers do not need to be saved when calling a function.

#### Global pointer (R8)

This register is reserved for storing a global pointer. The gcc33 does not use this register.

#### Scratch register for expanding extended instructions (R9)

Provided for use in assemble, this register is used by the Instruction Extender ext33 as it expands an extended instruction. The gcc33 does not use this register.

#### Registers for storing returned values (R10, R11)

These registers are used to store returned values. They are used as scratch registers before storing a returned value.

#### Registers for passing arguments (R12 to R15)

These registers are used to store arguments when calling a function. Arguments exceeding the four words are stored in the stack before being passed. They are used as scratch registers before storing arguments.

Note: When creating assembler subroutines that are called from C routines, pay attention to the register usage.

- The R4 to R7 registers can be used without saving/restoring the contents.
- The R10 and R11 registers can be used without saving/restoring the contents until a returned value is set in the register before returning.
- Before the R12 to R15 registers can be used, the stored arguments must be used or saved in other locations. It is necessary to restore the contents before returning.
- Try to use the R8 and R9 registers as little as possible.
- Before the R0 to R3 registers can be used, the contents must be saved to stack using the pushn instruction. Also, the saved contents must be restored from the stack using the popn instruction.

## 6.5.4 Function Call

### The way arguments are passed

When calling a function, arguments up to four words are stored in registers for passing argument (R12 to R15) while larger arguments are stored in the stack frame of the calling function (explained in the next section) before they are passed.

Example: `func(w1, w2, w3, w4, w5, w6); ...w1→R12, w2→R13, w3→R14, w4→R15, w5&w6→Stack`  
(wN: arguments equal to or smaller than word size)

Basically, arguments are stored in R12 to R15 in the order that they are specified.

### Data size of argument

Arguments in data size of 4 bytes or less are handled in units of words (4 bytes) irrespective of the data type. The char and short types of data are sign-extended; the unsigned char and unsigned short types are zero-extended. Only the double type is handled in units of 8 bytes. Unless two registers among R12 to R15 are available when passing an argument of the double type, it is passed via the stack.

Example: `func(w1, d2, d3, w4); ...w1→R12, d2(L)→R13, d2(H)→R14, w4→R15, d3→Stack`  
(wN: arguments equal to or smaller than word size; dN: arguments of double type)

### Handling of structure arguments (Note)

If the argument is structure data, the values of structure members are passed via a stack.

```
Example: struct _foo {
        int      a;
        short    b;
        char     c;
    };

    callee(struct _foo foo, int d);
```

In the above example, only d is stored in the register for passing argument (R12) and all the members of foo are stored in the stack.

### Passing argument to a function that returns structure (Note)

When calling a function that returns structure data, the structure address where the result is stored is set in the R12 register as the first argument before being passed to the called function. Consequently, the arguments written in the source are successively carried down by one.

If the structure is not used as a returned value, the compiler assigns dummy structure data to the local variable area of the calling function and passes the address of this location.

The called function returns the pointer passed in the first argument to the calling function as a return value.

### Saving registers

If a called function modifies the R0 to R3 registers, the function has to save and restore the register values.

The R4 to R7 and R10 to R15 registers can be used without such a restriction.

The R8 and R9 can also be used freely, if it does not conflict with the processing of Instruction Extender ext33.

### Returned values

The word size or less of returned value is stored in the R10 register.

The double-word size (double) of returned value is stored in the R10 (low-order word) and R11 (high-order word) registers.

**Note:** When a source program in which a structure is passed to or returned from a subroutine, the actual code is created so that all the members of the structure are copied using the memcopy function. This is undesirable since it increases the code size, lowers the execution speed and causes bugs in the compiler. Therefore, pointers should be used for passing structures as much as possible.

## 6.5.5 Stack Frame

When calling a function, the gcc33 creates the stack frame shown in Figure 6.5.5.1. The start address of the stack frame is always a word (32-bit) boundary address.

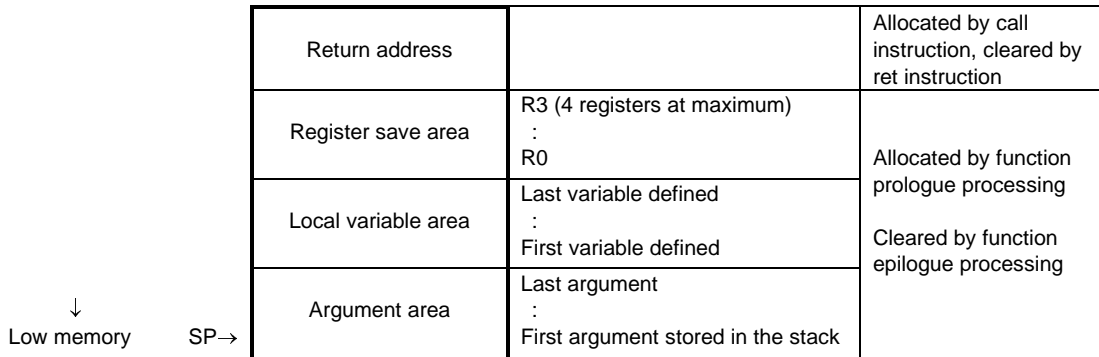


Fig. 6.5.5.1 Stack frame

### Return address

This is the return address (1 word) to the calling function.

### Register save area

If any registers from R0 to R3 are used by the calling function, they are saved to this area in order of register numbers beginning with the highest.

If none of the registers from R0 to R3 is used by the calling function, this area is not allocated.

### Local variable area

If there are any local variables defined in the called function that cannot be stored in registers, an area is allocated in the stack frame. Then they are saved sequentially beginning with the last-declared variable at boundary addresses (4-byte boundaries for the double type) according to the data types.

Example: {

```

char    c;
short   s;
int     i;
      :
}

```

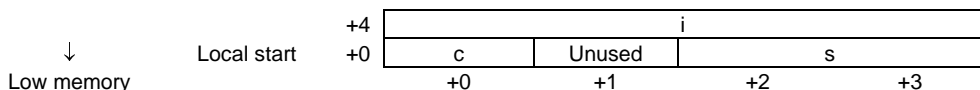


Fig. 6.5.5.2 Example of local variables saved to stack

This area is not allocated if there is no local variable that needs to be saved in the stack.

### Argument area

If there are any arguments for another function call in the called function that cannot be stored in the registers for passing argument, an area is allocated in the stack frame (see the preceding section). All arguments are located at 4-byte boundaries. The 32 low-order double-type bits are saved at low-order addresses, and the 32 high-order bits are saved at the high-order addresses.

This area is not allocated if there is no function call.

### Allocating and clearing the stack area

A stack area for the return address is allocated and the address is saved to this area by the call instruction. The address is popped from the stack and the area is cleared by the ret instruction.

All other areas are allocated in the prologue processing of the function, and are cleared in the epilogue processing.

## 6.6 Debugging Information

If the startup option `-g` is specified (by checking `[debug info]` in `wb33`), the `gcc33` inserts assembler pseudo-instructions in the output file as debugging information in order to allow for C source level and symbolic debugging.

- Notes:
- This debugging information is required before C source level or symbolic debugging can be performed with the Debugger `db33`.
  - Make sure the debugging information is created by only specifying the `-g` option, and not by any other method. Also, be sure not to correct the debugging information that is output. Corrections could cause the `as33`, `lk33`, `db33` or `dis33` to malfunction.
  - Unless the `-g` option is specified in the `lk33` even though it may be specified in the `gcc33` (same applies for `pp33`), all debugging information will be cut when linking.
  - Unless the `-g` option is specified in the `as33` even though it may be specified in the `gcc33`, all symbol information will be cut. The source information is not cut.
  - If the `-g` option is specified in the `as33` without specifying it in the `gcc33`, symbol names and address-only symbol information are added during assembly.

### 6.6.1 Source Information

The following three debug pseudo-instructions are output in order for the C source to be displayed in the debugger.

1) **.file** "<path\_name>"

This indicates the beginning of a file. It is inserted at the start position of the file. <path\_name> is the file's path name.

2) **.endfile**

This indicates the end of a file. It is inserted at the end of the file.

3) **.loc** <line\_no>

This indicates the line information of the source file. It is inserted at the beginning of the assembly code corresponding to each C source line. <line\_no> is the source's line number.

### 6.6.2 Symbol Information

Information on all functions and variables are output as a `.def` pseudo-instruction. The following shows the format of this `.def` pseudo-instruction.

General format: `.def <symbol>, <parameter>, [<parameter>, .....,] endef`

The contents of the `.def` pseudo-instruction thus output are shown below for each type of symbol.

#### Automatic variable, structure, union, or enum-type member, argument

**.def** <sym>, val <expr1>, scl <expr2>, type <expr3>, endef

- <sym> Symbol name in the C source (variable name/member name)
- <expr1> Automatic variable/argument (stack): Offset from the stack pointer (decimal)  
Automatic variable/argument (register): Register number (decimal)  
Structure or union member: Offset from the beginning of structure or union (decimal)  
enum-type member: Value indicating a member (decimal)
- <expr2> Storage class of <sym> (decimal)
- <expr3> Data type of <sym> (hex)

This pseudo-instruction indicates that <sym> is an automatic variable or a structure, union or enum-type member.

**Static variable, global variable, function**

```
.def <sym1>, val <sym2>, scl <expr1>, type <expr2>, endif
  <sym1>   Symbol name in the C source (variable name/member name)
  <sym2>   Relocatable symbol name corresponding to <sym1>
  <expr1>  Storage class of <sym2> (decimal)
  <expr2>  Data type of <sym2> (hex)
```

This pseudo-instruction indicates that <sym2> is a function, static variable, or global variable that corresponds to the C source's variable name <sym1>.

**Tag declaration of structure, union or enum type (start)**

```
.def <sym>, scl <expr1>, type <expr2>, size <expr3>, endif
  <sym>    Tag name of structure, union or enum type in the C source
  <expr1>  Storage class of <sym> (decimal)
  <expr2>  Data type of <sym> (hex)
  <expr3>  Data size of <sym> (decimal)
```

This pseudo-instruction indicates that the declared tag name is the structure, union or enum type of <sym>, and that member information exists in the next or later .def pseudo-instruction.

**Bit field member (structure or union member)**

```
.def <sym>, val <expr1>, scl <expr2>, type <expr3>, size <expr4>, endif
  <sym>    Bit field member name
  <expr1>  Bit offset from the beginning of structure or union (decimal)
  <expr2>  Storage class of <sym> (decimal)
  <expr3>  Data type of <sym> (hex)
  <expr4>  Bit size of <sym> (decimal)
```

This pseudo-instruction indicates that <sym> is a bit field member.

**Tag declaration of structure, union, or enum type (end)**

```
.def <sym1>, val <expr1>, scl <expr2>, tag <sym2>, size <expr3>, endif
  <sym1,2> Tag name of structure, union or enum type in the C source
  <expr1>  Data size of <sym1> (decimal)
  <expr2>  102 (fixed)
  <expr3>  Data size of <sym1> (decimal)
```

This pseudo-instruction indicates that the declared tag name is the structure, union or enum type of <sym1>, and that the member information is ended in the immediately preceding .def pseudo-instruction.

**Structure, union or enum-type variable****(automatic variable, argument, structure or union member)**

```
.def <sym1>, val <expr1>, scl <expr2>, tag <sym2>, size <expr3>, type <expr4>, endif
  <sym1>   Symbol name in the C source (variable name/member name)
  <sym2>   Tag name of the structure, union or enum-type variable indicated by <sym1>
  <expr1>  Automatic variable/argument: Offset from the stack pointer (decimal)
           Structure or union member: Offset from the beginning of structure or union (decimal)
  <expr2>  Storage class of <sym1> (decimal)
  <expr3>  Data size of structure, union or enum type <sym2> (decimal)
  <expr4>  Data type of <sym1> (hex)
```

This pseudo-instruction indicates that <sym1> is the structure, union or enum-type data of structure/union automatic variable or structure/union member of tag name <sym2>.



**Structure, union or enum-type variable (static variable, global variable)**

```
.def <sym1>, val <sym2>, scl <expr1>, tag <sym3>, size <expr2>, type <expr3>, endef
<sym1>    Symbol name in the C source (variable name/member name)
<sym2>    Relocatable symbol name corresponding to <sym1>
<sym3>    Tag name of the structure, union or enum-type variable indicated by <sym2>
<expr1>   Storage class of <sym2> (decimal)
<expr2>   Data size of structure, union or enum type of <sym3> (decimal)
<expr3>   Data type of <sym2> (hex)
```

This pseudo-instruction indicates that <sym2> is a structure, union or enum type static variable or a structure or union global variable corresponding to the C source's variable name <sym1>.

**Array (automatic variable, argument, structure or union member)**

```
.def <sym>, val <expr1>, scl <expr2>, dim <expr_list>, size <expr3>, type <expr4>, endef
<sym>     Symbol name in the C source (variable name/member name)
<expr1>   Automatic variable/argument: Offset from the stack pointer (decimal)
           Structure or union member: Offset from the beginning of a structure or union (decimal)
           Argument (passed via register): Register number where the beginning element of the array
           is stored (decimal)
<expr2>   Storage class of <sym> (decimal)
<expr3>   Data size of array (decimal)
<expr4>   Data type of array element (hex)
<expr_list> List of values indicating the dimension of array (decimal, 4-dimension at maximum)
           Example: int array[2][3] → dim 2 3
```

This pseudo-instruction indicates that <sym> is the array data of an automatic array variable or a structure or union member.

**Array (static variable, global variable)**

```
.def <sym1>, val <sym2>, scl <expr1>, dim <expr_list>, size <expr2>, type <expr3>, endef
<sym1>    Symbol name in the C source (variable name/member name)
<sym2>    Relocatable symbol name corresponding to <sym1>
<expr1>   Storage class of <sym2> (decimal)
<expr2>   Data size of array (decimal)
<expr3>   Data type of array element (hex)
<expr_list> List of values indicating the array dimension (decimal, 4-dimension maximum)
```

This pseudo-instruction indicates that <sym2> is a static array or a global array variable that corresponds to the C source's variable name <sym1>.

**Structure, union or enum-type array (automatic variable, structure or union member)**

```
.def <sym1>, val <expr1>, scl <expr2>, tag <sym2>, dim <expr_list>, size <expr3>, type <expr4>, endef
<sym1>    Symbol name in the C source (variable name/member name)
<sym2>    Tag name of the structure, union or enum type indicated by <sym1>
<expr1>   Automatic variable/argument: Offset from the stack pointer (decimal)
           Structure or union member: Offset from the beginning of the structure or union (decimal)
<expr2>   Storage class of <sym1> (decimal)
<expr3>   Data size of the structure, union or enum-type array of <sym2> (decimal)
<expr4>   Data type of the array element of <sym1> (hex)
<expr_list> List of values indicating the array dimension (decimal, 4-dimension maximum)
```

This pseudo-instruction indicates that <sym1> is the structure, union or enum-type array data of tag name <sym2>.

**Structure, union or enum-type array (static variable, global variable)**

```
.def <sym1>,val <sym2>,scl <expr1>,tag <sym3>,dim <expr_list>,size <expr2>,type <expr3>,endif
<sym1>   Symbol name in the C source (variable name/member name)
<sym2>   Relocatable symbol name corresponding to <sym1>
<sym3>   Tag name of the structure, union or enum-type variable indicated by <sym2>
<expr1>  Storage class of <sym2> (decimal)
<expr2>  Data size of the structure, union, or enum type array of <sym3> (decimal)
<expr3>  Data type of the array element of <sym2> (hex)
<expr_list> List of values indicating the array dimension (decimal, 4-dimension at maximum)
```

This pseudo-instruction indicates that <sym2> is a static array or a global array variable corresponding to the C source's variable name <sym1>.

**typedef (when using standard type)**

```
.def <sym>, scl <expr1>, type <expr2>, endif
<sym>   Name of data type newly defined by typedef
<expr1> Storage class of the defined data type (decimal)
<expr2> Original data type for the newly defined data type (hex)
```

This pseudo-instruction indicates that <sym> has been defined as a new data type by typedef.

**typedef (when using a nonstandard type)**

```
.def <sym1>, scl <expr1>, tag <sym2>, size <expr2>, type <expr3>, endif
<sym1>   Name of data type newly defined by typedef
<sym2>   Tag name of original structure, union or enum type for the newly defined data type
<expr1>  Storage class of the defined data type (decimal)
<expr2>  Data size of structure, union or enum type of <sym2> (decimal)
<expr3>  Data type of structure, union or enum type of <sym2> (hex)
```

This pseudo-instruction indicates that <sym1> has been defined as a new data type by typedef.

**User-defined label**

```
.def <sym1>, val <sym2>, scl <expr1>, type <expr2>, endif
<sym1>   User-defined label name
<sym2>   Local label name corresponding to the user-defined label name
<expr1>  Storage class of user-defined label name (decimal)
<expr2>  Data type of user-defined label name (hex)
```

This pseudo-instruction indicates that <sym1> is a user-defined label name corresponding to the local label <sym2> generated by the gcc33.

**Beginning and end of function or block**

```
.def <sym>, scl <expr>, type 0x0, endif
<sym>   "ent": Beginning of function
        "end": End of function
        "begin": Beginning of block
        "bend": End of block
<expr>  101: Beginning of function
        111: End of function
        100: Beginning of block
        110: End of block
```

This pseudo-instruction indicates that the current position is the beginning or end of a function or block.

**Values representing storage classes and data types**

The values representing storage classes and data types are defined as follows:

**Values of storage classes (scl)**

1	Automatic variable
2	Global symbol (function/variable)
3	Local symbol (function/variable)
4	Register variable
6	User-defined label
8	Structure member
9	Argument (passed via stack)
10	Structure tag
11	Union member
12	Union tag
13	Type defined by typedef
15	Enum-type tag
16	Enum-type member
17	Argument (passed via register)
18	Bit field
100	Start position of block (begin)
101	Start position of function (ent)
102	End of structure, union or enum type definition
110	End position of block (bend)
111	End position of function (end)

**Values of data types (type)**

A		B	
0x0	User-defined label	0x0	Any type other than pointer, function or array
0x1	void	0x1	Pointer
0x2	char	0x2	Function
0x3	short	0x3	Array
0x4	int		
0x5	long		
0x6	float		
0x7	double		
0x8	struct		
0x9	union		
0xA	enum		
0xB	Enum member		
0xC	unsigned char		
0xD	unsigned short		
0xE	unsigned int		
0xF	unsigned long		

The values of data types are calculated using the equation below:

$$A + (B(1) \ll 4) + (B(2) \ll 6) \dots + (B(N) \ll (2 + 2 * N))$$

For example, a function that returns a pointer to a structure takes on the following value:

$$0x8 + (0x2 \ll 4) + (0x1 \ll 6) = 0x68$$

## **6.7 *Functions of gcc33 and Usage Precautions***

---

- The calloc function cannot be used in this compiler.
- For other details about the gcc, refer to the documents for the gcc.  
The documents can be acquired from the GNU mirror sites located in various places around the world through Internet, etc.

## Chapter 7 Emulation Library

This chapter explains the emulation library of the E0C33 Family C Compiler Package, including floating-point number and integral remainder calculating functions.

### 7.1 Overview

The E0C33 Family C Compiler Package contains a floating-point calculation library (fp.lib) that supports the arithmetic operation, comparison, and type conversion of single-precision (32-bit) and double-precision (64-bit) floating-point numbers which conforms to IEEE format, and an integral remainder calculation library (idiv.lib) that supports the remainder calculation of integers.

The C Compiler gcc33 calls up functions from these libraries when a floating-point number or integral remainder calculation is performed.

Since library functions exchange data via a designated general-purpose register, they can be called from an assembly source.

To use emulation library functions, specify fp.lib and idiv.lib as libraries during linkage. Be sure to specify these libraries in the order of fp.lib and idiv.lib.

All emulation library functions have been created and tuned by an assembly source.

### 7.2 Floating-point Calculation Library (fp.lib)

#### 7.2.1 Function List

Table 7.2.1.1 below lists the floating-point calculation library (fp.lib) functions.

Table 7.2.1.1 Floating-point calculation library (fp.lib) functions

Classification	Function name	Functionality	
Double-precision floating-point calculation	__adddf3	Addition	(%r11, %r10) ← (%r13, %r12) + (%r15, %r14)
	__subdf3	Subtraction	(%r11, %r10) ← (%r13, %r12) - (%r15, %r14)
	__muldf3	Multiplication	(%r11, %r10) ← (%r13, %r12) * (%r15, %r14)
	__divdf3	Division	(%r11, %r10) ← (%r13, %r12) / (%r15, %r14)
	__negdf2	Sign inversion	(%r11, %r10) ← -(%r13, %r12)
Single-precision floating-point calculation	__addsf3	Addition	%r10 ← %r12 + %r13
	__subsf3	Subtraction	%r10 ← %r12 - %r13
	__mulsf3	Multiplication	%r10 ← %r12 * %r13
	__divsf3	Division	%r10 ← %r12 / %r13
	__negsf2	Sign inversion	%r10 ← -%r12
Type conversion	__fixunssf2	double → unsigned int	%r10 ← (unsigned int) (%r13, %r12)
	__fixdfsi	double → int	%r10 ← (int) (%r13, %r12)
	__floatsidf	int → double	(%r11, %r10) ← (double) %r12
	__fixunssf2	float → unsigned int	%r10 ← (unsigned int) %r12
	__fixfsi	float → int	%r10 ← (int) %r12
	__floatsisf	int → float	%r10 ← (float) %r12
	__truncdfsf2	double → float	%r10 ← (float) (%r13, %r12)
	__extendsfdf2	float → double	(%r11, %r10) ← (double) %r12
Floating-point comparison	__fcmpd	Comparison of double type	%psr change ← (%r13, %r12) - (%r15, %r14)
	__fcmps	Comparison of float type	%psr change ← %r12 - %r13

- If the operation resulted in an overflow or underflow, infinity or negative infinity (see next section) is returned.
- The comparison function changes the C, V, Z or N flag of the PSR depending on the result of op1 - op2, as shown below. Other flags are not changed.

Comparison result	C	V	Z	N
op1 > op2	0	0	0	0
op1 = op2	0	0	1	0
op1 < op2	1	0	0	1

- During type conversion, values are rounded.

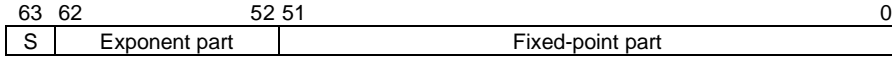
## 7.2.2 Floating-point Format

The C Compiler gcc33 supports the float type (single-precision, 32 bit) and the double type (double-precision, 64 bit) floating-point numbers conforming to IEEE standards.

The following shows the internal format of floating-point numbers.

### Format of double-precision floating-point number

The real number of the double type consists of 64 bits, as shown below.



Bit 63: Sign bit (1 bit)

Bits 62–52: Exponent part (11 bits)

Bits 51–0: Fixed-point part (52 bits)

When this type of value is stored in a register, it occupies two registers. For example, the result of a floating-point calculation is stored in the R11 and R10 registers.

R11 register: Sign bit, exponent part, and 20 high-order bits of fixed-point part (51:32)

R10 register: 32 low-order bits of fixed-point part (31:0)

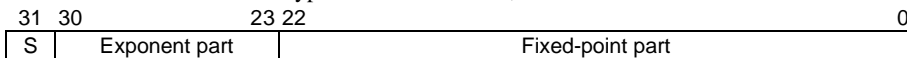
The following shows the relationship of the effective range, floating-point representation, and internal data of the double type.

+0:	0. 0e+0	0x00000000 00000000
-0:	-0. 0e+0	0x80000000 00000000
Maximum normalized number:	1. 79769e+308	0x7fefffff ffffffff
Minimum normalized number:	2. 22507e-308	0x00100000 00000000
Maximum unnormalized number:	2. 22507e-308	0x000fffff ffffffff
Minimum unnormalized number:	4. 94065e-324	0x00000000 00000001
Infinity:		0x7ff00000 00000000
Negative infinity:		0xfff00000 00000000

Values 0x7ff00000 00000001 to 0x7fffffff ffffffff and 0xfff00000 00000001 to 0xffffffff ffffffff are not recognized as numeric values.

### Format of single-precision floating-point number

The real number of the float type consists of 32 bits, as shown below.



Bit 31: Sign bit (1 bit)

Bits 30–23: Exponent part (8 bits)

Bits 22–0: Fixed-point part (23 bits)

The float type data can be stored in one register.

The following shows the relationship of the effective range, floating-point representation, and internal data of the float type.

+0:	0. 0e+0f	0x00000000
-0:	-0. 0e+0f	0x80000000
Maximum normalized number:	3. 40282e+38f	0x7f7fffff
Minimum normalized number:	1. 17549e-38f	0x00800000
Maximum unnormalized number:	1. 17549e-38f	0x007fffff
Minimum unnormalized number:	1. 40129e-45f	0x00000001
Infinity:		0x7f800000
Negative infinity:		0xff800000

Values 0x7f800001 to 0x7fffffff and 0xff800001 to 0xffffffff are not recognized as numeric values.

### Note

The floating-point numbers in the gcc33 differ from the IEEE-based FPU in precision and functionality, including the manner in which infinity is handled.

## 7.3 Integral Remainder Calculation Library (*idiv.lib*)

Table 7.3.1 below lists the integral remainder calculation library (*idiv.lib*) functions.

Table 7.3.1 Integral remainder calculation library (*idiv.lib*) functions

Classification	Function name	Functionality	
Integral division	<code>__divsi3</code>	Signed integral division	<code>%r10 ← %r12 / %r13</code>
	<code>__udivsi3</code>	Unsigned integral division	<code>%r10 ← %r12 / %r13</code>
Remainder	<code>__modsi3</code>	Signed modulo arithmetic	<code>%r10 ← %r12 % %r13</code>
	<code>__umodsi3</code>	Unsigned modulo arithmetic	<code>%r10 ← %r12 % %r13</code>

- These functions do not check the value of the input register. For this reason, if the R13 register is set to 0, a zero-division exception occurs in the `div0s` or `div0u` instruction in the function.

## 7.4 Floating-point Calculation Library (*fpp.lib*)

The *fpp.lib* library in this package consists of the same functions as the *fp.lib* floating-point calculation library. However, the following functions in the *fpp.lib* have higher operating accuracy than those of the *fp.lib*.

`__adddf3`, `__subdf3`, `__muldf3`, `__divdf3`

The functions in the *fp.lib* discard the digits under the effective range of the fixed-point part, while these four functions in the *fpp.lib* calculate the under part and reflect the rounded off results to the LSB of the fixed-point part.

They are effective when a higher operating accuracy is required in arithmetic functions such as `sin`, `cos` and `tan`.

## Chapter 8 ANSI Library

This chapter explains the ANSI library included in the E0C33 Family C Compiler Package.

### 8.1 Overview

---

The E0C33 Family C Compiler Package contains an ANSI library.

Each function in this library has ANSI-standard functionality. However, some file-related functions are dummy functions due to embedded microcomputer specifications.

There are five types of ANSI library files, which are installed in the lib directory.

io.lib lib.lib math.lib ctype.lib string.lib

In addition, the following header files which contain definitions of each function are installed in the include directory.

stdio.h stdlib.h time.h math.h errno.h float.h limits.h ctype.h string.h stdarg.h

Refer to the sample file located in the "cc33\sample\ansilib\" directory for how to use the library functions.

Note: When specifying library files including emulation library files (fp.lib, idiv.lib) during linkage, please follow the specification priority shown below:

io.lib lib.lib math.lib ctype.lib string.lib fp.lib idiv.lib

The file io.lib contains functions that call the lib.lib or math.lib functions. Also, lib.lib calls the math.lib functions. Reference between library files is only valid for functions in the library file that is specified later. Therefore, if library files are arranged in a different order, a warning may be generated during linkage.



## 8.2 ANSI Library Function List

### 8.2.1 Input/Output Functions (io.lib)

The table below lists the input/output functions included in io.lib.

Table 8.2.1.1 Input/output functions

Header file: `stdio.h`

Function	Functionality
<code>FILE *fopen(char *filename, char *mode);</code>	Dummy
<code>FILE *freopen(char *filename, char *mode, FILE *stream);</code>	Dummy
<code>int fclose(FILE *stream);</code>	Dummy
<code>int fflush(FILE *stream);</code>	Dummy
<code>int fseek(FILE *stream, long int offset, int origin);</code>	Dummy
<code>long int ftell(FILE *stream);</code>	Dummy
<code>void rewind(FILE *stream);</code>	Dummy
<code>int fgetpos(FILE *stream, fpos_t *ptr);</code>	Dummy
<code>int fsetpos(FILE *stream, fpos_t *ptr);</code>	Dummy
<code>size_t fread(void *ptr, size_t size, size_t count, FILE *stream);</code>	Input array element from stdin.
<code>size_t fwrite(void *ptr, size_t size, size_t count, FILE *stream);</code>	Output array element to stdout.
<code>int fgetc(FILE *stream);</code>	Input one character from stdin.
<code>int getc(FILE *stream);</code>	Input one character from stdin.
<code>int getchar( );</code>	Input one character from stdin.
<code>int ungetc(int c, FILE *stream);</code>	Push one character back to input buffer.
<code>char *fgets(char *s, int n, FILE *stream);</code>	Input character string from stdin.
<code>char *gets(char *s);</code>	Input character string from stdin.
<code>int fputc(int c, FILE *stream);</code>	Output one character to stdout.
<code>int putchar(int c);</code>	Output one character to stdout.
<code>int fputs(char *s, FILE *stream);</code>	Output character string to stdout.
<code>int puts(char *s);</code>	Output character string to stdout.
<code>int remove(char *filename);</code>	Dummy
<code>int rename(char *oldname, char *newname);</code>	Dummy
<code>void setbuf(FILE *stream, char *buf);</code>	Dummy
<code>int setvbuf(FILE *stream, char *buf, int type, size_t size);</code>	Dummy
<code>FILE *tmpfile( );</code>	Dummy
<code>char *tmpnam(char *buf);</code>	Dummy
<code>int feof(FILE *stream);</code>	Dummy
<code>int ferror(FILE *stream);</code>	Dummy
<code>void clearerr(FILE *stream);</code>	Dummy
<code>void perror(char *s);</code>	Output error information to stdout.
<code>int fscanf(FILE *stream, char *format, ...);</code>	Input from stdin with format specified.
<code>int scanf(char *format, ...);</code>	Input from stdin with format specified.
<code>int sscanf(char *s, char *format, ...);</code>	Input from character string with format specified.
<code>int fprintf(FILE *stream, char *format, ...);</code>	Output to stdout with format specified.
<code>int printf(char *format, ...);</code>	Output to stdout with format specified.
<code>int sprintf(char *s, char *format, ...);</code>	Output to array with format specified.
<code>int vfprintf(FILE *stream, char *format, va_list arg);</code>	Output conversion result to stdout.
<code>int vprintf(FILE *stream, char *format, va_list arg);</code>	Output conversion result to stdout.
<code>int vsprintf(char *s, char *format, va_list arg);</code>	Output conversion result to array.

## 8.2.2 Utility Functions (lib.lib)

The table below lists the utility functions included in lib.lib.

Table 8.2.2.1 Utility functions

Header file: **stdlib.h**

Function	Functionality
<code>void *malloc(size_t size);</code>	Allocate area.
<code>void *calloc(size_t elt_count, size_t elt_size);</code>	Allocate array area.
<code>void free(void *ptr);</code>	Clear area.
<code>void *realloc(void *ptr, size_t size);</code>	Change area size.
<code>int system(char *command);</code>	Dummy
<code>void exit(int status);</code>	Terminate program normally.
<code>void abort( );</code>	Terminate program abnormally.
<code>int atexit(void (*func)(void));</code>	Dummy
<code>char *getenv(char *str);</code>	Dummy
<code>void *bsearch(void *key, void *base, size_t count, size_t size, int (*compare)(void *, void *));</code>	Binary search.
<code>void qsort(void *base, size_t count, size_t size, int (*compare)(void *, void *));</code>	Quick sort.
<code>int abs(int x);</code>	Return absolute value (int type).
<code>long int labs(long int x);</code>	Return absolute value (long type).
<code>div_t div(int n, int d);</code>	Divide int type.
<code>ldiv_t ldiv(int n, int d);</code>	Divide long type.
<code>int rand( );</code>	Return pseudo-random number.
<code>void srand(unsigned int seed);</code>	Set seed of pseudo-random number.
<code>long int atol(char *str);</code>	Convert character string into long type.
<code>int atoi(char *str);</code>	Convert character string into int type.
<code>double atof(char *str);</code>	Convert character string into double type.
<code>double strtod(char *str, char **ptr);</code>	Convert character string into double type.
<code>long int strtol(char *str, char **ptr, int base)</code>	Convert character string into long type.
<code>unsigned long int strtoul(char *str, char **ptr, int base);</code>	Convert character string into unsigned long type.

## 8.2.3 Date and Time Functions (lib.lib)

The table below lists the date and time functions included in lib.lib.

Table 8.2.3.1 Date and time functions

Header file: **time.h**

Function	Functionality
<code>clock_t clock( );</code>	Dummy
<code>char *asctime(struct tm *ts);</code>	Dummy
<code>char *ctime(time_t *timeptr);</code>	Dummy
<code>double difftime(time_t t1, time_t t2);</code>	Dummy
<code>struct tm *gmtime(time_t *t);</code>	Convert calendar time to standard time.
<code>struct tm *localtime(time_t *t);</code>	Dummy
<code>time_t mktime(struct tm *tmptr);</code>	Convert standard time to calendar time.
<code>time_t time(time_t *tptr);</code>	Return current calendar time.

## 8.2.4 Mathematical Functions (math.lib)

The table below lists the mathematical functions included in math.lib.

Table 8.2.4.1 Mathematical functions

Header files: **math.h, errno.h, float.h, limits.h**

Function	Functionality
double fabs(double x);	Return absolute value (double type).
double ceil(double x);	Round up double-type decimal part.
double floor(double x);	Round down double-type decimal part.
double fmod(double x, double y);	Calculate double-type remainder.
double exp(double x);	Exponentiate ( $e^x$ ).
double log(double x);	Calculate natural logarithm.
double log10(double x);	Calculate common logarithm.
double frexp(double x, int *nptr);	Return mantissa and exponent of floating-point number.
double ldexp(double x, int n);	Return floating-point number from mantissa and exponent.
double modf(double x, double *nptr);	Return integer and decimal parts of floating-point number.
double pow(double x, double y);	Calculate $x^y$ .
double sqrt(double x);	Calculate square root.
double sin(double x);	Calculate sine.
double cos(double x);	Calculate cosine.
double tan(double x);	Calculate tangent.
double asin(double x);	Calculate arcsine.
double acos(double x);	Calculate arccosine.
double atan(double x);	Calculate arctangent.
double atan2(double y, double x);	Calculate arctangent of $y/x$ .
double sinh(double x);	Calculate hyperbolic sine.
double cosh(double x);	Calculate hyperbolic cosine.
double tanh(double x);	Calculate hyperbolic tangent.

## 8.2.5 Character Functions (string.lib)

The table below lists the character functions included in string.lib.

Table 8.2.5.1 Character functions

Header file: **string.h**

Function	Functionality
char *memchr(char *s, int c, int n);	Return specified character position in the storage area.
int memcmp(char *s1, char *s2, int n);	Compare storage areas.
char *memcpy(char *s1, char *s2, int n);	Copy the storage area.
char *memmove(char *s1, char *s2, int n);	Copy the storage area (overlapping allowed).
char *memset(char *s, int c, int n);	Set character in the storage area.
char *strcat(char *s1, char *s2);	Concatenate character strings.
char *strchr(char *s, int c);	Return specified character position found first in the character string.
int strcmp(char *s1, char *s2);	Compare character strings.
char *strcpy(char *s1, char *s2);	Copy character string.
size_t *strcspn(char *s1, char *s2);	Return number of characters from the beginning of the character string until the specified character appears (multiple choices).
char *strerror(int code);	Return error message character string.
size_t strlen(char *s);	Return length of character string.
size_t strncat(char *s1, char *s2, int n);	Concatenate character strings (number of characters specified).
int strncmp(char *s1, char *s2, int n);	Compare character strings (number of characters specified).
char *strncpy(char *s1, char *s2, int n);	Copy the character string (number of characters specified).
char *strpbrk(char *s1, char *s2);	Return specified character position (multiple choices) found first in the character string.
char *strrchr(char *s, int c);	Return specified character position found last in the character string.
size_t strspn(char *s1, char *s2);	Return number of characters from the beginning of the character string until the non-specified character appears (multiple choices).
char *strstr(char *s1, char *s2);	Return position where the specified character string appeared first.
char *strtok(char *s1, char S2);	Divide the character string into tokens.

\* All functions except strerror have been created and tuned by an assembly source.

## 8.2.6 Character Type Determination/Conversion Functions (ctype.lib)

The table below lists the character functions included in ctype.lib.

Table 8.2.6.1 Character type determination/conversion functions

Header file: **ctype.h**

Function	Functionality
int isalnum(char c);	Determine character type (decimal or alphabet).
int isalpha(char c);	Determine character type (alphabet).
int iscntrl(char c);	Determine character type (control character).
int isdigit(char c);	Determine character type (decimal).
int isgraph(char c);	Determine character type (graphic character).
int islower(char c);	Determine character type (lowercase alphabet).
int isprint(char c);	Determine character type (printable character).
int ispunct(char c);	Determine character type (delimiter).
int isspace(char c);	Determine character type (null character).
int isupper(char c);	Determine character type (uppercase alphabet).
int isxdigit(char c);	Determine character type (hexadecimal).
int tolower(char c);	Convert character type (uppercase alphabet → lowercase).
int toupper(char c);	Convert character type (lowercase alphabet → uppercase).

## 8.2.7 Variable Argument Macros (stdarg.h)

The table below lists the variable argument macros defined in stdarg.h.

Table 8.2.7.1 Variable argument macros

Header file: **stdarg.h**

Macro	Functionality
void va_start(va_list ap, type lastarg);	Initialize the variable argument group.
type va_arg(va_list ap, type);	Return the actual argument.
void va_end(va_list ap);	Return normally from the variable argument function.

## 8.3 Declaring and Initializing Global Variables

The ANSI library functions reference the global variables listed in Table 8.3.1. Since these variables are not defined in the library, be sure to declare and initialize them before calling a library function in the C source program.

Table 8.3.1 Global variables required of declaration

Global variable	Initial setting	Related header file/function
<b>FILE _iob[FOPEN_MAX +1];</b> FOPEN_MAX=3, Defined in stdio.h File structure data for standard input/output streams	_iob[N]._flg = _UGETN; _iob[N]._buf = 0; _iob[N]._fd = N; (N=0-2) _iob[0]: Input data for stdin _iob[1]: Output data for stdout _iob[2]: Output data for stderr	stdio.h, smcvals.h fgets, fread, fscanf, getc, getchar, gets, scanf, ungetc, perror, fprintf, fputs, fwrite, printf, putc, putchar, puts, vfprintf, vprintf
<b>FILE *stdin;</b> Pointer to standard input/output file structure data _iob[0]	stdin = &_iob[0];	stdio.h fgets, fread, fscanf, getc, getchar, gets, scanf, ungetc
<b>FILE *stdout;</b> Pointer to standard input/output file structure data _iob[1]	stdout = &_iob[1];	stdio.h fprintf, fputs, fwrite, printf, putc, putchar, puts, vfprintf, vprintf
<b>FILE *stderr;</b> Pointer to standard input/output file structure data _iob[2]	stderr = &_iob[2];	stdio.h fprintf, fputs, fwrite, printf, perror, putc, putchar, puts, vfprintf, vprintf
<b>int errno;</b> Variable to store error number	errno = 0;	errno.h fopen, freopen, fseek, fsetpos, perror, remove, rename, tmpfile, tmpnam, fprintf, printf, sprintf, vprintf, vfprintf, fscanf, scanf, sscanf atof, atoi, calloc, div, ldiv, malloc, realloc, strtod, strtol, strtoul acos, asin, atan2, ceil, cos, cosh, exp, fabs, floor, fmod, frexp, ldexp, log, log10, modf, pow, sin, sinh, sqrt, tan
<b>unsigned int seed;</b> Variable to store seed of random number	seed = 1;	stdlib.h rand, srand
<b>unsigned char * _STACK_TOP;</b> Stack top address	_STACK_TOP = (unsigned char *) <Stack top address>;	stdlib.h calloc, free, malloc, realloc
<b>unsigned char * _STACK_BOTTOM;</b> Stack bottom address	_STACK_BOTTOM = (unsigned char *) <Stack bottom address>;	stdlib.h calloc, free, malloc, realloc
<b>unsigned char *ucNxtA1cP;</b> Pointer that indicates the heap area allocated next	ucNxtA1cP = (unsigned char *) <Stack bottom address>;	stdlib.h calloc, free, malloc, realloc
<b>unsigned char *ucBefA1cP;</b> Pointer that indicates the beginning of previously allocated heap area	ucBefA1cP = (unsigned char *) NULL;	stdlib.h calloc, free, malloc, realloc
<b>unsigned char *end_alloc;</b> Pointer that indicates the end address of heap area	end_alloc = (unsigned char *) <Stack top address>;	stdlib.h calloc, malloc, realloc
<b>time_t gm_sec;</b> Elapsed time of timer function in seconds from 0:00:00 on January 1, 1970	gm_sec = -1;	time.h time

\* For an example of a source file that declares and initializes these global variables, refer to lib.c in the sample\ansilib\ directory.

## 8.4 Lower-level Functions

---

The following three functions (read, write, and \_exit) are the lower-level functions called by a library function. Since these functions depend on hardware, they are not provided in the library. If these functions are desired, define them in the user program.

For an example of a C source file that defines these functions, refer to sys.c in the sample\ansilib\ directory.

### 8.4.1 "read" Function

#### Contents of read function

- Format:           **int read(int fd, char \*buf, int nbytes);**
- Argument:        int fd;        File descriptor denoting input  
   When called from a library function, 0 (stdin) is passed.
- char \*buf;     Pointer to the buffer that stores input data
- int nbytes;    Number of bytes transferred
- Functionality:   This function reads up to nbytes of data from the user-defined input buffer, and stores it in the buffer indicated by buf.
- Returned value:  Number of bytes actually read from the input buffer  
   If the input buffer is empty (EOF) or nbytes = 0, 0 is returned.  
   If an error occurs, -1 is returned.
- Library functions that call the read function:
- Direct call:  fread, getc, \_doscan (\_doscan is a scanf-series internal function)
- Indirect call: fgetc, fgets, getchar, gets (calls getc)  
   scanf, fscanf, sscanf (calls \_doscan)

#### Definition of input buffer

- Format:           **unsigned char READ\_BUF[65];**   (Variable name is arbitrary; size is fixed to 65 bytes)  
                   **unsigned char READ\_EOF;**
- Buffer contents:  The size of the input data (1 to max. 64) is stored at the beginning of the buffer (READ\_BUF[0]). 0 denotes EOF.  
                   The input data is stored in READ\_BUF[1], and the following locations.  
                   READ\_EOF stores the status that indicates whether EOF is reached.

#### Precautions on using a simulated I/O

When using the debugger's simulated I/O, define in the read function the global label "READ\_FLASH" that is required for the debugger to update the input buffer, then create the function so that new data will be read into the input buffer at that position. (For details about the simulated I/O function, refer to the chapter where the debugger is discussed.)

## 8.4.2 "write" Function

### Contents of write function

Format: **int write(int fd, char \*buf, int nbytes);**

Argument: int fd; File descriptor denoting output  
When called from a library function, 1 (stdout) or 2 (stderr) is passed.  
char \*buf; Pointer to the buffer that stores output data  
int nbytes; Number of transferred bytes

Functionality: The data stored in the buffer indicated by buf is written as much as indicated by nbytes to the user-defined output buffer.

Returned value: Number of bytes actually written to the output buffer  
If data is written normally, nbytes is returned.  
If a write error occurs, a value other than nbytes is returned.

Library function that calls the write function:

Direct call: fwrite, putc, \_doprint (\_doprint is printf-series internal function)

Indirect call: fputc, fputs, putchar, puts (calls putcc)

printf, fprintf, sprintf, vprintf, vfprintf (calls \_doprint)

perror (calls fprintf)

### Definition of output buffer

Format: **unsigned char WRITE\_BUF[65];** (Variable name is arbitrary; size is fixed to 65 bytes)

Buffer content: The size of the output data (1 to max. 64) is stored at the beginning of the buffer (WRITE\_BUF[0]). 0 denotes EOF.

The output data is stored in WRITE\_BUF[1], and the following locations.

### Precautions on using simulated I/O

When using the debugger's simulated I/O, define in the write function the global label "WRITE\_FLASH" that is required for the debugger to update the output buffer, and create a function so that data will be output from the output buffer at that position. (For details about the simulated I/O function, refer to the chapter where the debugger is discussed.)

## 8.4.3 "\_exit" Function

### Contents of \_exit function

Format: **void \_exit(void);**

Functionality: Performs program terminating processing.

Argument/ Returned value: None

Library function that calls \_exit function:

Direct call: abort, exit

## Chapter 9 Preprocessor

This chapter describes the functions of the Preprocessor pp33.

### 9.1 Functions

The preprocessor pp33 (hereafter called "pp33") is the C compiler package's first tool to process the assembly source file; therefore, it provides the assembler as33 with additional functions. It expands the additional function part described in the assembly source file to mnemonics that can be assembled. The functions provided by the pp33 are as follows:

- Macro definition and macro invocation
- Definition of Define name
- Operators
- Insertion of other file
- Conditional assembly
- Addition of debugging information for assembly source display

### 9.2 Input/Output Files

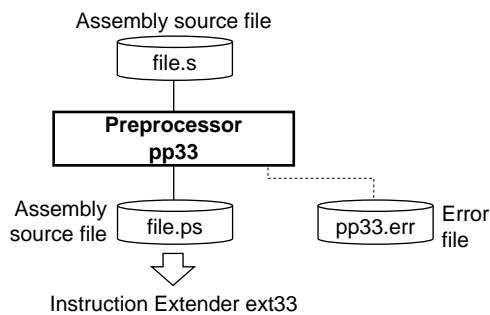


Fig. 9.2.1 Flowchart

#### 9.2.1 Input File

##### Source file

File format:	Text file
File name:	<file name>.s (Other extensions than ".s" can be used.)
Description:	File in which the assembly source program is described. Instructions for the pp33 and the extended instructions can be described there.

#### 9.2.2 Output Files

##### Assembly source file

File format:	Text file
File name:	<file name>.ps (The <file name> is the same as that of the input file.)
Output destination:	Current directory
Description:	File in which instructions for the pp33 are expanded into an assembling format.

##### Error file

File format:	Text file
File name:	pp33.err
Output destination:	Current directory
Description:	File that is output when the startup option (-e) is specified, and describes the contents which the pp33 delivers through the Standard Output (stdout), such as an error message.



## 9.3 Starting Method

---

### 9.3.1 Startup Format

#### General form of command line

**pp33 ^ [<startup option>] ^ [<file name>]**

^ denotes a space.

[ ] indicates the possibility to omit.

<file name>: Specify an assembly source file name including the extension (.s).

#### Operations on work bench

Select options and a source file, then click the [PP33] button.

In the command line, only one source file can be specified at a time.

The wb33 allows multiple files to be selected, in which case the pp33 is executed as many times as the number of files selected.

### 9.3.2 Startup Options

The pp33 comes provided with the following three types of startup options:

#### -d <Define name>

Function: Definition of Define name

Specification on wb33: Enter in the [define] text box.

Explanation:

- Works in the same manner as you describe "#define <Define name>" at top of the source. It is an option to control the conditional assembly at the startup. However, unlike the #define definition, it does not perform replacement in the source.
- One or more spaces are necessary between -d and the <Define name>.
- Refer to Section 9.5.2 for formats and restrictions of definable names.
- To define two or more Define names, repeat the specification of "-d <Define name>". For the wb33, separate each <Define name> with a comma (,) as you input them.

#### -g

Function: Addition of debugging information

Specification on wb33: Check [debug info].

Explanation:

- Creates an output file containing debugging information.
- Always specify this function when you perform the assembly source level debugging.
- Refer to Section 9.7 for debugging information.

#### -e

Function: Output of error files

Specification on wb33: None

Explanation:

- Delivers also in a file (pp33.err) the contents that are output by the pp33 via the Standard Output (stdout), such as error messages.

When entering options in the command line, you need to place one or more spaces before and after the option.

Example: c:\cc33\pp33 -g -e -d TEST1 -d TEST2 test.s

## 9.4 Messages

---

The pp33 delivers its messages through the Standard Output (stdout).

If the pp33 is started up by using the wb33's [PP33] button, the message is output to "wb33.err". When execution is completed, a message is displayed in the output window (default).

### End message

The pp33 outputs only the following end message when it ends normally.

```
Pre Processor Completed
```

### Usage output

If no file name was specified or an option was not specified correctly, the pp33 ends after delivering the following message concerning the usage:

```
Pre Processor 33 Ver x.xx
Copyright (C) SEIKO EPSON CORP. 199x
Usage:
  pp33 [options] filename
Options:
  -e : produce log file (pp33.err)
  -g : generate debug information in output file
  -d string : define string
Output:
  Assembler source file for ext33 (.ps)
Example:
  pp33 -e -g -d TYPE1 test.s
```

### When error/warning occurs

If an error is produced, an error message will appear before the end message shows up.

```
Example: test.s(431): Error: Invalid Syntax.
         Pre Processor Completed
```

In the case of an error, the pp33 ends without creating an output file.

If a warning is issued, a warning message will appear before the end message shows up.

```
Example: test.s(211): Warning: Multi define symbol.
         Pre Processor Completed
```

In the case of a warning, the pp33 ends after creating an output file.

For details on errors and warnings, refer to Section 9.12 "Error/Warning Messages".

## 9.5 Preprocessor Pseudo-Instructions

The principal mission of the pp33 is to expand the preprocessor pseudo-instructions, explained below, to mnemonics that can be processed by the Assembler as33.

For clear discrimination from the assembler pseudo instructions, the preprocessor pseudo-instructions all begin with a sharp (#). Describe the instructions always from top of the lines.

The pseudo-instructions in themselves are all in lowercase characters only. Parameters can use both uppercase and lowercase characters, which are discriminated, respectively.

The lines of preprocessor pseudo-instructions also follow the notation rules of statements (see Chapter 4).

### 9.5.1 Include Instruction (#include)

The include instruction inserts the contents of a file in any location of a source file. It results useful when the same source is shared in common among several source files.

#### Instruction format

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

- A drive name or path name can as well be specified as the <file name>.
- One or more spaces are necessary between the instruction and the "<file name>".

Sample descriptions:

```
#include "sample.def"
```

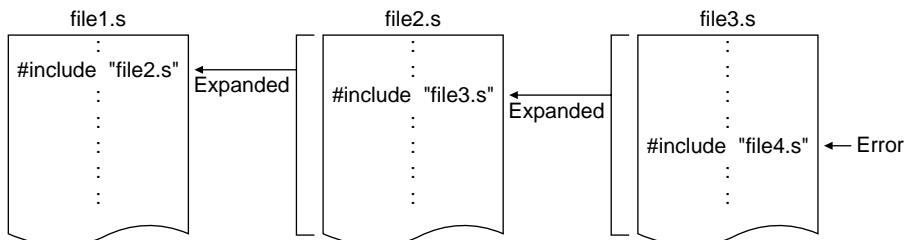
#### Expansion rule

The specified file is inserted in the location where #include was described.

For sample expansions, refer to Section 9.10 "Sample Executions".

#### Precautions

- Only files created in text file format can be inserted.
- Nesting is limited to maximum 2 levels. If this limit is surpassed, an error will result.



"file3.s" can be included in "file2.s", but "file4.s" cannot be included in "file3.s".

Fig. 9.5.1.1 Nesting levels of include

- When adding a relative path to the file name, specify the relative path from the directory in which the current source file exists.

## 9.5.2 Define Instruction (#define)

Any substitute character string can be left defined as a Define name by the define instruction (#define), and the details of that definition can be referred to from various parts of the program using the Define name.

### Instruction format

```
#define <Define name> [<Substitute character string>]
```

<Define name>:

- The first character is limited to a–z, A–Z and \_.
- The second and the subsequent characters can use a–z, A–Z, 0–9 and \_.
- Uppercase and lowercase characters are discriminated.
- One or more spaces or tab settings are necessary between the instruction and the Define name.

<Substitute character string>:

- The usable characters are limited to a–z, A–Z, 0–9, \_, % and . (period). They must not contain any space or comma (,).
- Values, operators, mnemonics, labels, and register names also can be specified.
- Uppercase and lowercase characters are discriminated.
- One or more spaces or tab settings are necessary between the Define name and the substitute character string.
- The substitute character string can be omitted. In that case, NULL is defined in lieu of the substitute character string. It can be used for the conditional assembly instruction.

Sample definitions:

```
#define TYPE1
#define L1 LABEL_01
#define li ld.w
#define r1 %r1
#define Mr1 [%r1] ...Error [ ] and [ ]+ cannot be used.
```

### Expansion rule

If a Define name defined appears in the source, the pp33 substitutes a defined character string for that Define name.

Sample expansion:

```
#define gp %r8
:
ld.w [gp], %r1 ...Expanded to "ld.w [%r8], %r1".
```

When a number is specified for the substitute character string, the following rule is applied:

- The pp33 converts the defined number into a signed 32-bit data and expands it as a hexadecimal number.
- #define allows the substitute character strings to describe in an expression using operators. The Define names that have been defined can also be used as terms of the expression.

Sample expansion:

```
Before expansion
#define A 0x12
#define B A*2
:
ld.w %r2, A+B ...Expanded to "ld.w %r2, 0x36".
```

**Precautions**

- The pp33 only permits back reference of a Define name. The definition needs to have been completed before making the reference.
- Once a Define name is defined, it cannot be canceled. However, redefinition can be made using a Define name.

Example: #define XH %ahr  
 #define XHigh XH  
 ld.w XHigh, %r1 ...Expanded to "ld.w %ahr, %r1".

- When the same Define name is defined twice or more, a warning message will appear and the redefined character string is validated.
- No other characters than delimiters (space, tab, line feed, and comma) can be added before and after a Define name in the source, unless they are enclosed in [ ] or [ ]+. However, a symbol mask (@..) described behind is valid.

Examples: #define H %ah  
 ld.w Hr, %r1 ;Hr = %ahr? ...Specification like this is invalid.  
 #define L LABEL  
 ext L@h ...Replaced with "ext LABEL@h".

- The pp33 does not check the validity of a statement following the replacement of the character string.
- The pp33 handles the defined numbers as 32-bit data. If the specified number or the calculation result is a negative value, it is delivered as a decimal number with a minus sign. If the value is positive, it is delivered as a hexadecimal number. Pay attention to the immediate data size, especially when it has a minus value.

Example: #define NUM -1 ...-1 is handled as 0xffffffff.  
 ld.w %r1, NUM ...It will be expanded as "ld.w %r1, -1".  
 ld.w %r1, NUM^L ...It will be expanded as "ld.w %r1, 0x3f".

### 9.5.3 Macro Instructions (#macro ... #endm)

Any statement string can be left defined as a macro using the macro instruction (#macro), and the content of that definition can be invoked from different parts of the program with the macro name. Unlike a subroutine, the part that is invoking a macro is replaced with the content of the definition by the pp33.

#### Instruction format

```
#macro    <Macro name>  [$1] [$2] . . . [$32]
          <Statement string>
```

```
#endm
```

<Macro name>:

- The first character is limited to a–z, A–Z and \_.
- The second and the subsequent characters can use a–z, A–Z, 0–9, and \_.
- Uppercase and lowercase characters are discriminated.
- One or more spaces or tab settings are necessary between the instruction and the macro name.

\$1–\$32:

- Dummy parameter symbols for macro definition. They are described when a macro to be defined needs parameters. Not more than 32 symbols can be specified.
- No other symbols than \$1 to \$32 can be used. You need to begin always with \$1 and to arrange them in an ascending order (\$1 → \$32).
- One or more spaces or tab settings are necessary between the macro name and \$1. When describing multiple parameters, a comma (,) is necessary between one parameter and another.

<Statement string>:

- The following statements can be described:
  - Basic instruction (mnemonic and operand)
  - Extended instruction (see Instruction Extender)
  - Conditional assembly instruction
  - Internal branch label\*
  - Comments
- The following statements cannot be described:
  - Preprocessor pseudo-instructions (excluding conditional assembly instruction)
  - Assembler pseudo-instructions
  - Other labels than internal branch labels
  - Macro invocation

\* Internal branch label

A macro is spread over to several locations in the source. Therefore, if you describe a label in a macro, a double definition will result, with an error issued. So, use internal branch labels which are only valid within a macro.

- A maximum of 64 internal branch labels can be described per macro.
- The labels should be arranged like this: \$\$1–\$\$64 in order of description. (Each macro should begin with \$\$1.)

Sample definition:

```
#define   Areg      %r1
#macro   ADD       $1, $2
          ld.w     Areg, $1
          add      Areg, $2
#ifdef   DEBUG
          cmp      Areg, 0x1
#else
          cmp      Areg, 0x2
#endif
          xjrne   $$1
```

```

        ld.w    [%r2], 0b11
$$1:
        ld.w    %r3, [%r2]+
        jr     LABEL1
#endm

```

### Expansion rules

When a defined macro name appears in the source, the pp33 inserts a statement string defined in that location.

If there are actual parameters described in that process, the dummy parameters (\$1–\$32) will be replaced with the actual parameters in the same order as the latter are arranged.

The internal branch labels are replaced, respectively, with \_\_L0001–\_\_L9999 from top of the source in the same order as they appear.

Sample expansion:

When the macro ADD shown above is defined:

Macro invocation

```

#define  DEBUG
      :
      ADD  1, 2
      :

```

After expansion

```

      :
      ld.w  Areg, $1  ;   ADD  1, 2
      add  Areg, $2
;#ifdef  DEBUG
      cmp  Areg, 0x1
;#else
      ;   cmp  Areg, 0x2
;#endif
      xjrne  __L0001
      ld.w  [%r2], 0b11
__L0001:
      ld.w  %r3, [%r2]+
      jr   LABEL1

```

("\_\_L0001" denotes the case where an internal branch label is expanded for the first time in the source.)

### Precautions

- The pp33 only permits back reference of a macro invocation. The definition needs to have been completed before making the reference.
- Once a defined macro name is defined, it cannot be canceled. If the same macro name is defined twice or more, a warning message will appear and the redefined macro is validated.
- No other characters than delimiters (space, tab, line feed, and commas) can be added before and after a dummy parameter in a statement, unless they are enclosed in [ ] or [ ]+. However, a symbol mask (@..) described behind is valid.
- The same character string as that of the #define and #define instruction cannot be used as a macro name.
- When the number of dummy parameters differs from that of actual parameters, an error will result.
- A maximum of 32 parameters and a maximum of 64 internal branch labels can be specified per macro. If these limits are surpassed, an error will result.
- "\_\_L####" used for the internal branch labels should not be employed as other label or symbol.
- Maximum 9999 internal branch labels can be expanded within one source file. If this limit is exceeded, an error will result.

## 9.5.4 Conditional Assembly Instructions

### (`#ifdef ... #else ... #endif`, `#ifndef... #else ... #endif`)

A conditional assembly instruction determines whether assembling should be performed within the specified range, dependent on whether the specified name (Define name) is defined or not.

#### Instruction formats

```
Format 1)  #ifdef  <Name>
           <Statement string 1>
           [#else
           <Statement string 2>]
           #endif
```

If the `<Name>` is defined, `<Statement string 1>` will be subjected to the assembling.

If the `<Name>` is not defined, and `#else ... <Statement string 2>` is described, then `<Statement string 2>` will be subjected to the assembling. `#else ... <Statement string 2>` can be omitted.

```
Format 2)  #ifndef <Name>
           <Statement string 1>
           [#else
           <Statement string 2>]
           #endif
```

If the `<Name>` is not defined, `<Statement string 1>` will be subjected to the assembling.

If the `<Name>` is defined, and `#else ... <Statement string 2>` is described, `<Statement string 2>` will be subjected to the assembling. `#else ... <Statement string 2>` can be omitted.

`<Name>`:

- Conforms to the restrictions on Define name. (See `#define`.)

`<Statement string>`:

- All statements, excluding conditional assembly instructions, can be described.

Sample description:

```
#ifdef      TYPE1
            ld.w    %r1, 0x12
#else
            ld.w    %r1, 0x13
#endif
```

#### Name definition

Name definition needs to have been completed by either of the following methods, prior to the execution of a conditional assembly instruction:

- 1) To define by using the startup option (`-d`) of the `pp33`.

Example: `pp33 -d TYPE1 sample.s`

- 2) To define in the source file by using the `#define` instruction.

Example: `#define TYPE1`

The `#define` statement is valid even in a file to be included, provided that it goes before the conditional assembly instruction that uses its Define name. A name defined after a conditional assembly instruction will be regarded as undefined.

When a name is going to be used only in conditional assembly, no substitute character string needs to be specified.



### Expansion rule

A statement string subjected to the assembling is expanded according to the expansion rule of the other preprocessor instructions. (If no preprocessor instruction is contained, the statement will be output in a file as is.)

Statement strings not subjected to the assembling are delivered as comments.

### Precautions

- A name specified in the condition is evaluated with discrimination between uppercase and lowercase. The condition is deemed to be satisfied only when there is the same Define name defined.
- The `#ifdef` (`#ifndef`) instruction cannot be used for a statement string in a conditional assembly instruction, but the `#define`, `#macro` and `#include` instructions can be employed.

## 9.6 Operators

An expression that consists of operators and numbers can be used for specifying an immediate data.

The pp33 handles expressions in signed 32-bit data.

When writing expressions, do not insert a space between a term and an operator.

### Types of operators

		Examples
+	Addition, Plus sign	+0xff, 1+2
-	Subtraction, Minus sign	-1+2, 0xffff-0b111
*	Multiplication	0xf*5
/	Division	0x123/0x56
%%	Residue	0x123%%0x56
>>	Shifting to right	1>>2
<<	Shifting to left	0x113<<3
&	Bit AND	0b1101&0b111
	Bit OR	0x123 0xff
^	Bit XOR	12^35
~	Bit inversion	~0x1234
^H	Acquires bit field (31:19)	0x1234^H
^M	Acquires bit field (18:6)	0x1234^M
^L	Acquires bit field (5:0)	0x1234^L
^AH	Acquires bit field (25:13)	0x1234^AH
^AL	Acquires bit field (12:0)	0x1234^L
()	Parenthesis	1+(1+2*5)

### Priority

The operators have the priority shown below. If there are two or more operators with the same priority in an expression, the preprocessor calculates the expression from the left.

- |    |  |               |
|----|--|---------------|
| 1. | ()   | High priority |
| 2. | + (plus sign), - (minus sign), ~<br>^H, ^M, ^L, ^AH, ^AL | ↑             |
| 3. | *, /, %%   |               |
| 4. | +, -   |               |
| 5. | <<, >>   |               |
| 6. | &  |               |
| 7. | ^  | ↓             |
| 8. |  | Low priority  |

### Terms in expression

The following contents can be written in the terms of an expression.

- Binary, decimal, or hexadecimal number in the effective range of values represented by 32 bits  
Unsigned integer: 0 to 4294967295 (0x0 to 0xffffffff)  
Signed integer: 0 to 2147483647 (0x0 to 0x7fffffff), -1 to -2147483648 (0xffffffff to 0x80000000)
- Define names defined for numbers (names defined by #define)
- Symbol  
If the symbol is not a Define name, the expression is limited to the following format:  
SYMBOL [+SYMBOL...] + numeric expression or SYMBOL [+SYMBOL...] - numeric expression

**Examples**

```
#define BAR 0x1
ld.w %r0, BAR+2      ...ld.w %r0, 0x3
xcall LABEL+BAR*2    ...xcall LABEL+0x2
xld.w %r1, [FOO+BAR+1] ...xld.w %r1, [FOO+0x2]
xld.w %r1, [BAR+FOO+1] ...An error will result if FOO is not a Define name.
```

**Precautions**

- Since the operation is internally performed as 32 signed bits, caution is required depending on the type of operation.

Pay attention to the calculation results of the `>>`, `/` and `%%` operators using hexadecimal numbers.

Examples:

```
#define NUM1 0xffffffff/2    ...-2/2 = -1 (0xffffffff)
                                The / and %% operators can only be used within the signed
                                32-bit range.
#define NUM2 0xffffffff>>1  ...-2>>1 = -1 (0xffffffff)
                                Mask as (0xffffffff>>1)&0x7fffffff.
```

- The calculation result is delivered as a decimal number with a minus sign if it is negative, or a hexadecimal number if it is positive.

Examples:

```
add %r0, -2+1      ...It will be expanded as "add %r0, -1".
add %r0,(-2+1)&0x3f  ...It will be expanded as "add %r0, 0x3f".
```

- Do not insert a space or a TAB between an operator and a term (number, Define name).

Examples:

```
ld.w %r0, 1+1      ...OK
ld.w %r0, 1 + 1    ...NG
ld.w %r0, (1+NUM1)*2  ...OK
ld.w %r0, (1 + NUM1)*2  ...NG
```

## 9.7 Debugging Information

When the startup option `-g` is specified ([debug info] checked on the work bench), the `pp33` inserts assembler pseudo-instructions in the output file, as the debugging information designed to correspond with the assembly source level debugging.

- Notes:
- This debugging information is necessary to perform debugging on the Debugger `db33`, with the assembly source displayed.
  - Make sure the debugging information is created by only specifying the `-g` option, and not by any other method. Also, be sure not to correct the debugging information that is output. Corrections could cause the `as33`, `lk33`, `db33` or `dis33` to malfunction.
  - Unless the `-g` option is specified in the `lk33` even though it may be specified in the `pp33` (same applies for `gcc33`), all debugging information will be cut during linkage.
  - The source information created by specifying the `-g` option in the `pp33` is not cut even when the `-g` option is not specified in the `as33`.
  - The assembler level symbol information (symbol names and addresses only) is created when the `-g` option is specified in the `as33`.

### Assembler pseudo-instructions to be delivered

The following three types of debugging pseudo-instructions are delivered. The characters other than those in the underlined parts are fixed.

#### 1) `.file "PATH_NAME"`

Indicates the beginning of a file. Inserted at top of the current file or in the start position of an included file. `PATH_NAME` is the file path name.

#### 2) `.endfile`

Indicates the end of a file. Inserted at the end of the current file

The `.file` pseudo-instruction indicating the restart of the original file is inserted at the end of the include file. The `.endfile` pseudo-instruction is not inserted, however.

#### 3) `.loc LINE_NO`

Indicates the line information of the source file. Added only to the mnemonic statement (line assembled to the object code). `LINE_NO` is a source line number.

### Sample output

Startup command: `pp33 -g base_file.s`

Before processing:

- Source file "base\_file.s"
 

```

; file start
#include "inc.def"
        ld.w      %r1, [%r7]
        ld.w      [%r3], %r1

```
- Included file "inc.def"
 

```

; This is an empty file.

```

After processing:

- Assembly source file "base\_file.ms"
 

```

.file      "base_file.s"          Start of "base_file.s"
; file start                       (Debugging information is not added to comments.)
#include "inc.def"
.file      "inc.def"              Start of "inc.def"
; This is an empty file.
.file      "base_file.s"          Resuming of "base_file.s"
.loc       3                       Line No. 3 (base_file.s)
        ld.w      %r1, [%r7]
        .loc      4                       Line No. 4 (base_file.s)
        ld.w      [%r3], %r1
.endfile                          End of "base_file.s"

```

## 9.8 Comment Adding Function

The Preprocessor instructions are all expanded to codes that can be assembled, and delivered in the output file. Even after that, those instructions are rewritten with comments beginning with a semicolon (;), so that the original instructions can be identified. However, note that replacements of Define names and expressions will not subsist as comments.

The comment is added to the first line following the expansion. In case the original statement is accompanied by a comment, that comment is also added.

A macro definition should have a semicolon (;) placed at top of the line.

Example:

Before expansion

```
#define      R0      %r0

#macro      ADDM    $1
            ld.w    R0, $1
            add     R0, [%r4]
            ld.w    [%r5], R0

#endm

            ADDM    0x10      ; [%r5] = [%r4] + 0x10
```

After expansion (no debugging information)

```
;;#define      R0      %r0

;;#macro      ADDM    $1
;;            ld.w    R0, $1
;;            add     R0, [%r4]
;;            ld.w    [%r5], R0
;;#endm

            ld.w    %r0, 0x10 ;      ADDM    0x10      ; [%r5] = [%r4] + 0x10
            add     %r0, [%r4]
            ld.w    [%r5], %r0
```

## 9.9 Other Functions

### 9.9.1 ASCII to HEX Conversion

The pp33 has the function to convert an ASCII character enclosed with ' ' in source files into a hexadecimal number. The corresponding parts of the output assembly source file is replaced with the hexadecimal ASCII codes.

Sample conversions:

```
ld.w  %r1, '1'      →      ld.w  %r1, 0x31
ld.w  %r1, '1'+1    →      ld.w  %r1, 0x32      ... Numeric operators can be used.
```

Note: Only one ASCII character can be converted.

'\t' and '\n' can also be used as 0x9 and 0xA, respectively.

### 9.9.2 Comment Line

The pp33 allows comment lines that begin with "//" or "/\*" as well as one that begins with semicolon (;).

The first "/" character will be converted into ";;".

Sample conversions:

```
//comment ..... →      ;/comment .....
/*comment ..... →      ;*comment .....
```

## 9.10 Process Flow

The following lists the instruction process flow by the pp33:

1. The statements in the conditional assembly instructions (`#ifdef`, `#ifndef`) are skipped if the condition is unmatched.
2. Comments and the `.ascii` pseudo-instruction statements are delivered without conversion.
3. Each source line is separated into token and Define names are replaced with the contents defined by `#define`. A space, TAB, ";", "[", "]", "@", ",", or an operator is used as a delimiter for separation.
4. Expressions that consist of numbers and operators are calculated and then replaced with the results.
5. The preprocessor pseudo-instructions such as `#macro` and `#include` are processed.

## 9.11 Sample Executions

### Input source file (pp.s)

```

; pp.s      1997.2.20
; sample source for pp33

#include "pp.def"           ; include file
#define SP_IRAM             ; definition for #ifdef
#ifdef SP_IRAM              ; condition assemble
    #define SP_INIT_ADDR 0x400 ; set number to defnum symbol
#else
    #define SP_INIT_ADDR 0x880000
#endif
#define BLK_ADDR 0x0+0x10   ; defnum symbol can use with arithmetic operators
                             ; operators : +, -, *, /, %, >>, <<, &, !, ^, ~, ^H, ^M, ^L, ^AH, ^AL, (, )
#define gpr %r8             ; define replace define symbol to string
#define GP_INIT_ADDR 0x0    ; define treat 0x0 as string, not number

#macro FILL_AREA $1 $2 $3   ; macro argument is $1, $2, --- $32
    xld.w    %r1, $1        ; $1 is start address
    xld.w    %r2, $2        ; $2 is fill pattern (8bit)
    xld.w    %r3, $3        ; $3 is fill size (byte address)
$$1:        ; $$1 -- $$64 is local jump label
    cmp      %r3, 0
    jreq     $$2
    ld.b     [%r1]+, %r2
    sub      %r3, 1
    jp      $$1
$$2:
#endm

    .word BOOT

BOOT:
    ext      SP_INIT_ADDR^H
    ext      SP_INIT_ADDR^M
    ld.w     %r0, SP_INIT_ADDR^L
    ld.w     %sp, %r0
    ld.w     gpr, GP_INIT_ADDR
    FILL_AREA BLK_ADDR 0b01010101 10 ; fill 0x10-0x1f with 0x55
    FILL_AREA BLK_ADDR+0x10 0 10    ; fill 0x20-0x2f with 0x00
    jp      BOOT

```

### Included file (pp.def)

```

; pp.def
; This is empty file

```

```

Output file (pp.ms) when "pp33 -g pp.s" is executed
.file "pp.s"
: pp.s 1997.2.20
: sample source for pp33
#include "pp.def" ; include file
.file "pp.def"
: pp.def
: This is empty file
.file "pp.s"
#define SP_IRAM ; definition for #ifdef
#ifdef SP_IRAM ; condition assemble
: #define SP_INIT_ADDR 0x400 ; set number to defnum symbol
#else
: #define SP_INIT_ADDR 0x880000
#endif
#define BLK_ADDR 0x0+0x10 ; defnum symbol can use with arithmetic operators
; operators : +, -, *, /, %, >>, <<, &, !, ^, ~, ^H, ^M, ^L, ^AH, ^AL, (, )
#define gpr %r8 ; define replace define symbol to string
#define GP_INIT_ADDR 0x0 ; define treat 0x0 as string, not number
: #macro FILL_AREA $1 $2 $3 ; macro argument is $1, $2, --- $32
: xld.w %r1, $1 ; $1 is start address
: xld.w %r2, $2 ; $2 is fill pattern (8bit)
: xld.w %r3, $3 ; $3 is fill size (byte address)
: $$1: ; $$1 -- $$64 is local jump label
: cmp %r3, 0
: jreq $$2
: ld.b [%r1]+, %r2
: sub %r3, 1
: jp $$1
: $$2:
: #endm

.word BOOT
BOOT:
.loc 31
ext 0x0
.loc 32
ext 0x10
.loc 33
ld.w %r0, 0x0
.loc 34
ld.w %sp, %r0
.loc 35
ld.w %r8, 0x0
.loc 36
xld.w %r1, 0x10 ; $1 is start address ; FILL_AREA BLK_ADDR
0b01010101 10 ; fill 0x10-0x1f with 0x55
xld.w %r2, 0b01010101 ; $2 is fill pattern (8bit)
xld.w %r3, 10 ; $3 is fill size (byte address)
__L0001: ; $$1 -- $$64 is local jump label
cmp %r3, 0
jreq __L0002
ld.b [%r1]+, %r2
sub %r3, 1
jp __L0001
__L0002:
.loc 37
xld.w %r1, 0x20 ; $1 is start address ; FILL_AREA
BLK_ADDR+0x10 0 10 ; fill 0x20-0x2f with 0x00
xld.w %r2, 0 ; $2 is fill pattern (8bit)
xld.w %r3, 10 ; $3 is fill size (byte address)
__L0003: ; $$1 -- $$64 is local jump label
cmp %r3, 0
jreq __L0004
ld.b [%r1]+, %r2
sub %r3, 1
jp __L0003
__L0004:
.loc 38
jp BOOT
.endfile

```

## 9.12 Error/Warning Messages

Error and warning messages are displayed/output through the Standard Output (stdout). If you specify the `-e` option, the messages will also be delivered in the "pp33.err" file.

If the pp33 is started up using the wb33's [PP33] button, the message is output to "wb33.err". When execution is completed, a message is displayed in the output window (default).

### 9.12.1 Errors

The errors produced in the pp33 are classified into two groups: system errors and preprocessor errors.

If a system error occurs, the pp33 will immediately terminate the processing after displaying an error message. No assembly source file will be output.

Table 9.12.1.1 List of system error messages

Error message	Content
Error: Cannot open file.	Cannot open the source file or included file. The file does not exist in the specified directory.
Error: Cannot allocate memory.	Cannot secure memory space.
Error: Cannot open output file.	Cannot open the output file.
Error: Cannot open working file.	Cannot open the provisional working file.
Error: Cannot write file.	Cannot write to the file.
Error: Cannot close file.	Cannot close the file.
Error: Cannot read file. Line size is too long.	The statement is too long to be read. The maximum number of characters that can be read in a line is 255.
Error: Preprocessor limit: macro label number full.	The number of internal branch labels surpassed the limit during micro expansion. The maximum number of internal branch labels that can be expanded at a time is 9999, including the included file.

The preprocessor errors are produced when the source contains a syntax or description that cannot be processed by the pp33. Even when these errors occur, the processing will be carried on till the last line of the input file, unless a system error is produced. However, no assembly source file will be delivered.

Table 9.12.1.2 List of preprocessor error messages

Error message	Content
<file name>(line No.): Error: Invalid syntax.	There is a syntactic error. The preprocessor pseudo-instruction was described in a wrong format.
<file name>(line No.): Error: Nesting level too deep.	The limit of nesting (2 levels) was surpassed in the #include pseudo-instruction.
<file name>(line No.): Error: Unknown preprocessor instruction.	There is an unknown preprocessor instruction.
<file name>(line No.): Error: Too many macro parameters.	33 or more formal parameters were defined.
<file name>(line No.): Error: Invalid macro parameter.	Dummy parameters were arranged abnormally or the number of actual parameters differs from that of dummy parameters. Arrange the dummy parameters successively from \$1 to \$32.
<file name>(line No.): Error: Invalid macro label.	Internal branch labels in the micro definition are abnormal. Internal branch labels are limited to \$1 to \$64 (64 labels). Arrange them successively from \$1.
<file name>(line No.): Error: Invalid expression.	The operator description format is illegal.
<file name>(line No.): Error: Multi symbol.	Duplicated definition of the same name was done by the #define and #defnum pseudo-instruction.



## 9.12.2 Warning

Even when a warning appears, the pp33 will keep on processing, and completes the processing after displaying a warning message, unless any error is produced in addition. The assembly source file will be output.

Table 9.12.2.1 Warning message

Warning message	Content
<file name>(line No.): Warning: Multi define symbol.	Multiple instances of the same macro name, define name, or numeric define name are defined. The last name defined is valid, with the others invalidated. If the same name is used in the define or numeric define definition and the macro definition, the define or numeric define name is given priority, and no warning is generated.

## 9.13 Precautions

---

- (1) The pp33 only checks the grammar necessary for Preprocessing. Notice that it does not check mnemonics, operands, extended instructions and assembler pseudo-instructions, including the validity following the expansion.
- (2) If you want to display the assembly source on the screen when debugging it with the db33, be sure to specify the -g option before executing the pp33. Note also that unless the -g option is specified in the lk33, all debugging information is cut during linkage.  
Make sure the debugging information is created by only specifying the -g option, and not by any other method. Also be sure not to correct the debugging information that is output. Corrections could cause the as33, lk33, db33 or dis33 to malfunction.

## Chapter 10 Instruction Extender

This chapter describes the functions of the Instruction Extender ext33.

### 10.1 Functions

The Instruction Extender ext33 (hereafter called the "ext33") is a software tool to process the assembly source files created by the C Compiler gcc33 and Preprocessor pp33. Specifically, it expands the extended instructions written in the assembly source file into an assemble-ready mnemonic code as its output. Immediate extension by the ext instruction or an operation requiring multiple instructions can be written in one extended instruction. Therefore, when creating an assembly source, you need not be concerned with restrictions to the immediate size during programming.

The ext33 provides the following two optimize functions that can be specified with its startup option:

- Optimization to delete unnecessary ext instructions  
Optimization based on symbol information after linkage is also available.
- Optimization by the global pointer  
The number of instructions necessary to reference a global variable can be reduced.

### 10.2 Input/Output Files

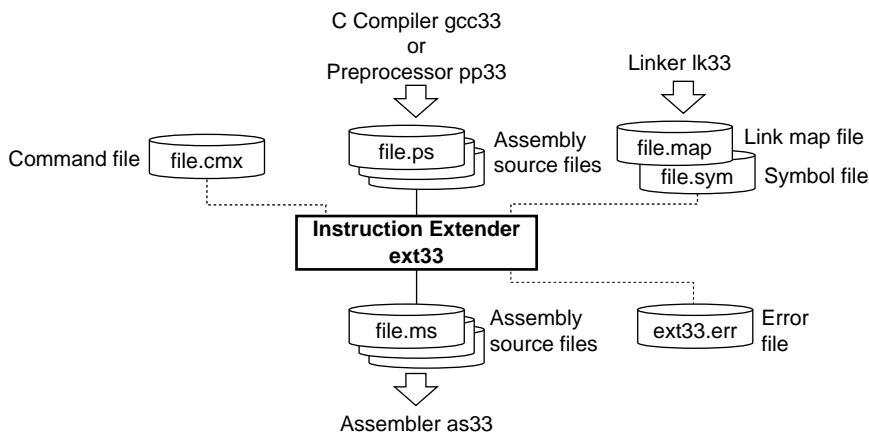


Fig. 10.2.1 Flowchart

#### 10.2.1 Input Files

**Assembly source file** (when the -c option is not specified)

File format: Text file  
 File name: <file name>.ps (Other extensions than ".ps" can be used excluding ".ms".)  
 Description: Files that are delivered from the gcc33 or the pp33 can be input.

**Command file** (when the -c option is specified)

File format: Text file  
 File name: <file name>.cmx  
 Description: File in which the startup options and input files for the ext33 are described. (See Section 10.4.)

**Link map file**

File format: Text file  
 File name: <file name>.map  
 Description: File that contains the map information indicating section addresses located by linkage. The link map file that is output by the Linker lk33 is used for code optimization.

**Symbol file**

File format:	Text file
File name:	<file name>.sym
Description:	File that contains the information of the symbols defined in all linked modules and the address information. The symbol file that is output by the Linker lk33 is used for code optimization.

**10.2.2 Output Files****Assembly source file**

File format:	Text file
File name:	<file name>.ms (The <file name> is the same as that of the input file.)
Output destination:	Current directory
Description:	File in which the extended instructions are expanded into an assembling format.

**Error file**

File format:	Text file
File name:	ext33.err
Output destination:	Current directory
Description:	File that is output when the startup option (-e) is specified, and describes the contents which the ext33 delivers through the Standard Output (stdout), such as an error message.

**10.3 Starting Method**

---

**10.3.1 Startup Format****General form of command line**

Format 1) **ext33 ^ [<startup option>] ^ [<source file name>]**

Format 2) **ext33 ^ [<startup option>] ^ -c ^ <command file name>**

^ denotes a space.

[ ] indicates the possibility to omit.

<source file name>: Specify assembly source file name(s) including the extension.

<command file name>: Specify a command file name including the extension.

**Operations on work bench**

Select options and input files, then click the [EXT33] button.

Multiple source files can be specified in the command line. All specified files can be processed simultaneously. Although the wb33 also allows multiple files to be selected, the ext33 need to be executed as many times as the number of files selected. If files are acquired from a command file, they all are processed simultaneously.

**10.3.2 Startup Options**

The ext33 comes provided with the following six types of startup options:

**-c <command file name>**

Function: Executes a command file.

Specification on wb33: Check [use .cmx file].

Explanation:

- This option acquires the startup option and input file name from the specified command file. The startup option also can be specified in the command line without including it in a command file.

**-gp <address>**

Function: Uses a global pointer.

Specification on wb33: Check [global pointer optimize], then input <address> in the text box.

Explanation:

- This option optimizes code generation by using a global pointer.
- The specified <address> is the address of the global pointer. Specify <address> in hexadecimal (0x####) using lower-case letters (0–9, a–f).
- For details about the global pointer, refer to Section 8.7.2.

**-lk <file name>**

Function: Optimizes instructions based on symbol information.

Specification on wb33: Check [symbol,map optimize]. The <file name> is taken from the contents of the text box in the execution window.

Explanation:

- This option optimizes the ext instruction based on the valid symbol information by reading the symbol and link map files generated by the linker.
- The symbol file (.sym) and link map file (.map) are specified by <file name> (object file name). No extension is required.
- For details about optimization by symbol information, refer to Section 8.7.3.

**-near**

Function: Expands a branch instruction into two instructions.

Specification on wb33: Check [far call is 2 inst].

Explanation:

- This option expands an extended branch instruction to a nonexistent label in the processed file into two instructions (one ext instruction + branch instruction, signed 22-bit displacement).
- Unless -near is specified, the above instruction is expanded into three instructions (two ext instructions + branch instruction, signed 32-bit displacement).
- For details about the optimization of branch instructions, refer to Section 8.7.1.

**-j <threshold value>**

Function: Specifies the threshold of optimized branching.

Specification on wb33: Check [change threshold], then input <threshold> in the text box.

Explanation:

- This option sets a threshold to determine the number of instructions expanded from an extended branch instruction.
- The effective range of <threshold> is 0x100 to 0x1fffff. Specify it in hexadecimal (0x####) using lower-case letters (0–9, a–f).
- Unless -j is specified, the threshold is set to the default value of 0x180000.
- For details about the optimization of branch instructions, refer to Section 8.7.1.

**-e**

Function: Output of error files

Specification on wb33: None

Explanation:

- Delivers also in a file (ext33.err) the contents that are output by the ext33 via the Standard Output (stdout), such as error messages.

When entering options in the command line, you need to place one or more spaces before and after the option.

Examples: c:\cc33\ext33 -gp 0x0 -lk test -near -j 0x180000 -e test1.ps test2.ps  
 c:\cc33\ext33 -gp 0x0 -c test.cmx

## 10.4 *Command File*

---

The ext33 allows the contents of the command line to be input from a command file (.cmx) specified by the -c option.

In a command file, write the options you want to specify and the source files to be input, each entry in one line. A comment can also be entered by inserting a semicolon (;) at the beginning of a line.

Example: sample.cmx

```

; This is a sample command file.           ← Comment line
-gp 0x80000
-near
-lk sample
-e
sample1.ps
sample2.ps
sample3.ps

```

If the same option that is included in a command file is specified from the command line, the first option encountered is recognized as the valid option.

Example: ext33 -gp 0x0 -c sample.cmx (Specifies sample.cmx in the above example)

In this example, the "-gp 0x0" option is recognized as the valid option, and the "-gp 0x80000" option in the command file is ignored.

## 10.5 Messages

The ext33 delivers its messages through the Standard Output (stdout).

If the ext33 is started up by using the wb33's [EXT33] button, the message is output to "wb33.err". When execution is completed, a message is displayed in the output window (default).

### During execution

When two or more input files are specified, the file name being executed is displayed.

```
file1.ps
file2.ps
:
```

When only one file is specified, the file name does not appear.

### End message

The ext33 outputs only the following end message when it ends normally.

```
Extend Completed
```

### Usage output

If no file name was specified or an option was not specified correctly, the ext33 ends after delivering the following message concerning the usage:

```
Extender 33 Ver x.xx
Copyright (C) SEIKO EPSON CORP. 199x
Usage:
  ext33 [options] filename
  ext33 [options] -c commandfile (.cmx)
Options:
  -e           : produce log file (ext33.err)
  -lk program : optimize with program information (program.sym, program.map)
  -gp address  : optimize with global pointer (0x0 - 0xffffffff)
  -near        : specifies all xjmp and xcall extract 2 instruction
  -j threshold : specifies jump optimization threshold (0x100 - 0x1fffff)
Output:
  Assembler source file for as33(.ms)
Example:
  ext33 -e -lk test -gp 0x8000 test.ps
```

### When error/warning occurs

If an error is produced, an error message will appear before the end message shows up.

```
Example: test.ps(431): Error: Invalid Syntax.
        Extend Completed
```

In the case of an error, the ext33 ends without creating an output file.

If a warning is issued, a warning message will appear before the end message shows up.

```
Example: Warning: Map file test.map does not exist.
        Extend Completed
```

In the case of a warning, the ext33 ends after creating an output file.

For details on errors and warnings, refer to Section 10.10 "Error/Warning Messages".

## 10.6 Extended Instructions

The ext33 expands the extended instructions, explained below, to mnemonics that can be processed by the Assembler as33.

Extended instructions allow an operation that normally requires using multiple instructions including the ext instruction to be written in one instruction. They are expanded into the absolutely necessary minimum basic instructions according to instruction functionality and the operand's immediate size.

### Symbols used in explanation

immX	Unsigned X-bit immediate
signX	Signed X-bit immediate
symbol	Symbol to indicate memory address
label	Jump address label
(X:Y)	Bit field from bit X to bit Y

### 10.6.1 Arithmetic Operation Instructions

#### Types and functions of extended instructions

Extended instruction	Function	Expansion format
xadd %rd, %rd, imm32	$\%rd \leftarrow \%rd + imm32$	(1)
xsub %rd, %rd, imm32	$\%rd \leftarrow \%rd - imm32$	(1)
xadd %sp, %sp, imm32	$\%sp \leftarrow \%sp + imm32$	(2)
xsub %sp, %sp, imm32	$\%sp \leftarrow \%sp - imm32$	(2)
xadd %rd, %rs, imm32	$\%rd \leftarrow \%rs + imm32$	(3)
xsub %rd, %rs, imm32	$\%rd \leftarrow \%rs - imm32$	(3)
xadd %rd, %sp, imm32	$\%rd \leftarrow \%sp + imm32$	(4)
xsub %rd, %sp, imm32	$\%rd \leftarrow \%sp - imm32$	(4)
xadd %rd, %rd, %sp	$\%rd \leftarrow \%rd + \%sp$	(5)
xsub %rd, %rd, %sp	$\%rd \leftarrow \%rd - \%sp$	(5)
xadd %sp, %sp, %rs	$\%sp \leftarrow \%sp + \%rs$	(6)
xsub %sp, %sp, %rs	$\%sp \leftarrow \%sp - \%rs$	(6)

These extended instructions allow a 32-bit immediate to be specified directly in an add or subtract operation. Furthermore, they support addition or subtraction between a stack pointer SP and a general-purpose register.

#### Basic instructions after expansion

xadd	Expanded into the add instruction
xsub	Expanded into the sub instruction

#### Expansion formats

##### (1) xOP %rd, %rd, imm32 (OP = add, sub)

Example: xadd %rd, %rd, imm32

$imm32 \leq 0x3f$	$0x3f < imm32 \leq 0x7fff$	$imm32 > 0x7fff$
add %rd, imm32(5:0)	ext imm32(18:6) add %rd, imm32(5:0)	ext imm32(31:19) ext imm32(18:6) add %rd, imm32(5:0)

##### (2) xOP %sp, %sp, imm32 (OP = add, sub)

Example: xadd %sp, %sp, imm32

$imm32 \leq 0xfff$	$0xfff < imm32 \leq 0x7fff$	$imm32 > 0x7fff$
add %sp, imm32(11:2)	ld.w %r9, %sp ext imm32(18:6) add %r9, imm32(5:0) ld.w %sp, %r9	ld.w %r9, %sp ext imm32(31:19) ext imm32(18:6) add %r9, imm32(5:0) ld.w %sp, %r9

**(3) xOP %rd, %rs, imm32 (OP = add, sub)**

Example: xadd %rd, %rs, imm32

$imm32 \leq 0x1fff$	$0x1fff < imm32 \leq 0x3fffff$	$imm32 > 0x3fffff$
ext imm32(12:0) add %rd, %rs	ext imm32(25:13) ext imm32(12:0) add %rd, %rs	ld.w %rd, %rs ext imm32(31:19) ext imm32(18:6) add %rd, imm32(5:0)

**(4) xOP %rd, %sp, imm32 (OP = add, sub)**

Example: xadd %rd, %sp, imm32

$imm32 \leq 0x3f$	$0x3f < imm32 \leq 0x7fff$	$imm32 > 0x7fff$
ld.w %rd, %sp add %rd, imm32(5:0)	ld.w %rd, %sp ext imm32(18:6) add %rd, imm32(5:0)	ld.w %rd, %sp ext imm32(31:19) ext imm32(18:6) add %rd, imm32(5:0)

**(5) xOP %rd, %rd, %sp (OP = add, sub)**

Example: xadd %rd, %rd, %sp

ld.w %r9, %sp add %rd, %r9
-------------------------------

**(6) xOP %sp, %sp, %rs (OP = add, sub)**

Example: xadd %sp, %sp, %rs

ld.w %r9, %sp add %r9, %rs ld.w %sp, %r9
--

## 10.6.2 Comparison Instructions

### Types and functions of extended instructions

Extended instruction	Function	Expansion format
xcmp %rd, sign32	%rd - sign32 (Sets/resets C, V, Z and N flags in PSR)	(1)
xcmp %rd, %sp	%rd - %sp (Sets/resets C, V, Z and N flags in PSR)	(2)
xcmp %sp, %rs	%sp - %rs (Sets/resets C, V, Z and N flags in PSR)	(3)

These extended instructions allow you to compare a general-purpose register and a signed 32-bit immediate or a stack pointer SP and general-purpose register.

### Basic instruction after expansion

xcmp Expanded into the cmp instruction

### Expansion formats

**(1) xcmp %rd, sign32**

$-32 \leq sign32 \leq 31$	$-262144 \leq sign32 < -32$ or $31 < sign32 \leq 262143$	$sign32 < -262144$ or $262143 < sign32$
cmp %rd, sign32(5:0)	ext sign32(18:6) cmp %rd, sign32(5:0)	ext sign32(31:19) ext sign32(18:6) cmp %rd, sign32(5:0)

**(2) xcmp %rd, %sp**

ld.w %r9, %sp cmp %rd, %r9
-------------------------------

**(3) xcmp %sp, %rs**

ld.w %r9, %sp cmp %r9, %rs
-------------------------------



## 10.6.3 Logic Operation Instructions

### Types and functions of extended instructions

Extended instruction	Function	Expansion format
xand %rd, %rd, sign32	%rd ← %rd & sign32	(1)
xoor %rd, %rd, sign32	%rd ← %rd   sign32	(1)
xxor %rd, %rd, sign32	%rd ← %rd ^ sign32	(1)
xand %rd, %rs, sign32	%rd ← %rs & sign32	(2)
xoor %rd, %rs, sign32	%rd ← %rs   sign32	(2)
xxor %rd, %rs, sign32	%rd ← %rs ^ sign32	(2)
xnot %rd, sign32	%rd ← !sign32	(3)

These extended instructions allow a signed 32-bit immediate to be specified directly in a logical operation.

### Basic instructions after expansion

xand	Expanded into the and instruction
xoor	Expanded into the or instruction
xxor	Expanded into the xor instruction
xnot	Expanded into the not instruction

### Expansion formats

#### (1) xOP %rd, %rd, sign32 (OP = and, oor, xor)

Example: xand %rd, %rd, sign32

-32 ≤ sign32 ≤ 31	-262144 ≤ sign32 < -32 or 31 < sign32 ≤ 262143	sign32 < -262144 or 262143 < sign32
and %rd, sign32(5:0)	ext sign32(18:6) and %rd, sign32(5:0)	ext sign32(31:19) ext sign32(18:6) and %rd, sign32(5:0)

#### (2) xOP %rd, %rs, sign32 (OP = and, oor, xor)

Example: xand %rd, %rs, sign32

0x0 ≤ sign32 ≤ 0x1fff (within 13 bits)	0x1fff < sign32 ≤ 0x3fffff (within 26 bits)	0x3fffff < sign32 < 0xfffc0000 (26 bits < sign32 < -262144)
ext sign32(12:0) and %rd, %rs	ext sign32(25:13) ext sign32(12:0) and %rd, %rs	ld.w %rd, %rs ext sign32(31:19) ext sign32(18:6) and %rd, sign32(5:0)
0xfffc0000 ≤ sign32 < 0xffffffe0 (-262144 ≤ sign32 < -32)	0xffffffe0 ≤ sign32 ≤ 0xfffffff (-32 ≤ sign32 ≤ -1)	
ld.w %rd, %rs ext sign32(18:6) and %rd, sign32(5:0)	ld.w %rd, %rs and %rd, sign32(5:0)	

#### (3) xnot %rd, sign32

-32 ≤ sign32 ≤ 31	-262144 ≤ sign32 < -32 or 31 < sign32 ≤ 262143	sign32 < -262144 or 262143 < sign32
not %rd, sign32(5:0)	ext sign32(18:6) not %rd, sign32(5:0)	ext sign32(31:19) ext sign32(18:6) not %rd, sign32(5:0)

## 10.6.4 Shift & Rotate Instructions

### Types and functions of extended instructions

Extended instruction	Function	Expansion format
x srl    %rd, %rs	Logical shift to right	(1)
x sll    %rd, %rs	Logical shift to left	(1)
x sra    %rd, %rs	Arithmetic shift to right	(1)
x sla    %rd, %rs	Arithmetic shift to left	(1)
x rr     %rd, %rs	Rotation to right	(1)
x rl     %rd, %rs	Rotation to left	(1)
x srl    %rd, imm5	Logical shift to right	(2)
x sll    %rd, imm5	Logical shift to left	(2)
x sra    %rd, imm5	Arithmetic shift to right	(2)
x sla    %rd, imm5	Arithmetic shift to left	(2)
x rr     %rd, imm5	Rotation to right	(2)
x rl     %rd, imm5	Rotation to left	(2)

These extended instructions allow a shift or rotate operation to be performed in up to 31 bits.

### Basic instructions after expansion

x srl	Expanded into the srl instruction
x sll	Expanded into the sll instruction
x sra	Expanded into the sra instruction
x sla	Expanded into the sla instruction
x rr	Expanded into the rr instruction
x rl	Expanded into the rl instruction

### Expansion formats

#### (1) xOP    %rd, %rs        (OP = srl, sll, sra, sla, rr, rl)

Example: x srl    %rd, %rs

ld.w	%r9, %rs	; Stores Shift count
and	%r9, 0x1f	; Checks Shift count (Shift count < 31)
cmp	%r9, 0x8	; while (Shift count > 0x8)
jrle	4	; {
srl	%rd, 0x8	;        %rd ← %rd shift 0x8
jp.d	-3	;        Shift count -= 0x8
sub	%r9, 0x8	; }
rl	%rd, %r9	; %rd ← %rd shift Shift count

#### (2) xOP    %rd, imm5        (OP = srl, sll, sra, sla, rr, rl)

Example: x srl    %rd, imm5

imm5 ≤ 8	8 < imm5 < 16	imm5 = 16
srl    %rd, imm5(3:0)	srl    %rd, 0x8	srl    %rd, 0x8
	srl    %rd, imm5(2:0)	srl    %rd, 0x8
16 < imm5 ≤ 24	imm5 > 24	
srl    %rd, 0x8	srl    %rd, 0x8	
srl    %rd, 0x8	srl    %rd, 0x8	
srl    %rd, imm5(3:0)	srl    %rd, 0x8	
	srl    %rd, imm5(2:0)	

## 10.6.5 Data Transfer Instructions (between Stack and Register)

### Types and functions of extended instructions

Extended instruction	Function	Expansion format
xld.b %rd, [%sp+imm32]	%rd ← B[%sp+imm32] (with sign extension)	(1)
xld.ub %rd, [%sp+imm32]	%rd ← B[%sp+imm32] (with zero extension)	(1)
xld.h %rd, [%sp+imm32]	%rd ← H[%sp+imm32] (with sign extension)	(2)
xld.uh %rd, [%sp+imm32]	%rd ← H[%sp+imm32] (with zero extension)	(2)
xld.w %rd, [%sp+imm32]	%rd ← W[%sp+imm32]	(3)
xld.b [%sp+imm32], %rs	B[%sp+imm32] ← %rs(7:0)	(1)
xld.h [%sp+imm32], %rs	H[%sp+imm32] ← %rs(15:0)	(2)
xld.w [%sp+imm32], %rs	W[%sp+imm32] ← %rs	(3)
xld.w [%sp+imm32], %sp	W[%sp+imm32] ← %sp	(4)

These extended instructions allow you to directly specify a displacement of up to 32 bits. Specification of imm32 can be omitted.

### Basic instructions after expansion

xld.b	Expanded into the ld.b instruction
xld.ub	Expanded into the ld.ub instruction
xld.h	Expanded into the ld.h instruction
xld.uh	Expanded into the ld.uh instruction
xld.w	Expanded into the ld.w instruction

### Expansion formats

If imm32 is omitted, the ext33 assumes that [%sp+0x0] is specified as it expands the instruction.

#### (1) Byte data transfer (xld.b, xld.ub)

Example: xld.b %rd, [%sp+imm32]

imm32 ≤ 0x3f	0x3f < imm32 ≤ 0x7fff	imm32 > 0x7fff
ld.b %rd, [%sp+imm32(5:0)]	ext imm32(18:6) ld.b %rd, [%sp+imm32(5:0)]	ext imm32(31:19) ext imm32(18:6) ld.b %rd, [%sp+imm32(5:0)]

#### (2) Half word data transfer (xld.h, xld.uh)

Example: xld.h %rd, [%sp+imm32]

imm32 ≤ 0x7f	0x7f < imm32 ≤ 0x7fff	imm32 > 0x7fff
ld.h %rd, [%sp+imm32(6:1)]	ext imm32(18:6) ld.h %rd, [%sp+imm32(5:0)]	ext imm32(31:19) ext imm32(18:6) ld.h %rd, [%sp+imm32(5:0)]

#### (3) Word data transfer (xld.w)

Example: xld.w %rd, [%sp+imm32]

imm32 ≤ 0xff	0xff < imm32 ≤ 0x7fff	imm32 > 0x7fff
ld.w %rd, [%sp+imm32(7:2)]	ext imm32(18:6) ld.w %rd, [%sp+imm32(5:0)]	ext imm32(31:19) ext imm32(18:6) ld.w %rd, [%sp+imm32(5:0)]

#### (4) Word data transfer using SP as the source (xld.w [%sp+imm32], %sp)

imm32 ≤ 0xff	0xff < imm32 ≤ 0x7fff	imm32 > 0x7fff
ld.w %r9, %sp ld.w [%sp+imm32(7:2)], %r9	ld.w %r9, %sp ext imm32(18:6) ld.w [%sp+imm32(5:0)], %r9	ld.w %r9, %sp ext imm32(31:19) ext imm32(18:6) ld.w [%sp+imm32(5:0)], %r9

## 10.6.6 Data Transfer Instructions (between Memory and Register)

### Types and functions of extended instructions

Extended instruction	Function	Expansion format
xld.b %rd, [symbol±imm32]	%rd ← B[symbol±imm32] (with sign extension)	(1)
xld.ub %rd, [symbol±imm32]	%rd ← B[symbol±imm32] (with zero extension)	(1)
xld.h %rd, [symbol±imm32]	%rd ← H[symbol±imm32] (with sign extension)	(1)
xld.uh %rd, [symbol±imm32]	%rd ← H[symbol±imm32] (with zero extension)	(1)
xld.w %rd, [symbol±imm32]	%rd ← W[symbol±imm32]	(1)
xld.b [symbol±imm32], %rs	B[symbol±imm32] ← %rs(7:0)	(1)
xld.h [symbol±imm32], %rs	H[symbol±imm32] ← %rs(15:0)	(1)
xld.w [symbol±imm32], %rs	W[symbol±imm32] ← %rs	(1)
xld.w [symbol±imm32], %sp	W[symbol±imm32] ← %sp	(2)
xld.b %rd, [imm32]	%rd ← B[imm32] (with sign extension)	(3)
xld.ub %rd, [imm32]	%rd ← B[imm32] (with zero extension)	(3)
xld.h %rd, [imm32]	%rd ← H[imm32] (with sign extension)	(3)
xld.uh %rd, [imm32]	%rd ← H[imm32] (with zero extension)	(3)
xld.w %rd, [imm32]	%rd ← W[imm32]	(3)
xld.b [imm32], %rs	B[imm32] ← %rs(7:0)	(3)
xld.h [imm32], %rs	H[imm32] ← %rs(15:0)	(3)
xld.w [imm32], %rs	W[imm32] ← %rs	(3)
xld.w [imm32], %sp	W[imm32] ← %sp	(4)
xld.b %rd, [%rb+symbol±imm32]	%rd ← B[%rb+symbol±imm32] (with sign extension)	(5)
xld.ub %rd, [%rb+symbol±imm32]	%rd ← B[%rb+symbol±imm32] (with zero extension)	(5)
xld.h %rd, [%rb+symbol±imm32]	%rd ← H[%rb+symbol±imm32] (with sign extension)	(5)
xld.uh %rd, [%rb+symbol±imm32]	%rd ← H[%rb+symbol±imm32] (with zero extension)	(5)
xld.w %rd, [%rb+symbol±imm32]	%rd ← W[%rb+symbol±imm32]	(5)
xld.b [%rb+symbol±imm32], %rs	B[%rb+symbol±imm32] ← %rs(7:0)	(5)
xld.h [%rb+symbol±imm32], %rs	H[%rb+symbol±imm32] ← %rs(15:0)	(5)
xld.w [%rb+symbol±imm32], %rs	W[%rb+symbol±imm32] ← %rs	(5)
xld.w [%rb+symbol±imm32], %sp	W[%rb+symbol±imm32] ← %sp	(6)
xld.b %rd, [%rb+imm32]	%rd ← B[%rb+imm32] (with sign extension)	(7)
xld.ub %rd, [%rb+imm32]	%rd ← B[%rb+imm32] (with zero extension)	(7)
xld.h %rd, [%rb+imm32]	%rd ← H[%rb+imm32] (with sign extension)	(7)
xld.uh %rd, [%rb+imm32]	%rd ← H[%rb+imm32] (with zero extension)	(7)
xld.w %rd, [%rb+imm32]	%rd ← W[%rb+imm32]	(7)
xld.b [%rb+imm32], %rs	B[%rb+imm32] ← %rs(7:0)	(7)
xld.h [%rb+imm32], %rs	H[%rb+imm32] ← %rs(15:0)	(7)
xld.w [%rb+imm32], %rs	W[%rb+imm32] ← %rs	(7)
xld.w [%rb+imm32], %sp	W[%rb+imm32] ← %sp	(8)

\* "symbol±imm32" means that "symbol+imm32" and "symbol-imm32" can be specified.

These extended instructions allow memory locations to be accessed by specifying the address with a symbol or 32-bit immediate. However, the postincrement function ([ ]+) cannot be used.

### Basic instructions after expansion

xld.b	Expanded into the ld.b instruction
xld.ub	Expanded into the ld.ub instruction
xld.h	Expanded into the ld.h instruction
xld.uh	Expanded into the ld.uh instruction
xld.w	Expanded into the ld.w instruction

Expansion formats

- (1) `xld.* %rd, [symbol+imm32]`    `xld.* %rd, [symbol-imm32]`    (\*=**b, ub, h, uh, w**)  
`xld.* [symbol+imm32], %rs`    `xld.* [symbol-imm32], %rs`    (\*=**b, h, w**)

- When `[symbol+imm32]` is specified

Example: `xld.w %rd, [symbol+imm32]`

When global pointer is not specified:

<code>symbol+imm32 ≤ 0x1f</code>	<code>0x1f &lt; symbol+imm32 ≤ 0x3fff</code>	<code>symbol+imm32 &gt; 0x3fff</code>
<code>ld.w %r9, symbol+imm32@l</code> <code>ld.w %rd, [%r9]</code>	<code>ext symbol+imm32@m</code> <code>ld.w %r9, symbol+imm32@l</code> <code>ld.w %rd, [%r9]</code>	<code>ext symbol+imm32@h</code> <code>ext symbol+imm32@m</code> <code>ld.w %r9, symbol+imm32@l</code> <code>ld.w %rd, [%r9]</code>
Unknown symbol		
<code>lxt symbol+imm32@h</code> <code>ext symbol+imm32@m</code> <code>ld.w %r9, symbol+imm32@l</code> <code>ld.w %rd, [%r9]</code>		

When global pointer (gp) is specified:

(`sign32 = -gp+imm32`)

<code>symbol+sign32 = 0x0</code>	<code>0x0 &lt; symbol+sign32 ≤ 0x1fff</code>	<code>0x1fff &lt; symbol+sign32 ≤ 0x3fffff</code>
<code>ld.w %rd, [%r8]</code>	<code>ext symbol+sign32@al</code> <code>ld.w %rd, [%r8]</code>	<code>ext symbol+sign32@ah</code> <code>ext symbol+sign32@al</code> <code>ld.w %rd, [%r8]</code>
<code>symbol+sign32 &gt; 0x3fffff</code>	Unknown symbol	<code>gp &gt; symbol+imm32</code>
<code>ext symbol+imm32@h</code> <code>ext symbol+imm32@m</code> <code>ld.w %r9, symbol+imm32@l</code> <code>ld.w %rd, [%r9]</code>	<code>ext symbol+sign32@ah</code> <code>ext symbol+sign32@al</code> <code>ld.w %rd, [%r8]</code>	Expanded into a format without gp specification according to the <code>symbol+imm32</code> value.

Specification of `imm32` can be omitted. If `imm32` is omitted, the `ext33` assumes that `[symbol+0x0]` is specified as it expands the instruction.

- When `[symbol-imm32]` is specified

Example: `xld.w %rd, [symbol-imm32]`

The instruction is always expanded into the following format.

<code>ext symbol-imm32@h</code>
<code>ext symbol-imm32@m</code>
<code>ld.w %r9, symbol-imm32@l</code>
<code>ld.w %rd, [%r9]</code>

- (2) `xld.w [symbol+imm32], %sp`    `xld.w [symbol-imm32], %sp`

- When `[symbol+imm32]` is specified

When global pointer is not specified:

<code>symbol+imm32 ≤ 0x1f</code>	<code>0x1f &lt; symbol+imm32 ≤ 0x3fff</code>	<code>symbol+imm32 &gt; 0x3fff</code>
<code>ld.w %r9, %sp</code> <code>pushn %r0</code> <code>ld.w %r0, symbol+imm32@l</code> <code>ld.w [%r0], %r9</code> <code>popn %r0</code>	<code>ld.w %r9, %sp</code> <code>pushn %r0</code> <code>ext symbol+imm32@m</code> <code>ld.w %r0, symbol+imm32@l</code> <code>ld.w [%r0], %r9</code> <code>popn %r0</code>	<code>ld.w %r9, %sp</code> <code>pushn %r0</code> <code>ext symbol+imm32@h</code> <code>ext symbol+imm32@m</code> <code>ld.w %r0, symbol+imm32@l</code> <code>ld.w [%r0], %r9</code> <code>popn %r0</code>
Unknown symbol		
<code>ld.w %r9, %sp</code> <code>pushn %r0</code> <code>ext symbol+imm32@h</code> <code>ext symbol+imm32@m</code> <code>ld.w %r0, symbol+imm32@l</code> <code>ld.w [%r0], %r9</code> <code>popn %r0</code>		

When global pointer (gp) is specified:

(sign32 = -gp+imm32)

symbol+sign32 = 0x0	0x0 < symbol+sign32 ≤ 0x1fff	0x1fff < symbol+sign32 ≤ 0x3fffff
ld.w %r9, %sp ld.w [%r8], %r9	ld.w %r9, %sp ext symbol+sign32@al ld.w [%r8], %r9	ld.w %r9, %sp ext symbol+sign32@ah ext symbol+sign32@al ld.w [%r8], %r9
symbol+sign32 > 0x3fffff	Unknown symbol	gp > symbol+imm32
ld.w %r9, %sp pushn %r0 ext symbol+imm32@h ext symbol+imm32@m ld.w %r0, symbol+imm32@l ld.w [%r0], %r9 popn %r0	ld.w %r9, %sp pushn %r0 ext symbol+imm32@h ext symbol+imm32@m ld.w %r0, symbol+imm32@l ld.w [%r0], %r9 popn %r0	Expanded into a format without gp specification according to the symbol+imm32 value.

Specification of imm32 can be omitted. If imm32 is omitted, the ext33 assumes that [symbol+0x0] is specified as it expands the instruction.

- **When [symbol-imm32] is specified**

The instruction is always expanded into the following format.

ld.w %r9, %sp
pushn %r0
ext symbol-imm32@h
ext symbol-imm32@m
ld.w %r0, symbol-imm32@l
ld.w [%r0], %r9
popn %r0

(3) **xld.\* %rd, [imm32] (\*=b, ub, h, uh, w) xld.\* [imm32], %rs (\*=b, h, w)**

Example: xld.w %rd, [imm32]

When global pointer is not specified:

imm32 ≤ 0x1f	0x1f < imm32 ≤ 0x3fff	imm32 > 0x3fff
ld.w %r9, imm32(5:0) ld.w %rd, [%r9]	ext imm32(18:6) ld.w %r9, imm32(5:0) ld.w %rd, [%r9]	ext imm32(31:19) ext imm32(18:6) ld.w %r9, imm32(5:0) ld.w %rd, [%r9]

When global pointer (gp) is specified:

(sign32 = -gp+imm32)

sign32 = 0x0	0x0 < sign32 ≤ 0x1fff	0x1fff < sign32 ≤ 0x3fffff
ld.w %rd, [%r8]	ext sign32(12:0) ld.w %rd, [%r8]	ext sign32(25:13) ext sign32(12:0) ld.w %rd, [%r8]
sign32 > 0x3fffff	gp > imm32	
ext imm32(31:19) ext imm32(18:6) ld.w %r9, imm32(5:0) ld.w %rd, [%r9]	Expanded into a format without gp specification according to the imm32 value.	

(4) **xld.w [imm32], %sp**

When global pointer is not specified:

imm32 ≤ 0x1f	0x1f < imm32 ≤ 0x3fff	imm32 > 0x3fff
ld.w %r9, %sp pushn %r0 ld.w %r0, imm32(5:0) ld.w [%r0], %r9 popn %r0	ld.w %r9, %sp pushn %r0 ext imm32(18:6) ld.w %r0, imm32(5:0) ld.w [%r0], %r9 popn %r0	ld.w %r9, %sp pushn %r0 ext imm32(31:19) ext imm32(18:6) ld.w %r0, imm32(5:0) ld.w [%r0], %r9 popn %r0

When global pointer (gp) is specified:

(sign32 = -gp+imm32)

sign32 = 0x0	0x0 < sign32 ≤ 0x1fff	0x1fff < sign32 ≤ 0x3fffff
ld.w %r9, %sp ld.w [%r8], %r9	ld.w %r9, %sp ext sign32(12:0) ld.w [%r8], %r9	ld.w %r9, %sp ext sign32(25:13) ext sign32(12:0) ld.w [%r8], %r9
sign32 > 0x3fffff	gp > imm32	
ld.w %r9, %sp pushn %r0 ext imm32(31:19) ext imm32(18:6) ld.w %r0, imm32(5:0) ld.w [%r0], %r9 popn %r0	Expanded into a format without gp specification according to the imm32 value.	

- (5) **xld.\* %rd, [%rb+symbol+imm32]**      **xld.\* %rd, [%rb+symbol-imm32]**      (\*=b, ub, h, uh, w)  
**xld.\* [%rb+symbol+imm32], %rs**      **xld.\* [%rb+symbol-imm32], %rs**      (\*=b, h, w)

• **When [%rb+symbol+imm32] is specified**

Example: xld.w %rd, [%rb+symbol+imm32]

symbol+imm32 = 0x0	0x0 < symbol+imm32 ≤ 0x1fff	0x1fff < symbol+imm32 ≤ 0x3fffff
ld.w %rd, [%rb]	ext symbol+imm32@al ld.w %rd, [%rb]	ext symbol+imm32@ah ext symbol+imm32@al ld.w %rd, [%rb]
symbol+imm32 > 0x3fffff	Unknown symbol	
ext symbol+imm32@h ext symbol+imm32@m ld.w %r9, symbol+imm32@l add %r9, %rb ld.w %rd, [%r9]	ext symbol+imm32@h ext symbol+imm32@m ld.w %r9, symbol+imm32@l add %r9, %rb ld.w %rd, [%r9]	

Specification of imm32 can be omitted. If imm32 is omitted, the ext33 assumes that [%rb+symbol+0x0] is specified as it expands the instruction.

• **When [%rb+symbol-imm32] is specified**

Example: xld.w %rd, [%rb+symbol-imm32]

The instruction is always expanded into the following format.

ext symbol-imm32@h ext symbol-imm32@m ld.w %r9, symbol-imm32@l add %r9, %rb ld.w %rd, [%r9]
---

- (6) **xld.w [%rb+symbol+imm32], %sp**      **xld.w [%rb+symbol-imm32], %sp**

• **When [%rb+symbol+imm32] is specified**

symbol+imm32 = 0x0	0x0 < symbol+imm32 ≤ 0x1fff	0x1fff < symbol+imm32 ≤ 0x3fffff
ld.w %r9, %sp ld.w [%rb], %r9	ld.w %r9, %sp ext symbol+imm32@al ld.w [%rb], %r9	ld.w %r9, %sp ext symbol+imm32@ah ext symbol+imm32@al ld.w [%rb], %r9
symbol+imm32 > 0x3fffff and %rb ≠ %r0	symbol+imm32 > 0x3fffff and %rb = %r0	
ld.w %r9, %sp pushn %r0 ext symbol+imm32@h ext symbol+imm32@m ld.w %r0, symbol+imm32@l add %r0, %rb ld.w [%r0], %r9 popn %r0	ld.w %r9, %sp pushn %r1 ext symbol+imm32@h ext symbol+imm32@m ld.w %r1, symbol+imm32@l add %r1, %rb ld.w [%r1], %r9 popn %r1	

Unknown symbol, %rb ≠ %r0	Unknown symbol, %rb = %r0
ld.w %r9, %sp	ld.w %r9, %sp
pushn %r0	pushn %r1
ext symbol+imm32@h	ext symbol+imm32@h
ext symbol+imm32@m	ext symbol+imm32@m
ld.w %r0, symbol+imm32@l	ld.w %r1, symbol+imm32@l
add %r0, %rb	add %r1, %rb
ld.w [%r0], %r9	ld.w [%r1], %r9
popn %r0	popn %r1

Specification of imm32 can be omitted. If imm32 is omitted, the ext33 assumes that [%rb+symbol+0x0] is specified as it expands the instruction.

- **When [%rb+symbol-imm32] is specified**

The instruction is always expanded into one of the following formats.

%rb ≠ %r0	%rb = %r0
ld.w %r9, %sp	ld.w %r9, %sp
pushn %r0	pushn %r1
ext symbol-imm32@h	ext symbol-imm32@h
ext symbol-imm32@m	ext symbol-imm32@m
ld.w %r0, symbol-imm32@l	ld.w %r1, symbol-imm32@l
add %r0, %rb	add %r1, %rb
ld.w [%r0], %r9	ld.w [%r1], %r9
popn %r0	popn %r1

(7) **xld.\* %rd, [%rb+imm32] (\*=b, ub, h, uh, w) xld.\* [%rb+imm32], %rs (\*=b, h, w)**

Example: xld.w %rd, [%rb+imm32]

imm32 = 0x0	0x0 < imm32 ≤ 0x1fff	0x1fff < imm32 ≤ 0x3fffff
ld.w %rd, [%rb]	ext imm32(12:0) ld.w %rd, [%rb]	ext imm32(25:13) ext imm32(12:0) ld.w %rd, [%rb]
imm32 > 0x3fffff		
ext imm32(31:19) ext imm32(18:6) ld.w %r9, imm32(5:0) add %r9, %rb ld.w %rd, [%r9]		

Specification of imm32 can be omitted. If imm32 is omitted, the ext33 assumes that [%rb+0x0] is specified as it expands the instruction.

(8) **xld.w [%rb+imm32], %sp**

imm32 = 0x0	0x0 < imm32 ≤ 0x1fff	0x1fff < imm32 ≤ 0x3fffff
ld.w %r9, %sp ld.w [%rb], %r9	ld.w %r9, %sp ext imm32(12:0) ld.w [%rb], %r9	ld.w %r9, %sp ext imm32(25:13) ext imm32(12:0) ld.w [%rb], %r9
imm32 > 0x3fffff and %rb ≠ %r0	imm32 > 0x3fffff and %rb = %r0	
ld.w %r9, %sp pushn %r0 ext imm32(31:19) ext imm32(18:6) ld.w %r0, imm32(5:0) add %r0, %rb ld.w [%r0], %r9 popn %r0	ld.w %r9, %sp pushn %r1 ext imm32(31:19) ext imm32(18:6) ld.w %r1, imm32(5:0) add %r1, %rb ld.w [%r1], %r9 popn %r1	

Specification of imm32 can be omitted. If imm32 is omitted, the ext33 assumes that [%rb+0x0] is specified as it expands the instruction.



## 10.6.7 Immediate Data Load Instructions

### Types and functions of extended instructions

Extended instruction	Function	Expansion format
xld.w %rd, symbol±imm32	%rd ← symbol±imm32	(1)
xld.w %rd, sign32	%rd ← sign32	(2)

\* "symbol±imm32" means that "symbol+imm32" and "symbol-imm32" can be specified.

These extended instructions allow a 32-bit immediate to be loaded directly into a general-purpose register. A symbol also can be used for immediate specification.

### Basic instruction after expansion

xld.w Expanded into the ld.w instruction

### Expansion formats

(1) xld.w %rd, symbol+imm32      xld.w %rd, symbol-imm32

- When symbol+imm32 is specified

symbol+imm32 ≤ 0x1f	0x1f < symbol+imm32 ≤ 0x3fff	symbol+imm32 > 0x3fff
ld.w %rd, symbol+imm32	ext symbol+imm32@m ld.w %rd, symbol+imm32@l	ext symbol+imm32@h ext symbol+imm32@m ld.w %rd, symbol+imm32@l
Unknown symbol		
ext symbol+imm32@h ext symbol+imm32@m ld.w %rd, symbol+imm32@l		

Specification of imm32 can be omitted. If imm32 is omitted, the ext33 assumes that symbol+0x0 is specified as it expands the instruction.

- When symbol-imm32 is specified

The instruction is always expanded into the following format.

ext symbol-imm32@h
ext symbol-imm32@m
ld.w %rd, symbol-imm32@l

(2) xld.w %rd, sign32

-32 ≤ sign32 ≤ 31	-262144 ≤ sign32 < -32 or 31 < sign32 ≤ 262143	sign32 < -262144 or 262143 < sign32
ld.w %rd, sign32(5:0)	ext sign32(18:6) ld.w %rd, sign32(5:0)	ext sign32(31:19) ext sign32(18:6) ld.w %rd, sign32(5:0)

## 10.6.8 Bit Operation Instructions

### Types and functions of extended instructions

Extended instruction	Function	Expansion format
xbtst [symbol±imm32], imm3	B[symbol±imm32] bit test	(1)
xbclr [symbol±imm32], imm3	B[symbol±imm32] bit clear	(1)
xbset [symbol±imm32], imm3	B[symbol±imm32] bit set	(1)
xbnot [symbol±imm32], imm3	B[symbol±imm32] bit negation	(1)
xbtst [imm32], imm3	B[imm32] bit test	(2)
xbclr [imm32], imm3	B[imm32] bit clear	(2)
xbset [imm32], imm3	B[imm32] bit set	(2)
xbnot [imm32], imm3	B[imm32] bit negation	(2)
xbtst [%rb+symbol±imm32], imm3	B[%rb+symbol±imm32] bit test	(3)
xbclr [%rb+symbol±imm32], imm3	B[%rb+symbol±imm32] bit clear	(3)
xbset [%rb+symbol±imm32], imm3	B[%rb+symbol±imm32] bit set	(3)
xbnot [%rb+symbol±imm32], imm3	B[%rb+symbol±imm32] bit negation	(3)
xbtst [%rb+imm32], imm3	B[%rb+imm32] bit test	(4)
xbclr [%rb+imm32], imm3	B[%rb+imm32] bit clear	(4)
xbset [%rb+imm32], imm3	B[%rb+imm32] bit set	(4)
xbnot [%rb+imm32], imm3	B[%rb+imm32] bit negation	(4)
xbtst [%sp+imm32], imm3	B[%sp+imm32] bit test	(5)
xbclr [%sp+imm32], imm3	B[%sp+imm32] bit clear	(5)
xbset [%sp+imm32], imm3	B[%sp+imm32] bit set	(5)
xbnot [%sp+imm32], imm3	B[%sp+imm32] bit negation	(5)

\* "symbol±imm32" means that "symbol+imm32" and "symbol-imm32" can be specified.

These extended instructions allow a memory address for manipulating bits to be specified with a symbol or 32-bit immediate.

Note: The second operand (imm3) used to specify a bit number does not cause an error in the ext33 providing that it is within the range of values represented by unsigned 32 bits. It is output as the operand of a basic instruction directly as is. Note that the effective range of the basic instructions is 3 unsigned bits.

### Basic instructions after expansion

xbtst	Expanded into the btst instruction
xbclr	Expanded into the bclr instruction
xbset	Expanded into the bset instruction
xbnot	Expanded into the bnot instruction

### Expansion formats

(1) xOP [symbol+imm32], imm3    xOP [symbol-imm32], imm3    (OP = btst, bclr, bset, bnot)

- When [symbol+imm32] is specified

Example: xbtst [symbol+imm32], imm3

When global pointer is not specified:

symbol+imm32 ≤ 0x1f	0x1f < symbol+imm32 ≤ 0x3fff	symbol+imm32 > 0x3fff
ld.w    %r9, symbol+imm32@l btst    [%r9], imm3	ext    symbol+imm32@m ld.w    %r9, symbol+imm32@l btst    [%r9], imm3	ext    symbol+imm32@h ext    symbol+imm32@m ld.w    %r9, symbol+imm32@l btst    [%r9], imm3
Unknown symbol		
ext    symbol+imm32@h ext    symbol+imm32@m ld.w    %r9, symbol+imm32@l btst    [%r9], imm3		

When global pointer (gp) is specified:

(sign32 = -gp+imm32)

symbol+sign32 = 0x0	0x0 < symbol+sign32 ≤ 0x1fff	0x1fff < symbol+sign32 ≤ 0x3fffff
btst [%r8], imm3	ext symbol+sign32@al btst [%r8], imm3	ext symbol+sign32@ah ext symbol+sign32@al btst [%r8], imm3
symbol+sign32 > 0x3fffff	Unknown symbol	gp > symbol+imm32
ext symbol+imm32@h ext symbol+imm32@m ld.w %r9, symbol+imm32@l btst [%r9], imm3	ext symbol+sign32@ah ext symbol+sign32@al btst [%r8], imm3	Expanded into a format without gp specification according to the symbol+imm32 value.

Specification of imm32 can be omitted. If imm32 is omitted, the ext33 assumes that [symbol+0x0] is specified as it expands the instruction.

• When [symbol-imm32] is specified

The instruction is always expanded into the following format.

ext symbol-imm32@h
ext symbol-imm32@m
ld.w %r9, symbol-imm32@l
btst [%r9], imm3

(2) xOP [imm32], imm3 (OP = btst, bclr, bset, bnot)

Example: xbtst [imm32], imm3

When global pointer is not specified:

imm32 ≤ 0x1f	0x1f < imm32 ≤ 0x3fff	imm32 > 0x3fff
ld.w %r9, imm32(5:0) btst [%r9], imm3	ext imm32(18:6) ld.w %r9, imm32(5:0) btst [%r9], imm3	ext imm32(31:19) ext imm32(18:6) ld.w %r9, imm32(5:0) btst [%r9], imm3

When global pointer (gp) is specified:

(sign32 = -gp+imm32)

sign32 = 0x0	0x0 < sign32 ≤ 0x1fff	0x1fff < sign32 ≤ 0x3fffff
btst [%r8], imm3	ext sign32(12:0) btst [%r8], imm3	ext sign32(25:13) ext sign32(12:0) btst [%r8], imm3
sign32 > 0x3fffff	gp > imm32	
ext imm32(31:19) ext imm32(18:6) ld.w %r9, imm32(5:0) btst [%r9], imm3	Expanded into a format without gp specification according to the imm32 value.	

(3) xOP [%rb+symbol+imm32], imm3 xOP [%rb+symbol-imm32], imm3 (OP = btst, bclr, bset, bnot)

• When [%rb+symbol+imm32] is specified

Example: xbtst [%rb+symbol+imm32], imm3

symbol+imm32 = 0x0	0x0 < symbol+imm32 ≤ 0x1fff	0x1fff < symbol+imm32 ≤ 0x3fffff
btst [%rb], imm3	ext symbol+imm32@al btst [%rb], imm3	ext symbol+imm32@ah ext symbol+imm32@al btst [%rb], imm3
symbol+imm32 > 0x3fffff	Unknown symbol	
ext symbol+imm32@h ext symbol+imm32@m ld.w %r9, symbol+imm32@l add %r9, %rb btst [%r9], imm3	ext symbol+imm32@h ext symbol+imm32@m ld.w %r9, symbol+imm32@l add %r9, %rb btst [%r9], imm3	

Specification of imm32 can be omitted. If imm32 is omitted, the ext33 assumes that [%rb+symbol+0x0] is specified as it expands the instruction.

- When [%rb+symbol-imm32] is specified

Example: `xbtst [%rb+symbol-imm32], imm3`

The instruction is always expanded into the following format.

ext	symbol-imm32@h
ext	symbol-imm32@m
ld.w	%r9, symbol-imm32@l
add	%r9, %rb
btst	[%r9], imm3

(4) **xOP [%rb+imm32], imm3** (OP = `btst`, `bclr`, `bset`, `bnot`)

Example: `xbtst [%rb+imm32], imm3`

imm32 = 0x0	0x0 < imm32 ≤ 0x1fff	0x1fff < imm32 ≤ 0x3fffff
btst [%rb], imm3	ext imm32(12:0) btst [%rb], imm3	ext imm32(25:13) ext imm32(12:0) btst [%rb], imm3
imm32 > 0x3fffff		
ext imm32(31:19) ext imm32(18:6) ld.w %r9, imm32(5:0) add %r9, %rb btst [%r9], imm3		

Specification of imm32 can be omitted. If imm32 is omitted, the ext33 assumes that [%rb+0x0] is specified as it expands the instruction.

(5) **xOP [%sp+imm32], imm3** (OP = `btst`, `bclr`, `bset`, `bnot`)

Example: `xbtst [%sp+imm32], imm3`

imm32 = 0x0	0x0 < imm32 ≤ 0x3f	0x3f < imm32 ≤ 0x7ffff
ld.w %r9, %sp btst [%r9], imm3	ld.w %r9, %sp add %r9, imm32(5:0) btst [%r9], imm3	ld.w %r9, %sp ext imm32(18:6) add %r9, imm32(5:0) btst [%r9], imm3
imm32 > 0x7ffff		
ld.w %r9, %sp ext imm32(31:19) ext imm32(18:6) add %r9, imm32(5:0) btst [%r9], imm3		

## 10.6.9 Branch Instructions

### Types and functions of extended instructions

Extended instruction	Function	Expansion format
xcall label+imm32	Subroutine call	(1)
xcall.d label+imm32	Subroutine call (with delayed branch operation)	(1)
xjp label+imm32	Unconditional jump	(1)
xjp.d label+imm32	Unconditional jump (with delayed branch operation)	(1)
xjreq label+imm32	Conditional jump	(1)
xjreq.d label+imm32	Conditional jump (with delayed branch operation)	(1)
xjrne label+imm32	Conditional jump	(1)
xjrne.d label+imm32	Conditional jump (with delayed branch operation)	(1)
xjrgt label+imm32	Conditional jump	(1)
xjrgt.d label+imm32	Conditional jump (with delayed branch operation)	(1)
xjrge label+imm32	Conditional jump	(1)
xjrge.d label+imm32	Conditional jump (with delayed branch operation)	(1)
xjrft label+imm32	Conditional jump	(1)
xjrft.d label+imm32	Conditional jump (with delayed branch operation)	(1)
xjrle label+imm32	Conditional jump	(1)
xjrle.d label+imm32	Conditional jump (with delayed branch operation)	(1)
xjrugt label+imm32	Conditional jump	(1)
xjrugt.d label+imm32	Conditional jump (with delayed branch operation)	(1)
xjrue label+imm32	Conditional jump	(1)
xjrue.d label+imm32	Conditional jump (with delayed branch operation)	(1)
xjrult label+imm32	Conditional jump	(1)
xjrult.d label+imm32	Conditional jump (with delayed branch operation)	(1)
xjrle label+imm32	Conditional jump	(1)
xjrle.d label+imm32	Conditional jump (with delayed branch operation)	(1)
xcall sign32	Subroutine call	(2)
xcall.d sign32	Subroutine call (with delayed branch operation)	(2)
xjp sign32	Unconditional jump	(2)
xjp.d sign32	Unconditional jump (with delayed branch operation)	(2)
xjreq sign32	Conditional jump	(2)
xjreq.d sign32	Conditional jump (with delayed branch operation)	(2)
xjrne sign32	Conditional jump	(2)
xjrne.d sign32	Conditional jump (with delayed branch operation)	(2)
xjrgt sign32	Conditional jump	(2)
xjrgt.d sign32	Conditional jump (with delayed branch operation)	(2)
xjrge sign32	Conditional jump	(2)
xjrge.d sign32	Conditional jump (with delayed branch operation)	(2)
xjrft sign32	Conditional jump	(2)
xjrft.d sign32	Conditional jump (with delayed branch operation)	(2)
xjrle sign32	Conditional jump	(2)
xjrle.d sign32	Conditional jump (with delayed branch operation)	(2)
xjrugt sign32	Conditional jump	(2)
xjrugt.d sign32	Conditional jump (with delayed branch operation)	(2)
xjrue sign32	Conditional jump	(2)
xjrue.d sign32	Conditional jump (with delayed branch operation)	(2)
xjrult sign32	Conditional jump	(2)
xjrult.d sign32	Conditional jump (with delayed branch operation)	(2)
xjrle sign32	Conditional jump	(2)
xjrle.d sign32	Conditional jump (with delayed branch operation)	(2)

These extended instructions allow a branch destination to be specified using a label with displacement included or a signed 32-bit immediate. The branch conditions of these conditional jump instructions are the same as those of the basic instructions.

**Basic instructions after expansion**

xcall/xcall.d	Expanded into the call/call.d instruction
xjp/xjp.d	Expanded into the jp/jp.d instruction
xjreq/xjreq.d	Expanded into the jreq/jreq.d instruction
xjrne/xjrne.d	Expanded into the jrne/jrne.d instruction
xjrgt/xjrgt.d	Expanded into the jrgt/jrgt.d instruction
xjrge/xjrge.d	Expanded into the jrge/jrge.d instruction
xjrle/xjrle.d	Expanded into the jrle/jrle.d instruction
xjrugt/xjrugt.d	Expanded into the jrugt/jrugt.d instruction
xjruge/xjruge.d	Expanded into the jruge/jruge.d instruction
xjrult/xjrult.d	Expanded into the jrult/jrult.d instruction
xjrle/xjrle.d	Expanded into the jrle/jrle.d instruction

**Expansion formats****(1) xOP label+imm32 (OP = call, call.d, jp, jp.d, jr\*, jr\*.d)**

Example: xcall label+imm32

When expanded into 1 instruction	When expanded into 2 instructions	When expanded into 3 instructions
call label+imm32	ext label+imm32@rm call label+imm32@rl	ext label+imm32@rh ext label+imm32@rm call label+imm32@rl

Specification of imm32 can be omitted. If imm32 is omitted, the ext33 assumes that "label+0x0" is specified as it expands the instruction. For details on how the number of instructions when expanded is determined, refer to Section 10.7.1, "Optimizing Relative Branch Instruction".

**(2) xOP sign32 (OP = call, call.d, jp, jp.d, jr\*, jr\*.d)**

Example: xcall sign32

-256 ≤ sign32 ≤ 254	-2097152 ≤ sign32 < -256 or 254 < sign32 ≤ 2097150	sign32 > 2097150 or sign32 < -2097152
call sign32(8:1)	ext sign32(21:9) call sign32(8:1)	ext sign32(31:22)<<0x3 ext sign32(21:9) call sign32(8:1)

## 10.7 Optimize Function

### 10.7.1 Optimizing Relative Branch Instruction

The E0C33000's basic relative branch instruction allows branch to any location within a signed 9-bit (LSB = 0) relative address range by using a single instruction. Branching to an address beyond this range requires extension by the ext instruction.

The ext33 allows you to write an extended branch instruction without worrying about the ext instruction as described in Section 10.6.9. Thus instruction is then expanded so that control is transferred a relative distance to the branch destination label by using the fewest possible instructions.

The following extended branch instructions can be optimized:

```
xcall xjp xjreq xjrne xjrgt xjrge xjrld xjrle xjrugt xjruge xjrult xjrule
xcall.d xjp.d xjreq.d xjrne.d xjrgt.d xjrge.d xjrld.d xjrle.d xjrugt.d xjruge.d xjrult.d xjrule.d
```

The basic branch instructions written by adding the ext instruction are not optimized.

The number of instructions (1 to 3 instructions) that derive from expansion are determined according to the following conditions:

- Relative distance between the extended branch instruction and branch destination label
- Whether the extended branch instruction and branch destination label exist in the same file
- Whether the -near option is specified
- Whether there is symbol file/link map file specification by the -lk option
- Relative distance determination threshold 0x180000 (default) or value specified by the -j option

The number of instructions derived by expansion are determined in the following manner by resolving the above conditions:

Table 10.7.1.1 Number of instructions derived by expansion

Relative distance (absolute value) *1	Instruction and label positions	-near flag	Number of expanded instructions
0 to 126 *2	In the same file	–	1
	In different files	–	2
To threshold value *3	–	–	2
To 0x7fffff *4	–	Specified	2
	–	Not specified	3
Unknown relative distance	–	Specified	2
	–	Not specified	3

\*1: The value indicates the number of instructions from the extended branch instruction to the branch destination label.

\*2: Up to 125 when branching toward to a higher address.

\*3: Up to the threshold value - 1 when branching toward to a higher address in the same file.

\*4: Up to 0x7fffff when branching toward to a higher address in the same file.

The threshold value is half of the value specified using -j option. When using the -j option's default value of 0x180000, the threshold value will be 0xc0000. Values in ( ) apply when branching to a lower address. The threshold value may be decreased due to distance judgment when branching toward to a lower address.

The following shows the basic format after expansion.

Example: xcall sign32

1 instruction	2 instructions	3 instructions
call sign32(8:1)	ext sign32(21:9)	ext sign32(31:22)<<0x3
	call sign32(8:1)	ext sign32(21:9)
		call sign32(8:1)

An expansion result different from those shown above may be obtained depending on method of label specification. For details, refer to Section 10.6.9.

When the `-lk` option is specified, branching to a different file based on the postlink symbol information can be optimized. However, since execution by cashing into the internal RAM is possible, branching to a label in a different file – although in the object file it may be branching within the 1-instruction range – is expanded as consisting of at least two instructions.

## 10.7.2 Optimization by the Global Pointer

Memory access by an extended instruction using a symbol is accomplished by using the R9 register as a scratch register, and is expanded in the following manner:

```
Example: xld.w  [i],%r10  (Example of expansion before linkage)
             ext        i+0x0@h
             ext        i+0x0@m
             ld.w      %r9,i+0x0@l
             ld.w      [%r9],%r10
```

When accessing a memory-resident global variable, for example, the number of instructions derived from expansion can be reduced by setting the start address of the global variable area as a global pointer in advance.

Since the `ext33` uses the R8 register as a global pointer, the address of the global pointer must be set to the R8 register in the initialize routine in advance. Note that the memory accessible range is limited to within +26-bit space from the global pointer address.

Specification for the `ext33` is made by using the `-gp <address>` option (`[global pointer optimize]` in the `wb33`). When this option is specified, the `ext33` assumes that the R8 register is set as a global pointer to the specified `<address>` as it processes the program.

If a global pointer is specified, the above example will be expanded as follows:

```
Example: xld.w  [i],%r10  (Example of expansion before linkage)
             ext        i+0x0@ah
             ext        i+0x0@al
             ld.w      [%r8],%r10
```

In this case, since no scratch register is used, the number of instructions for each access can be reduced by one.

## 10.7.3 Optimization by Symbol Information

When creating one program by linking multiple relocatable modules, it should be noted that the absolute address of each instruction in the assembly source is not determined until after the modules are linked. Optimization at this phase is limited to those instructions that can be solved within the same file.

For this reason, the `ext33` is designed in such a way that symbol information can be obtained from the symbol and link map files output by the linker by specifying the `-lk` option. Since each of these files contains information about the absolute addresses of the symbols that are determined after linkage, the symbols defined in other files can be referenced.

Therefore, make sure all source files including the `ext33` are processed temporarily way up to the linking phase, then re-execute the `ext33` after specifying the `-lk` option. The `ext33` optimizes the extended instructions that are used to access memory locations using indeterminate symbols as it generates an assembly source file. Then, after this is done, assemble and link the source files one more time.

To generate the symbol and link map files, you need to specify the `-m` and `-s` options when starting up the linker.

If the variable `i`, which was used as an example in the preceding section, is assumed to be located in the 4th byte from the address indicated by the global pointer, then the first `ext` instruction will be deleted as follows:

```
Example: xld.w  [i],%r10  (Example of expansion using postlink symbol information)
             ext        i+0x0@al
             ld.w      [%r8],%r10
```

Furthermore, if there is a memory access that is out of the accessible range by the global pointer, this function disables the global pointer optimization.

The `ext33` checks to see if the symbol and map files bearing the file name specified by the `-lk` option are created in the same directory. If one or both of the two files cannot be found, the `ext33` outputs a warning and stops performing `-lk` option-based optimization. Therefore, these files must always be stored under the same name in the same directory.



## 10.8 Other Functions

---

### 10.8.1 Comment Adding Function

When expanding extended instructions to create an assembly source file, the ext33 replaces these extended instructions with comments that begin with a semicolon (;) in order to show information about the contents of the extended instructions before they are expanded. The comments are added after the first line of instruction derived by expansion. If there is any comment accompanying the original statement, that comment is included among the added comments.

Example: Before expansion

```
xcall    main    ; goto main
```

After expansion

```
ext      main@rm ;          xcall    main    ; goto main
call     main@rl
```

### 10.8.2 Classification of Local Symbols

The ext33 classifies the local labels that are valid in only the files generated by the Preprocessor and Compiler, then changes the labels "\_\_L???" used in extended relative branch instructions to "\_\_LX???" The changed labels have already been referenced in their file.

As a result, the Assembler as33 interprets the information on labels beginning with "\_\_LX" as being local; therefore, this information is not output to the object file.

Local symbols used in any other instructions are not changed even if they are labeled "\_\_L???".

### 10.8.3 Syntactic Check

The ext33 only checks the syntax of extended instructions and the assembler pseudo-instructions listed below.

#### Assembler pseudo-instructions checked by ext33

.org, .space, .align, .comm, .lcomm, .set pseudo-instructions

These instructions are checked to see if the address-specifying operand is within the effective range of values represented by 32 bits.

.ascii pseudo-instruction

This instruction is checked to see if character strings are enclosed with double quotations (").

An error results if any extended instruction or one of the above assembler pseudo-instructions is written in a syntactically incorrect manner. If the operand value is invalid, a warning is output, and processing continues.

The syntax of the basic instructions is not checked. Nor is the validity of the instructions derived by expansion checked.

## 10.9 Sample Execution

### Input file (ext.ps)

```
; ext.ps 1997.2.23
; sample source for ext33 extended instructions
; not for execution, just for ext33 extension only
```

```
.word BOOT
```

```
BOOT:
```

```
; summary of major patterns
```

```
; xld.w
```

```
xld.w    %r8, 0           ; immediate load
xld.w    %r1, DATA1     ; symbol immediate load
xld.w    %r2, DATA1+4   ; symbol+offset

xadd     %r1, %r2, 0x12345678 ; 3 operand for xadd, xsub

xand     %r14, %r15, 0xff000000 ; 3 operand for xand, xoor, xxor
xnot     %r8, 0b1111100000

xsrl     %r3, 8           ; immediate shift
xrr      %r7, %r8        ; register shift
                        ; for xsrl, xsll, xsra, xsla, xrr, xrl
```

```
xld.w    %r1, [0x1234568] ; immediate address
xld.ub   %r5, [DATA1]     ; symbol address
xbstst   [COMM1+0x400], 2 ; symbol address + offset
xld.uh   %r10, [%sp+0x222] ; sp relative
xld.b    [%sp], %r7       ; sp relative
xld.uh   %r1, [%r15+0x1234568] ; register + immediate address
xld.h    %r5, [%r11+DATA1] ; register + symbol address
xbset    [%r9+COMM1+0x400], 2 ; register + symbol address + offset
                        ; for xld.w, xld.uh, xld.h, xld.ub, xld.b,
                        ; xbst, xbcrl, xbst, xbnot

xjp      -2               ; immediate relative
xjrgt.d  BOOT             ; symbol relative
                        ; for xjp, xjreq, xjrne, xjrgt, xjrge, xjrlt, xjrle,
                        ; xjrugt, xjruge, xjrult, xjrule, xcall
```

```
; more detail samples
```

```
; xld.w load immediate to register operation
```

```
xld.w    %r8, 0           ; decimal
xld.w    %r0, 0x12345678 ; hex
xld.w    %r0, 0b10101    ; binary
xld.w    %r1, DATA1     ; symbol
xld.w    %r2, DATA1+4   ; symbol+offset (hex, dec, bin)
xld.w    %r2, DATA1+0x5
xld.w    %r2, DATA1+0b110
```

```
; xadd, xsub add and sub, arithmetic operations
```

```
xadd     %r1, %r2, 0x12345678 ; 3 operand No. 1
xsub     %r2, %r1, 0x12345     ; 3 operand No. 2
xadd     %r0, %r1, 1          ; 3 operand No. 3
xsub     %r2, %r2, 5          ; 3 operand No. 4
xadd     %r1, %r2, %sp        ; for C compiler
xsub     %sp, %sp, %r1        ; for C compiler
```

## CHAPTER 10: INSTRUCTION EXTENDER

```
; xand, xoor, xxor, xnot
;
;          and, or, xor, and not, logical operations
;
;          xand      %r14, %r15, 0xff000000 ; 3 operand No. 1
;          xoor      %r12, %r11, 0xfedc    ; 3 operand No. 2
;          xxor      %r9, %r9, -1          ; 3 operand No. 3
;          xnot      %r8, 0b1111100000
;
; xsrl, xsll, xsra, xsla, xrr, xrl      shift operations
;
;          xsrl      %r3, 8                ; immediate shift No. 1
;          xsll      %r4, 15               ; immediate shift No. 2
;          xsra      %r5, 17               ; immediate shift No. 3
;          xsla      %r6, 31               ; immediate shift No. 4
;          xrr       %r7, %r8              ; register shift
;
; xld.w, xld.uh, xld.h, xld.ub, xld.b, xbset, xbcclr, xbtst, xbnot
;
;          load, bit operation from/to absolute address
;
;          xld.w     %r1, [0x1234568]      ; immediate address No. 1
;          xld.uh    %r2, [0xABC]         ; immediate address No. 2
;          xld.h     [10], %r3            ; immediate address No. 3
;          xld.ub    %r4, [0]             ; immediate address No. 4
;          xld.b     %r5, [DATA1]         ; symbol address No. 1
;          xbnot     [COMM1], 1           ; symbol address No. 2
;          xbtst     [COMM1+0x400], 2     ; symbol address + offset No. 1
;          xbset     [COMM1+0x10], 3      ; symbol address + offset No. 2
;          xbcclr    [COMM1+1], 4        ; symbol address + offset No. 3
;
; xld.w, xld.uh, xld.h, xld.ub, xld.b, xbset, xbcclr, xbtst, xbnot
;
;          load, bit operation from/to SP relative address
;
;          xld.w     %r15, [%sp+0x4444444] ; sp relative No. 1
;          xld.uh    %r10, [%sp+0x222]     ; sp relative No. 2
;          xld.b     [%sp], %r7           ; sp relative No. 3
;          xbset     [%sp+0x14], 5         ; sp relative No. 4
;
; xld.w, xld.uh, xld.h, xld.ub, xld.b, xbset, xbcclr, xbtst, xbnot
;
;          load, bit operation from/to register relative address
;
;          xld.w     %r1, [%r15+0x1234568] ; + immediate address No. 1
;          xld.uh    %r2, [%r14+0xABC]     ; + immediate address No. 2
;          xld.h     [%r13+10], %r3        ; + immediate address No. 3
;          xld.ub    %r4, [%r12]          ; + immediate address No. 4
;          xld.b     %r5, [%r11+DATA1]     ; + symbol address No. 1
;          xbnot     [%r10+COMM1], 1       ; + symbol address No. 2
;          xbtst     [%r9+COMM1+0x400], 2  ; + symbol address + offset No. 1
;          xbset     [%r8+COMM1+0x10], 3   ; + symbol address + offset No. 2
;          xbcclr    [%r7+COMM1+1], 4     ; + symbol address + offset No. 3
;
; xld.w      load word operation from sp register for C support
;
;          xld.w     [%sp], %sp
;          xld.w     [%sp+0x2468], %sp
;          xld.w     [0x12340], %sp
;          xld.w     [COMM1], %sp
;          xld.w     [COMM1+4], %sp
;          xld.w     [%r5], %sp
;          xld.w     [%r6+0b1100], %sp
;          xld.w     [%r7+DATA1], %sp
;          xld.w     [%r7+DATA1+200], %sp
;
; xjp, xjreq, xjrne, xjrgt, xjrge, xjrslt, xjrle, xjrugt, xjruge, xjrult
; xjrule, xcall and with .d      relative branches
```

```
NEAR:
    xjp.d      -2          ; immediate relative No.1
    xjreq      800        ; immediate relative No.2
    xjrne      0x1000000  ; immediate relative No.3
    xjrgt.d    BOOT       ; symbol relative No.1
    xjrge      COMM1     ; symbol relative No.2
    xjrge      NEAR       ; symbol relative No.3
```

```
.data
DATA1:
    .word      0x12345678
    .comm      COMM1 4
```

### Output file (ext.ms) when "ext33 -gp 0x0 text.ps" is executed

(\* In actual operation, a comment may be output at a different position.)

```
; ext.ps 1997.2.23
; sample source for ext33 extended instructions
; not for execution, just for ext33 extension only
```

```
.word BOOT
BOOT:
```

```
; summary of major patterns
```

```
; xld.w
```

```
ld.w      %r8, 0x0          ; xld.w  %r8, 0          ; immediate load
ext       DATA1+0x0@h      ; xld.w  %r1, DATA1     ; symbol immediate load
ext       DATA1+0x0@m
ld.w      %r1, DATA1+0x0@l
ext       DATA1+0x4@h      ; xld.w  %r2, DATA1+4   ; symbol+offset
ext       DATA1+0x4@m
ld.w      %r2, DATA1+0x4@l

ld.w      %r1, %r2          ; xadd   %r1, %r2, 0x12345678 ; 3 operand for xadd, xsub
ext       0x246
ext       0x1159
add       %r1, 0x38

ld.w      %r14, %r15        ; xand   %r14, %r15, 0xff000000 ; 3 operand for xand, xoor, xoror
ext       0x1fe0
ext       0x0
and       %r14, 0x0
ext       0xf              ; xnot   %r8, 0b1111100000
not      %r8, 0x20

srl       %r3, 0x8          ; xsrl   %r3, 8          ; immediate shift
ld.w      %r9, %r8          ; xrr    %r7, %r8        ; register shift
and       %r9, 0x1f
cmp       %r9, 0x8
jrle     4
rr       %r7, 0x8
jp.d     -3
sub      %r9, 0x8
rr       %r7, %r9
; for xsrl, xsll, xsra, xsla, xrr, xrl

ext       0x91a            ; xld.w  %r1, [0x1234568]   ; immediate address
ext       0x568
ld.w      %r1, [%r8]
ext      DATA1+0x0@ah      ; xld.ub %r5, [DATA1]     ; symbol address
ext      DATA1+0x0@al
ld.ub    %r5, [%r8]
```

```

ext      COMM1+0x400@ah      ; xbtst  [COMM1+0x400], 2      ; symbol address + offset
ext      COMM1+0x400@a1
btst    [%r8], 0x2
ext      0x8                 ; xld.uh  %r10, [%sp+0x222]    ; sp relative
ld.uh   %r10, [%sp+0x22]
ld.b    [%sp+0x0], %r7      ; xld.b   [%sp], %r7          ; sp relative
ext      0x91a              ; xld.uh  %r1, [%r15+0x1234568] ; register + immediate address
ext      0x568
ld.uh   %r1, [%r15]
ext      DATA1+0x0@h      ; xld.h   %r5, [%r11+DATA1]   ; register + symbol address
ext      DATA1+0x0@m
ld.w    %r9, DATA1+0x0@l
add     %r9, %r11
ld.h    %r5, [%r9]
ext      COMM1+0x400@h      ; xbsset  [%r9+COMM1+0x400], 2
                                           ; register + symbol address + offset

ext      COMM1+0x400@m
ld.w    %r9, COMM1+0x400@l
add     %r9, %r9
bset    [%r9], 0x2
                                           ; for xld.w, xld.uh, xld.h, xld.ub, xld.b,
                                           ; xbsset, xbc1r, xbtst, xbnot

jp      0xff                ; xjp     -2                  ; immediate relative
ext     BOOT@rh            ; xjrgt.d BOOT                ; symbol relative
ext     BOOT@rm
jrgt.d  BOOT@rl
                                           ; for xjp, xjreq, xjrne, xjrgt, xjrge, xjr1t, xjr1e,
                                           ; xjrugt, xjruge, xjrult, xjrule, xcall

```

; more detail samples

; xld.w load immediate to register operation

```

ld.w    %r8, 0x0            ; xld.w   %r8, 0              ; decimal
ext     0x246               ; xld.w   %r0, 0x12345678    ; hex
ext     0x1159
ld.w    %r0, 0x38          ; xld.w   %r0, 0b10101      ; binary
ld.w    %r0, 0x15          ; xld.w   %r1, DATA1       ; symbol
ext     DATA1+0x0@h
ext     DATA1+0x0@m
ld.w    %r1, DATA1+0x0@l
ext     DATA1+0x4@h      ; xld.w   %r2, DATA1+4     ; symbol+offset (hex, dec, bin)
ext     DATA1+0x4@m
ld.w    %r2, DATA1+0x4@l
ext     DATA1+0x5@h      ; xld.w   %r2, DATA1+0x5
ext     DATA1+0x5@m
ld.w    %r2, DATA1+0x5@l
ext     DATA1+0x6@h      ; xld.w   %r2, DATA1+0b110
ext     DATA1+0x6@m
ld.w    %r2, DATA1+0x6@l

```

; xadd, xsub add and sub. arithmetic operations

```

ld.w    %r1, %r2           ; xadd    %r1, %r2, 0x12345678 ; 3 operand No. 1
ext     0x246
ext     0x1159
add     %r1, 0x38
ext     0x9                ; xsub    %r2, %r1, 0x12345    ; 3 operand No. 2
ext     0x345
sub     %r2, %r1
ext     0x1                ; xadd    %r0, %r1, 1         ; 3 operand No. 3
add     %r0, %r1
sub     %r2, 0x5           ; xsub    %r2, %r2, 5        ; 3 operand No. 4
ld.w    %r9, %sp          ; xadd    %r1, %r2, %sp     ; for C compiler

```

```

add      %r1,%r9
ld.w    %r9,%sp      ; xsub      %sp,%sp,%r1      ; for C compiler
sub     %r9,%r1
ld.w    %sp,%r9

; xand, xoor, xxor, xnot
;
;      and, or, xor, and not, logical operations

ld.w    %r14,%r15    ; xand      %r14,%r15,0xff000000 ; 3 operand No. 1
ext     0x1fe0
ext     0x0
and     %r14,0x0
ext     0x7          ; xoor      %r12,%r11,0xfedc      ; 3 operand No. 2
ext     0x1edc
or      %r12,%r11
xor     %r9,0x3f     ; xxor     %r9,%r9,-1      ; 3 operand No. 3
ext     0xf          ; xnot     %r8,0b1111100000
not     %r8,0x20

; xsrl, sll, xsra, xsla, xrr, xrl      shift operations

srl     %r3,0x8      ; xsrl     %r3,8          ; immediate shift No. 1
sll     %r4,0x8      ; sll      %r4,15         ; immediate shift No. 2
sll     %r4,0x7
sra     %r5,0x8      ; xsra     %r5,17         ; immediate shift No. 3
sra     %r5,0x8
sra     %r5,0x1
sla     %r6,0x8      ; xsla     %r6,31         ; immediate shift No. 4
sla     %r6,0x8
sla     %r6,0x8
sla     %r6,0x7
ld.w    %r9,%r8      ; xrr      %r7,%r8       ; register shift
and     %r9,0x1f
cmp     %r9,0x8
jrle   4
rr      %r7,0x8
jp.d   -3
sub     %r9,0x8
rr      %r7,%r9

; xld.w, xld.uh, xld.h, xld.ub, xld.b, xbsset, xbcrl, xbtst, xbnot
;
;      load, bit operation from/to absolute address

ext     0x91a        ; xld.w    %r1,[0x1234568] ; immediate address No. 1
ext     0x568
ld.w    %r1,[%r8]
ext     0xabc        ; xld.uh   %r2,[0xABC]    ; immediate address No. 2
ld.uh   %r2,[%r8]
ext     0xa          ; xld.h    [10],%r3       ; immediate address No. 3
ld.h    [%r8],%r3
ld.ub   %r4,[%r8]   ; xld.ub   %r4,[0]       ; immediate address No. 4
ext     DATA1+0x0@ah ; xld.b    %r5,[DATA1]   ; symbol address No. 1
ext     DATA1+0x0@al
ld.b    %r5,[%r8]
ext     COMM1+0x0@ah ; xbnot    [COMM1],1     ; symbol address No. 2
ext     COMM1+0x0@al
bnot    [%r8],0x1
ext     COMM1+0x400@ah ; xbtst    [COMM1+0x400],2 ; symbol address + offset No. 1
ext     COMM1+0x400@al
btst    [%r8],0x2
ext     COMM1+0x10@ah ; xbsset   [COMM1+0x10],3  ; symbol address + offset No. 2
ext     COMM1+0x10@al
bset    [%r8],0x3
ext     COMM1+0x1@ah ; xbcrl    [COMM1+1],4    ; symbol address + offset No. 3
ext     COMM1+0x1@al

```

```

        bclr      [%r8], 0x4

; xld.w, xld.uh, xld.h, xld.ub, xld.b, xbsset, xbcrl, xbtst, xbnor
;
        load, bit operation from/to SP relative address

        ext      0x88          ; xld.w   %r15, [%sp+0x4444444] ; sp relative No. 1
        ext      0x1111
        ld.w     %r15, [%sp+0x4]
        ext      0x8          ; xld.uh   %r10, [%sp+0x222]      ; sp relative No. 2
        ld.uh   %r10, [%sp+0x22]
        ld.b     [%sp+0x0], %r7 ; xld.b   [%sp], %r7          ; sp relative No. 3
        ld.w     %r9, %sp      ; xbsset  [%sp+0x14], 5      ; sp relative No. 4
        add     %r9, 0x14
        bset     [%r9], 0x5

; xld.w, xld.uh, xld.h, xld.ub, xld.b, xbsset, xbcrl, xbtst, xbnor
;
        load, bit operation from/to register relative address

        ext      0x91a        ; xld.w   %r1, [%r15+0x1234568] ; + immediate address No. 1
        ext      0x568
        ld.w     %r1, [%r15]
        ext      0xabc        ; xld.uh   %r2, [%r14+0xABC]      ; + immediate address No. 2
        ld.uh   %r2, [%r14]
        ext      0xa          ; xld.h   [%r13+10], %r3    ; + immediate address No. 3
        ld.h    [%r13], %r3
        ld.ub   %r4, [%r12]    ; xld.ub  %r4, [%r12]      ; + immediate address No. 4
        ext     DATA1+0x0@h   ; xld.b   %r5, [%r11+DATA1] ; + symbol address No. 1
        ext     DATA1+0x0@m
        ld.w    %r9, DATA1+0x0@l
        add    %r9, %r11
        ld.b    %r5, [%r9]
        ext     COMM1+0x0@h    ; xbnor   [%r10+COMM1], 1    ; + symbol address No. 2
        ext     COMM1+0x0@m
        ld.w    %r9, COMM1+0x0@l
        add    %r9, %r10
        bnot   [%r9], 0x1
        ext     COMM1+0x400@h   ; xbtst   [%r9+COMM1+0x400], 2 ; + symbol address + offset No. 1
        ext     COMM1+0x400@m
        ld.w    %r9, COMM1+0x400@l
        add    %r9, %r9
        btst  [%r9], 0x2
        ext     COMM1+0x10@h    ; xbsset  [%r8+COMM1+0x10], 3    ; + symbol address + offset No. 2
        ext     COMM1+0x10@m
        ld.w    %r9, COMM1+0x10@l
        add    %r9, %r8
        bset   [%r9], 0x3
        ext     COMM1+0x1@h     ; xbcrl   [%r7+COMM1+1], 4      ; + symbol address + offset No. 3
        ext     COMM1+0x1@m
        ld.w    %r9, COMM1+0x1@l
        add    %r9, %r7
        bclr   [%r9], 0x4

; xld.w
        load word operation from sp register for C support

        ld.w    %r9, %sp      ; xld.w   [%sp], %sp
        ld.w    [%sp+0x0], %r9
        ld.w    %r9, %sp      ; xld.w   [%sp+0x2468], %sp
        ext     0x91
        ld.w    [%sp+0x28], %r9
        ld.w    %r9, %sp      ; xld.w   [0x12340], %sp
        pushn  %r0
        ext     0x48d
        ld.w    %r0, 0x0
        ld.w    [%r0], %r9
        popn   %r0
    
```

```

ld.w      %r9, %sp          ; xld.w  [COMM1], %sp
pushn    %r0
ext      COMM1+0x0@h
ext      COMM1+0x0@m
ld.w     %r0, COMM1+0x0@l
ld.w     [%r0], %r9
popn     %r0
ld.w     %r9, %sp          ; xld.w  [COMM1+4], %sp
pushn    %r0
ext      COMM1+0x4@h
ext      COMM1+0x4@m
ld.w     %r0, COMM1+0x4@l
ld.w     [%r0], %r9
popn     %r0
ld.w     %r9, %sp          ; xld.w  [%r5], %sp
ld.w     [%r5], %r9
ld.w     %r9, %sp          ; xld.w  [%r6+0b1100], %sp
ext      0xc
ld.w     [%r6], %r9
ld.w     %r9, %sp          ; xld.w  [%r7+DATA1], %sp
pushn    %r0
ext      DATA1+0x0@h
ext      DATA1+0x0@m
ld.w     %r0, DATA1+0x0@l
add      %r0, %r7
ld.w     [%r0], %r9
popn     %r0
ld.w     %r9, %sp          ; xld.w  [%r7+DATA1+200], %sp
pushn    %r0
ext      DATA1+0xc8@h
ext      DATA1+0xc8@m
ld.w     %r0, DATA1+0xc8@l
add      %r0, %r7
ld.w     [%r0], %r9
popn     %r0

; xjp, xjreq, xjrne, xjrgt, xjrge, xjrle, xjrle, xjrugt, xjruge, xjrult
; xjrule, xcall and with .ld          relative branches

NEAR:
xjp.d    -2                ; immediate relative No.1
ext      0x1                ; xjreq  800          ; immediate relative No.2
jreq     0x90
ext      0x20                ; xjrne  0x1000000 ; immediate relative No.3
ext      0x0
jrne     0x0
xjrgt.d  BOOT              ; symbol relative No.1
ext      COMM1@rh          ; xjrge  COMM1      ; symbol relative No.2
ext      COMM1@rm
jrge     COMM1@rl
jruge    NEAR              ; xjruge  NEAR      ; symbol relative No.3

.data
DATA1:
.word    0x12345678

.comm   COMM1 4

```



## 10.10 Error/Warning Messages

Error and warning messages are displayed/output through the Standard Output (stdout). If you specify the `-e` option, the messages will also be delivered in the "ext33.err" file.

If the ext33 is started up using the wb33's [EXT33] button, the message is output to "wb33.err". When execution is completed, a message is displayed in the output window (default).

### 10.10.1 Errors

The errors produced in the ext33 are classified into two groups: system errors and extender errors.

System errors refer to those that make it impossible to carry on the processing. If a system error occurs, the ext33 will immediately terminate the processing after displaying an error message. No assembly source file will be output.

Table 10.10.1.1 List of system error messages

Error message	Content
Error: Cannot allocate memory.	Cannot secure memory space.
Error: Cannot open input file "<file name>".	Cannot open the source file. The file does not exist in the specified directory.
Error: Invalid input filename "<file name>".	Cannot open the source file. An illegal input file name was specified.
Error: Cannot open output file "<file name>".	Cannot open the output file.
Error: Cannot open error file "ext33.err".	Cannot open the error file.
Error: Cannot open command file "<file name>".	Cannot open the command file. The file does not exist in the specified directory.
Error: Cannot write error file.	Cannot write to the error file.
Error: Cannot write output file "<file name>".	Cannot write to the output file.

The extender errors are produced when the command line, command file or source file contains a syntax or description that cannot be processed by the ext33. No assembly source file will be output.

Table 10.10.1.2 List of extender error messages

Error message	Content
Error: Too long filename "<file name>".	File name is excessively long. A file name, including path, must be 255 characters or less.
Error: Invalid command file description "<description>".	<description> in command file is invalid. Write one option and one input file in each line. Begin a comment with ";" and do not write it on the same line as the other option or input file.
Error: Invalid command file format.	Command file format is invalid. The file format specified here is not that of a text file.
Error: Invalid jump threshold "<parameter>".	Parameter of the -j option is invalid. The threshold must be specified within a range of 0x100 to 0x1ffff.
Error: Invalid GP address "<parameter>".	Parameter of the -gp option is invalid. Specify an effective hexadecimal address.
Error: Too many input files.	There are too many input files. More than 682 input files are specified.
Error: No input file is specified.	No input file is specified.
<file name>(line No.): Error: Invalid syntax. *1	Extended instruction has a syntax error.

\*1 When the source file that contains debugging information is input, the ext33 displays "near <file name>(line No.>)" after the message. It consists of the original source file name (\*.c, \*.s) and the line number indicated in the debugging information.

## 10.10.2 Warning

Even when a warning appears, the ext33 will keep on processing, and completes the processing after displaying a warning message, unless any error is produced in addition. The assembly source file will be output.

Table 10.10.2.1 List of warning messages

Warning messages	Content
Warning: Map file "<file name>" does not exist.	Link map file cannot be found. Processing is continued after the specification of the -lk option is invalidated.
Warning: Symbol file "<file name>" does not exist.	Symbol file cannot be found. Processing is continued after the specification of the -lk option is invalidated.
<file name>: Warning: No map information in map file.	Map information corresponding to the input file does not exist in the link map file. Processing is continued after the specification of the -lk option is invalidated.
<file name>: Warning: Invalid map file format.	This is an invalid map file format. Processing is continued after the specification of the -lk option is invalidated.
<file name>: Warning: Invalid symbol file format.	This is an invalid symbol file format. Processing is continued after the specification of the -lk option is invalidated.
<file name>: Warning: Cannot find the symbol "<symbol>" in symbol table. *1	Information of <symbol> cannot be found in the symbol file. Processing is continued by assuming that <symbol> is undefined.
<file name>: Warning: Operand exceeds maximum address. *1	Operand address exceeds the effective range of values represented by 32 bits.
<file name>: Warning: Invalid address operand. *1	Invalid boundary address is specified. An address whose LSB is not 0 is specified in an extended instruction that handles half word data. An address whose two lower bits are not 0 is specified in an extended instruction that handles word data.
<file name>: Warning: Invalid operand value. *1	A value exceeding the effective range is specified in the operand of an extended instruction.

\*1 When the source file that contains debugging information is input, the ext33 displays "near <file name>(line No.>)" after the message. It consists of the original source file name (\*.c, \*.s) and the line number indicated in the debugging information.

## 10.11 Precautions

- (1) In the ext33, the general-purpose register R8 is reserved for use as a global pointer and the register R9 is reserved for use as a scratch register for extended instructions. Do not use these two registers when creating assembly sources.
- (2) The ext33 only performs the syntactic check that is necessary for the expansion and optimization of extended instructions. The validity of the instructions derived by expansion is not checked. Nor does the ext33 check the mnemonics, operands, or assembler pseudo-instructions (except a few).

## Chapter 11 Assembler

This chapter describes the functions of the Assembler as33.

For the syntax of the assembly sources, refer to Section 4.3, "Grammar of Assembly Source".

### 11.1 Functions

The Assembler as33 (hereafter called the "as33") assembles (translates) assembly source files that are delivered by the Instruction Extender and creates object files in the machine language.

The functions and features of the as33 are summarized below:

- Supports the absolute assembling and the relocatable assembling.
- Allows to develop programs by module.
- Can deliver debugging information for purposes of symbolic debugging.

### 11.2 Input/Output Files

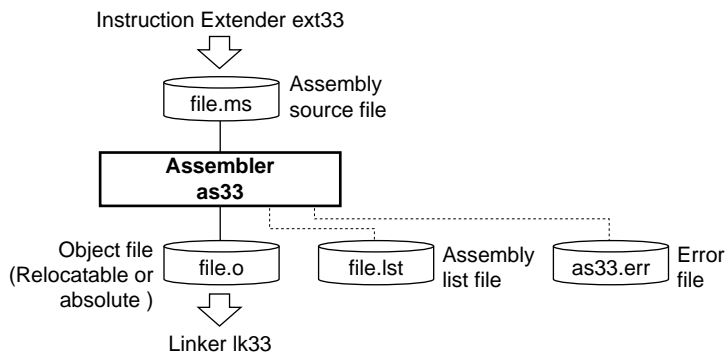


Fig. 11.2.1 Flowchart

#### 11.2.1 Input File

##### Assembly source file

File format: Text file

File name: <File name>.ms (Other extenders than ".ms" can be used. A path can also be specified.)

Description: File in which a source program is described. Usually, a file delivered by the Instruction Extender ext33 is input there.

If source files were created that only describe basic instructions and assembler pseudo-instructions, they can be input into the as33 directly.

#### 11.2.2 Output Files

##### Object file

File format: Binary file in srf33 format

File name: <File name>.o (The <File name> is the same as that of the input file.)

Output destination: Current directory

Description: File in which symbol information and debugging information are added to the program code (machine language).

For the srf33 format, refer to Appendix, "srf33 File Structure".

**Assembly list file**

File format:	Text file
File name:	<File name>.lst (The <File name> is the same as that of the input file.)
Output destination:	Current directory
Description:	Assembly source file in which assembled results (address and object code) are added to each line. It is delivered when the startup option (-l) is specified.

For specific examples, refer to Section 11.9 "Assembly List File".

**Error file**

File format:	Text file
File name:	as33.err
Output destination:	Current directory
Description:	File delivered when the startup option (-e) is specified. It records error messages and other information which the as33 delivers via the Standard Output (stdout).

## 11.3 Starting Method

---

### 11.3.1 Startup Format

**General form of command line**

**as33 ^ [<startup option>] ^ [<file name>]**

^ denotes a space.

[ ] indicates the possibility to omit.

<file name>: Specify an assembly source file name including the extension.

**Operations on work bench**

Select options and a source file, then click the [AS33] button.

In the command line, only one source file can be specified at a time.

The wb33 allows multiple files to be selected, in which case the as33 is executed as many times as the number of files selected.

### 11.3.2 Startup Options

The as33 comes provided with the following three types of startup options:

**-g**

Function: Addition of debugging information

Specification on wb33: Check [debug info].

Explanation:

- Creates an output file containing symbolic debugging information.
- Always specify this function when you perform symbolic debugging.
- Even if this option is not selected, the debugging information added in the C Compiler gcc33 or the Preprocessor pp33 for source display will not be cut off.

**-l**

Function: Output of assembly list file

Specification on wb33: Check [list file].

Explanation:

- Outputs an assembly list file.

**-e**

Function: Output of error files

Specification on wb33: None

Explanation:

- Delivers also in a file (as33.err) the contents that are output by the as33 via the Standard Output (stdout), such as error messages.

When entering options in the command line, you need to place one or more spaces before and after the option.

Example: c:\cc33\as33 -g -e -l test.ms

## 11.4 Messages

---

The as33 delivers its messages through the Standard Output (stdout).

If the as33 is started up by using the wb33's [AS33] button, the message is output to "wb33.err". When execution is completed, a message is displayed in the output window (default).

### End message

The as33 outputs only the following end message when it ends normally.

```
Assembly Completed
```

### Usage output

If no file name was specified or an option was not specified correctly, the as33 ends after delivering the following message concerning the usage:

```
Assembler 33 Ver x.xx
Copyright (C) SEIKO EPSON CORP. 199x
Usage:
  as33 [options] <filename>
Options:
  -e : produce log file (as33.err)
  -g : generate debug information in object file
  -l : produce list file
Output:
  object file (.o)
  list file  (.lst)
  log file   (as33.err)
Example:
  as33 -e -g -l sample.ms
```

### When error/warning occurs

If an error is produced, an error message will appear before the end message shows up.

```
Example: test.ms(7): Error: Invalid instruction syntax.
        Assembly Completed
```

In the case of an error, the as33 ends without creating an output file.

If a warning is issued, a warning message will appear before the end message shows up.

```
Example: test.ms(12): Warning: Numeric range.
        Assembly Completed
```

In the case of a warning, the as33 ends after creating an output file.

For details on errors and warnings, refer to Section 11.10 "Error/Warning Messages".

## 11.5 Relocatable Assembling and Absolute Assembling

The as33 supports both the relocatable assembling and the absolute assembling. It even allows to develop software with relocatable modules and absolute modules existing in a mixed fashion.

### 11.5.1 Relocatable Assembling

The relocatable assembling is a method that assembles files without fixing addresses, so that the program will be able to work wherever on the memory the modules may be mapped. Therefore, no absolute address specification will be made in the source files. The C Compiler delivers assembly source files in the relocatable format.

In the relocatable assembling, the assembler performs the processing by applying the relative addresses from top of each section to the individual codes.

This information is output in the object file along with the codes, and all the addresses are determined by the processing of the Linker lk33.

Modules assembled by this method can freely be combined with other modules. This will let you use general-purpose modules thus assembled relocatably as a library in the software development for other models. For relocation by the Linker lk33, refer to Chapter 12, "Linker".

Refer to Chapter 15, "Librarian", for making libraries using the output objects.

The `.abs` pseudo-instruction, `.org` pseudo-instruction, and `.set` pseudo-instruction cannot be employed in the relocatable source files.

### 11.5.2 Absolute Assembling

The absolute assembling is a method that assembles source files by priory specifying the addresses where codes are to be mapped.

However, since the usefulness of the source files for various purposes is lost when they are linked with other modules, do not use this method unless you are creating a simple test program.

The absolute assembling is specified by the `.abs` pseudo-instruction written in the first line of the source file, and the address are set by using the `.org` pseudo-instruction. The `.org` pseudo-instruction and the `.set` pseudo-instruction to define absolute addresses can be employed only in the absolute source.

Example:

```

.abs                ...Specifies the absolute assembling.
.code
.org      0x80000    ...Maps the following codes from address 0x80000.
.word    0x80004
BOOT:
  ext     0x20
  ld.w   %r8,0x0
  ld.w   %sp,%r8   ; set SP
  ld.w   %r8,0x0   ; set global pointer
  :
  .data
  .org   0xC0000    ...Maps the following data from address 0xC0000.
  .word  0x12345678

```

This method causes all the codes in that file to have absolute addresses. It cannot make part of a file relocatable. However, even when a program is created in the form of one absolute source file, it needs to be passed through the Linker lk33 in order to obtain an execution file in which final addresses are defined. (In case of one file, remove the check on the [use .cm file] in the Work Bench wb33 for linking.)

Make sure there is only one instance of the CODE section, DATA section, and BSS section in each absolute file. Basically, create a relocatable assembly source file, then relocate it by using the map function of the lk33.

## 11.6 Scope

Symbols defined in each source file can freely be referred to within that file. Such reference range of symbols is termed scope.

This scope remains the same both in the relocatable and the absolute assembling. Usually, reference can be made only within a defined file. If a symbol that does not exist in that file is referenced, the as33 creates the object file assuming that the symbol is an undefined symbol, leaving the problem to be solved by the lk33.

If your development project requires the use of multiple source files, it is necessary for the scope to be extended to cover other source files. The as33 has the pseudo-instructions (.global, .comm) that can be used for this purpose. Use these instructions to declare that the symbol is a global symbol, so that it can be referenced in other source files. Symbols that can be referenced in only the file where they are defined are called "local symbols". Symbols that are declared to be global are called "global symbols". Local symbols – even when symbols of the same name are specified in two or more different files – are handled as different symbols. Global symbols – if defined as overlapping in multiple files – cause a warning to be generated in the lk33.

Example:

file1: file in which global symbol is defined

```
.global    SYMBOL                ...Global declaration of a symbol which is to be defined in this file.
SYMBOL:
          :
          :
LABEL:    ...Local symbol
          :
          :                      (Can be referred to only in this file)
.comm    VAR1    4
```

file2: file in which a global symbol is referred

```
ext      SYMBOL@rh
ext      SYMBOL@m
call     SYMBOL@r1                ...Symbol externally referred
          :
ext      VAR1@h
ext      VAR1@m
ld.w     %r1, VAR1@l                ...Symbol externally referred
LABEL:   ...Local symbol
          :                      (Treated as a different symbol from LABEL of file1)
```

The as33 regards the symbols SYMBOL and VAR1 in the file2 as those of undefined addresses in the assembling, and includes that information in the object file it delivers. Those addresses are finally determined by the processing of the linker.

## 11.7 Definition of Sections

---

In addition to the programs that control the CPU and peripheral circuits, the source file contains permanently fixed data, such as character generators, which does not require initialization, symbols for the variables stored in RAM and I/O memory control registers. These data and symbols, which bear different attributes, must finally be relocated into the corresponding physical memory locations by the linker, for example, programs must be relocated into the program area in ROM, and fixed data into the data area in ROM. For this reason, the Assembler is designed in such a way that the object code is classified by attribute into each section.

The following three sections exist:

1. **CODE section**      Block for programs
2. **DATA section**      Block for the data to be written into ROM
3. **BSS section**        Block that is mapped into RAM, etc.

To allow to specify these sections in assembly source files, the as33 comes provided with pseudo-instructions. Since the Compiler generates pseudo-instructions, you need not be concerned about sections when programming the C source.

### CODE section

The `.code` pseudo-instruction defines a CODE section. A statement from this instruction to an instruction that defines some other section is assumed to be a program code/data, and is an object for the CODE section. The source file will be regarded as a CODE section by default. Therefore, the part that goes from top of the file, to another section will be processed as CODE section.

### DATA section

The `.data` pseudo-instruction defines a DATA section. A statement from this instruction to an instruction that defines some other section is assumed to be data, and is an object for the DATA section. Therefore, nothing but the symbols to reference addresses and the pseudo-instructions to define data (`.word`, `.half`, `.byte`, `.ascii`, `.space`), those to define alignment (`.align`), and comments can be written in this area.

Although data can be written in the CODE section too, if you want the data blocks to be stored separately from programs after they are linked, data must be written in the DATA section.

### BSS section

The `.comm` pseudo-instruction and the `.lcomm` pseudo-instruction are designed to define the symbol and size of a variables area. When either one of the instructions is described, the symbol will be set in a BSS section. Although the BSS section basically consists in a RAM area, it can as well be used as a data memory area, such as I/O memory. Code definition in this area is meaningless in embedded type microcomputers, such as those of the E0C33 Family. When some other instruction or definition follows the `.comm` or `.lcomm` pseudo-instruction, the section changes to the type defined prior to the BSS section.

Although this section has no actual data as an object, it is required to generate symbol and map information.



**Section management for relocatable source**

In the relocatable assembling, identical sections will be joined together in order to create an object file composed of three integrated sections of CODE, DATA and BSS. Even for a section having no data described or no definition made, the section information will be delivered in the object file.

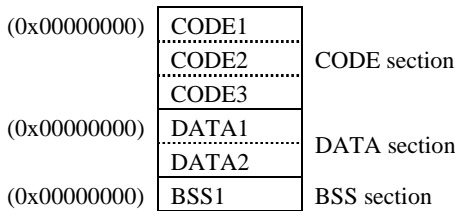
**Sample definition of sections**

```

:
CODE1 (Program)
:
.data
:
DATA1 (Data definition)
:
.comm      RAM0,1
:
BSS1 (RAM area definition)
:
.code
:
CODE2 (Program)
:
.data
:
DATA2 (Data definition)
:
.code
:
CODE3 (Program)
:

```

If you define sections in the manner shown above, the as33 will create an absolute object file composed in the following manner:



## Section management for absolute source

When a source program is divided with the pseudo-instructions mentioned above or the `.org` pseudo-instruction in the absolute assembling, the assembler will create the object file by treating all of the divided parts as independent sections. Moreover, when the types of sections are to be modified with the section defining pseudo-instructions, the start address of each individual section has to be specified using the `.org` pseudo-instruction.

### Sample definition of sections

```
.abs
.code
.org      0x80100
:
CODE1 (Program)      → Section 1
:
.data
.org      0x80f00  ...If this specification is omitted, a DATA section begins from the address
:              following CODE1.
DATA1 (Data definition)  → Section 2
:
.org      0x0
.comm    RAM0,1      → Section 3 (BSS area definition)
```

If you define sections in the manner indicated above, the `as33` will create an absolute object file having three sections.

### Precaution

When there appears in a section a statement which is designed for other section, an error will be issued.

```
Example:  .data
          ld.w    %r1, %r0      ...Error
```

## 11.8 *Assembler Pseudo-Instructions*

---

The assembler pseudo-instructions are not converted to execution codes, but they are designed to control the assembler or to set data.

For discrimination from other instructions, all the assembler pseudo-instructions begin with a period (.). Describe all the instructions in lowercase. Parameters are discriminated between uppercase and lowercase.

### 11.8.1 Absolute Assembling Pseudo-Instruction (.abs)

#### **.abs pseudo-instruction**

##### **Instruction format**

**.abs**

##### **Function**

Specifies the absolute assembling. With this specification done, the as33 performs assembling by handling the file as an absolute file. The top of a file is at address 0x0 by default.

##### **Precautions**

- The .abs pseudo-instruction needs to be specified ahead of other basic instructions and pseudo-instructions. Describe it in the first line of a file.
- The pseudo-instructions (.org, .set) dedicated to the absolute assembling cannot be used without the .abs pseudo-instruction described. If they are used in such situation, an error will result.

## 11.8.2 Section Defining Pseudo-Instructions (.code, .data)

### .code pseudo-instruction

#### Instruction format

**.code**

#### Function

Declares the start of a CODE section. Statements following this instruction are assembled as those to be mapped in the CODE section, until another section is declared.

The CODE section is set by default in the as33. Therefore, the .code pseudo-instruction can be omitted at top of a source file. Always describe it when you change a section to a CODE section. For details on the sections, refer to Section 11.7 "Definition of Sections".

#### Precautions

- A CODE section can be divided among multiple locations of a source file for purposes of definition (describing the .code pseudo-instruction in the respective start positions).  
However, not that multiple CODE section cannot be defined in an absolute source file. The total of sections that can be defined in one source file is maximum 3 in the absolute assembling.
- In the case of an absolute source, be sure to specify an address by the .org pseudo-instruction in the line preceding or following the .code pseudo-instruction.

### .data pseudo-instruction

#### Instruction format

**.data**

#### Function

Declares the start of a DATA section. Statements following this instruction are assembled as those to be mapped in the DATA section, until another section is declared.

For details on the sections, refer to Section 11.7 "Definition of Sections".

#### Precautions

- In a DATA section, nothing other than the data defining pseudo-instructions (.word, .half, .byte, .ascii and .space), .alignment pseudo-instructions (.align), location counter control pseudo-instructions (.org), symbols, and comments can be described. If anything else is described, it will result in an error.
- A DATA section can be divided among multiple locations of a source file for purposes of definition (describing the .data pseudo-instruction in the respective start positions).  
However, not that multiple DATA section cannot be defined in an absolute source file. The total of sections that can be defined in one source file is maximum 3 in the absolute assembling.
- In the case of an absolute source, be sure to specify an address by the .org pseudo-instruction in the line preceding or following the .data pseudo-instruction.

### 11.8.3 Area Securing Pseudo-Instructions (.comm, .lcomm)

#### .comm pseudo-instruction

##### Instruction format

**.comm** <Symbol>[,] <Size>

<Symbol>: Symbols for memory access (address reference)

- The 1st character is limited to a–z, A–Z and \_.
- The 2nd and the subsequent character can use a–z, A–Z, 0–9 and \_.
- Up to 32 characters can be used for symbol names.
- Uppercase and lowercase are discriminated.
- One or more spaces, tabs or a comma (,) are necessary between instruction and symbol.

<Size>: Number of bytes of the area to be secured

- Only decimal, binary and hexadecimal numbers can be described.
- One or more spaces, tabs or a comma (,) are necessary between symbol and size.

Sample description:

```
.comm FOO 4
```

##### Function

Sets an area of the specified size in the BSS section, and creates a symbol indicating its top address with the specified name. By using this symbol, you can describe an instruction to access the memory. The symbols created by the .comm pseudo-instruction become global symbols, which can be referred to externally from other modules.

Only the .comm and .lcomm pseudo-instructions are processed as BSS sections. If some other statement follows the .comm or .lcomm pseudo-instruction, the previous section type applies from that point.

For details on the sections, refer to Section 11.7 "Definition of Sections".

##### Precautions

- A BSS section can be divided among multiple locations of a source file for purposes of definition (describing the .comm pseudo-instruction in the respective start positions).

However, not that multiple BSS section cannot be defined in an absolute source file. The total of sections that can be defined in one source file is maximum 256 in the absolute assembling.

- The address to be assigned the symbol is adjusted to the boundary according to the data size.

Data size: 1	Byte boundary
2	Half word boundary
3 or more	Word boundary

## .lcomm pseudo-instruction

### Instruction format

**.lcomm <Symbol>[,] <Size>**

<Symbol>: Symbols for memory access (address reference)

- The 1st character is limited to a–z, A–Z and \_.
- The 2nd and the subsequent character can use a–z, A–Z, 0–9 and \_.
- Up to 32 characters can be used for symbol names.
- Uppercase and lowercase are discriminated.
- One or more spaces, tabs or a comma (,) are necessary between instruction and symbol.

<Size>: Number of bytes of the area to be secured

- Only decimal, binary and hexadecimal numbers can be described.
- One or more spaces, tabs or a comma (,) are necessary between symbol and size.

Sample description:

```
.lcomm BAR 0x10
```

### Function

Sets an area of the specified size in the BSS section, and creates a symbol indicating its top address with the specified name. By using this symbol, you can describe an instruction to access the memory. The symbols created by the .lcomm pseudo-instruction are local symbols, which cannot be referred to from other modules. Only the .lcomm and .comm pseudo-instructions are processed as BSS sections. If some other statement follows the .lcomm or .comm pseudo-instruction, the previous section type applies from that point. For details on the sections, refer to Section 11.7 "Definition of Sections".

### Precautions

- A BSS section can be divided among multiple locations of a source file for purposes of definition (describing the .lcomm pseudo-instruction in the respective start positions).

However, not that multiple BSS section cannot be defined in an absolute source file. The total of sections that can be defined in one source file is maximum 256 in the absolute assembling.

- The address to be assigned the symbol is adjusted to the boundary according to the data size.

Data size: 1	Byte boundary
2	Half word boundary
3 or more	Word boundary

## 11.8.4 Location Counter Control Pseudo-Instruction (.org)

### .org pseudo-instruction

#### Instruction format

**.org <Address>**

<Address>: Absolute address specification

- Only decimal, binary and hexadecimal numbers can be described.
- The addresses that can be specified are from 0 to 0xffffffff.
- One or more spaces, tabs or a comma (,) are necessary between the instruction and the address.

Sample description:

```
.org 0
.org 0x80000
```

#### Function

Specifies an absolute address in an absolute assembly source file. The as33 performs assembling by assuming that statements following this instruction start from the specified address.

#### Precautions

- The .org pseudo-instruction cannot be used (within a relocatable source), if the .abs pseudo-instruction was not described. If used under such condition, an error will result.
- The .org pseudo-instruction specifies a section start address with the operand value. Note, however, if an odd address is specified, the address may be adjusted to the boundary address according to the subsequent instruction or definition.

## 11.8.5 Symbol Defining Pseudo-Instruction (.set)

### .set pseudo-instruction

#### Instruction format

**.set <Symbol>[,] <Address>**

<Symbol>: Symbols for memory access (address reference)

- The 1st character is limited to a–z, A–Z and \_.
- The 2nd and the subsequent character can use a–z, A–Z, 0–9 and \_.
- Uppercase and lowercase are discriminated.
- One or more spaces, tabs or a comma (,) are necessary between the instruction and the symbol.

<Address>: Absolute address specification

- Only decimal, binary, and hexadecimal numbers can be described.
- The addresses that can grammatically be specified are from 0 to 0xffffffff.
- One or more spaces, tabs or a comma (,) are necessary between the instruction and the address.

Sample description:

```
.set DATA1 0x80000
```

#### Function

Defines an absolute address (32-bit) for a symbol.

#### Precautions

- The .set pseudo-instruction cannot be used (within a relocatable source), if the .abs pseudo-instruction was not described. If used in such situation, an error will result.
- The set symbol becomes a local symbol. To use it as a global symbol, global declaration using the .global pseudo-instruction is necessary.

#### Reference

To define general-use data and character strings, use the #define pseudo-instruction of the preprocessor. (See Section 9.5.2.)



## 11.8.6 Data Defining Pseudo-Instruction (.word, .half, .byte, .ascii, .space)

### .word pseudo-instruction

#### Instruction format

Format 1) **.word** <Data>[[,] <Data> ... [,] <Data>]

Format 2) **.word** <Symbol>

<Data>: Word data (32 bits)

- Only decimal, binary and hexadecimal numbers can be described.
- The data that can be specified are from 0 to 0xffffffff.
- One or more spaces, tabs or a comma (,) are necessary between the instruction and the first data and between one data and another.

<Symbol>: Symbol name that has been defined

Sample description:

```
.word 0x10000000 0x20000000 0x30000000 0x40000000
.word 256
.word FOO
```

#### Function

Format 1) Defines word data. Data can be defined in a CODE section or DATA section.

Format 2) Defines the symbol value as a word data. Data can be defined in a CODE section or DATA section.

#### Precautions

- The .word pseudo-instruction can be used in a CODE section or a DATA section.
- Two or more data can be defined at a time in Format 1. However, one line is limited to 255 characters, including blank characters.
- The defined data is located beginning with a word boundary address unless it is immediately preceded by the .align pseudo-instruction. If the current position is not a word boundary address, 0x00 is set in the interval from that position to the nearest word boundary address.

### .half pseudo-instruction

#### Instruction format

**.half** <Data>[[,] <Data> ... [,] <Data>]

<Data>: Half word data (16 bits)

- Only decimal, binary and hexadecimal numbers can be described.
- The data that can be specified are from 0 to 0xffff.
- One or more spaces, tabs or a comma (,) are necessary between the instruction and the first data and between one data and another.

Sample description:

```
.half 0xfffc 0xfffd 0xfffe 0xffff
.half 256
```

#### Function

Defines half word data. Data can be defined in a CODE section or DATA section.

#### Precautions

- The .half pseudo-instruction can be used in a CODE section or a DATA section.
- Two or more data can be defined at a time. However, one line is limited to 255 characters, including blank characters.
- The defined data is located beginning with a half word boundary address, unless it is immediately preceded by the .align pseudo-instruction. If the current position is an odd address, 0x00 is set at the current position.

## .byte pseudo-instruction

### Instruction format

**.byte** <Data>[[,] <Data> . . . [,] <Data>]

<Data>: Byte data (8 bits)

- Only decimal, binary and hexadecimal numbers can be described.
- The data that can be specified are from 0 to 0xff.
- One or more spaces, tabs or a comma (,) are necessary between the instruction and the first data and between one data and another.

Sample description:

```
.byte 0xfc 0xfd 0xfe 0xff
.byte 255
```

### Function

Defines byte data. Data can be defined in a CODE section or DATA section.

### Precautions

- The .byte pseudo-instruction can be used in a CODE section or a DATA section.
- Two or more data can be defined at a time. However, one line is limited to 255 characters, including blank characters.
- The defined data is located at the current address, unless it is immediately preceded by the .align pseudo-instruction. If byte data is defined at an even address of the CODE section and an instruction is written next, 0x00 is set at an odd address next to the data-defined address to ensure that the instruction will begin with a half word boundary.

## .ascii pseudo-instruction

### Instruction format

**.ascii** "<Character string>"

<Character string>:

ASCII character string

- The character code that can be specified are from 0 to 0xff.
- ASCII characters and an escape sequence that begins with a symbol "\" can be written in a character string. For example, if you want to set double quotations in a character string, write \"; to set a \, write \\.
- One or more spaces, tabs or a comma (,) are necessary between the instruction and the character string.

Sample description:

```
.ascii "abcd \"E\" fg" (=abcd "E" fg)
```

### Function

Defines a character string. Data can be defined in a CODE section or DATA section.

### Precautions

- The .ascii pseudo-instruction can be used in a CODE section or a DATA section.
- The defined data is located beginning with the current address first, unless it is immediately preceded by the .align pseudo-instruction.

**.space pseudo-instruction****Instruction format****.space <Size>**

&lt;Size&gt;: Number of bytes to be filled with 0x0

- Only decimal numbers can be described.
- The size that can be specified are from 1 to 2147483647.
- One or more spaces, tabs or a comma (,) are necessary between the instruction and the size.

Sample description:

`.space 16`**Function**

An area of the specified size is set to 0x0. Such an area can be defined in a CODE or a DATA section.

**Precautions**

- The `.space` pseudo-instruction can be used in a CODE section or a DATA section. If used in a BSS section, an error will result.
- An area of the specified size beginning from the current address is set to 0x0, unless it is immediately preceded by the `.align` pseudo-instruction.

**Regarding the alignment of definition data**Unless it is immediately preceded by the `.align` pseudo-instruction, data is located beginning with a boundary address matched to the data size by a data definition pseudo-instruction.

In the CODE section, instructions are located beginning with a half word boundary. Therefore, it is possible that a blank space occurs in an interval from the last address of defined data to the next instruction or data. The blank addresses are filled with 0x0.

Examples:

<code>jp</code>	<code>SYMBOL</code>	
<code>.byte</code>	<code>0x41</code>	<code>...0x0</code> is set after <code>0x41</code> .
<code>ld.w</code>	<code>%r1, %r7</code>	
<code>.word</code>	<code>0x00</code>	
<code>.byte</code>	<code>0x01</code>	<code>...Three bytes of 0x0</code> are set after <code>0x01</code> .
<code>.word</code>	<code>0x02</code>	

## 11.8.7 Alignment Pseudo-Instruction (.align)

### .align pseudo-instruction

#### Instruction format

**.align** <Boundary specifying value>

<Boundary specifying value>:

Value to specify a boundary

- The values that can be specified are 0 to 8.
- Specify alignment to a  $2^0$  through  $2^8$  byte boundary.
- One or more spaces, tabs or a comma (,) are necessary between the instruction and specification value.

Sample description:

```
.align 2      (aligned to a 4-byte boundary)
```

#### Function

The data that appears immediately after this pseudo-instruction is aligned to a  $2^N$  byte boundary. (N = boundary specification value)

#### Precaution

The .align pseudo-instruction is valid for only the immediately following data definition pseudo-instruction. Therefore, when defining data that requires alignment, you need to use the .align pseudo-instruction for each data definition pseudo-instruction.

## 11.8.8 Global Declaring Pseudo-Instruction (.global)

### .global pseudo-instruction

#### Instruction format

**.global** <Symbol>

<Symbol>: Symbol to be defined in the current file

- One or more spaces, tabs or a comma (,) are necessary between the instruction and the symbol.

Sample description:

```
.global SUB1
```

#### Function

Makes global declaration of a symbol. The declaration made in a file with a symbol defined converts that symbol to a global symbol which can be referred to from other modules.

#### Precautions

- The symbols referenced from other modules must be declared to be global. The symbols defined by the .comm pseudo-instruction are global symbols, so there is no need to use the .global pseudo-instruction to make a declaration.
- The symbols not declared in the current file are processed as global symbols that are declared in some other file.

## 11.8.9 List Control Pseudo-Instructions (.list, .nolist)

### .nolist pseudo-instruction

#### Instruction format

**.nolist**

#### Function

Controls output to the assembly list file.

The .nolist pseudo-instruction stops output to the assembly list file after it is issued.

By default (unless otherwise specified) all statements are output to the assembly list file.

#### Precaution

The as33 delivers assembly list files only when it is started up with the -l option specified. Therefore, this instruction is invalid, if the -l option was not specified.

### .list pseudo-instruction

#### Instruction format

**.list**

#### Function

Controls output to the assembly list file.

The .list pseudo-instruction resumes from there the output which was stopped by the .nolist pseudo-instruction.

#### Precaution

The as33 delivers assembly list files only when it is started up with the -l option specified. Therefore, this instruction is invalid, if the -l option was not specified.

## 11.8.10 Debugging Pseudo-Instructions (.file, .endfile, .loc, .def)

### .file, .endfile, .loc pseudo-instructions

#### Instruction formats

- (1) **.file**        "**<File name>**"
- (2) **.endfile**
- (3) **.loc**         **<Line No.>**

#### Function

These pseudo-instructions are used to add source information to the object file, and are generated by the C Compiler gcc33 and the Preprocessor pp33.

The as33 outputs object files in srf33 format, including debugging information conforming to these instructions. This debugging information is necessary to perform source level debugging by the Debugger db33. Even when the -g option of the as33 is not specified, the debugging information will not be cut off.

The .file pseudo-instruction outputs information indicating the source file's start position. The code following this pseudo-instruction is the content of the file specified by <file name>. It is inserted at the beginning of the source file or an include file at a place where a file is changed.

The .endfile pseudo-instruction outputs information indicating the end position of the file. It is not inserted at the end position of an include file.

The .loc pseudo-instruction outputs information indicating the line numbers of instructions in the source file. It is not added to comments or anywhere other than instruction lines.

### .def pseudo-instruction

#### Instruction format

**.def**   **<Symbol name>**   **[, <Parameter>, . . . <Parameter>], endef**

#### Function

This pseudo-instruction is used to add the C source's symbol information to the object file, and is generated for each symbol by the gcc33. The pp33 does not generate this information.

When the -g option is specified, the as33 outputs the object file in srf33 format, including the debugging information that conforms to this instruction. This debugging information is required when the Debugger db33 symbolic-debug the C source. If the -g option of the as33 is not specified, this debugging information is cut.

<parameter> indicates such information as symbol type and storage class. For details, refer to Section 6.6, "Debugging Information".

### Insertion of debugging information by C Compiler and Preprocessor

When the -g option is specified ([debug info] checked on the work bench) as a start option of the gcc33 and pp33, the gcc33 and pp33 will insert debugging pseudo-instructions in the output file (assembly source file). The pp33 does not insert the .def pseudo-instruction.

Therefore, you do not have to describe these pseudo-instructions in creating source files.

## 11.9 Assembly List File

The assembly list file is an assembly source file that carries assembled results (addresses and object codes) added to the first half of each line. It is delivered only when the startup option (-l) is specified.

If the .nolist pseudo-instruction is written in the source, no contents from that position to the end of the file or the .list pseudo-instruction are output.

Its file format is a text file, and the file name, <File name>.lst. (The <File name> is the same as that of the input source file.)

The format of each line of the assembly list file is as follows:

Address    Code    Source Statement

Example:

```

                .file      "boot.s"
; boot.s 1997. 2. 13
; boot program

;#define SP_INI 0x0800      ; sp is in end of 2KB internal RAM
;#define GP_INI 0x0000      ; global pointer %r8 is 0x0

                .code
00000000 00000000      .word BOOT; BOOT VECTOR
                BOOT:
                .loc      10
00000004 c020          ext      0x20      ;          xld.w      %r8, 0x800
00000006 6c08          ld.w     %r8, 0x0
                .loc      11
00000008 a081          ld.w     %sp, %r8   ; set SP
                .loc      12
0000000a 6c08          ld.w     %r8, 0x0   ; set global pointer
                .loc      13
0000000c c000          ext      main@rh   ;          xcall     main      ; goto main
0000000e c000          ext      main@rm
00000010 1c00          call    main@rl
                .loc      14
00000012 1e00          jp      BOOT      ;          xjp      BOOT      ; infinity loop

                .data
00000000 00000001      .word 1 2
                00000002

00000000 00          .comm tmp 4
                .endfile

```

### Content of address

In the case of an absolute module, an absolute address will be delivered in hexadecimal number.

In the case of a relocatable module, a relative address will be delivered in hexadecimal number from the top of each section.

### Content of code

CODE section: The instruction codes and the defined data are delivered in hexadecimal numbers.

DATA section: The data defined by the data defining pseudo-instruction are delivered.

BSS section: Irrespective of the size of the secured area, 00 is always delivered here.

\* Only the address defined for a symbol (top address of the secured area) is delivered as the address of the BSS section.

### Precaution

The assembler sets the operand (immediate data) of the code that refers to a symbol to 0. The immediate data will be decided by the linker.

## 11.10 Error/Warning Messages

Error and warning messages are displayed/output through the Standard Output (stdout). If you specify the `-e` option, the messages will also be delivered in the "as33.err" file.

If the as33 is started up using the wb33's [AS33] button, the message is output to "wb33.err". When execution is completed, a message is displayed in the output window (default).

### 11.10.1 Errors

The errors produced in the as33 are classified into two groups: system errors and assembler errors.

Table 11.10.1.1 List of system error messages

Error message	Content
<file name>: Error: Out of memory	Cannot secure memory space.
<file name>: Error: File open error	Cannot open the file.
<file name>: Error: File access error	Cannot read/write from/to the opened file.

The assembler errors are produced when the source contains a syntax or description which cannot be processed by the as33. No object file will be delivered. An assembly list file will be delivered, including error messages.

Table 11.10.1.2 List of assembler error messages

Error message	Content
Error: Invalid file name.	The source file has the extension (.o) same as the output file.
Error: Filename length limit exceeded - 255.	The file name exceeds 255 characters.
Error: Directory path length limit exceeded - 255.	The path (directory and file name) exceeds 255 characters.
Error: Line length limit exceeded - 255.	The line exceeds the limit number of characters.
Error: Symbol name length limit exceeded - 32.	The symbol name surpassed the limit number of characters.
Error: Token length limit exceeded - 64.	The numeric data exceeds the limit number of digit.
Error: Multiple statements on the same line.	There are more than two statements described in one line.
Error: Invalid statement syntax.	There is an illegal character in the statement.
Error: Invalid instruction. - "<instruction>"	Non-existing instruction was used.
Error: Invalid register. - "<register>"	The specified register name has an error.
Error: Invalid directive. - "<directive>"	A non-existing pseudo-instruction was used.
Error: Invalid symbol mask. - "<mask>"	The symbol mask has a description error.
Error: Invalid instruction syntax.	The description of the instruction has an error.
Error: Invalid directive syntax.	The description of the pseudo-instruction has an error.
Error: Multiply declared symbol. - "<symbol>"	The same symbol name was declared in multiple locations.
Error: Multiply defined symbol. - "<symbol>"	The same symbol name was defined in multiple locations.
Error: Incorrect section type for statement.	There is an impermissible statement described in the current section.
Error: Memory mapping conflict.	The address is duplicated.
Error: Section count limit exceeded - 256.	The limit number of sections was surpassed.

- \* "Error" is preceded by an input file name and a line number displayed in the form of "<file name>(<line No.>)". If the source file that includes the debugging information is input, the message is followed by "near <file name>(<line No.>)". This consists of the original source file name (\*.c, \*.s) and line number indicated by the debugging information.



## 11.10.2 Warning

When a warning appears, the assembler will keep on processing, and terminates the processing after displaying a warning message, unless any other error is produced. An object file and assembly list file will be delivered.

Table 11.10.2.1 List of warning messages

Warning message	Content
Warning: Numeric range.	The operand value exceed the specifiable range.
Warning: Unknown escape sequence.	An invalid escape sequence is used in the .ascii pseudo-instruction.
Warning: Escape sequence out of range for character.	The character code represented in oct or hex in the .ascii pseudo-instruction exceeds 0xff.
Warning: \x used with no following hex digits.	No value written in hex is found for the hex value specification in the .ascii pseudo-instruction.

\* "Warning" is preceded by an input file name and a line number displayed in the form of "<file name>(<line No.>)." If the source file that includes the debugging information is input, the message is followed by "near <file name>(<line No.>)." This consists of the original source file name (\*.c, \*.s) and line number indicated by the debugging information.

## 11.11 Precautions

---

- (1) The maximum number of object files that can be linked are 4,000 files including library modules. If this limit is exceeded, an error occurs in the linker.
- (2) When performing C source-level or symbolic debugging with the db33, always be sure to specify the -g option before you execute the as33.

Even when the -g option is specified in the gcc33 (the same applies in the case of the pp33), all symbol information is cut unless the -g option is specified in the as33. The source information is not cut. Conversely, if the -g option is specified in the as33 but no -g option is specified in the gcc33 (the same applies in the case of the pp33), symbol information consisting only of symbol names and addresses is added during assembly. Furthermore, unless the -g option is specified in the lk33, all debugging information is cut during linkage.

Make sure that the debugging information (debug pseudo-instructions) in the source file is created only by specifying the -g option of the gcc33 and pp33, and not by any other method. Also be sure not to correct the debugging information that is output. Corrections could cause the as33, lk33, db33 or dis33 to malfunction.

## Chapter 12 Linker

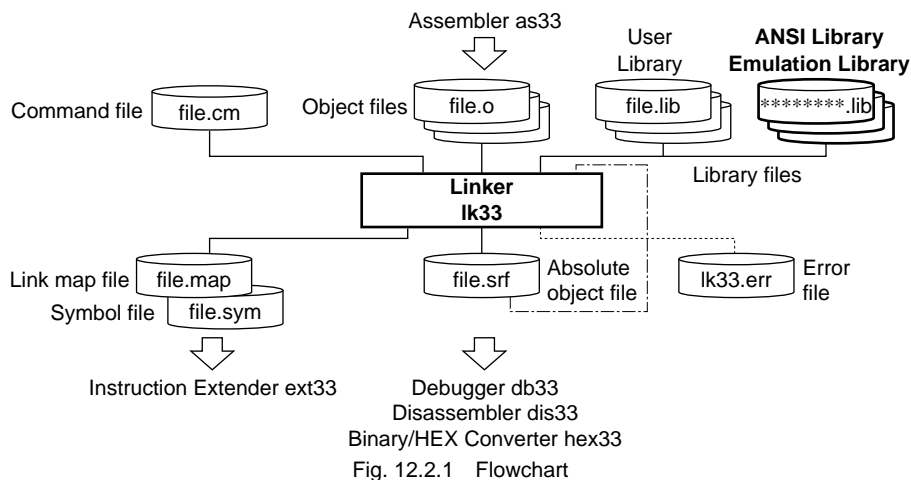
This chapter describes the functions of the Linker lk33.

### 12.1 Functions

The Linker lk33 (hereafter called the "lk33") is a software that generates executable object files. It provides the following functions:

- Links together multiple object modules including libraries to create one executable object file.
- Resolves external reference from one module to another.
- Relocates relative addresses to absolute addresses.
- Delivers debugging information, such as line numbers and symbol information, in the object file created after linking.
- Capable of outputting link map files and symbol files.

### 12.2 Input/Output Files



#### 12.2.1 Input Files

##### Object file

File format: Binary file in srf33 format

File name: <File name>.o

Description: Object file of individual modules created by the Assembler as33.

##### Library file

File format: Binary file in library format

File name: <File name>.lib

Description: ANSI library files, emulation library files and user library files created by the Librarian lib33.

##### Linker command file

File format: Text file

File name: <File name>.cm

Description: File to specify object file names to be input and the start address of each section.

Since the template of a command file is created by [Make edit] of the wb33, correct it with a general-purpose editor before use.

It is input to the lk33 when the -c startup option is specified.

For its contents, refer to Section 12.5 "Linker Commands".

**Absolute object file**

File format:	Binary file in srf33 format
File name:	<File name>.srf
Description:	Executable object file generated by the Linker lk33. This file can be linked only when the <code>-inlink</code> option is specified. Note that ".srf" files cannot be relocated by specifying an address since those absolute addresses have been decided. Furthermore, the absolute object file for the loader that was generated with the <code>-ld</code> option cannot be linked.

**12.2.2 Output Files****Absolute object file**

File format:	Binary file in srf33 format
File name:	<File name>.srf
Output destination:	Current directory
Description:	Object file in executable format that can be input in the Debugger db33. All the modules comprising one program are linked together in the file, and the absolute addresses that all the codes will be mapped are determined. It also contains the necessary debugging information in srf33 format. The file name is decided in the following manner: In case of one single module: The same name as the input object file. When link command files are input: The same name as that of the file to be linked first, or a name specified by the command.

For the contents of the output object file, refer to Appendix, "srf33 File Structure".

**Link map file**

File format:	Text file
File name:	<File name>.map
Output destination:	Current directory
Description:	Mapping information file showing from which address of a section each input file was mapped. The file is delivered when the <code>-m</code> startup option is specified. This file is used to optimize the codes by the Instruction Extender ext33.

**Symbol file**

File format:	Text file
File name:	<File name>.sym
Output destination:	Current directory
Description:	Symbols defined in all the modules and their address information are delivered in this file. The file is delivered when the <code>-s</code> startup option is specified. This file is used to optimize the codes by the Instruction Extender ext33.

**Error file**

File format:	Text file
File name:	lk33.err
Output destination:	Current directory
Description:	File delivered when the startup option ( <code>-e</code> ) is specified. It records the information which the lk33 outputs to the Standard Output (stdout), such as error messages.

## 12.3 Starting Method

---

### 12.3.1 Startup Format

#### General form of command line

Format 1) Linking of multi modules

**lk33 ^ [<Startup option>] ^ -c ^ <Linker command file name>**

Format 2) Linking of single module

**lk33 ^ [<Startup option>] ^ <Input object file name>**

^ denotes a space.

[ ] indicates the possibility to omit.

The extension should be included in the file name.

Format 2 can also specify two or more object files. Note, however, that the limitation of DOS command lines applies to the number of characters in Format 2.

#### Operations on work bench

Format 1) Linking of multi modules

Select options and a command file, then click the [LK33] button. The [use .cm file] button must be selected.

Format 2) Linking of single module

Remove the check on the [use .cm file] button and select object files (.o), then click the [LK33] button.

### 12.3.2 Startup Options

The lk33 comes provided with the following five types of startup options:

#### -g

Function: Addition of debugging information

Specification on wb33: Check [debug info].

Explanation:

- Creates an output file containing debugging information.
- Always specify this function when you perform source level debugging. Failure to specify it will cut off the debugging information which was added by the C Compiler gcc33, Preprocessor pp33 and Assembler as33.

#### -c <Linker command file name>

Function: Specification of linker command file

Specification on wb33: Check [use .cm file].

Explanation:

- Inputs a linker command file.

#### -s

Function: Output of symbol file

Specification on wb33: Check [symbol,map file].

Explanation:

- Outputs a symbol file.

#### -m

Function: Output of link map file

Specification on wb33: Check [symbol,map file].

Explanation:

- Outputs a link map file.

#### -e

Function: Output of error file

Specification on wb33: None

Explanation:

- Delivers also in a file (lk33.err) the contents to be output by the lk33 through the Standard Output (stdout), such as error messages.

The options can be described in the command file except for the -c option.

When inputting options in the command line, one or more spaces are necessary before and after the option.

Example: `c:\cc33\lk33 -g -e -s -m -c test.cm`

Note: When specifying a command file with the -c option, write other options at positions preceding the -c option or within the command file. Any options entered after the -c option are ignored.

## 12.4 Messages

---

The lk33 delivers its messages through the Standard Output (stdout).

If the lk33 is started up by using the wb33's [LK33] button, the message is output to "wb33.err". When execution is completed, a message is displayed in the output window (default).

### End message

The lk33 outputs only the following end message when it ends normally.

```
Link Completed
```

### Usage output

If no file name was specified or an option was not specified correctly, the lk33 ends after delivering the following message concerning the usage:

```
Linker 33 Ver x.xx
Copyright (C) SEIKO EPSON CORP. 199x
Usage:
  lk33 [options] <filenames>
Options:
  -e : produce log file (lk33.err)
  -g : generate debug information in output file
  -s : generate symbol information (.sym)
  -m : generate map information (.map)
  -c CommandFile : execute lk33 commands from CommandFile (.cm)
Output:
  SRF33 object file (.srf)
Example:
  lk33 -e -g -s -m -c test.cm
```

### When error/warning occurs

If an error is produced, an error message will appear before the end message shows up.

Example: Error: Too many global bss symbol.

```
Link Completed
```

In the case of an error, the lk33 ends without creating an output file.

If a warning is issued, a warning message will appear before the end message shows up.

Example: test.o: Warning: Unresolved external symbol 'SUB1'.

```
Link Completed
```

In the case of a warning, the lk33 ends after creating an output file, but that operation is not guaranteed.

For details on errors and warnings, refer to Section 12.13 "Error/Warning Messages".

## 12.5 Linker Commands

Besides the startup options, the lk33 offers linker commands that can be specified in the linker command file. To link multiple modules, it is necessary to create a linker command file and input it in the lk33 by the -c option.

### 12.5.1 Linker Command File

To simplify the keystroke in the command line at the time of startup, you can execute the link processing through the lk33 by inputting a linker command file that holds the necessary specifications described.

#### Sample linker command file

```

;Map set
-code 0x0080000          ; CODE section start address
-data 0x0081000         ; DATA section start address
-bss 0x0000000         ; BSS section start address

-code 0x0080100 {test2.o} ; Fixing of CODE section start position of test2.o

-defsym BOOT = 0x0080000 ; Setting of global symbol

;Library path
-l C:\CC33\lib          ; Library search path

;Executable file
-o test.srf             ; Output file name

;Object files
test1.o                ; Input files
test2.o

;Library files
io.lib                 ; Library files
lib.lib
math.lib
string.lib
ctype.lib
fp.lib
idiv.lib

```

Create the linker command file in line with the following rules:

#### File format

The linker command file is a general text format as shown above.

Create it on a general-purpose editor. Or a template is created by the Make file editor of the wb33, so use that file after correcting it.

The extension of the file name should be described as ".cm".

#### Command description

All commands should begin with a hyphen (-). Each individual command needs to be delineated with more than one space, tab, or line feed. For better visibility, it is recommended to describe each command in a separate line.

Up to 2048 characters can be described in one line.

- Notes:
- Describe all commands in lowercase characters. Uppercase characters will not be accepted.
  - A numeric value to specify an address should be described in the hexadecimal format (0x####). Decimal and binary notations will not be accepted.
  - When a command which is only permitted in single setting is specified in a duplicated manner, the last entered command will be effective.  
Example: `-code 0x0080000`  
`-code 0x0080100` ...`-code 0x0080100` is effective.  
If the command is duplicate-sensitive (e.g., `-code{ }`, `-ucode{ }`), an error results.
  - The following characters can only be accepted for symbol names, U section names and file names:  
1st character: a–z, A–Z  
2nd and the subsequent character: a–z, A–Z, 0–9, \_  
"." and "\" can aslo be used for file (path) names.

**Specification of input and library files**

Make sure the object or library file names to be input are written at the end of the link command file. Also, be sure to write the library file after the object file. File location by linkage is performed in such a way that unless otherwise specified, the files are located in the order they are written.

Write each file name including the extension (.o, .lib, .srf).

Specifying only a library file without writing an object file name is not permitted.

**Comment**

A comment can be described in the linker command file.

As in the source file, the character string from a semicolon (;) to the end of the line is regarded as a comment.

**Blank line**

A blank line carrying only blank characters and a line feed will be ignored. It need not be converted to a comment.

## 12.5.2 Linker Command List

The following 21 types of linker commands are provided for the lk33 (including the startup options that can be specified in a command line):

Table 12.5.2.1 Linker command list

Command	Function
-c	Specifies a command file. *1, *2
-m	Outputs a link map file. *1
-s	Outputs a symbol file. *1
-g	Outputs debugging information. *1
-e	Outputs an error file. *1
-w	Sets the warning level.
-l	Specifies a library search path.
-o	Sets an output file name.
-defsym	Sets a global symbol.
-d	Deletes the duplicated global BSS area.
-code	Sets a relocatable CODE section start address.
-data	Sets a relocatable DATA section start address.
-bss	Sets a relocatable BSS section start address.
-ucode	Sets a virtual/shared CODE section start address.
-udata	Sets a virtual/shared DATA section start address.
-ubss	Sets a shared BSS section start address.
-objsym	Creates section symbols.
-section	Specifies an output section.
-ld	Outputs an srf33 file for the loader.
-inlink	Enables linking .srf files.

\*1: Startup options

\*2: Cannot be used in the command file.

The following explains each linker command. (For details on startup options, refer to Section 12.3.2.) Actual usage examples and link results are described in the next section.

### -w command

Format: **-w**

Sample description: **-w**

Explanation:

- If the -w command is specified, no warning is output for duplicate global labels in the BSS section.

Default: If this command is not specified, all warning messages are output.

### -l command

Format: **-l <Library search path>**

Sample description: **-l c:\cc33\lib**

Explanation:

- This command specifies the directory where libraries can be searched.
- At least one space or tab is required between -l and <Library search path>.
- Up to four library search paths can be specified. To specify multiple directories, specify the -l option for each directory.

Default: Unless this command is specified, only the current directory is searched. A path name can be included in each written library file name.

### -o command

Format: **-o <Output file name>.srf**

Sample description: **-o test.srf**

Explanation:

- This command specifies an output file name.
- At least one space or tab is required between -o and <Output file name>.

Default: Unless this command is specified, the linker uses the first file name that appears in the input object files written in the command file to generate the output file.



**-defsym command**

Format: **-defsym <Symbol name> = <Value>**

Sample description: **-defsym BOOT = 0x0080000**

Explanation:

- Use this command to define the value of a global symbol.
- At least one space or tab is required between **-defsym** and **<Symbol name>**.
- The maximum number of symbols that can be defined by this command is 256.

Default: Unless this command is specified, no global symbol is set.

**-d command**

Format: **-d**

Sample description: **-d**

Explanation:

- If multiple areas bearing the same global symbol name are set in the BSS section, this command deletes all but one area.
- The area that remains valid is the largest one which appears first among the input object file names specified.

Default: Unless this command is specified, the areas with invalid symbols are not deleted.

Note: A warning is issued if global symbols of the same name are defined.  
This command is valid in only the BSS section; it is ignored in all other sections.

**-code command**

Format 1: **-code <Address>**

Format 2: **-code <Section name>**

Sample description: **-code 0x0c00000**  
**-code EXTERNAL\_ROM**

Explanation:

- This command sets the start address of an area where a relocatable CODE section is located. The CODE sections in the files specified in format 3 or 4, and those in absolute object files are unaffected.
- **<Section name>** can only be specified when the section name and start address are set by the **-section** command.
- At least one space or tab is required between **-code** and **<Address/Section name>**.
- Specify a 4-byte boundary address for **<Address>**. If something else is specified, a warning is issued, in which case the two low-order bits of the specified address are ignored.

Default: Unless this command is specified, the CODE section begins from 0x0080000.

Format 3: **-code <Address> {<File name> ... <File name> }**

Format 4: **-code <Section name> {<File name> ... <File name> }**

Sample description: **-code 0x0080100 {test1.o test2.o}**  
**-code BLOCK2 {test1.o, test2.o}**

Explanation:

- This command locates the CODE sections of the relocatable object files specified in { } sequentially in the order the files are specified beginning with a specified address.
- When specifying multiple files, insert at least one space or tab between each **<File name>**.
- Others are the same as in format 1 or 2.

**-data command**

Format 1: **-data <Address>**

Format 2: **-data <Section name>**

Sample description: **-data 0x0081000**  
**-data DATA1**

Explanation:

- This command sets the start address of an area where a relocatable DATA section is located. The DATA sections in the files specified in format 3 or 4 and those in absolute object files are unaffected.
- **<Section name>** can only be specified when the section name and start address are set by the **-section** command.
- At least one space or tab is required between **-data** and **<Address/Section name>**.

- Specify a 4-byte boundary address for <Address>. If something else is specified, a warning is issued, in which case the two low-order bits of the specified address are ignored.
- Default: Unless this command is specified, the DATA section is located after the CODE section that is located at the highest address.
- Format 3: **-data <Address> {<File name> ... <File name> }**
- Format 4: **-data <Section name> {<File name> ... <File name> }**
- Sample description: -data 0x0080100 {test1.o test2.o}  
-data DATA1 {test1.o test2.o}
- Explanation:
  - This command locates the DATA sections of the relocatable object files specified in { } sequentially in the order the files are specified beginning with a specified address.
  - When specifying multiple files, insert at least one space or tab between each <File name>.
  - Others are the same as in format 1 or 2.

**-bss command**

- Format 1: **-bss <Address>**
- Format 2: **-bss <Section name>**
- Sample description: -bss 0x0000100  
-bss VARIABLES
- Explanation:
  - This command sets the start address of an area where a relocatable BSS section is located. The BSS sections in the files specified in format 3 or 4 and those in absolute object files are unaffected.
  - <Section name> can only be specified when the section name and start address are set by the -section command.
  - At least one space or tab is required between -bss and <Address/Section name>.
  - Specify a 4-byte boundary address for <Address>. If something else is specified, a warning is issued, in which case the two low-order bits of the specified address are ignored.
- Default: Unless this command is specified, is the BSS section begins from 0x0000000.
- Format 3: **-bss <Address> {<File name> ... <File name> }**
- Format 4: **-bss <Section name> {<File name> ... <File name> }**
- Sample description: -bss 0x0000100 {test1.o test2.o}
- Explanation:
  - This command locates the BSS sections of the relocatable object files specified in { } sequentially in the order the files are specified beginning with the specified address.
  - When specifying multiple files, insert at least one space or tab between each <File name>.
  - Others are the same as in format 1 or 2.

**-ucode command**

- Format 1: **-ucode <Address>**
- Format 2: **-ucode <Section name>**
- Sample description: -ucode 0x1000  
-ucode CACHE
- Explanation:
  - This command sets the start address of a virtual CODE section. The CODE sections of the relocatable object file for which format 1 and 2 settings of the -code command are applied are linked by resolving the symbol addresses in such a way that they can be located and executed beginning with the specified address. The row data positions are left intact as specified by -code, and are not modified. The CODE sections of absolute object files are unaffected. Specify this command when executing a program written in ROM after transferring it to RAM.
  - <Section name> can only be specified when the section name is set by the -section command.
  - At least one space or tab is required between -ucode and <Address/Section name>.
  - Specify a 4-byte boundary address for <Address>. If something else is specified, a warning is issued, in which case the two low-order bits of the specified address is ignored.
- Default: Unless this command is specified, no virtual CODE section is set.

Format 3: **-ucode <Address> {<File name> ... <File name> }**

Format 4: **-ucode <Section name> {<File name> ... <File name> }**

Sample description: `-ucode 0x1000 {test1.o test2.o}`  
`-ucode CACHE {test1.o test2.o}`

- Explanation:
- This command sets the start address of a shared CODE section. The CODE sections of all relocatable object files specified in { } are linked by resolving the symbol addresses in such a way that they can be located and executed in the same area (the shared area that begins from a specified address or the specified start address of the shared section). This command should prove effective when one RAM area is shared by multiple specified object codes, and execution is repeated by sending the code to the RAM area via a time-multiplexed transfer.
  - At least one space or tab is required between each <File name>.
  - Others are the same as in format 1 or 2.

#### **-udata command**

Format 1: **-udata <Address>**

Format 2: **-udata <Section name>**

Sample description: `-udata 0x1000`  
`-udata INITDATA`

- Explanation:
- This command sets the start address of a virtual DATA section. The DATA sections of the relocatable object file for which the format 1 and 2 settings of the -data command are applied are linked by resolving the symbol addresses in such a way that they can be located and executed beginning with the specified address. The row data positions are left intact as specified by -data and not modified. The DATA sections of absolute object files are unaffected. Specify this command when using data written in ROM (e.g., variables requiring initialization) after transferring it to RAM.
  - <Section name> can only be specified when a section name is set by the -section command.
  - At least one space or tab is required between -udata and <Address/Section name>.
  - Specify a 4-byte boundary address for <Address>. If something else is specified, a warning is issued, in which case the two low-order bits of the specified address are ignored.

Default: Unless this command is specified, no virtual DATA section is set.

Format 3: **-udata <Address> {<File name> ... <File name> }**

Format 4: **-udata <Section name> {<File name> ... <File name> }**

Sample description: `-udata 0x1000 {test1.o test2.o}`  
`-udata INITDATA {test1.o test2.o}`

- Explanation:
- This command sets the start address of a shared DATA section. The DATA sections of all relocatable object files specified in { } are linked by resolving the symbol addresses in such a way that they can be located and executed in the same area (the shared area that begins from the specified address or the specified start address of the shared section). This command should prove effective in cases in which one RAM area is shared by multiple specified object data, and execution is repeated by sending data to the RAM area via a time-multiplexed transfer.
  - When specifying multiple files, insert at least one space or tab between each <File name>.
  - Others are the same as in format 1 or 2.

#### **-ubss command**

Format 1: **-ubss <Address> {<File name> ... <File name> }**

Format 2: **-ubss <Section name> {<File name> ... <File name> }**

Sample description: `-ubss 0x1000 {test1.o test2.o}`  
`-ubss TMP {test1.o test2.o}`

- Explanation:
- This command sets the start address of a shared BSS section. The BSS sections of all relocatable object files specified in { } are linked by resolving the symbol addresses in such a way that they can be located and used in the same area (the shared area that begins from the specified address or the specified start address of the shared section). The BSS sections of absolute object files are unaffected. This command should prove effective in cases in which one RAM area is shared by multiple specified object data, and the data is used separately via a time-multiplexed transfer.
  - <Section name> can only be specified when the section name is set by the -section command.
  - At least one space or tab is required between -ubss and <Address/Section name>.
  - At least one space or tab is required between each <File name>.
  - Specify a 4-byte boundary address for <Address>. If something else is specified, a warning is issued, in which case the two low-order bits of the specified address are ignored.

Default: Unless this command is specified, no shared BSS section is set.

### **-objsym command**

Format: **-objsym**

Sample description: -objsym

- Explanation:
- This command creates three types of symbols indicating the start address, end address, and size of each section located in the input file (refer to Section 12.8). These symbols can be used in the source file when creating a routine for transfer to virtual or shared sections.

Default: Unless this command is specified, no symbol is created.

### **-section command**

Format 1: **-section <Section name>**

Format 2: **-section <Section name> = <Address>**

Sample description: -section TMP  
-section CACHE = 0x1000

- Explanation:
- This command defines a section name.
  - Format 1 is used for virtual and shared sections, where an address following the last address of the default BSS area (specified by the -bss command) is the start address of the virtual or shared section. Format 2 defines a section name and its start address, which can be used to specify any section.
  - At least one space or tab is required between -section and <Section name>.
  - The size of an area is determined by setting a section specifying command, and three types of symbols are created indicating the start address, the end address after being located (at maximum use), and the size (refer to Section 12.8). These symbols can be used in the source file when creating a routine for transfer to virtual or shared sections.

### **-ld**

Format: **-ld**

Sample description: -ld

- Explanation:
- Creates an srf33 file for the loader ld33 instead of a standard absolute object file. Refer to readme.txt (English) or readmeja.txt (Japanese) located in the "utility\ld33\" directory for the loader ld33.

Default: Unless this command is specified, a standard absolute object file is created.

Note: When creating this file, the -ucode, -udata and -ubss commands for specifying U sections cannot be used. Furthermore, 2-pass make optimization cannot be performed.

### **-inlink**

Format: **-inlink**

Sample description: -inlink

- Explanation:
- Enables absolute object files (.srf) as link files. When this command is specified, .srf files can be linked similar to the absolute object files generated by the as33. However, absolute object files for the loader that are created with the -ld command cannot be linked.

Default: Unless this command is specified, ".srf" files cannot be linked

## 12.6 Locating Sections

### Standard location of relocatable file

Each relocatable object file has one CODE, one DATA, and one BSS section.

When multiple relocatable object files are linked, sections of the same type in all files except a specified file are combined into one section. Entries within each section are arranged in the order in which the input files are written in the command file (or command line).

The start address of each section (absolute address after linkage) can be specified by a linker command. If this address is not specified, it is determined in the following manner:

**CODE section:** The CODE section is located beginning with address 0x0080000 (area 3, start address of internal ROM).

**DATA section:** After all CODE sections in the input files are located, the DATA section is located immediately following the CODE section that is located at the highest address.

**BSS section:** The BSS section is located beginning with address 0x0000000 (area 0, start address of internal RAM).

For example, if two relocatable object files, sample1.o and sample2.o, are linked without specifying a section address, each section in these files is located as shown below. Each section in each object file starts at a 4-byte boundary address.

Example: lk33 sample1.o sample2.o

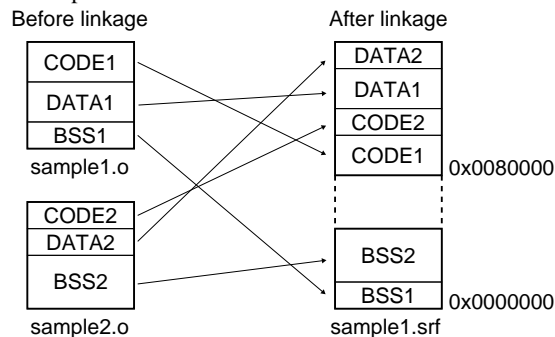


Fig. 12.6.1 Example of standard linkage

When an absolute object file is generated by the lk33, the CODE, DATA, and BSS sections are output to the file in that order. For the contents of the object files thus output, refer to Appendix, "srf33 File Structure".

### To specify the start address of a relocatable section...

If you want sections to be located beginning with an address that is not the default address shown above, use the `-code`, `-data`, or `-bss` commands to specify the start address of each type of section.

The command formats are shown below:

- code** <Address> Sets a CODE section.
- data** <Address> Sets a DATA section.
- bss** <Address> Sets a BSS section.

For example, if you want the start addresses of the CODE and the BSS sections in Figure 12.6.1 to be changed to address 0x0c00000 and address 0x0010000, respectively, input a command file like the one shown below before linking the object files.

Example: Command file

```
-code    0x0c00000
-bss     0x0010000
sample1.o
sample2.o
```

## To locate sections in a specific file beginning with a specific address...

You can specify a relocatable object file by using the `-code { }`, `-data { }`, and `-bss { }` commands, so that the sections in only that file will be located beginning with the specified address. Multiple files can be specified, in which case the sections are located in the order that they are specified beginning with the start address.

The command formats are shown below:

```
-code <Address> {<File name> ...} Sets a CODE section.
-data <Address> {<File name> ...} Sets a DATA section.
-bss <Address> {<File name> ...} Sets a BSS section.
```

Example: Command file

```
-code    0x0080000
-bss     0x0000000
-code    0x0c00000 {sample2.o sample3.o}
sample1.o
sample2.o
sample3.o
```

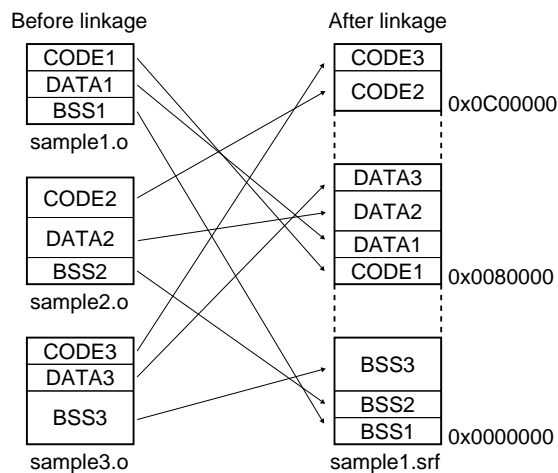


Fig. 12.6.2 Example of how sections in a specific file are located

In all relocatable object files other than those specified with the `-code { }`, `-data { }`, and `-bss { }` commands, sections are located at default addresses or addresses set by `-code <Address/Section name>`, `-data <Address/Section name>`, and `-bss <Address/Section name>`.

The `-code { }`, `-data { }`, and `-bss { }` commands cannot be used to specify a library file.

## Specifying the start address with a section name

The `-section` command can be used to define a section name and its start address.

The command format is shown below:

```
-section <Section name> = <Address>
```

The `<Section name>` defined by the `-section` command can be used in place of `<Address>` specified by the `-code`, `-data`, and `-bss` commands or the `-code { }`, `-data { }`, and `-bss { }` commands described above. However, these addresses must first be defined by the `-section` command before they can be used.

Example: `-section CODE1 = 0x0080100`

```
-section DATA1 = 0x0081000
```

```
-code CODE1 {test1.o} ; Locates the CODE section of the test1.o in section CODE1.
```

```
-data DATA1 {test1.o} ; Locates the DATA section of the test1.o in section DATA1.
```

The start address specified for each section by the `-section` command is the start address of each section. Specification of `<Address>` in the `-section` command cannot be omitted.

When a section name is defined, the lk33 generates three types of section symbols indicating its start address, end address, and size (refer to Section 12.8). These symbols can be used at the source as global symbols.

### Locating absolute files

An absolute object file has the absolute address of each of its sections already determined before linkage, therefore the sections in this file are located at those addresses preferentially over other sections. (The lk33 locates the absolute object file before relocatable object files.) The absolute object files are unaffected by commands that specify addresses, such as the `-code`, `-data`, or `-bss` command.

Example: Command file

```
-code    0x0081000
sample1.o    ; Absolute object file
sample2.o    ; Relocatable object file
```

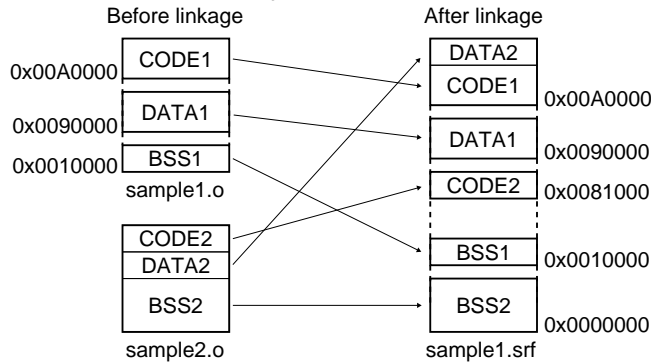


Fig. 12.6.3 Example of how an absolute object file is located

### Priority of address specification

Address specifications are resolved according to the following order of priorities:

1. Settings in absolute object file
2. Settings by `-code { }`, `-data { }`, and `-bss { }` commands
3. Settings by `-code`, `-data`, and `-bss` commands
4. Default settings

Sections in absolute object files are located at the addresses specified in the source preferentially over all other sections.

Sections in relocatable object files are located in such a way that those specified by the `-code { }`, `-data { }`, and `-bss { }` commands are located before other sections, and the sections in remaining other files are located beginning with the addresses set by the `-code`, `-data`, and `-bss` commands or the default addresses.

Sections in two or more files having the same priority are located in the order the file names are written in the command file (or command line) in the upward address direction.

### Section alignment

Sections are always located at 4-byte boundaries irrespective of their types. If the specified start address of a section does not reside on a 4-byte boundary, a warning is issued, in which case the two low-order bits of the address are treated as 0.

## 12.7 Virtual and Shared (U) Sections

Virtual and shared sections (U sections) do not have real data created by the linker, therefore real data is copied from some other area into this area before it is actually accessed or executed. The U sections are used for this purpose. The U sections are normally located in RAM.

If symbol information is created at addresses where real data is stored such as when using variables in the C language which have the initial values or copying a program from external memory into RAM for high-speed operation, the program or data cannot be operated or used in RAM. In such a case, set an area that is actually executed as a U section, then logically locate the program or data in that area before linking the modules. The symbol information of the modules located in the U section will be generated as the internal addresses of the U section.

**Note:** The expression "locate in a U section" actually means a logical location for obtaining the execution address from the U section. Although the `-ucode`, `-udata`, or other similar commands are used for logical location into a U section, the addresses at which real data is stored are determined by the `-code` or `-data` commands (commands that do not specify a file name) or by default settings.

The program or data located in a U section must be copied in advance to an area where the program or data is actually used. Because run-time relocation is required, no absolute object file can be located in a U section.

There are two types of U sections: a virtual section and a shared section.

### Virtual section

Equivalents of the `-code` and `-data` commands that locate the real data of the CODE and DATA sections are the commands `-ucode` and `-udata` which are used for the U sections.

These commands can be used in the following formats to specify the start addresses of the virtual CODE and virtual DATA sections.

**-ucode** <Address> Sets a virtual CODE section.

**-udata** <Address> Sets a virtual DATA section.

When one of these commands is specified, symbols are interpreted assuming that the default CODE section (specified by `-code <Address>`) or DATA section (specified by `-data <Address>`) is executed in the virtual section that starts from a specified address.

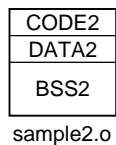
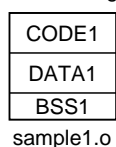
Example: Command file

```
-udata 0x01000
```

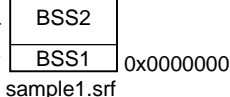
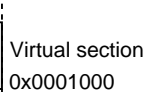
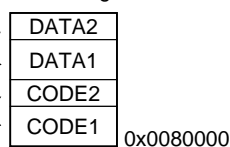
```
sample1.o
```

```
sample2.o
```

Before linkage



After linkage



At execution

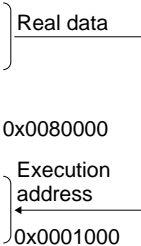
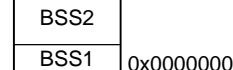
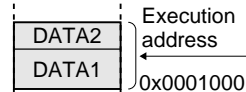
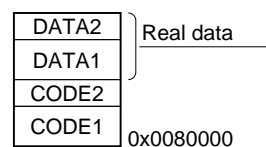


Fig. 12.7.1 Example of how data is located in a virtual section



### Shared section

By specifying a relocatable object file with the `-ucode { }`, `-udata { }`, or `-ubss { }` commands, you can locate only the sections in that file, parallel to a shared section. Multiple files are located beginning with the same start address.

- ucode** <Address> {<File name> ...} Sets a shared CODE section.
- udata** <Address> {<File name> ...} Sets a shared DATA section.
- ubss** <Address> {<File name> ...} Sets a shared BSS section.

Example: Command file

```
-ucode 0x0a000 {sample1.o sample2.o}
sample1.o
sample2.o
```

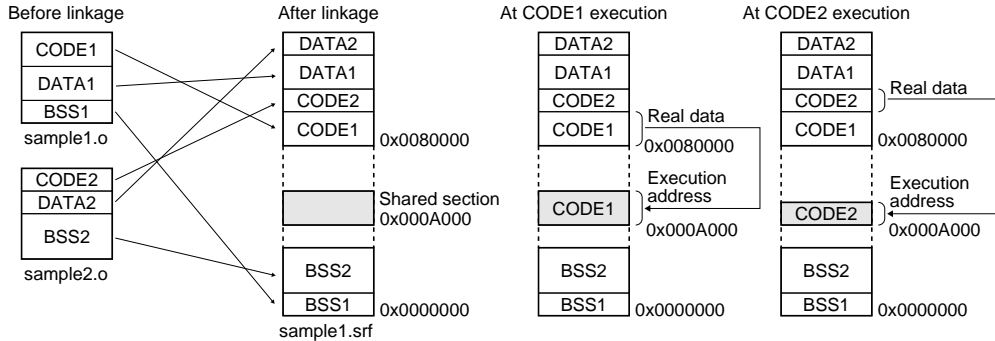


Fig. 12.7.2 Example of how data is located in a shared section

### Named U section

A U section can have its name defined by the `-section` command. The command format is shown below.

- Format 1) `-section <Section name>`
- Format 2) `-section <Section name> = <Address>`

This command creates one U section that is assigned a specified name. Multiple U sections can be defined in this way. For this purpose, write the `-section` command as many times as necessary.

The names defined here can be used in place of `<Address>` of the `-ucode`, `-udata`, and `-ubss` commands or the `-ucode { }`, `-udata { }`, and `-ubss { }` commands described above. However, these addresses must first be defined by the `-section` command before they can be used.

Example:

```
-section CACHE = 0x300
-section INITDATA = 0x100
-ucode CACHE {test1.o} ; Locates the CODE section of the test1.o to U section CACHE.
-udata INITDATA {test1.o} ; Locates the DATA section of the test1.o to U section INITDATA.
```

The start address of a defined U section is set as follows:

- Format 1) Address immediately following the end of the default BSS section. This address is determined after locating all BSS sections in the input object files.
- Format 2) Specified address

When a section name is defined, the `lk33` generates three types of section symbols indicating its start address, end address, and size (see the next section). Since these symbols can be used in the source as global symbols, you can use them to copy real data.

## Location and size of U section

A U section starts from the address specified by a command. The size of a virtual section equals the total size of the located sections. The size of a shared section is that of the largest section of multiple shared sections.

When defining a U section, be careful with its address because the addresses overlapping between U sections or between a U section and another normal section are accepted.

## Section alignment

U sections are always located at 4-byte boundaries irrespective of their types. If the specified start address of a U section does not reside on a 4-byte boundary, a warning is generated, in which case the two low-order bits of the address are treated as 0.

## How to use U sections

To allocate real program codes and variables successively into U sections, specification in the order as below is effective unless the absolute addresses are otherwise specified.

↑ RAM: Lower address

- (1) Variables without an initial value (BSS)
- (2) Variables with an initial value (DATA)
- (3) Program codes 1 (replication of a CODE area in the ROM)
- (4) Program codes 2 (replication of a CODE area in the ROM)

:

↓ RAM: Higher address

To allocate as above, specify with the following command in the command file:

### Sample command file

```
-section INITDATA
-section CACHE1
-section CACHE2

-udata INITDATA
-ucode CACHE1 {main.o}
-ucode CACHE2 {sub.o}

main.o
sub.o
```

The link map file after linking shows that the blocks (1) to (4) have been located in the order from lower address to higher address.

### Sample link map file

```
Code Section mapping
Address  Vaddress  Size      File          ID  Attr
00080000 0000001c  00000040  main.o ←(3)   0  REL
00080040 0000005c  00000020  sub.o ←(4)    0  REL

Data Section mapping
Address  Vaddress  Size      File          ID  Attr
00080060 00000014  00000008  main.o ←(2)   1  REL
00080068 0000001c  00000000  sub.o ←(2)    1  REL

Bss Section mapping
Address  Vaddress  Size      File          ID  Attr
00000000 -----  00000004  main.o ←(1)   2  REL
00000004 -----  00000010  sub.o ←(1)    2  REL
```

## 12.8 Section Symbols

The lk33 can generate section symbols (global symbols) that indicate the addresses and sizes of located sections. These symbols can be used in the source.

### Default DATA section symbol

Unless the `-data` command is specified, the default data section (located following the CODE section) has the following section symbols automatically generated:

<code>__START_DEFAULT.DATA</code>	Defines the start address of the default DATA section.
<code>__END_DEFAULT.DATA</code>	Defines the end address of a default DATA section.
<code>__SIZEOF_DEFAULT.DATA</code>	Defines the size (in bytes) of the default DATA section.

### Creating section symbols using the `-objsym` command

When the `-objsym` command is written in a command file, the lk33 generates the following symbols for each section in each input file to indicate section information after relocation:

<code>__START_&lt;File name&gt;_&lt;Section&gt;</code>	Defines the start address of a relocated section.
<code>__END_&lt;File name&gt;_&lt;Section&gt;</code>	Defines an address next to the end address of a relocated section.
<code>__SIZEOF_&lt;File name&gt;_&lt;Section&gt;</code>	Defines the size (in bytes) of a relocated section.

Example: If the input file name is "test.o", the start address of the CODE section is 0x80000, and the size is 0x100

Symbol name	Symbol value
<code>__START_test_code</code>	0x80000
<code>__END_test_code</code>	0x80100
<code>__SIZEOF_test_code</code>	0x100

The path and extension of `<File name>` are cut irrespective of how they are written in a command file. The start address and size defined to a symbol are the same as the contents output to the link map file (real data address), and the end address is determined by the "start address" + "size".

### Section name by `-section` command

When a section name is defined by the `-section` command, the following symbols are created to indicate the information of the named section regardless of whether a `-objsym` command is specified:

<code>__START_&lt;Section name&gt;</code>	Defines the start address of a named section.
<code>__END_&lt;Section name&gt;</code>	Defines the end address of a named section.
<code>__SIZEOF_&lt;Section name&gt;</code>	Defines the size (in bytes) of a named section.

Here, `<Section name>` is the name specified by the `-section` command.

The start address defined to a symbol is the address that is specified by the `-section` command or an address immediately following the end of a default BSS section.

The size in the case of virtual sections (`-ucode`, `-udata`) equals the total size of the located sections. The size in the case of shared sections (`-ucode { }`, `-udata { }`, `-ubss { }`) is the same as that of the largest section among those located.

The end address is determined by the "start address" + "size".

**Example using section symbols in the assembly source**

Example: When transferring the CODE section of test.o to U section "CACHE1"

**Transfer routine in the source file**

```
xld.w    %r12, __START_CACHE1
xld.w    %r13, __START_test1_code
xld.w    %r14, __SIZEOF_test1_code
```

**HCOPY\_LOOP:**

```
ld.uh   %r4, [%r13]+      ; Transfers the instruction code.
ld.h    [%r12]+, %r4
sub     %r14, 2
jrgt   HCOPY_LOOP
```

**Setting U section by using the linker command**

```
-objsym
-section CACHE1 = 0x300
-ucode CACHE1 {test.o}
test.o
```

Note: Do not define symbols in the source that are assigned the same name as the section symbols used.

## 12.9 Linking Libraries

Libraries are linked after all other input object files are located.

### Searching the library module

Only when an unresolved external reference symbol exists in the input object, the lk33 searches library files in the order that they are written in a command file. It then reads in and links only the first module found that has the unresolved symbol. When an external reference is resolved, the lk33 stops searching for the subsequent modules or library files. Consequently, even when a specified library file contains multiple instances of the symbol to be searched, all but the first-found module are ignored. If the library does not exist in the current directory, the directory specified by the `-I` command is searched.

External reference between library files can be resolved only in the currently processed library file or in the library files to be searched next. External references defined in an already searched library file are not resolved. Therefore, be careful with the order in which library files are specified.

### Location of library modules

The addresses at which library modules are located cannot be specified. Each section in the linked library modules is located in the default section of the same type (i.e., a section in which all unspecified relocatable sections are located).

Example: Command file

```
-I c:\cc33\lib
sample1.o      ; Relocatable object file
sample2.o      ; Relocatable object file
sample.lib     ; Library file (module n is used)
```

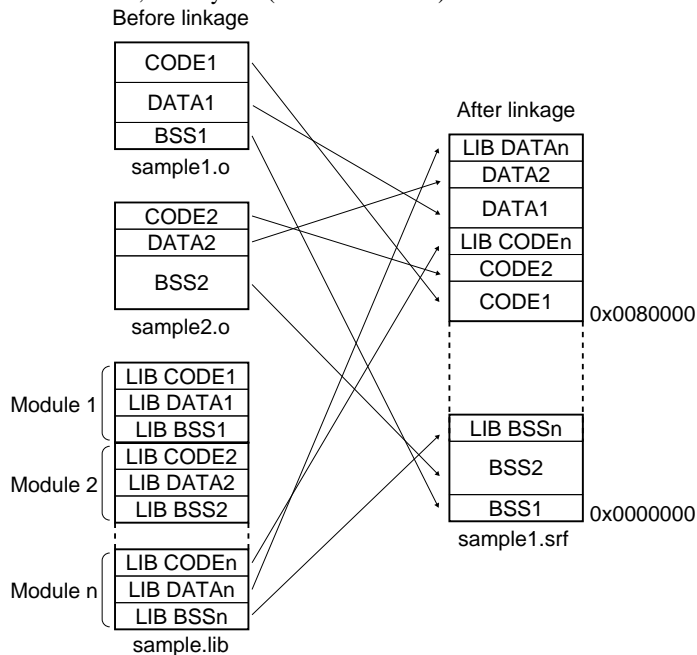


Fig. 12.9.1 Example of how library modules are located

### Location of library modules in the U section

No library file can be specified in the `-code { }`, `-data { }`, and `-bss { }` commands, as well as in the `-ucode { }`, `-udata { }`, and `-ubss { }` commands.

If you want to use a library file in a U section, restore the desired library module to the relocatable object file before linking.

## 12.10 Resolving Symbols

---

### Procedure for resolving global symbols

The lk33 follows the procedure described below to resolve symbols:

- (1) The lk33 adds global symbol information to the internal table sequentially in the order of the input object files specified. If an undefined symbol is referenced, the lk33 searches the table and when the matching symbol information is found, applies the content of that information. If no information is found in the table, the symbol is kept pending until it is defined in a subsequent input file.
- (2) If two or more global symbols of the same name are defined, a warning is output. The lk33 uses the first symbol information encountered when searched in order of input files as the valid symbol, and continues processing. (This does not apply to the global symbols defined in the BSS section.)
- (3) Only when an unresolved external reference is found after searching all input object files, the lk33 searches library files in the order in which they are entered. When the desired symbol definition is found, the lk33 links that module. If a symbol of the same name is defined in a multiple library module, a warning is generated, in which case the first encountered module is linked.
- (4) If an unresolved external reference in the library module to be linked is defined in one of the input object files or in the currently processed library file, this definition is applied. If it is not defined, the unresolved external reference is kept pending until it is defined in a subsequent library file.  
An external reference between library files can be resolved only in the currently processed file or in the library files to be searched next. External references defined in an already searched library file are not resolved.
- (5) If an undefined external reference still exists after all library files are searched, a warning is output, and the lk33 does not relocate the instructions which reference the symbol. The bits in the instruction codes that require modification become 0x0.

### Global symbols defined twice or more in the BSS section

If two or more global symbols of the same name are defined by the .comm pseudo-instruction, the symbol definition with the largest size is assumed to be the valid symbol. If there are two or more symbol definitions with the largest size, the definition first encountered during the search in the order of input files is assumed to be the valid symbol. The lk33 outputs a warning and continues processing. No warning is output if the -w command is specified.

The -d command deletes the areas of the symbols that have been invalidated by the above processing. The deleted areas in the BSS section are closed up before sections are relocated.

The -d command is valid only in the BSS section, and does not delete invalid symbol areas in any other section.

## 12.11 Link Map File

The link map file serves to refer to the mapping information of modules of each section.

Furthermore, this file can be input to the Instruction Extender ext33 along with a symbol file for code optimization.

It is output if you specify the -m option in the command file or command line.

The file format is a text file, and its file name is "<File name>.map". (<File name> is the same as that of the output object file.)

### Sample link map file

```
Code Section mapping
Address  Vaddress      Size      File          ID  Attr
00080000 -----      00000028  boot.o        0  REL
00080028 000002c4      0000005c  main.o        0  REL
00080084 00000274      00000050  foo.o         0  REL
000800d4 00000274      00000040  bar.o         0  REL

Data Section mapping
Address  Vaddress      Size      File          ID  Attr
00080114 00000a00      00000000  boot.o        1  REL
00080114 00000a00      00000038  main.o        1  REL
0008014c 00000a38      0000000c  foo.o         1  REL
00080158 00000a44      00000028  bar.o         1  REL

Bss Section mapping
Address  Vaddress      Size      File          ID  Attr
00000000 -----      00000000  boot.o        2  REL
00000000 -----      00000230  main.o        2  REL
00000230 -----      00000024  foo.o         2  REL
00000254 -----      00000020  bar.o         2  REL
```

### Contents of link map file

- Address** Indicates the start address of each section. Sections that have a value in Vaddress indicate an address where real data is stored.
- Vaddress** Indicates the execution address in the U section.
- Size** Indicates the section size.
- File** Indicates the file names of the linked module.
- ID** Indicates the section ID of each section in each object file.
- Attr** Indicates the attribute of the section:
  - REL: Means that it is a relocatable section.
  - ABS: Means that it is an absolute section.

## 12.12 Symbol File

The symbol file serves to refer to the symbols defined in all the modules and their address information.

Furthermore, this file can be input to the Instruction Extender ext33 along with a link map file for code optimization.

It is output if you specify the -s option in the command file or command line.

The file format is a text file, and its file name is "<File name>.sym". (<File name> is the same as that of the output object file.)

### Sample symbol file

Symbol	File	Section	Type	Address
DATA1	comm2.o	data	global	00080006
i	comm2.o	bss	global	00000100
buffer	main.o	bss	global	00000030
BOOT	main.o	code	global	00080000
main	main.o	code	global	00080100
i	lk21.o	data	local	00081008
Name	word.o	data	global	0008100c
:	:	:	:	:
__START_CACHE1	\$\$lk33\$\$	uscn	global	00000400
__END_CACHE1	\$\$lk33\$\$	uscn	global	000009ff
__SIZEOF_CACHE1	\$\$lk33\$\$	uscn	global	000005ff
:	:	:	:	:

### Contents of symbol file

- Symbol** Indicates all the defined symbols in order of processing.  
The section symbols created by the lk33 are output at the end of the file.
- File** Indicates the name of the object file in which symbols are defined.  
The section symbol is \$\$lk33\$\$.
- Section** Indicates the section type.  
code: CODE section  
data: DATA section  
bss: BSS section  
uscn: U section
- Type** Indicates the attribute of symbols.  
global: Means a global symbol.  
local: Means a local symbol.
- Address** Indicates the absolute address defined for a symbol.



## 12.13 Error/Warning Messages

Error and warning messages are displayed/output through the Standard Output (stdout). If you specify the `-e` option, the messages will also be delivered in the "lk33.err" file.

If the lk33 is started up using the wb33's [LK33] button, the message is output to "wb33.err". When execution is completed, a message is displayed in the output window (default).

### 12.13.1 Errors

The errors produced in the lk33 are classified into three groups: system errors, command file errors, and linker errors.

In case an error occurs, the lk33 will immediately terminate the processing after displaying an error message. No object file will be output. The link map file and the symbol file will be delivered only in the part which was processed prior to the occurrence of the error.

Table 12.13.1.1 List of system error messages

Error message	Content
<file name>: Error: Cannot open input file.	Cannot open the input file (*.o, *.lib).
<file name>: Error: Cannot open output file.	Cannot open the output file.
Error: Cannot open working file.	Cannot open the temporary file.
Error: Cannot allocate memory.	Cannot secure memory space.
Error: Cannot read a file.	Cannot read the file.

Table 12.13.1.2 List of command file error messages

Error message	Content
link command: Error: Unknown option near line = #.	There is a linker command described that cannot be recognized, in the proximity of line # inside linker command file.
link command: Error: Invalid parameter near line = #.	There is an error in the description of a linker command parameter, in the proximity of line # inside linker command file.
link command: Error: Not define section name near line = #.	The section name specified near the # line in the linker command file is not defined.
link command: Error: Uninitialized section name near line = #.	The section name specified near the # line in the linker command file does not have its address defined.

Table 12.13.1.3 List of linker error messages

Error message	Content
<file name>: Error: Code section map out of range.	The mapping address of the CODE section deviates the linkable range. Specification by the <code>-code</code> command exceeds the 0x0–0xffffffff range, or the mapping address of data exceeded the above range in the linking process.
<file name>: Error: Data section map out of range.	The mapping address of the DATA section deviates the linkable range. Specification by the <code>-data</code> command exceeds the 0x0–0xffffffff range, or the mapping address of data exceeded the above range in the linking process.
<file name>: Error: Bss section map out of range.	The mapping address of the BSS section deviates the linkable range. Specification by the <code>-bss</code> command exceeds the 0x0–0xffffffff range, or the mapping address of data exceeded the above range in the linking process.
Error: Too many object files.	The number of object files to be linked exceeded the limit (4,000 files).
Error: Too many output sections.	The number of output sections exceeded the limit (256 sections).
Error: Too many input sections.	The number of sections to be located in a default section exceeded the limit (4,000 sections).
Error: Too many library files.	The number of library files exceeded the limit (256 files).
Error: Too many global bss symbol.	The number of global symbols in the BSS section exceeded the limit (1,024 symbols).
Error: Too many object symbols.	The number of object symbols exceeded the limit (36,000 symbols).
Error: Too many U sections.	The number of U sections exceeded the limit (256 sections).

Error message	Content
Error: Too many section symbols.	The number of section symbols exceeded the limit (256 symbols).
Error: Too many U section symbols.	The number of U section symbols exceeded the limit (256 symbols).
Error: No object files.	The object file to be linked is not specified.
<file name>: Error: Not SRF33 Object file or library file.	The input file is not an srf33 object file or a library file.
Error: Chain information size is greater than file size. Error: Chain seek address is greater than file size. Error: Undefined symbol type. Error: Undefined relocation type.	There is a problem in the srf33 object file. Redo the processing from the C Compiler gcc33 or Preprocessor pp33.

## 12.13.2 Warning

Even when a warning appears, the lk33 continues with the processing. It completes the processing after displaying a warning message, unless an error takes place in addition. Object file, link map file, and symbol file will all be delivered, but the operation of the program is not guaranteed.

Table 12.13.2.1 List of warning messages

Warning message	Content
<file name1>: Warning: '<symbol>' already defined in '<file name2>'.	<symbol> defined in <file name1> was already defined in <file name2>. Correct it in the source file. The symbol definition that appears first according to the order of files to be linked is effective. If the -w command is specified, this warning is not output for duplicate global labels in the BSS section.
<file name>: Warning: unresolved external symbol '<symbol>'.	An undefined symbol was referred. The lk33 does not relocate the instructions which refer the symbol. The bits in the instruction codes that require modification become 0x0.
<file name>: Warning: Out of relocation range, Address = <address>.	The instruction exceeds the relocatable range. Only the bits that can be inserted in the instruction are modified after the relocation addresses are calculated.
<file name>: Warning: Section mapping conflict, Address = <address>.	The section is duplicated from <address>.
link command: Warning: Cannot mapping to USection '<file name>(<section>)'.	The <section> of the <file name> written in the command file cannot be relocated.
link command: Warning: Cannot find relocatable section '<file name>(<section>)'.	The relocatable section <section> of <file name> cannot be found.
link command: Warning: Alignment <address 1> -> <address 2>.	The <address 1> specified in the command file is adjusted for alignment as <address 2>.
Warning: Cannot access section information.	Section information cannot be accessed. The srf33 object file is erratic.
Warning: Invalid loader information.	The -ucode, -udata or -ubss command has been specified with the -ld command. Or, the input file contains a relocation type that is not allowed for ld33.

## 12.14 Precautions

---

- (1) The address range in which sections can be located is 0x0 to 0xffffffff. No error will occur in the linker as long as all modules are located within this range. However, care must be taken because the memory capacity will be limited depending on the microcomputer model to be developed.
- (2) The maximum number of object files that can be linked is 4,000. If this limit is exceeded, an error results.
- (3) The number of sections and U sections that can be output are both 256. If there are more sections to be output, an error results.
- (4) The maximum number of library files that can be specified when linking is 256. If this limit is exceeded, an error results. Note also that only up to four library search paths can be specified by the `-l` command.
- (5) Up to 256 section symbols and 256 U section symbols can be generated. If this limit is exceeded, an error results.

## Chapter 13 Disassembler

This chapter describes the functions of Disassembler dis33.

### 13.1 Functions

The Disassembler dis33 (hereafter called the "dis33") inputs the object files in the srf33 format and outputs the disassembled contents of the object's code part in list form, while corresponding to the source one to one. It also delivers a dump output of the data part. It is effective to verify a program following its linking or debugging. The output can be selected by specifying an appropriate startup option.

### 13.2 Input/Output Files

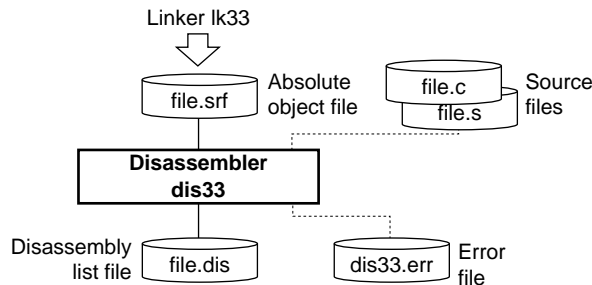


Fig. 13.2.1 Flowchart

#### 13.2.1 Input Files

##### Object file

File format: Binary file in srf33 format  
 File name: <File name>.srf, or <File name>.o  
 Description: Object file created by the Linker lk33 or Assembler as33. To deliver the source in a mixed output, the file needs to contain debugging information.

##### Source file

File format: Text file  
 File name: <File name>.c and <File name>.s  
 Description: When the source is delivered in a mixed output, the source file will also be input according to the source file name information contained in the object file mentioned above.

#### 13.2.2 Output Files

##### Disassembler file

File format: Text file  
 File name: <File name>.dis (<File name> is the same as that of the input file.)  
 Output destination: Current directory  
 Description: Disassembled contents of the input file are delivered. For contents of the output, refer to Section 13.5 "Disassembling Output".

##### Error file

File format: Text file  
 File name: dis33.err  
 Output destination: Current directory  
 Description: File that is delivered when the startup option (-e) is specified. It records the information which the dis33 outputs to the Standard Output (stdout), such as error messages.

## 13.3 Starting Method

---

### 13.3.1 Startup Format

#### General form of command line

**dis33 ^ [<startup option>] ^ <file name>**

^ denotes a space.

[ ] indicates the possibility to omit.

<file name>: Specify an srf33 object file name including the extension.

#### Operations on work bench

Select options and an object file, then click the [DIS33] button.

### 13.3.2 Startup Options

The dis33 comes provided with the following five types of startup options:

#### -m

Function: Source mixed output

Specification on wb33: Check [src mix].

Explanation:

- Reads a source file and delivers source codes in correspondence with disassembled codes.
- To specify this option, the object file to be input needs to contain debugging information.

#### -c

Function: Output of code section only

Specification on wb33: Check [code only].

Explanation:

- Delivers only the disassemble result of the code section. Does not dump data.

#### -d

Function: Output of data section only

Specification on wb33: Check [data only].

Explanation:

- Delivers only the dump result of the data section. Does not disassemble the code section.

#### -a <start address> <end address>

Function: Specification of address range

Specification on wb33: Check [addr range] and enter the start and end addresses in the text box.

Explanation:

- Specifies an address range to be disassembled.
- One address range can only be specified. The start and end addresses must be specified as a hexadecimal number. "-a" , <start address> and <end address> must be separated with one or more spaces.
- When a value exceeding 28 bits (0x0fffffff) is specified for the address, it is handled as 0c0fffffff. When the start address is higher than the end address or the address is specified not a hexadecimal number, an error occurs and the usage message will be displayed.
- If this option is not specified, all the addresses in the specified file will be disassembled.

#### -e

Function: Output of error files

Specification on wb33: None

Explanation:

- Delivers also in a file (dis33.err) the contents to be output by the dis33 through the Standard Output (stdout), such as error messages.

When entering an option in the command line, one or more spaces are necessary before and after the option.

Example: c:\cc33\dis33 -e -m test.srf

**Combination of -m, -c and -d**

The explanations given above refer to the case where only one function is specified. When the functions are specified in combination with one another, they change as detailed further below. There is no rule established for the order of combination.

Without any option specified	Delivers a disassembling output of the code section and dump output.
-c -m	Delivers a mixed output of the code section.
-d -m	Delivers only a dump output of the data section.
-c -d	Delivers an empty file.
-c -d -m	Delivers an empty file.

## 13.4 Messages

---

The dis33 delivers its messages through the Standard Output (stdout).

If the dis33 is started up by using the wb33's [DIS33] button, the message is output to "wb33.err". When execution is completed, a message is displayed in the output window (default).

**End message**

The dis33 outputs only the following end message when it ends normally.

```
Disassemble complete
```

**Usage output**

If no file name was specified or an option was not specified correctly, the dis33 ends after delivering the following message concerning the usage:

```
Disassembler 33 Ver x.xx
Copyright (C) SEIKO EPSON CORP. 199x.
Usage:
dis33 [options] <file name>
Options:
-e      : produce log file (dis33.err)
-m      : generate disassemble code with source mix
-c      : generate disassemble code only
-d      : generate data dump only
-a address1 address2
        : specify disassemble area
        address1 - start address, hexadecimal number
        address2 - end address, hexadecimal number
Output:
Disassemble file (.dis)
Log file (dis33.err)
Example:
dis33 -e -m -a 0x80000 0x8ffff sample.srf
```

**When error/warning occurs**

If an error is produced, an error message will appear before the end message shows up.

Example: Error: Can't open file, test.srf.

```
Disassemble complete
```

In the case of an error, the dis33 ends without creating an output file.

If a warning is issued, a warning message will appear before the end message shows up.

Example: Warning: No debug information.

```
Disassemble complete
```

In the case of a warning, the dis33 ends after creating an output file.

For details on errors and warnings, refer to Section 13.6 "Error/Warning Messages".

## 13.5 Disassembling Output

### 13.5.1 Mix Output

When the mixed output (-m option) is specified, the code output is delivered with the contents of the source file added.

The dis33 acquires and reads the source file name from the debugging information of the srf33 file entered in it. Therefore, the source section will not be output, if you did not specify the -g option designed for addition of debugging information in the processing of the C Compiler gcc33 or Preprocessor pp33, Assembler as33 and Linker lk33.

It will also deliver a hexadecimal dump output after outputting the code section, if there is a data section and the -c option is not specified. (For details, refer to Section 13.5.3 "Data Output".)

#### Output format

```

***** Disassemble code and source code *****
Addr  Code  Unassemble      Line      Source
                                     -- <Source file name*> --
:      :      :           :           :
      (Disassembly code)                (Source code)
:      :      :           :           :

***** Data *****
Addr   +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +A +B +C +D +E +F
00800000                (Data)
:       : : : : : : : : : : : : : : : :

```

**Addr** Indicates the address of a code/data in hexadecimal number.

**Code** Indicates an object code in hexadecimal number.

**Unassemble** Indicates a mnemonic code resulting from disassembling.

**Line** Indicates a line number of the source file in decimal number.

**Source** Indicates a statement in the source file.

\* The names of the source file/included file which were referred to are delivered in their respective start positions.

#### Sample output

##### Source file "sample1.s"

```

#define SP_INI 0x0800      ; sp is in end of 2KB RAM
#define GP_INI 0x0000      ; global pointer %r8 is 0x0

```

```

.code
.word BOOT                ; BOOT VECTOR
BOOT:
  xld.w    %r8, SP_INI
  ld.w     %sp, %r8        ; set SP
  ld.w     %r8, GP_INI    ; set global pointer
  xcall    main            ; goto main
  xjp      BOOT           ; infinity loop

```

```
#include sample2.s
```

##### Source file "sample2.s"

```
#include sample3.s
```

##### Source file "sample3.s"

```
.data
.word 1 2
```

**Source file "sample4.c"**

```

int i;

main()
{
    int j;

    i = 0;
    for (j=0 ; ; j++)
    {
        i++;
    }
}

```

**Disassembling output**

\*\*\*\* Disassemble code and source code \*\*\*\*

Addr	Code	Unassemble	Line	Source												
00080000	0004	***														
00080002	0008	***														
			---	sample1.s ---												
			00001	#define SP_INI 0x0800 ; sp is in end of 2KB RAM												
			00002	#define GP_INI 0x0000 ; global pointer %r8 is 0x0												
			00003													
			00004	.code												
			00005	.word BOOT ; BOOT VECTOR												
			00006	BOOT:												
			00007	xld.w %r8, SP_INI												
00080004	C020	ext 0x20														
00080006	6C08	ld.w %r8, 0x0														
00080008	A081	ld.w %sp, %r8	00008	ld.w %sp, %r8 ; set SP												
0008000A	6C08	ld.w %r8, 0x0	00009	ld.w %r8, GP_INI ; set global pointer												
0008000C	C000	ext 0x0	00010	xcall main ; goto main												
0008000E	1C03	call 0x3														
00080010	1EFA	jp 0xfa	00011	xjp BOOT ; infinity loop												
00080012	0000	nop														
			---	sample4.c ---												
			00001	int i;												
			00002													
			00003	main()												
			00004	{												
			00005	int j;												
			00006													
			00007	i = 0;												
00080014	6C0B	ld.w %r11, 0x0														
00080016	C004	ext 0x4														
00080018	3C8B	ld.w [%r8], %r11														
			00008	for (j=0 ; ; j++)												
			00009	{												
			00010	i++;												
0008001A	C004	ext 0x4														
0008001C	308A	ld.w %r10, [%r8]														
0008001E	601A	add %r10, 0x1														
00080020	C004	ext 0x4														
00080022	3C8A	ld.w [%r8], %r10														
00080024	1EFB	jp 0xfb	00008	for (j=0 ; ; j++)												
			00011	}												
00080026	0640	ret	00012	}												
			****	Data ****												
Addr	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
00080020									01	00	00	00	02	00	00	00

The source lines corresponding to the codes and the contents of the sources preceding them are delivered.

Sources without actual codes and included files without actual codes will not be delivered either.

If one source line is expanded into codes of two or more lines as in the case of a for statement, such a source line may appear at various places.

If the source file does not exist, "no source" is output to the source field.



## 13.5.2 Code Output

When mixed output (-m option) is not specified, the code section of an input srf33 file is disassembled in order of the addresses, and delivered in the following format:

### Output format

```
***** Disassemble code *****
Addr Code  Unassemble
:       :       :
(Disassembly code)
:       :       :
```

**Addr** Indicates the address of a code in hexadecimal number.

**Code** Indicates an object code in hexadecimal number.

**Unassemble** Indicates a mnemonic code resulting from disassembling.

### Sample output

#### Source file

```
. abs
. code
. org 0x80000
. word BOOT

. org 0x80080
xjp      BOOT

. org 0x80100
BOOT:
xld.w    %r8, 0x800
ld.w     %sp, %r8
ld.w     %r8, 0x0
STNDBY:
halt
jp       STANDBY
```

#### Disassembling output

```
***** Disassemble code *****
Addr Code Unassemble
00080000 0100 ***
00080002 0008 ***

00080080 1E40 jp      0x40

00080100 C020 ext     0x20
00080102 6C08 ld.w    %r8, 0x0
00080104 A081 ld.w    %sp, %r8
00080106 6C08 ld.w    %r8, 0x0
00080108 0080 halt
0008010A 1E00 jp      0x0
```

A line is fed where the addresses are discontinuous.

### 13.5.3 Data Output

The data section is dumped out in hexadecimal numbers by the amount corresponding to its size in order of the addresses. If the input file has no data, no data output will take place. If you specify the -d option alone and input a file without data, an empty file will be delivered.

#### Output format

```

**** Data ****
Addr  +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +A +B +C +D +E +F
00800000          (Data)
:           : : : : : : : : : : : : : : :

```

**Addr** Indicates an address of data in hexadecimal number. It is the start address of that line (16 addresses).

**+0 to +F** Indicates data corresponding to 16 addresses in hexadecimal numbers. Address without data defined will remain in blank.

#### Sample output

##### Data definition in the source file

```

.data
.org      0x8008
CHAR1:
.half    0x9
.half    0xa
.half    0xb

.org      0x8104
CHAR2:
.word    0x3
.word    0x4
.word    0x5

```

##### Disassembling output

```

**** Data ****
Addr  +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +A +B +C +D +E +F
00008000          09 00 0A 00 0B 00

00008100          03 00 00 00 04 00 00 00 05 00 00 00

```

A line is fed where the addresses are discontinuous. Also, a space is inserted at addresses that do not have data.

## 13.6 Error/Warning Messages

Error and warning messages are displayed/output through the Standard Output (stdout). If you specify the `-e` option, the messages will also be delivered in the "dis33.err" file.

If the dis33 is started up using the wb33's [DIS33] button, the message is output to "wb33.err". When execution is completed, a message is displayed in the output window (default).

### 13.6.1 Errors

The errors produced in the dis33 are classified into two groups: system errors and input file format errors.

When an error occurs, the dis33 immediately terminates the processing after displaying an error message. It will not output any disassembler file.

Table 13.6.1.1 List of system error messages

Error message	Content
Error: Cannot open file, <file name>.	Cannot open the file.
Error: Cannot close file, <file name>.	Cannot close the file.
Error: Cannot write to file, <file name>.	Cannot write to the file.
Error: Cannot close SROFF file.	Cannot close the object file.
Error: Cannot load data, memory allocation failure.	The section information cannot be loaded due to a memory allocation error.
Error: Cannot load debug information, memory allocation failure.	The debug information cannot be loaded due to a memory allocation error.
Error: Cannot load data, file read failure, <file name>.	The section information cannot be loaded due to a file read error.
Error: Cannot load debug information, file read failure, <file name>.	The debug information cannot be loaded due to a file read error.

Table 13.6.1.2 List of input file error messages

Error message	Content
Error: Invalid file name <file name>.	".dis" is specified for the input file name.
Error: Too many sections. Error: Cannot load data, please check SROFF file. Error: Cannot load debug information, please check SROFF file.	These errors are produced when there is an error in the information contained in the input srf33 object file. In such case, check to make sure that the files in the phases ranging from the source through linking retain consistency, and redo the processing from the C compiler or preprocessor by using definitive source files.

### 13.6.2 Warning

Even if a warning is issued, the dis33 keeps on processing, and completes the processing after displaying a warning message, unless any error is produced in addition. It will output the disassembler file.

Table 13.6.2.1 Warning message

Warning message	Content
Warning: No debug information.	The input file does not contain debugging information. This warning is produced only when you activated a mixed output by specifying the <code>-m</code> option. Since there is no debugging information, the disassembler cannot output a source, but only delivers a disassembling output.
Warning: Cannot open file, <file name>.	Cannot open the source file for mixed output. "no source" is output for the source field.
Warning: Line number of source file is invalid.	The source lines are insufficient. The source might be modified.

## 13.7 Precautions

---

To obtain a source mixed output by the dis33, pay heed to the following aspects:

- (1) When describing a source, set the tab every 8 characters. If any other tab setting is made, the output position will appear displaced.
- (2) The dis33 acquires and reads a source file name from the debugging information data of the input srf33 file. Therefore, you need to input an object file created in the processing of the C Compiler gcc33 or Preprocessor pp33, Assembler as33 and Linker lk33, with the -g option specified for addition of debugging information. In the case where an object file which holds debugging information for source display is linked with another object file which does not have such information, only the file with debugging information will be delivered in a source mixed output.
- (3) Pay attention to the consistency of the source file to the object file to be input. If the source file is modified after the object file was created, you will not be able to obtain an output with correct correspondence between the codes and the source. Or, an error will result, and no output will be delivered.

## Chapter 14 Binary/HEX Converter

This chapter describes the functions of the Binary/HEX Converter hex33.

### 14.1 Functions

The Binary/HEX Converter hex33 (hereafter called the "hex33") inputs the object files in srf33 format and outputs a specified address range to a file after converting it into Motorola S3 format data. The areas in the specified address range that do not have data are filled with 0xff. For the addresses of converted data, you can specify absolute addresses or offset addresses from a specified start address.

Use the hex33 in the following cases:

- When creating the mask data by extracting the internal ROM data from completed srf33 object file
- When creating the data you want to be written into the external ROM of the target board or product
- When verifying the completed program with the Debugger for final acceptance

### 14.2 Input/Output Files

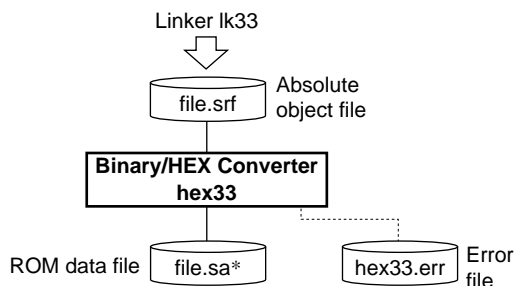


Fig. 14.2.1 Flowchart

#### 14.2.1 Input File

##### Object file

- File format: Binary file in srf33 format  
 File name: <File name>.srf  
 Description: Object file created by the Linker lk33.

#### 14.2.2 Output Files

##### HEX file

- File format: HEX file in Motorola S3 format  
 File name: <File name>.sa\* (<File name> is the same as that of the input file.)  
 Output destination: Current directory  
 Description: File in which the specified address range of the input file is converted in Motorola S3 format.

##### Error file

- File format: Text file  
 File name: hex33.err  
 Output destination: Current directory  
 Description: File that is delivered when the startup option (-e) is specified. It records the information which the hex33 outputs to the Standard Output (stdout), such as error messages.

## 14.3 Starting Method

---

### 14.3.1 Startup Format

#### General form of command line

**hex33 ^ [<startup option>] ^ <start address> ^ <end address> ^ <file name>**

^ denotes a space.

[ ] indicates the possibility to omit.

<start address>: Specify the conversion start address in a hexadecimal number.

<end address>: Specify the conversion end address in a hexadecimal number.

<file name>: Specify an srf33 object file name including the extension.

- \* A 32-bit boundary address should be specified for the <start address> and <end address>. The specified number is handled as a hexadecimal number even if "0x" is not prefixed.

#### Operations on work bench

Select options, conversion range and an object file, then click the [HEX33] button.

### 14.3.2 Startup Options

The hex33 comes provided with the following four types of startup options:

#### -x

Function: Adds a specified address to the extension.

Specification on wb33: Check [addr to name].

Explanation:

- If this option is specified, the hex33 adds a specified start and an end address to the extension of the file to be generated.  
Example: test.sa\_c00000\_c0ffff  
It can tell you which part of the data it is at a glance.

#### -z

Function: Outputs an absolute address.

Specification on wb33: Check [abs addr].

Explanation:

- If this option is specified, the hex33 uses absolute addresses for the converted address part when generating the output file. Unless this option is specified, the addresses are converted to offset addresses where the specified start address is assumed to be address 0.

Note:

- Be sure to specify this option when creating mask data.

#### -r

Function: Checks a section.

Specification on wb33: Check [abs addr].

Explanation:

- If this option is specified, the hex33 checks whether all converted sections are within a specified address range. If there is any section that exceeds the specified range, an error is assumed.

#### -e

Function: Outputs an error file.

Specification on wb33: None

Explanation:

- Delivers also in a file (hex33.err) the contents to be output by the hex33 through the Standard Output (stdout), such as error messages.

When entering an option in the command line, one or more spaces are necessary before and after the option.

Example: c:\cc33\hex33 -e -x -z test.srf

## 14.4 Messages

---

The hex33 delivers its messages through the Standard Output (stdout).

If the hex33 is started up by using the wb33's [HEX33] button, the message is output to "wb33.err". When execution is completed, a message is displayed in the output window (default).

### End message

The hex33 outputs only the following end message when it ends normally.

```
Conversion Completed
```

### Usage output

If no file name was specified or an option was not specified correctly, the hex33 ends after delivering the following message concerning the usage:

```
HEX Data Converter 33 Ver x.xx
Copyright (C) SEIKO EPSON CORP. 199x
Usage:
  hex33 [options] <start address> <end address> <file name>
Options:
  -e : produce log file (hex33.err)
  -x : add start and end addresses to the file name extension
  -z : make converted address absolute
  -r : check all data within start and end area
Output:
  hex file (.sa, .sa_<start address>_<end address>)
  log file (hex33.err)
Example:
  hex33 -x -z 80000 80fff sample.srf
```

### When error/warning occurs

If an error is produced, an error message will appear before the end message shows up.

Example: Error: Input file is not SRF33 file.

```
Conversion Completed
```

In the case of an error, the hex33 ends without creating an output file.

If a warning is issued, a warning message will appear before the end message shows up.

Example: Warning: Section information chain is not found.

```
Conversion Completed
```

In the case of a warning, the hex33 ends after creating an output file.

For details on errors and warnings, refer to Section 14.6 "Error/Warning Messages".

## 14.5 Contents of HEX File

### 14.5.1 Motorola S3 Format

The hex33 converts srf33 object files into the Motorola S3 format that supports 32-bit addressing.

The diagram below shows the format of each line in the output file.

S3	length(1)	address(4)	data(1)	----	data(1)	sum(1)	\n
----	-----------	------------	---------	------	---------	--------	----

:

S7	length(1)	00000000	sum(1)	\n
----	-----------	----------	--------	----

Numbers in ( ) are bytes.

**S3:** Indicates that the line is a data record.

**S7:** Indicates that the line is an end record (end of data).

**length:** Indicates the record length of "address + data + sum". The data records output by the hex33 are always 0x25, while the end records are 0x05.

**address:** Indicates the address where the head data in a record is placed.

**data:** This is 32-byte data. This is not included in the end record.

**sum:** This is a checksum (1's complement) from "length" to the last data.

**\n:** This is a return code.

The end records are always S70500000000FA.

### 14.5.2 Absolute Address Output

If the -z option is specified, an absolute address is placed in the "address" part of the output file.

#### Example of dump of an output file

```
S325000800000400080020C0086C81A0086C00C000C0021CF91E00020B6C00C000C08B3C006CFC
S325000800200C2E051C1060FD1E400240061C70081800C000C08A301A6000C000C08A3C400658
S32500080040FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFB2
S32500080060FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF92
.
S32500080FA0FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF43
S32500080FC0FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF23
S32500080FE0FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF03
S70500000000FA
```

Shown above is an example of a file that was created after specifying 0x80000 for the start address, 0x80fff for the end address, and the -z option.

Data records for 32 addresses each are generated, with the address part ranging from 00080000 to 00080fe0. All areas that do not have data are filled with 0xff.

### 14.5.3 Offset Address Output

If the -z option is omitted, an offset address from the specified start address is placed in the "address" part of the output file.

#### Example of dump of an output file

```
S325000000000400080020C0086C81A0086C00C000C0021CF91E00020B6C00C000C08B3C006C04
S325000000200C2E051C1060FD1E400240061C70081800C000C08A301A6000C000C08A3C400660
S32500000040FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFBA
S32500000060FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF9A
.
S32500000FA0FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF4B
S32500000FC0FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF2B
S32500000FE0FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0B
S70500000000FA
```

Shown above is a conversion result of the example in the preceding page generated without specifying the -z option. The addresses in the "address" part are offset addresses from address 0x80000.



## 14.6 Error/Warning Messages

Error and warning messages are displayed/output through the Standard Output (stdout). If you specify the `-e` option, the messages will also be delivered in the "hex33.err" file.

If the hex33 is started up using the wb33's [HEX33] button, the message is output to "wb33.err". When execution is completed, a message is displayed in the output window (default).

### 14.6.1 Errors

The errors produced in the hex33 are classified into two groups: system errors and input file format errors.

When an error occurs, the hex33 immediately terminates the processing after displaying an error message. It will not output any HEX file.

Table 14.6.1.1 List of system error messages

Error message	Content
Error: File open error.	Cannot open the file.
Error: File write error.	Cannot write to the file.
Error: File read error.	Cannot read the file.
Error: Memory allocation error.	Cannot secure memory space.

Table 14.6.1.2 List of input file error messages

Error message	Content
Error: "<file name>" file could not be opened.	Cannot open the file <file name>.
Error: Input file is not SRF33 file.	The input file is not an object file in srf33 format.
Error: Start address error.	The conversion start address is invalid. The specified address either exceeds the effective range of 0x00000000 to 0x0ffffff or does not reside on 32-byte boundaries.
Error: End address error.	The conversion end address is invalid. The specified address either exceeds the effective range of 0x00000000 to 0x0ffffff or does not reside on 32-byte boundaries.
Error: Out of area in address <address>.	The converted data exceeded a specified address range.
Error: Chain information size is greater than file size.	These errors occur when the input srf33 object files contain erroneous information. Check all the files from the source to the linked files for consistency. If necessary, reprocess from the C Compiler and Preprocessor using the final source files.
Error: Chain seek address is greater than file size.	
Error: File control flag error.	
Error: Section address error.	
Error: Section ID error.	
Error: Data conflicted at <address>.	

### 14.6.2 Warning

Even if a warning is issued, the hex33 keeps on processing, and completes the processing after displaying a warning message, unless any error is produced in addition. It will output the HEX file.

Table 14.6.2.1 Warning message

Warning message	Content
Warning: Section information chain is not found.	There is no section information chain in the input file. This warning is produced when the input srf33 object files contain erroneous information. Check all the files from the source to the linked files for consistency. If necessary, reprocess from the C Compiler and Preprocessor using the final source files.

## 14.7 Precautions

---

- (1) Specify hexadecimal 32-byte boundary addresses for the conversion start and end addresses.
- (2) When converting internal ROM data into mask data, for the conversion start end addresses, specify the start and end addresses of the internal ROM of the model being developed, and produce the output file in absolute addresses (by specifying the -z option). Even if the program size is small, the HEX file must be created for all areas to the end address of the internal ROM.

## Chapter 15 Librarian

This chapter describes the functions of the Librarian lib33.

### 15.1 Functions

The librarian lib33 (hereafter called the "lib33") is a software tool for editing the srf33 format library files. It allows you to create a library from the relocatable object files output by the Assembler as33. A library of general-purpose modules will help you reduce the time and cost required for developing a product using the EOC33 Family of microcomputers in the future.

The lib33 has the following features:

- Adds the relocatable object files output by the Assembler as33 to an existing library file.
- Creates a new library file from the relocatable object files output by the Assembler as33.
- Outputs a list of modules in a library file.
- Deletes specified modules from a library file.
- Restores specified modules in a library file to the original relocatable object file.

Once the created or edited library file is specified during linkage, only the necessary modules in that file are linked with other object files.

### 15.2 Input/Output Files

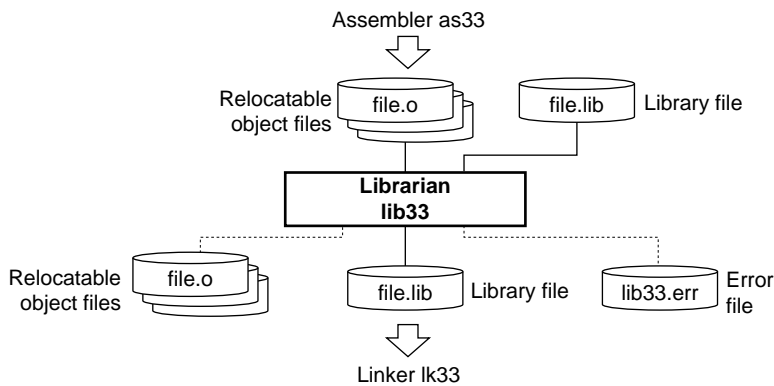


Fig. 15.2.1 Flowchart

#### 15.2.1 Input Files

##### Library file (except when creating a new library)

File format: Binary file in srf33 library format  
 File name: <File name>.lib  
 Description: This is a library file included with the package or one that is created by the lib33.

##### Relocatable object file (when creating a library or adding to a library)

File format: Binary file in srf33 format  
 File name: <File name>.o  
 Description: This is a relocatable object file created by the Assembler as33.

#### 15.2.2 Output Files

##### Library file (when creating a new library or adding/deleting modules)

File format: Binary file in srf33 library format  
 File name: <File name>.lib  
 Output destination: Current directory  
 Description: This file is comprised of multiple relocatable object modules. For the contents of a library file, refer to Appendix, "srf33 File Structure".

**Relocatable object file (when restoring modules)**

File format:	Binary file in srf33 format
File name:	<File name>.o
Output destination:	Current directory
Description:	The specified modules are restored to the original relocatable object file.

**Error file**

File format:	Text file
File name:	lib33.err
Output destination:	Current directory
Description:	File that is delivered when the startup option (-e) is specified. It records the information which the lib33 outputs to the Standard Output (stdout), such as error messages.

## 15.3 Starting Method

---

### 15.3.1 Startup Format

**General form of command line**

**lib33 ^ [<startup option>] ^ <library file name> ^ [<object file name>]**

^ denotes a space.

[ ] indicates the possibility to omit.

<library file name>: Specify a library file you want to edit including the file name extension.

<object file name>: Specify a file you want to be registered, deleted, or restored including the file name extension. Multiple object files can be specified.

**Operations on work bench**

Select options and input files, then click the [LIB33] button.

### 15.3.2 Startup Options

The lib33 comes provided with the following five types of startup options:

The actual method will be described later.

**-a**

Function: Adds an object file to a library.

Specification on wb33: Check [add]. (addition)

Explanation:

- If an existing library is specified, the specified object file is added to the library.
- If a new library file name is specified, a library file is created from the specified object file.

**-d**

Function: Deletes an object file from a library.

Specification on wb33: Check [del].

Explanation:

- The specified object file is deleted from the specified library.

**-l**

Function: Displays object files registered in a library.

Specification on wb33: Check [list].

Explanation:

- The object files in the specified library are listed in the order they are stored.
- These files are listed on a standard output device (stdout).

**-x**

Function: Restores library modules to the original object files.

Specification on wb33: Check [extract] (Only specified modules are restored) or [extract all] (All modules are restored)

- Explanation:
- If an object file name is specified, only the specified modules in the library are restored to the previous object file.
  - If no object file name is specified, all the modules in the library are restored to the previous object files.

The `-a`, `-d`, `-l` and `-x` options cannot be specified simultaneously; they can only be specified one at a time.

If none of these options are specified, a new library file is created from the specified object file (check [new] in the `wb33`).

**-e**

Function: Output of error files

Specification on `wb33`: None

- Explanation:
- Delivers also in a file (`lib33.err`) the contents to be output by the `lib33` through the Standard Output (`stdout`), such as error messages.

When entering an option in the command line, one or more spaces are necessary before and after the option.

Example: `c:\cc33\lib33 -e -a sample.lib test1.o test2.o`

## 15.4 Messages

---

The `lib33` delivers its messages through the Standard Output (`stdout`).

If the `lib33` is started up by using the `wb33`'s [LIB33] button, the message is output to "`wb33.err`". When execution is completed, a message is displayed in the output window (default).

### End message

The `lib33` outputs only the following end message when it ends normally.

```
Librarian Completed
```

### Usage output

If no file name was specified or an option was not specified correctly, the `lib33` ends after delivering the following message concerning the usage:

```
Librarian 33 Ver x.xx
Copyright (C) SEIKO EPSON CORP. 199x
Usage:
  lib33 [option] <library-file> <files>
Options:
  -e : produce log file (lk33.err)
  -a : add new files after an existing member of the library
  -d : delete members from the library
  -l : display a table listing the contents of library
  -x : extract members from the library
Output:
  E0C33 library file (.lib)
Example:
  lib33 -a libc.lib putc.o getc.o
```

### When error/warning occurs

If an error or a warning is produced, an error message or a warning message will appear before the end message shows up.

```
Example: test.lib: Error: Cannot open file
        Librarian Completed
```

In the case of an error, the `lib33` ends without creating an output file.

In the case of a warning, the `lib33` ends after creating an output file.

For details on errors and warnings, refer to Section 15.6 "Error/Warning Messages".

## 15.5 Library Editing Functions

---

### 15.5.1 Creating a New Library

To create a new library, execute the lib33 by specifying no option but -e.

**lib33 <Library file name> <Object file name> ... <Object file name>**

Example: c:\cc33\lib33 test.lib test1.o test2.o test3.o

A library file is created in the specified name. Object files are stored in it in the order in which they are specified.

When creating a library file in the wb33, check the [new] button in the [other options] window and input a library file name in the text box located below the button. (The extension ".lib" is unnecessary.) Choose the object files that you want to be registered in the library from the file list box of the [other options] window.

Notes:

- Only the srf33 format relocatable object files (\*.o) can be registered in a library. Absolute object files cannot be registered in a library.

- If an existing library name is specified, it is overwritten with the specified object files.
- The maximum number of object modules that can be registered in one library file is 256. If this limit is exceeded, an error results.

### 15.5.2 Adding Modules to a Library

To add object files to an existing library, execute the lib33 using the following startup command:

**lib33 -a <Library file name> <Object file name> ... <Object file name>**

Example: c:\cc33\lib33 -a test.lib test1.o test2.o test3.o

By specifying the -a option and an existing library file name, you can add the specified object files to that library. The object files are registered at the end of the library file in the order in which they are specified.

When adding modules to a library in the wb33, choose a library file name from the file list box of the execution window and object file names to be added from the file list box of the [other options] window. Since no directory can be specified for the object files, the object files you want must be prepared in the same directory as that of the library file before selecting them. When you check the [add] button to execute the command, the selected object files are registered in the library.

Notes:

- If a nonexistent library file name is specified, a new library file is created in that library name.

- Only the srf33 format relocatable object files (\*.o) can be registered in a library. Absolute object files cannot be registered in a library.
- If the specified object file has the same name as an already registered module, a warning is issued, but the specified file is added at the end of the library in the same name. Then, if the object file name is specified in a delete command, both modules are deleted.
- The maximum number of object modules that can be registered in one library file is 256. If this limit is exceeded, an error results.

### 15.5.3 Listing Registered Modules

To view a list of the object files registered in a library, execute the lib33 using the following startup command:

**lib33 -l <Library file name>**

```
Example: c:\cc33\lib33 -l string.lib
The files list in string.lib
strerror.o
strcat.o
strchr.o
:
memset.o
strtok.o
Librarian Completed
```

All object file names included in the specified library are listed in the order they are stored. This list is displayed on (output to) a standard output device (stdout). If the -e option is specified, the list is also output to the "lib33.err".

When executing this command in the wb33, choose <Library file name> from the file list box of the execution window and check the [list] button to execute the command.

### 15.5.4 Deleting Modules from a Library

The library modules that have become unnecessary can be deleted from the library file.

**lib33 -d <Library file name> <Object file name> ... <Object file name>**

```
Example: c:\cc33\lib33 -d test.lib test1.o test2.o
```

If multiple object files with the same name are registered in the specified library, all of them are deleted.

When executing this command in the wb33, choose <Library file name> from the file list box of the execution window and input the <Object file name> that you want deleted in the text box below the [del] button of the [other options] window (extension ".o" is unnecessary). Only one object file name can be specified in this text box at a time. If you want to delete multiple object files, execute the lib33 as many times as the number of files to be deleted or input all these file names from the command line.

When you check the [del] button to execute the command, the specified object file is deleted from the library.

### 15.5.5 Restoring Object Files

The modules registered in a library can be restored to the original srf33 format relocatable object files. To restore modules to the original object file, execute the following startup command.

**lib33 -x <Library file name> <Object file name> ... <Object file name>**

```
Example: c:\cc33\lib33 -x test.lib test1.o test2.o
```

If the command is executed without specifying <Object file name>, all modules in the library are restored to the original object files. Even when modules in a library are restored in this way, the contents of the library file remain intact. If multiple modules with the specified name are registered in the library file, the last module registered is restored.

When executing this command in the wb33, choose <Library file name> from the file list box of the execution window and input the <Object file name> that you want restored in the text box on the right side of the [extract all] button of the [other options] window (extension ".o" is unnecessary). Only one object file name can be specified in this text box at a time. If you want to restore multiple object files, execute the lib33 as many times as the number of files to be restored or input all these file names from the command line.

When you check the [extract] button to execute the command, the specified object file is restored.

If you want to restore all the modules, check the [extract all] button to execute the command.

## 15.6 Error/Warning Messages

Error and warning messages are displayed/output through the Standard Output (stdout). If you specify the `-e` option, the messages will also be delivered in the "lib33.err" file.

If the lib33 is started up using the wb33's [LIB33] button, the message is output to "wb33.err". When execution is completed, a message is displayed in the output window (default).

### 15.6.1 Errors

When an error occurs, the lib33 immediately terminates the processing after displaying an error message. It will not create any output file.

Table 15.6.1.1 List of error messages

Error message	Content
<file name>: Error: Cannot open file	Cannot open the file.
<file name>: Error: Not srf33 library file	The specified library file is not in the srf33 library format.
Error: Max object files	The number of objects in the library exceeded the upper limit. The maximum number of objects that can be registered in one library file is 256.

### 15.6.2 Warnings

Even if a warning is issued, the lib33 keeps on processing, and completes the processing after displaying a warning message, unless any error is produced in addition. It will create the output file, but the results are not guaranteed.

Table 15.6.2.1 List of warning messages

Warning message	Content
<file name>: Warning: Not srf33 object file	The specified object is not a srf33 format relocatable object file.
Warning: Multiple object file '<file name>'	The specified object file is already registered. Even in this case, the specified file is added at the end of the library.
Warning: Cannot find '<file name>' in library	The object file specified to be deleted or restored cannot be found in the library.

## 15.7 Precautions

- (1) Only the srf33 format relocatable object files (\*.o) can be registered in a library. Absolute object files cannot be registered in a library. Note also that the maximum number of object modules that can be registered in one library is 256.
- (2) If, after specifying an object file that has the same name as that of an already registered module, you execute a command to add it to a library, a warning is generated but the file itself is added at the end of the library in the same name. Then, when the object file name is specified in a delete command, both modules are deleted; therefore, be careful.



## Chapter 16 Debugger

This chapter describes how to use the Debugger db33.

### 16.1 Features

The Debugger db33 (hereinafter called the "db33") is used to debug your program after reading an object file in the srf33 format that is generated by the linker or a ROM data file in Motorola S3 format.

It has the following features and functions:

- Various data can be referenced at the same time using multiple windows.
- Frequently used commands can be executed from tool bars and menus using a mouse.
- In addition to using the ICE33, ICD33 or Debug Monitor to debug your program, a software emulator function is available that allows program debugging on personal computers.
- Supports C and assembly source level debugging functions.
- In addition to continuous program execution, two types of C source/ assembler level single-stepping are supported.
- Hardware and software break functions, as well as a data break function that allows the memory access condition to be specified are available.
- Furthermore, the following useful functions are also provided:
  - A real-time display function for showing flags and watch addresses on-the-fly
  - A time display function for showing execution time by both duration and number of cycles
  - A trace function that allows data to be searched and saved
  - An automatic command execution function using a command file
  - A simulated I/O function that allows input/output to be evaluated in the debugger

### 16.2 Input/Output Files

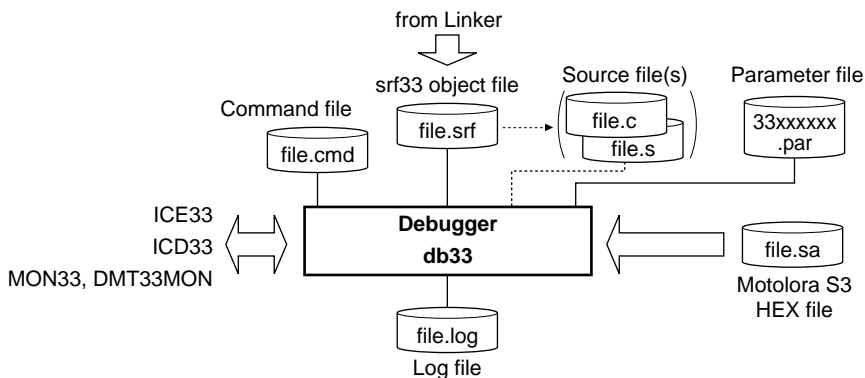


Fig. 16.2.1 Flowchart

#### 16.2.1 Input Files

##### Parameter file

File format: Text file

File name: <file name>.par

Description: This file contains memory information on each microcomputer model and is indispensable for starting the debugger. The [Par gen] button of the Work Bench wb33 allows you to create a parameter file that contains the basic parameters of the E0C33 Family.

The following files are read by the debugger according to command specification.

**Object file**

File format: Binary file in the srf33 format

File name: <file name>.srf

Description: This is an absolute object file generated by the Linker lk33. This file is read into the debugger by the lf command. By reading a file in the srf33 format that contains debug information, you can perform source level debugging.

**ROM data HEX file**

File format: HEX file in Motorola S3 format

File name: <file name>.sa

Description: This is a load image file of the ROM created by the Binary/HEX converter hex33, and is read into the debugger by the lh command. This file cannot be used for source level debugging because it has no debugging information, but you can use it to check the operation of final program data.

**Source file**

File format: Text file

File name: <file name>.c (C source), <file name>.s (assembly source)

Description: This is the source file of the above object file. It is read when the debugger performs source display.

**Command file**

File format: Text file

File name: <file name>.cmd

Description: This file contains a description of debug commands to be executed successively. By writing a series of frequently used commands in this file, you can save the time and labor required for entering commands from the keyboard. This command is read in and executed using the com or cmw command.

## 16.2.2 Output File

**Log file**

File format: Text file

File name: <file name>.log

Description: This file contains information on executed commands and execution results that are output to a file. Output of this file can be controlled by the log command.

## 16.3 Starting Method

---

### 16.3.1 Startup Format

#### General form of command line

**db33 ^ [<startup option>] ^ -p ^ <parameter file name>**

^ denotes a space.

[ ] indicates the possibility to omit.

#### Operation on work bench

Select the parameter file and options, then click the [DB33] button.

### 16.3.2 Startup Options

The db33 has eight startup options available.

#### **-p <file name>.par**

Function: Specifies a parameter file (essential setup item).

Specification on wb33: Select from the file list box in the execution window.

Explanation:

- Specify a parameter file (create a template by the wb33 and modify it according to the model to be developed).
- You cannot debug a program unless you have this file.

#### **-sim/-mon/-icd/-ice**

Function: Specifies debugger mode.

Specification on wb33: Select a mode from the combo box (ICD, MON, ICE, SIM).

Explanation:

- The db33 is started up in the specified mode.
  - sim Simulator mode
  - mon Debug monitor mode
  - icd ICD mode
  - ice ICE mode
- Unless this option is specified, the db33 is started in ICE mode.

#### **-c <file name>.cmd**

Function: Specifies a command file.

Specification on wb33: Check [db33 \*.cmd file] and select the command file from the list box.

Explanation:

- If you want a series of commands to be executed immediately after the db33 starts up, use this option to specify a command file that describes those commands.

#### **-w**

Function: Specifies a window at startup.

Specification on wb33: Check [1 win].

Explanation:

- If you specify this option, only the [Command] window opens when the db33 starts up. Specify this option when saving the log.
- Unless this option is specified, the [Command] window, [Source] window, and [Register] window open simultaneously.

#### **-comX**

Function: Specifies a communication port.

Specification on wb33: Select from a list box (com1, com2, com3 or com4).

Explanation:

- This option specifies the communication port through which a personal computer is communicated with the ICE33, ICD33 or MON33. Specify a port number (1–4) in the X part of this option. The port that can be used for this purpose varies among different personal computers.
- Unless this option is specified, the com1 port is used.

**-b <baud rate>**

- Function: Specifies a communication transmission rate.
- Specification on wb33: Select from a list box (4800, 9600, 19200, 38400, 57600 or 115200).
- Explanation:
- This option specifies the baud rate on the personal computer. For <baud rate>, select one from 4800, 9600, 19200, 38400, 57600 or 115200. The baud rate selections vary among different personal computers.
  - Unless this option is specified, the baud rate is set to 38400 bps in ICE or debug monitor mode and 115200 bps in ICD mode.
  - The baud rate on the ICE33/ICD33 is set using the DIP switch mounted on the ICE33/ICD33.

**-sf**

- Function: Specifies display with the small font.
- Specification on wb33: Check [small font].
- Explanation:
- The characters displayed in the db33 window are set to "Terminal 10pt".
  - Unless this option is specified, FixedSys 14pt is used.

**-lptX**

- Function: Specifies a parallel port.
- Specification on wb33: Select from a list box (no lpt, lpt1, or lpt2).
- Explanation:
- This option specifies the parallel port through which a personal computer is communicated with the ICE33 or ICD33. Specify a port number (1 or 2) in the X part of this option. The port that can be used for this purpose varies among different personal computers.
  - If this option is specified, the lf and lh commands download files at high speed using a parallel port.

Note: Do not use the COM and LPT ports for the db33 in other drivers and applications. Furthermore, make sure that the port has been enabled when using a note PC as some can disable COM ports.

When entering an option in a command line, make sure that there is at least one space before and after the option.

Example: c:\cc33\db33 -p 33104\_1.par -c startup.cmd -sf -com1 -b 38400

### 16.3.3 Startup Messages

When the db33 starts up, it outputs the following message in the [Command] window.

#### When starting up in ICE mode (with -ice option specified, default)

```

Debugger 33 Ver x.xx
Copyright (C) SEIKO EPSON CORP. 199x

Connecting with ICE ..... done
Reading parameter file ..... done
Initializing ..... done
Parameter file name      : xxxxxxxx.par
      Version           : xx
      Chip name         : xxxxx
BOOT address             : xxxxxxxx
PRC board version        : x.x
PRC board status         : xxxxxxxxxxxxxxxxx
ICE hardware version     : x.x
ICE software version     : x.x
DIAG test                : omitted
Emulation memory size    : xMB
Mapping.....done
>

```

When the db33 starts up in ICE mode, it first performs the tests and initializing operations described below then displays the above message.

1. Test the ICE33 connections.
2. Read a parameter file and check its contents.
3. Read and check the hardware version.
4. Initialize the ICE33.
5. Mapping
6. Reset the ICE33.
7. Initialize debugging information.

If an error or warning appears during any of the above processes, the db33 halts subsequent processing after displaying an error or warning message. In this case, quit the db33 temporarily, and eliminate the cause of error before restarting the db33.

Before starting up the db33 in ICE mode, please be sure to check the following:

- The ICE33 is connected to your computer with the designated RS-232C cable and parallel cable.
- Ports are specified correctly.
- The baud rate is set correctly.
- The PRC board is mounted correctly in place.
- The power of the ICE33 is turned on.
- The ICE33 and PRC board are not in a reset condition.
- The ICE33 is not in the free-run or self-diagnostic mode.
- The specified COM and LPT ports are not used in other applications.

### When starting up in ICD mode (with -icd option specified)

```

Debugger 33 Ver x.xx
Copyright (C) SEIKO EPSON CORP. 199x

Connecting with ICD ..... done
Reading parameter file ..... done
Initializing ..... done
Parameter file name      : xxxxxxxx.par
      Version           : xx
      Chip name        : xxxxx
ICD software version     : x.x
Target connection test   : OK
Mapping..... done
>

```

When the db33 starts up in ICD mode, it first performs the tests and initializing operations described below then displays the above message.

1. Test the ICD33 connections.
2. Read a parameter file and check its contents.
3. Initialize the ICD33.
4. Initialize debugging information.

If an error or warning appears during any of the above processes, the db33 halts subsequent processing after displaying an error or warning message. In this case, quit the db33 temporarily, and eliminate the cause of error before restarting the db33.

Before starting up the db33 in ICD mode, please be sure to check the following:

- The ICD33 is connected to your computer with the designated RS-232C cable and parallel cable.
- Ports are specified correctly.
- The baud rate is set correctly.
- The target board is connected correctly.
- The power of the ICD33 and target board are turned on.
- The specified COM and LPT ports are not used in other applications.

**When starting up in debug monitor mode (with -mon option specified)**

```

Debugger 33 Ver x.xx
Copyright (C) SEIKO EPSON CORP. 199x

Connecting with MON ..... done
Reading parameter file ..... done
Initializing ..... done
Parameter file name      : xxxxxxxx.par
          Version       : xx
          Chip name     : xxxxx
MON software version    : x.x
Mapping ..... done
>

```

When the db33 starts up in debug monitor mode, it first performs the tests described below then displays the above message.

1. Test the MON33 connections.
2. Read a parameter file and check its contents.
3. Initialize debugging information.

If an error or warning appears during any of the above processes, the db33 halts subsequent processing after displaying an error or warning message. In this case, quit the db33 temporarily, and eliminate the cause of error before restarting the db33.

Before starting up the db33 in debug monitor mode, please be sure to check the following:

- The DMT33MON board is connected to your computer with the designated RS-232C cable.
- The communication port is specified correctly.
- The baud rate is set correctly.
- The target board is connected correctly.
- The MON33 has been activated on the target board.

**When starting up in simulator mode (with -sim option specified)**

```

Debugger 33 Ver x.xx
Copyright (C) SEIKO EPSON CORP. 199x

Connecting with simulator ..... done
Reading parameter file ..... done
Initializing ..... done
Parameter file name      : xxxxxxxx.par
          Version       : xx
          Chip name     : xxxxx
BOOT address            : xxxxxxxx
Mapping ..... done
>

```

When started up in simulator mode, the db33 performs the following test and initialization before displaying the above messages.

1. Read a parameter file and check its contents.
2. Initialize the simulator.
3. Mapping
4. Reset the simulator.
5. Initialize debugging information.

If an error or warning is encountered during the above processing, the db33 halts subsequent processing after displaying an error or warning message. In this case, quit the db33 temporarily and create or select the correct parameter file before restarting the db33.

## Usage output

If no parameter file was specified or an option was not specified correctly, the db33 displays the following message concerning the usage:

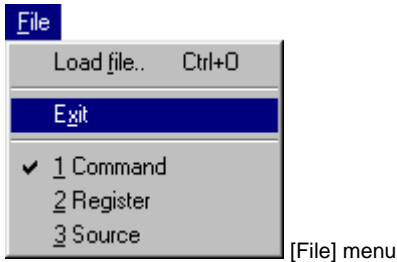
```
Debugger 33 Ver x.xx
Copyright (C) SEIKO EPSON CORP. 199x

- usage -
db33.exe -p parameter_file [-MODE] [-c command_file] [-w] [-comX] [-b baud_rate] [-sf] [lptX]
-p          ... parameter file
-MODE      ... select debugger mode
-sim       ... simulator mode
-mon       ... debug monitor mode
-icd       ... LCD mode
-ice       ... ICE mode
default    ... ICE mode
-c         ... command file
-w         ... open only command window
-comX (X:1-4) ... com port, default com1
-b         ... baud rate, 4800, 9600, 19200, 38400(default), 57600, 115200 [bps]
                default baud rate  ICE mode 38400
                default baud rate  LCD mode 115200
                default baud rate  debug monitor mode 38400
-sf        ... display with small font
-lptX (X:1-2) ... parallel port
>
```

When this message appears on the screen, temporarily quit the db33 and then start it up again correctly.

### 16.3.4 Method of Termination

To terminate the debugger, select [Exit] from the [File] menu.



You can also input the q command in the [Command] window to terminate the debugger.

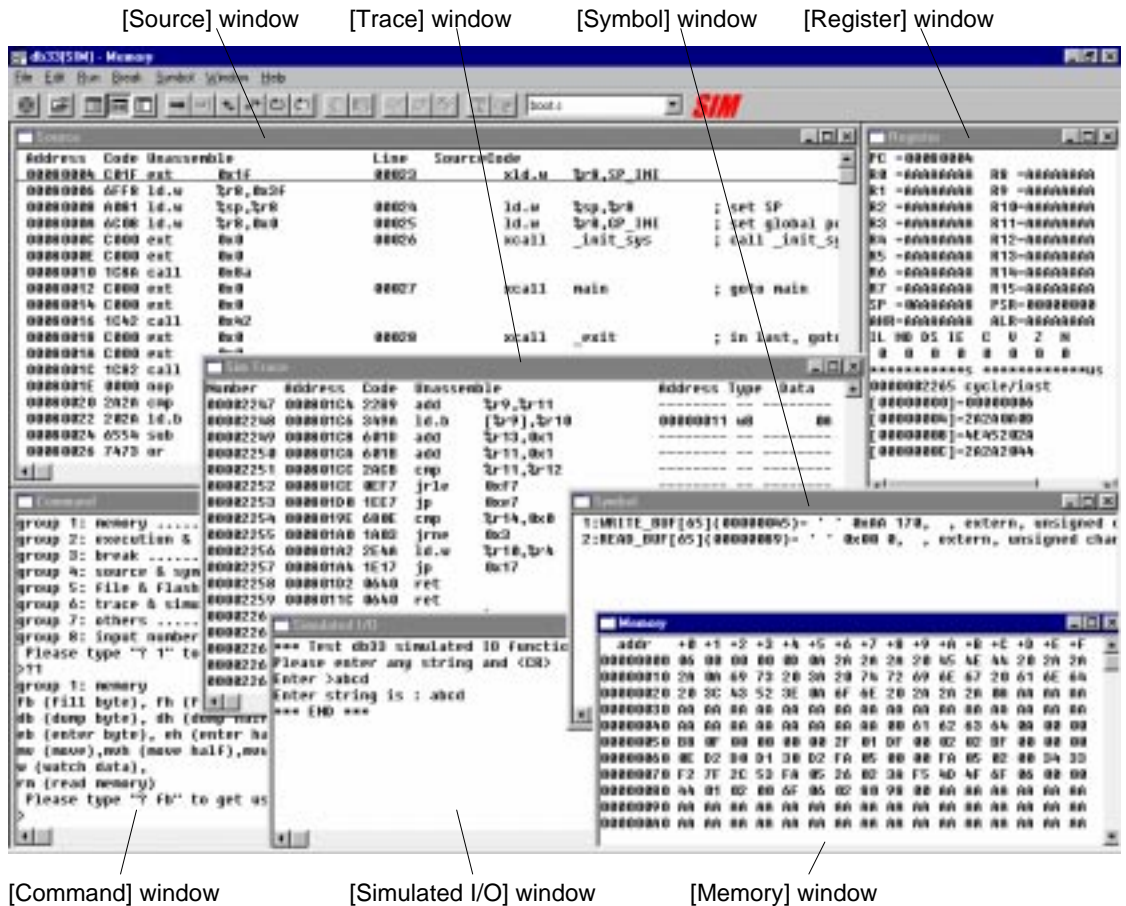
>q

## 16.4 Windows

This section describes the types of windows used by the db33.

### 16.4.1 Basic Structure of Window

The diagram below shows the window structure of the db33.



When the debugger starts up, the [Command], [Source], and [Register] windows are displayed by default. If you specify the `-w` option when starting up the debugger, only the [Command] window is displayed on the screen. Other windows are displayed by selecting the appropriate menu commands.



## Features common to all windows

### (1) Resizing and moving a window

Each window can be resized as needed by dragging the boundary of the window with the mouse. The [Minimize] and [Maximize] buttons work in the same way as in general Windows applications. Each window can be moved to your desired display position by dragging the window's title bar with the mouse. However, windows can only be resized and moved within the range of the application window.

All windows except [Command] can be closed as necessary.

### (2) Scrolling a window

All windows except for the [Register] window can be scrolled. Use one of the following three methods to scroll a window:

1. Click on an arrow button or enter an arrow key (cursor movement) to scroll a window one line at a time (except for the [Command] window).
2. Click on the scroll bar of a window to scroll it one page (current window size) at a time.
3. Drag the scroll bar handle of a window to move it to the desired area.

### (3) Other

The arrow keys can only be used in the [Command], [Source] and [Simulated I/O] windows. In all the windows, edit commands such as cut and paste cannot be used. However, the [Command] window supports a text-paste command.

## 16.4.2 [Command] Window

```

Command
>?
group 1: memory ..... fb,fh,fw / db,dh,dw / e
group 2: execution & register ..... g,s,n,rstc,rsth / int /
group 3: break ..... bp,bs,bc,bh,bhc,bd,bsq
group 4: source & symbol ..... u,sc,m / ss / sy,sa,sw
group 5: file & flash memory ..... lf,lh,lo / lf1,sf1,ef1
group 6: trace & simulated I/O ..... tm,td,ts,tf / stdin,sto
group 7: others ..... com,cmw,log / od,ct,ext
group 8: input number method ..... number,data,address,lir
Please type "? 1" to show group 1 or type "? fb" to get usage
>?1
group 1: memory
fb (fill byte), fh (fill half), fw (fill word),
db (dump byte), dh (dump half), dw (dump word),
eb (enter byte), eh (enter half), ew (enter word),
mv (move),
w (watch data)
Please type "? fb" to get usage of command "fb".
>

```

### Contents displayed

The [Command] window displays command execution results such as a db33 message, break information, map information, and trace search conditions. However, some command execution results are displayed in the other window. The contents of these execution results are displayed when their corresponding windows are open. If the corresponding window is closed, the execution result is displayed in the [Command] window. Only the contents displayed in the [Command] window are output to a log file. (Refer to the explanation of the log command.)

For the displayed command execution results, refer to the explanation of each command.

### Operation

#### Opening and closing the window

The [Command] window always opens when the db33 starts up. It cannot be closed, but can be minimized.

#### Entering commands

You can enter and execute all the debug commands in the [Command] window.

When the prompt ">" appears at the bottom line of the [Command] window, the system accepts a command entered from the keyboard.

If some other window is selected, click on the [Command] window. A cursor will blink behind the prompt, indicating that you are ready to input a command.

#### Command history

The ↑ and ↓ keys can be used to display up to 20 of the latest executed commands and the displayed command can be executed again by pressing [Enter].

See Section 16.7.1, "Entering Commands from Keyboard", for entering command.

### Restrictions

- The maximum number of characters that can be displayed or input in one line of the [Command] window is 255, including null characters.
- The maximum number of lines that can be scrolled is 256. The contents displayed or input preceding these lines are deleted.

### 16.4.3 [Source] Window

#### Contents displayed

##### Program code display

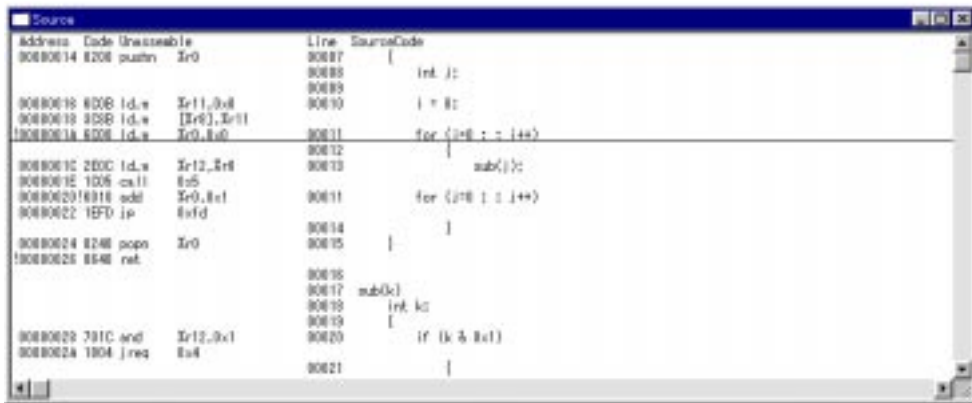
The [Source] window displays the contents of the program loaded. You can select one of the following three display modes:

##### 1. Mixed display (selected by [Mix] button or m command)



[Mix] button

In this mode, the window displays the addresses, codes, disassembled contents, and corresponding source line numbers and source statements.

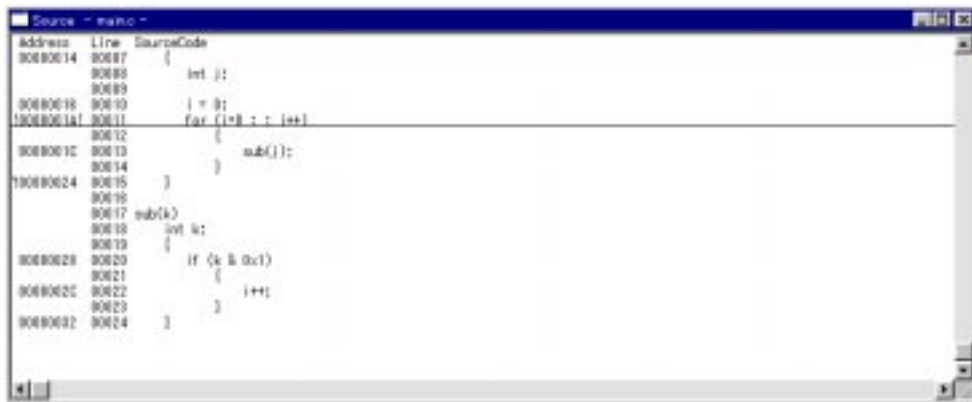


##### 2. Source display (selected by [Source] button or sc command)



[Source] button

In this mode, the window displays the addresses, corresponding source line numbers and source statements. The displayed address is the address of the first instruction in the source. Only the source that includes the current PC position is displayed.



### 3. Disassemble display (selected by [Unassemble] button or u command)



[Unassemble] button

In this mode, the window displays the addresses, codes, and disassembled contents. This format is selected when the db33 starts up.

```

Source
Address Code Unassemble
0080014 8208 pushl %r0
0080016 8209 id.w [%r1],%r8
0080018 820B id.w [%r6],%r11
008001A 820C id.w %r0,%r0
008001C 220C id.w %r12,%r8
008001E 1006 call %5
0080020 1618 addl %r0,%r1
0080022 1EFD jmp %rd
0080024 8248 popl %r0
0080026 8E48 ret
0080028 781C andl %r12,%r1
008002A 1804 imul %4
008002C 385A id.w [%r16],%r1
008002E 8B1A addl %r16,%r1
0080030 325A id.w [%r6],%r10
0080032 8E48 ret
0080034 FFFF see
0080036 FFFF see
0080038 FFFF see
008003A FFFF see
008003C FFFF see
  
```

#### Display of source line numbers and source statements

The source line numbers and source statements can only be displayed when the srf33 object file including debug information for source display is loaded. Furthermore, the source statements that are actually displayed from this file are those which have had the -g option specified by the C compiler or preprocessor.

The db33 displays the source lines corresponding to the address of each code and the source statements bearing information on the source lines up to the immediately preceding code. Therefore, uncoded source statements written after the end of code are not displayed.

#### Underlined display (current PC)

The underlined line indicates the line to be executed next (the line of the current PC address). In the mixed display and disassemble display modes, the entire line is underlined. In the source display mode, the entire line is underlined only when the current PC falls upon the address of the first instruction in the source. Otherwise, the address part is not underlined.

#### ! and ? (breakpoint)

The "!" displayed immediately before or after an address indicates that the address is a breakpoint. The "!" displayed immediately before an address indicates a software break and the "!" displayed immediately after an address indicates a hardware break.

If a breakpoint is set somewhere other than the first instruction address displayed in the source display mode, it is marked with "?" in place of "!".

#### Display of unused areas

The unmapped areas in the memory specified by a parameter file are marked with words "no map".

#### Others

If no source code corresponding to the address is loaded in the mixed display mode, "no source" is displayed in the source part.

## Operation

#### Opening and closing the window

The [Source] window opens simultaneously when you start up the db33. However, if the -w option is specified, this window is not opened automatically.

The [Source] window can be opened by using the [Source] command on the [Window] menu or can be closed by clicking on the close box.

**Changing display format**

Execute the m, sc or u command as explained above. (Contents will be displayed beginning with the current PC address.) Or click on each button. (The display will be switched over from the currently displayed address or line number to another.)

**Updating display contents**

The contents displayed in the [Source] window are updated automatically in the following cases:

- When a program is loaded (lf, lh or lfl command)
- When a program is executed (g, s or n command)
- When display format is changed (m, sc or u command)
- When the CPU is reset (rstc or rsth command)

When contents are redisplayed, the start position is the address indicated by the current PC. When the CPU is reset, contents are displayed beginning with the boot address.

The contents displayed in the [Source] window are not updated for any other reason even when the contents of program memory are modified by a command (e, f or mv). To bring up the last content on the screen, perform one of the following operations:

1. Execute one of the m, sc or u commands.
2. Click on the vertical scroll bar.

**Setting breakpoints**

Software breakpoints, hardware breakpoints and a temporary breakpoint can be specified from the window. Position the cursor at an address line where you want a breakpoint to be set (cannot be set at a source-only line). Then...

- Click on the [Soft PC break] or [Hard PC break] button. A software or hardware breakpoint will be set at that address. The breakpoint here can be cleared by performing the same operation at this address as you did above.



[Soft PC break] button



[Hard PC break] button

- When you click on the [Go to] button, the program starts running beginning with the current PC and breaks before executing the line at which the cursor is placed.



[Go to] button

**Displaying and registering symbols**

Symbols whose contents you want to be displayed and those registered in the [Symbol] window can be selected from the [Source] window.

When in the mix display or source display mode, position the cursor in or immediately before or after a symbol you want to choose. (The symbol name is taken including "\*", ".", ">", "[", and "]".) Then...

- Click on the [Symbol watch] button. The content of the symbol will be displayed in the [Command] window.



[Symbol watch] button (The [Watch] command on the [Symbol] menu can also be used.)

- When you click on the [Symbol add] button, the symbol is registered in the [Symbol] window.

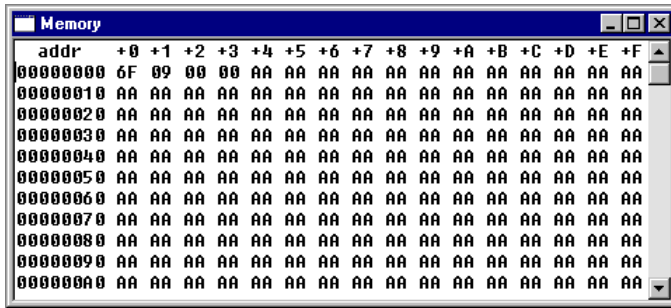


[Symbol add] button (The [Add] command on the [Symbol] menu can also be used.)

**Restrictions**

- The maximum number of characters that can be displayed in one line of the [Source] window is 255 (including the first ! or ?).
- The [Source] window can be used for display only; it cannot be used to input anything from the keyboard or edit the displayed contents.

## 16.4.4 [Memory] Window



### Contents displayed

The [Memory] window displays a dumped result the entire memory area in hexadecimal. The content of any desired memory location can be viewed by scrolling the display. The contents of unmapped memory addresses in each model are marked with asterisks (\*).

### Display format

Data can be displayed in units of bytes, half words, or words as set by the db, dh, or dw commands. Displaying in half word or word units are performed according to the endian format of the area set in the parameter file.

Examples: **Display in byte units (db command)**

```
addr  +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +A +B +C +D +E +F
00000000 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA
:
:
```

**Display in half word units (dh command)**

```
addr  +0 +2 +4 +6 +8 +A +C +E
00000000 AAAA AAAA AAAA AAAA AAAA AAAA AAAA AAAA
:
:
```

**Display in word units (dw command)**

```
addr  +0 +4 +8 +C
00000000 AAAAAAAAAA AAAAAAAAAA AAAAAAAAAA AAAAAAAAAA
:
:
```

### Operation

#### Opening and closing the window

The [Memory] window is not opened automatically when the debugger starts up. It can, however, be opened by choosing the [Memory] command from the [Window] menu.

To close the window, click on the close box of the window.

#### Display format and changing display start address

Execute the db, dh or dw command as explained earlier. These commands can also be used to specify the display start address.

#### Updating display contents

Even if the memory contents are modified by a command (e, f, or mv), the contents displayed in the [Memory] window are not updated at that point in time. The contents displayed in the [Memory] window are cleared when a file is loaded (lf, lh or lfl command).

To bring up the latest content on the screen, perform one of the following operations:

1. Execute one of the db, dh or dw commands.
2. Click on the vertical scroll bar.

The [Memory] window while the program is being executed (g, s, or n) is updated when a break occurs or every step. (default)

### Restriction

The [Memory] window is used for display only, and cannot be used to input anything from the keyboard or edit the displayed contents.

## 16.4.5 [Register] Window



### Contents displayed

The [Register] window displays the following contents:

#### Register contents

The contents of general-purpose registers (R0 to R15) and special registers (PC, SP, PSR, AHR, and ALR) are displayed. The contents of the PSR register are displayed individually for each flag.

#### Execution counter

The total number of cycles executed and a total execution time since the execution counter is actuated after a reset are displayed by adding up the count values. For details about the execution counter, refer to Section 16.8.5, "Executing Program".

### Watch memory data

The debugger allows you to specify four memory addresses and monitor the contents of those memory locations. The contents of these four watch data addresses (4 bytes each from a specified address) are displayed in the [Register] window. The default watch data addresses set at startup time are addresses 0, 4, 8, and C. The data is displayed in the endian format specified by the parameter file. If watch data addresses are specified using symbols (w command), symbol names also are displayed after data.

### ICE CPU status (in ICE mode)

In the ICE mode, the CPU status in the ICE33 is displayed on the right side of the PC display data. The display contents are listed below. The CPU status is not displayed in other modes.

- SLP** Indicates that the CPU is in the SLEEP mode.
- HLT** Indicates that the CPU is in the HALT mode.
- OSC3** Indicates that the CPU is operating with the high-speed clock (OSC3).
- OSC1** Indicates that the CPU is operating with the low-speed clock (OSC1).
- RESET** Indicates that the CPU is in a reset state.
- NO VCC** Indicates that no supply voltage is fed to the CPU.
- NO CLK** Indicates that no operating clock is fed to the CPU.
- WAIT** Indicates that the CPU is in a wait state.

### Updating display contents

The contents displayed in the [Register] window are updated automatically in the following cases:

- When register contents are dumped (rd command)
- When the CPU is reset (rstc or rsth command)
- When register values are changed (rs command)
- When watch data addresses are set (w command)
- When the execution counter display mode is changed (md command)
- When option data or flash memory contents are loaded (lo or lf command)
- After program execution is completed (g, s or n command)

The numeric value display part is left blank during continuous program execution (g command) in any mode other than on-the-fly mode or when single-stepping the program (s or n command) in the final step mode.

If the program is executed (g command) after turning the on-the-fly function on in the ICE mode, the display contents of the PC, PSR flag, and watch data are updated in real time every one to 0.1 second. All other contents are left blank until program execution breaks.

## Operation

### Opening and closing the window

The [Register] window opens automatically when you start up the db33. However, this window is not opened automatically if you specify the -w option at startup.

The [Register] window can be opened by using the [Register] command on the [Window] menu, and can be closed by clicking on the close box of the window.

## Restriction

The [Register] window is used for display only, and cannot be used to input anything from the keyboard or edit the displayed contents.

## 16.4.6 [Trace] Window

### Contents displayed

The [Trace] window displays the trace data that indicates the execution result of each instruction.

Refer to Section 16.8.7, "Trace Functions", for details of the trace mode and trace information.

Note that the debug monitor mode does not support the trace function.

#### (1) ICE mode

Cycle#	Address	Code	Unassemble	Address	Data	Clk	Type	TRC	File	Line	SourceCode
00061	000012E	5900	2	000012E	5900	2	DatM W I/O				
00060	000012E	8000	2	000012E	8000	2	DatM W I/O				
00059	0000064	E024	ext	0000064	E024	1	Inst # S00H		(area.s)	00060	xld.h [TTR],%r0
00058	0000065	E134	ext	0000065	E134	1	Inst # S00H		(area.s)		
00057	0000066	2000	ld.h	0000066	[%r0],%r0	1	Inst # S00H		(area.s)	00061	ld %r0
00056	0000066	0400	ld	0000066	%r0	1	Inst # S00H		(area.s)		
00055	000006C	0000	nap	000006C	0000	2	DatM W I/O		(area.s)	00062	nap
00054	000006C	0000	nap	000006C	0000	1	Inst # S00H		(area.s)		
00053	000006C	0000	nap	000006C	0000	2	UseR W S00H		(area.s)	00067	nap
00052	000006C	0000	nap	000006C	0000	3	UseR W S00H		(area.s)		
00051	000006C	0000	nap	000006C	0000	4	Inst # S00H		(area.s)	00068	retl
00050	000006C	0000	retl	000006C	0000	5	Inst # S00H		(area.s)	00069	retl
00049	000006C	0000	retl	000006C	0000	3	Inst # S00H		(area.s)	00070	retl
00048	000006C	0000	retl	000006C	0000	3	Inst # S00H		(area.s)	00071	retl
00047	000006C	0000	retl	000006C	0000	4	Inst # S00H		(area.s)	00072	retl
00046	000006C	0000	retl	000006C	0000	1	Inst # S00H		(area.s)	00073	retl
00045	000006C	0000	retl	000006C	0000	1	Inst # S00H		(area.s)	00074	retl
00044	000006C	0000	retl	000006C	0000	1	Inst # S00H		(area.s)	00075	retl

In ICE mode, the trace result can be displayed for up to 32,768 cycles by reading it from the ICE33 trace memory. Two methods of trace are supported: normal and single delay.

The following lists the trace contents:

- Executed cycle number
- Executed address, code, disassembled content
- Memory access contents (address, R/W and data)
- Input to TRCIN pin
- Bus operation type
- Source codes (tm command option)

#### (2) ICD mode

Cycle	Address	Code	Unassemble	Clk	Method	File	Line	SourceCode
000017	0602000	DFF8	ext	0602000	DFF8	DPC		
000016	0602000	DFF9	ext	0602000	DFF9	DPC		
000015	0602000	DFF9	ext	0602000	DFF9	DPC		
000014	0602000	1C44	call	0602000	1C44	DPC		
000013	0602000	6C0C	ld.u	0602000	%r12,%r0	DPC (sys.c)	00001	ldBytes = 0; /* no read now */
000012	0602000	C80C	ext	0602000	%r0	DPC (sys.c)	00006	for (;;)
000011	0602000	C80F	ld.u	0602000	%r15,%r0	DPC		
000010	0602000	6C14	ld.u	0602000	%r14,%r1	DPC		
000009	0602000	680E	cmp	0602000	%r14,%r0	DPC (sys.c)	00001	if (iReadBytes == 0) /* if require
000008	0602000	1000	jmpq	0602000	%r0	DPC		
000007	0602000	C300	ext	0602000	%r00	DPC (sys.c)	00006	if (READ_EOF == 1)
000006	0602000	C809	ext	0602000	%r0	DPC		
000005	0602000	2405	ld.u	0602000	%r5, [%r6]	DPC		
000004	0602000	6815	cmp	0602000	%r5,%r1	DPC		
000003	0602000	1000	jmpq	0602000	%r0	DPC		
000002	0602000	2408	ld.u	0602000	%r11, [%r15]	DPC (sys.c)	00101	iSize = READ_BUF[0];
000001	0602000	6800	cmp	0602000	%r11,%r0	DPC (sys.c)	00102	if (iSize > 0)
000000	0602000	0E14	jmpq	0602000	%r14	DPC		

In ICD mode, the trace result can be displayed by reading it from the ICD33 trace memory. Two methods of trace are supported: all trace mode and area trace mode.

The following lists the trace contents:

- Executed cycle number
- Executed address, code, disassembled content
- Number of clocks
- PC-analyze method
- Source line number and code



### (3) Simulator mode

Number	Address	Code	Disassemble	Address	Type	Data	File	Line	SourceCode
00000009	00000026	687D	add	%r0,%r1			(main.c)	00011	for (j=0 ; ; j++)
00000010	00000028	1EFD	jp	%r0			(main.c)	00013	sub(j);
00000011	00000022	250C	ld.u	%r12,%r0			(main.c)	00009	
00000012	00000024	1C05	call	%r5		00007F4 w 00000026	(main.c)	00009	if (k & %r1)
00000013	0000002E	781C	and	%r12,%r1			(main.c)	00009	
00000014	00000030	1806	jreq	%a			(main.c)	00004	}
00000015	00000034	06A9	ret			00007F4 rW 00000026	(main.c)	00011	for (j=0 ; ; j++)
00000016	00000026	687D	add	%r0,%r1			(main.c)	00011	for (j=0 ; ; j++)
00000017	00000028	1EFD	jp	%r0			(main.c)	00013	sub(j);
00000018	00000022	250C	ld.u	%r12,%r0			(main.c)	00009	
00000019	00000024	1C05	call	%r5		00007F4 w 00000026	(main.c)	00009	if (k & %r1)
00000020	0000002E	781C	and	%r12,%r1			(main.c)	00009	
00000021	00000030	1806	jreq	%a			(main.c)	00004	}
00000022	00000032	C809	ext	%0			(main.c)	00002	i++;
00000023	00000034	C809	ext	%0					
00000024	00000036	6C09	ld.u	%r9,%r0					
00000025	00000038	3006	ld.u	%r10,%r9		00000000 rW 00000002			
00000026	0000003A	687D	add	%r0,%r1					
00000027	0000003C	C809	ext	%0					

In simulator mode, when the trace function is turned on (tm command), all of the subsequent program execution is displayed as traced by the debugger (except for file output).

The following lists the trace contents:

- Number of executed instructions
- Executed address, code, and disassembled content
- Data memory access contents (address, R/W, and data)
- Source codes (tm command option)
- Register contents (tm command option)

## Operation

### Opening and closing the window

The [Trace] window does not open automatically when the debugger starts up. The [Trace] window can be opened by using the [Trace] command on the [Window] menu, and can be closed by clicking on the close box of the window.

### Updating display

#### ICE mode

The contents of the [Trace] window are cleared when you execute the target program.

To display the latest contents of this window, execute the td command or temporarily close the [Trace] window and then reopen it.

#### ICD mode

The contents of the [Trace] window are cleared when you execute the target program.

To display the latest trace information after the execution has suspended, execute the td command or temporarily close the [Trace] window and then reopen it.

Clicking the [Display trace] button while the program is being executed suspends tracing and displays the trace data in the trace memory to the [Trace] window. After that, clicking the [Resume trace] button clears the [Trace] window and resumes tracing.



[Display trace] button



[Resume trace] button

### Simulator mode

When the trace function is turned on (tm command), trace results are successively displayed as you execute the program (g, s, or n command) and the display is halted simultaneously when program execution stops.

If the trace function is turned off (default), the display will not be updated even when you execute a program.

## Restrictions

- The [Trace] window is used for display only, and cannot be used to input anything from the keyboard or edit the displayed contents.
- In the simulator mode, the contents of this window can be scrolled for up to 255 lines. In the ICE/ICD mode, all the contents of the ICE33/ICD33 trace memory can be displayed.

## 16.4.7 [Symbol] Window

```

Symbol
1:UDIV8(00080014)= 0x2EC52ED4, , extern, null
2:DIV8(00080036)= 0x2EC52ED4, , extern, null
3:UDIV16(0008005C)= 0x2EC52ED4, , extern, null
4:DIV16(00080090)= 0x2EC52ED4, , extern, null
5:SCAN0_32(0008012C)= 0x12108ACA, , extern, null
6:SCAN1_32(00080150)= 0x12108ECA, , extern, null
7:mirror32(000800C8)= 0x074098C4, , extern, null
8:bRemainder(000007F5)= ' ', 0xFF -1, sample.c/main, auto, char
9:ubRemainder(000007F4)= ' ', 0x1 1, sample.c/main, auto, unsigned char
10:uwRemainder(000007F6)= 0x0001 1, sample.c/main, auto, unsigned short
11:wRemainder(000007F8)= 0x0001 1, sample.c/main, auto, short

```

### Contents displayed

This window displays the contents of the symbols registered in the watch table. Up to 99 symbols can be registered. For the display format of symbols, refer to the explanation of the sa command.

### Updating display

The contents displayed in the [Symbol] window are updated automatically in the following cases:

- After program execution (g, s, or n command) is completed  
Contents are not updated during single-stepping (s or n command) in the final step display mode.
- When symbols are registered (sa command)  
Contents are redisplayed when an already registered symbol is specified by the sa command.
- When symbols are deleted (sd command)

The contents displayed in the [Symbol] window are not updated even when the contents of symbol addresses are modified by a command (e, f, or mv).

### Operation

#### Opening and closing the window

The [Symbol] window does not open automatically when the debugger starts up. The [Symbol] window can be opened by using the [Symbol] command on the [Window] menu, and can be closed by clicking on the close box of the window.

#### Registering symbols for display

To display the content of a symbol in the [Symbol] window, you must first register the symbol. Follow one of the two methods below to register a symbol:

1. Execute the sa command.
2. Change the [Source] window's display mode to "Mix" or "Source" and position the cursor in or immediately before or after a symbol name you want to be registered. Then click on the [Symbol add] button.



[Symbol add] button

(The [Add] command on the [Symbol] menu can also be used.)

#### Deleting registered symbols

When a symbol does not need to be watched, delete it from the window by following one of the two methods described below:

1. Execute the sd command.
2. Place the cursor at the symbol line in the [Symbol] window that you want deleted. Then click on the [Symbol delete] button.



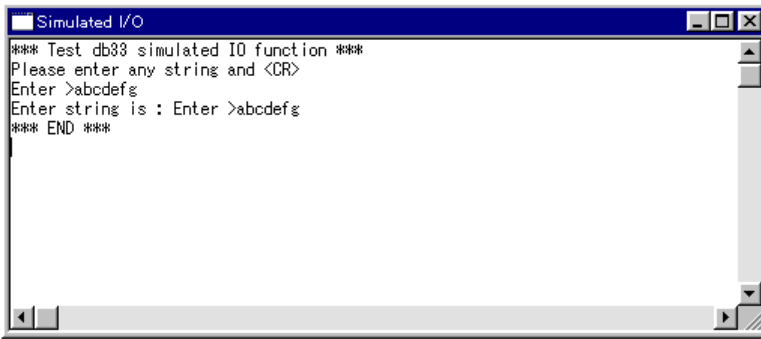
[Symbol delete] button

(The [Delete] command on the [Symbol] menu can also be used.)

### Note

The debugger reads the contents of symbols from the target in byte units and re-arranges the read data to byte, half-word or word data according to the symbol size before displaying. Note that data cannot be displayed if the big/little endian settings are different between the BCU on the target and the parameter file read in the debugger.

## 16.4.8 [Simulated I/O] Window



### Contents displayed

Using the simulated I/O function, this window displays the contents that are input from stdin and those that are output to stdout.

### Operation

#### Opening and closing the window

The [Simulated I/O] window is not opened when the debugger starts up. Choose the [StdIO] command from the [Window] menu to open it.

This window opens automatically in the following cases:

- When the input source of the data is set in the window by the stdin command and a breakpoint specified by the program is encountered
- When the output destination of data is set in the window by the stdout command and a breakpoint specified by the program is encountered

To close the window, click on its close box.

#### Entering data

The data taken in from stdin can be input from this window. For details, refer to Section 16.8.8, "Simulated I/O".

### Restriction

The maximum number of lines that can be displayed in the [Simulated I/O] window is 256.

## 16.5 Tool Bar

This section outlines the tool bar available with the db33.

### 16.5.1 Tool Bar Structure

The db33 has 14 buttons and a combo box in its tool bar, each one assigned to a frequently used command.



### 16.5.2 [Key break] Button



[Key break]

This button forcibly breaks execution of the target program. This function can be used to cause the program to break when the CPU is placed in standby (HALT or SLEEP) mode as you execute the target program or when the program has fallen into an endless loop.

Note that the debug monitor mode does not support this function.

### 16.5.3 [Load file] Button



[Load file]

This button reads an object file in the srf33 format into the debugger. It performs the same function when the lf command is executed. When you click [Load file], a dialog box for opening a file appears on the screen, allowing you to choose the file you want to be debugged.

### 16.5.4 [Source], [Mix] and [Unassemble] Buttons



[Source]

This button switches the display of the [Source] window to the source mode.



[Mix]

This button switches the display of the [Source] window to the mix mode (disassemble & source).



[Unassemble]

This button switches the display of the [Source] window to the disassemble mode.

The display is switched between these modes based on the source line number or address shown at the top of the [Source] window. These buttons are valid only when the [Source] window is open.

### 16.5.5 [Go], [Go to], [Step], [Next], [Reset cold] and [Reset hot] Buttons



[Go]

This button executes the target program beginning with the address indicated by the current PC. It performs the same function when the g command is executed.



[Go to]

This button executes the target program from the address indicated by the current PC to the cursor position in the [Source] window (i.e. the address of that line). It performs the same function when the g <address> command is executed.

To select this button, the [Source] window must be open and you must have clicked on the address line where you want the program to break. Selecting a break address by clicking on the address line is valid for only the lines that have actual code, and is invalid for the source-only lines.



[Step]

This button executes one instruction step of the target program beginning with the address indicated by the current PC. It performs the same function when the s command is executed.



[Next]

This button executes one instruction step of the target program beginning with the address indicated by the current PC. Functions and subroutines are executed as one step. This button performs the same function when the n command is executed.



[Reset cold]

This button cold-resets the CPU. It performs the same function when the `rstc` command is executed.



[Reset hot]

This button hot-resets the CPU. It performs the same function when the `rsth` command is executed.

### 16.5.6 [Soft PC break] and [Hard PC break] Buttons



[Soft PC break]

Use this button to set and reset a software breakpoint at the address where the cursor is located in the [Source] window. (See Section 16.8.6, "Break Functions" for details.) This function is valid only when the [Source] window is open. Note that selecting a break address by clicking on the address line is valid for only the lines that have actual code and is invalid for the source-only lines.



[Hard PC break]

Use this button to set and reset a hardware breakpoint at the address where the cursor is located in the [Source] window. (See Section 16.8.6, "Break Functions" for details.) This function is valid only when the [Source] window is open. Note that selecting a break address by clicking on the address line is valid for only the lines that have actual code and is invalid for the source-only lines.

### 16.5.7 [Symbol watch], [Symbol add] and [Symbol delete] Buttons



[Symbol watch]

The content of the symbol at the cursor position of the [Source] window is displayed in the [Command] window. It performs the same function when the `sw` command is executed. This function is valid only when the [Source] window is open.



[Symbol add]

The symbol at the cursor position of the [Source] window is registered in the [Symbol] window. It performs the same function when the `ss` command is executed. This function is valid only when the [Source] window is open.



[Symbol delete]

The symbol on the line where the cursor is positioned in the [Symbol] window is deleted from the [Symbol] window. It performs the same function when the `sd` command is executed. This function is valid only when the [Symbol] window is open.

### Obtaining symbol names

The [Symbol watch] and [Symbol add] buttons get the character string which is pointed with the cursor in the [Source] window and use it as a symbol name. However, the cursor must be placed in or immediately before or after the symbol name, and the character string must consist of the following characters only. Other characters including blank characters (space, etc.) are regarded as a delimiter of the character string.

`a-z A-Z 0-9 * . -> [ ]`

Example: "`↓`" indicates cursor position.

Obtained character string (symbol name)

<code>a↓ = b;</code>	<code>a</code>
<code>a = b*c↓;</code>	<code>c</code>
<code>a = b+*c↓;</code>	<code>*c</code>
<code>stru↓ct1-&gt;a = b;</code>	<code>struct1-&gt;a</code>
<code>stru↓ct1 -&gt; a = b;</code>	<code>struct1</code>

## 16.5.8 [Display trace] and [Resume trace] Buttons



[Display trace]

Clicking this button while the program is being executed in ICD mode suspends tracing and displays the trace data in the ICD33 trace memory to the [Trace] window.



[Resume trace]

Clicking this button while the program is being executed in ICD mode resumes tracing.

## 16.5.9 [Select source] Combo Box



[Select source]

This box is used to select the source file name of the program to be displayed in the [Source] window. The source file names listed in this box are obtained from the debugging information in the loaded object file. Therefore, this function can only be used when an srf33 object file with source information is loaded, otherwise this box displays "no source" and does not allow the selection. This box also deactivates when the [Source] window is closed or while command parameters are input in the guidance mode.

Source files can be selected regardless of the display mode for the [Source] window.

When a source file name is selected in this box, the [Source] window displays the codes from the top of the file.

This operation is not regarded as a command execution, so it does not appear in the command history or the log file.

When the loaded object file contains two or more sources, the source file names are listed in alphabetical order. If a multiple source file is included in some locations of the object, the file name appears only once.

This box is used only for selection and does not allow entering file names.

## 16.6 Menu

This section outlines the menu bar available with the db33.

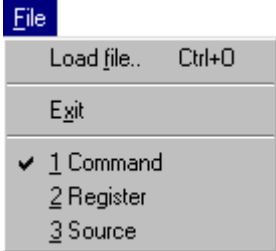
### 16.6.1 Menu Structure

The db33 menu bar has six menu items, each including frequently-used commands.

File Edit Run Break Symbol Window Help

Each menu command can be selected from the keyboard (by entering the menu and underlined command characters after pressing the [Alt] key), as well as selected with the mouse.

### 16.6.2 [File] Menu



#### [Load File...]

This menu command performs the same function as the [Load file] button on the tool bar. The keyboard shortcut [Ctrl]+[O] is also allowed for the selection.

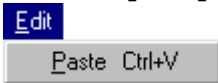
#### [Exit]

Terminates the db33.

#### Window list

Lists the currently opened window names (including minimized windows). The check mark indicates the active window. When a window name is selected in this list, the selected window will become active.

### 16.6.3 [Edit] Menu



#### [Paste]

This paste command is valid only for the [Command] window. The command copied from log or other files can be executed after pasting it to the [Command] window using this menu command. The keyboard shortcut [Ctrl]+[V] is also allowed.

### 16.6.4 [Run] Menu



#### [Go]

This menu command performs the same function as the [Go] button on the tool bar.

#### [Go to]

This menu command performs the same function as the [Go to] button on the tool bar.

#### [Step]

This menu command performs the same function as the [Step] button on the tool bar.

#### [Next]

This menu command performs the same function as the [Next] button on the tool bar.

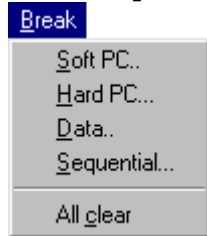
#### [Reset Cold]

This menu command performs the same function as the [Reset cold] button on the tool bar.

#### [Reset Hot]

This menu command hot-resets the CPU. It performs the same function when the rsth command is executed.

## 16.6.5 [Break] Menu



### [Soft PC...]

This menu command sets and resets software PC break addresses. It performs the same function as executing the bp command. When this command is selected, a dialog box appears on the screen, allowing you to set break addresses in up to 16 locations.

### [Hard PC...]

This menu command sets and resets a hardware PC break address. It performs the same function as executing the bh/bh2 command. When this command is selected, a dialog box appears on the screen, allowing you to set a hardware PC break address.

### [Data...]

This menu command sets data break conditions. It performs the same function as executing the bd command. When this command is selected, a dialog box appears on the screen, allowing you to set break conditions.

### [Sequential...]

This menu command sets sequential break conditions. It performs the same function as executing the bsq command. When this command is selected, a dialog box appears on the screen, allowing you to set break conditions. This menu command is valid only in the ICE mode.

### [All clear]

This menu command clears all the break conditions. It performs the same function as executing the bac command.

Refer to Section 16.8.6, "Break Functions", for details of each break function.

## 16.6.6 [Symbol] Menu



### [Watch]

This menu command performs the same function as the [Symbol watch] button on the tool bar. This function is valid only when the [Source] window is open.

### [Add]

This menu command performs the same function as the [Symbol add] button on the tool bar. This function is valid only when the [Source] window is open.

### [Delete]

This menu command performs the same function as the [Symbol delete] button on the tool bar. This function is valid only when the [Symbol] window is open.

## 16.6.7 [Window] Menu



### [Command]

This menu command activates the [Command] window.

### [Source]

This menu command opens the [Source] window. This menu command is invalid when the [Source] window is already open.

### [Memory]

This menu command opens the [Memory] window. This menu command is invalid when the [Memory] window is already open.

### [Register]

This menu command opens the [Register] window. This menu command is invalid when the [Register] window is already open.

### [Trace]

This menu command opens the [Trace] window. This menu command is invalid when the [Trace] window is already open.



**[Symbol]**

This menu command opens the [Symbol] window. This menu command is invalid when the [Symbol] window is already open.

**[StdIO]**

This menu command opens the [Simulated I/O] window. This menu command is invalid when the [Simulated I/O] window is already open.

### 16.6.8 [Help] Menu

**Help**

About db33...

**[About db33...]**

This menu command displays the version of the db33. By clicking on the [OK] button, the dialog box will close, returning to the debugger window.



## 16.7 Method for Executing Commands

All debug functions can be performed by executing debug commands. This section describes how to execute these commands. Refer to the description of each command for command parameters and other details.

To execute a debug command, activate the [Command] window and input the command from the keyboard. You can use the menu and tool bar to execute frequently-used commands.

### 16.7.1 Entering Commands from Keyboard

Select the [Command] window (by clicking somewhere on the [Command] window). When the prompt ">" appears on the last line in this window and a cursor is blinking behind it, the system is ready to accept a command from the keyboard. Input a debug command at the prompt position. The commands are not case-sensitive; they can be input in either uppercase or lowercase.

#### General command input format

>command [ parameter [parameter ... parameter ] ] ↵

- A space is required between a command and parameter.
- A space, comma (,) or tab is required between parameters.

Use the arrow keys (←, →), [Back Space] key, or [Delete] key to correct erroneous input.

When you hit the [Enter] key after entering a command, the system executes that command. (If a command you are entering is accompanied by guidance, the command is executed when you input the necessary data according to the displayed guidance.)

Input example:

>g↵ (Only a command is input.)

>com test. cmd↵ (A command and parameter are input.)

#### Command input accompanied by guidance

For commands that cannot be executed unless you specify a parameter or the commands that modify the existing data, a guidance mode is entered when only a command is input. In this mode, the system brings up a guidance field, so you input a parameter there.

Input example:

>com↵

File name ? : test. cmd↵ ...Input data according to the guidance (underlined part).

>

#### Commands requiring parameter input as a precondition

The com command shown in the above example reads a command file into the debugger. Commands like this that require an entered parameter as a precondition are not executed until you input the parameter and press the [Enter] key. If a command has multiple parameters to be input, the system brings up the next guidance, so be sure to input all necessary parameters sequentially. If you press the [Enter] key without entering a parameter in some guidance session of a command, the system assumes the command is canceled and does not execute it.

#### Commands that replace existing data after confirmation

The commands that rewrite memory or register contents one by one provide you with the option of skipping guidance (do not modify the contents), returning to the immediately preceding guidance, or terminating during the input session.

[Enter] key            Skips input.

[^] & [Enter] key    Returns to the immediately preceding guidance.

[q] & [Enter] key    Terminates the input session.

```

Input example:
>eb␣ ...Command to modify data memory.
Enter address ? :0␣ ...Inputs the start address.
00000000 02 :aa␣ ...Modifies address 0 to 0xAA.
00000001 00 :^␣ ...Returns to the immediately preceding address.
00000000 AA :10␣ ...Inputs address 0 back again.
00000001 00 :␣ ...Skips address 1. (Contents not modified)
00000002 00 :ff␣
00000003 00 :q␣ ...Terminates the input session.
>

```

### Successive execution using the [Enter] key

The commands listed below can be executed successively by using only the [Enter] key after executing once. Successive execution here means repeating the previous operation or continuous display of the previous contents.

Execution commands: g, s, n

Display commands: sc, m, u, db, dh, dw, od, td, sy, sw, com, cmw, ss

The successive execution function is terminated when some other command is executed.

For the com and cmw commands that execute a command file, all the commands described in the command file are executed. This function is useful to execute a series of commands successively. For example, after the command file that contains the s and db commands is executed once, pressing [Enter] executes the step and memory dump (byte) operations repeatedly.

### Command history

The [Command] window supports a command history function. Up to 20 of the latest executed commands can be redisplayed at the prompt position using the ↑ or ↓ key and the displayed command can be executed by pressing [Enter].

Furthermore, the [F3] key can be used to redisplay the previously executed command.

## 16.7.2 Parameter Input Formats

### Numeric value

The parameters to specify addresses or data in a command are set to be input in hexadecimal by default. The 0x normally added at the beginning of a hexadecimal number can be omitted for input here. The characters that are recognized as hexadecimal are numbers 0 to 9 and alphabets a to f and A to F only.

Some parameters used to specify a number of execution steps or step No. in a command are set to be input in decimal by default. The characters that can be used in these parameters are only numbers 0 to 9.

For details about these parameters, refer to the explanation of each command.

The numeric values in the following formats are always accepted regardless of the default settings:

**Numeric values that begin with 0x:** These values are regarded as hexadecimal numbers. Only the lowercase x is accepted, so a 0X is not recognized as a valid numeric value. The characters that can be used after the 0x are numbers 0 to 9 and letters a to f and A to F only.

**Numeric values that begin with +:** These values are regarded as decimal numbers. A negative number will result in an error. Only numbers 0 to 9 can be used after the +.

**Note:** If an hexadecimal number is input by omitting the "0x" for an address specifying parameter whose default input is in hexadecimal from, the db33 assumes that a symbol is specified, and searches for the symbol information first. Therefore, when using the symbols represented by a hexadecimal or decimal number, you want to specify an address by using a number, be sure to add the "0x" when you input it.

## Address specification by line number

The line numbers in the source file can be used to specify an address. However, this is limited to cases in which you are debugging a srf33 format object file that contains information on the source line numbers.

Use the following format to specify a line number:

Line number specification: [**<file name>**]#**<line No.>**

**<file name>**: Source file name

The **<file name>** can be omitted when specifying a symbol in the current file (one that contains a code corresponding to the current PC). If a symbol that does not exist in the current file is specified without entering a file name, an error results.

**<line No.>**: Line number

The **<line No.>** can only be specified in decimal form. Adding a "+" or "-" results in an error.

Examples:    main.c#100  
              #100

## Address specification by symbol

Symbols can be used to specify an address. However, this is limited to cases in which you are debugging a srf33 format object file that contains symbol information.

Use one of the following two formats to specify a symbol:

**Note:** The term "current", as used in the current source files and current functions in the explanation below, means that the file or function contains a code corresponding to the current PC.

Format 1: **<symbol>**

**<symbol>**: Symbol name

- A pointer (\*), structure member (->, .), or array ([, ]) can also be specified. Notation must conform to the C language syntax. "\*" can be specified up to three nest levels and "[ ]" can be specified up to the fourth dimension.
- The characters that can be used here are limited to numbers 0 to 9, letters a to z and A to Z, and the symbols ->, ., and \*. Upper-case and lower-case letters are distinguished.

Examples:    i  
              \*message1  
              struct1->member1  
              struct2[5]

When a symbol is specified in this format, the db33 searches for the symbol in the order shown below until it finds the address of that symbol.

1. Current block
2. Current function
3. Static symbol in the current source file
4. External symbol

If no corresponding symbol is found, it is assumed to be a hexadecimal number. In this case, an error results if any character other than 0 to 9, a to f or A to F is used.

Format 2: [**<file>**]/[**<function>**]/**<symbol>**

The parameters in [ ] can be omitted. However, "/" cannot be omitted.

**<file>**: Source file name

- When specifying the current source file, input a period (.).
- See "Types of specification" below for specification when it is omitted.

**<function>**: Function name

- When specifying the current function, input a period (.).
- See "Types of specification" below for specification when it is omitted.

**<symbol>**: Symbol name

- Up to three asterisks (\*) can be added at the beginning of a symbol name.
- Structure members (->, .) can be specified in up to 10 hierarchical levels.
- Arrays ([, ]) can be specified in up to the 4th dimension.
- The characters that can be used here are limited to numbers 0 to 9, letters a to z and A to Z, and the symbols ->, ., and \*. Upper-case and lower-case letters are distinguished. Parentheses ( ) cannot be specified.

Types of specification:

//symbol	Global symbol
./symbol	Auto/static symbol in the current function
./symbol	Static symbol in the current source file
file/symbol	Static symbol in the specified source file
/function/symbol	Auto/static symbol for the specified external function
./function/symbol	Auto/static symbol for the specified function in the current source file
file/function/symbol	Auto/static symbol for the specified function in the specified source file

### Precautions on specifying array

1) When type information is included

In a one-dimensional array, offsets are calculated according to the size of the type even if the specified element number is greater than the actual number of elements. This does not result in an error.

In two-dimensional or higher-order arrays, this relationship is checked, so that if a specified element number is greater than the actual number of elements, an error is assumed.

2) When no type information is included

In a one-dimensional array, offsets are calculated in byte units. This does not result in an error.

Two-dimensional or higher-order arrays cannot be specified.

### Other restrictions

If one of the following cases applies when specifying an address, an error is assumed because no address can be obtained:

- 1) When a register variable is specified (because no addresses are assigned)
- 2) When the specified pointer variable indicates an unmapped area

### Entering file name



















A file name can be input using up to 127 characters, including a path. When specifying a file name that does not exist in the current directory, be sure to add a path.

The characters that can be used here are limited to numbers 0 to 9, letters a to z and A to Z, and the symbols \_, ., and /. Upper-case and lower-case letters are distinguished.

### 16.7.3 Executing from Menu or Tool Bar

The menu and tool bar are assigned frequently-used commands as described in Sections 16.5 and 16.6. A command can be executed simply by selecting your desired menu command or clicking on the tool bar button. Table 16.7.3.1 lists the commands assigned to the menu and tool bar.

Table 16.7.3.1 Commands that can be specified from menu or tool bar

Command	Function	Menu	Button	Window
lf	Loads a srf33 file.	[File]-[Load File..]		-
m	Produces mixed display.	-		[Source]
sc	Produces source display.	-		[Source]
u	Produces disassemble display.	-		[Source]
g	Executes the program.	[Run]-[Go]		-
g <address>	Executes the program until <address>.	[Run]-[Go to]		[Source]
s	Instructs one step at a time.	[Run]-[Step]		-
n	Steps and skips.	[Run]-[Next]		-
rstc	Cold-resets the CPU.	[Run]-[Reset cold]		-
rsth	Hot-resets the CPU.	[Run]-[Reset hot]		-
bp	Sets software breakpoints.	[Break]-[Soft PC...]		[Source]
bh, bh2	Sets hardware breakpoints.	[Break]-[Hard PC...]		[Source]
bd	Sets data break conditions.	[Break]-[Data...]	-	-
bsq	Sets sequential break conditions.	[Break]-[Sequential...]	-	-
bac	Clears all break conditions.	[Break]-[All clear]	-	-
sw	Displays symbols.	[Symbol]-[Watch]		[Source]
sa	Registers the symbols to be monitored.	[Symbol]-[Add]		[Source]
sd	Deletes the registered symbols.	[Symbol]-[Delete]		[Symbol]
-	Displays ICD on-chip trace data	-		-
-	Resumes ICD on-chip tracing	-		-
-	Activates the [Command] window.	[Window]-[Command]	-	-
-	Opens the [Source] window.	[Window]-[Source]	-	-
-	Opens the [Memory] window.	[Window]-[Memory]	-	-
-	Opens the [Register] window.	[Window]-[Register]	-	-
-	Opens the [Trace] window.	[Window]-[Trace]	-	-
-	Opens the [Symbol] window.	[Window]-[Symbol]	-	-
-	Opens the [Simulated I/O] window.	[Window]-[StdIO]	-	-
-	Forcibly breaks program execution.	-		-

The window column of the above table indicates the window that must be opened before selecting the tool bar buttons or menu commands.

A command executed from the menu or tool bar is not displayed in the prompt section of the [Command] window. The execution result is displayed in the corresponding window.

## 16.7.4 Executing from Command File

Another method for executing commands is to use a command file that contains descriptions of a series of debug commands. By reading a command file into the debugger you can execute the commands written in it.

### Creating a command file

Create a command file as a text file using an editor.

Although there are no specific restrictions on the extension of a file name, Seiko Epson recommends using ".cmd".

### Example of a command file

The example below shows a command file included in the simulated I/O sample files.

Example: File name = simIO.cmd

```
lf simIO.srf      ...Loads the file.
rstc             ...Cold-resets the CPU.
stdout          ...Sets stdout conditions.
1
WRITE_FLASH
WRITE_BUF
1
stdin           ...Sets stdin conditions.
1
READ_FLASH
READ_BUF
1
bs _exit        ...Sets a software break point.
```

You can use a command file to write the commands that come with a guidance mode. In this case, be sure to break the line for each guidance input item as you write a command. In the above example, the contents following stdout and stdin are guidance items.

### Reading in and executing a command file

There are two methods to read a command file into the debugger and to execute it, as described below.

#### (1) Execution by the startup option

By specifying the `-c` option in the db33 startup command, you can execute one command file when the debugger starts up.

Example: Startup command of the db33

```
db33 -c startup.cmd -p 33104_1.par
```

#### (2) Execution by a command

The db33 has the `com` and `cmw` commands available that you can use to execute a command file.

The `com` command reads in a specified file and executes the commands in that file sequentially in the order they are written.

The `cmw` command performs the same function as the `com` command except that each command is executed at intervals specified by the `md` command (1 to 256 seconds).

Examples: `com startup.cmd`

```
cmw test.cmd
```

The commands written in the command file are displayed in the [Command] window.

### Successive execution using a command file

After a command file is executed once, pressing [Enter] alone can execute all the commands described in the command file repeatedly. The successive execution function is terminated when some other command is executed.

### Restriction

You can read in another command file from within a command file. However, nesting of these command files is limited to a maximum of five levels. An error is assumed and the subsequent execution is halted when the `com` or `cmw` command at the sixth level is encountered.

## 16.7.5 Log File

The executed commands and the execution results can be saved to a file in text format that is called a "log file". This file allows you to verify the debug procedures and contents.

### Command example

```
>log test.log
```

After being set to the log mode by the log command (after output starts), a log is saved until the log command is executed next.

### Contents saved to a log

The contents displayed in the [Command] window are saved to a log file. The results of commands executed from tool bars or menus, the execution results shown elsewhere, and all other contents not displayed in the [Command] window are not saved.

Therefore, if log management is desired, Seiko Epson recommends specifying the -w option before you start up the Debugger db33, and recording a log of execution results only in the [Command] window.



## 16.8 Debug Functions

This section outlines the debug features of the db33, classified by function.  
See Section 16.9, "Command Reference", for details about each debug command.

### 16.8.1 Debugger Mode

The db33 supports four debugger mode selectable with startup option.

**Note:** If the specified debugger mode option does not match with the connected debugging target system, a dialog box appears to show a warning message. In this case, terminate the db33 and then restart the db33 with the correct option specification.

#### ICE mode

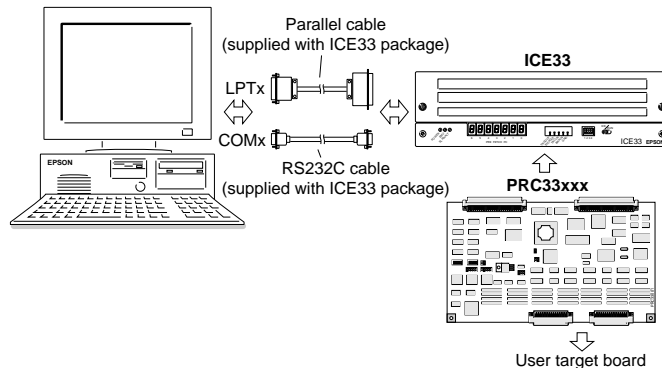


Fig. 16.8.1.1 Debugging system using ICE33

#### Specification at startup

Startup option: Specify `-ice` (can be omitted)

Specification on wb33: Select [ICE]

When the debugger starts up in ICE mode, "ICE" is displayed on the tool bar.

The ICE mode is used to debug a program using the ICE33 in-circuit emulator. In this mode, therefore, program execution and trace utilizes the internal memory of the ICE33. All functions available with the ICE33 can be utilized. It is also possible to debug hardware functions after connecting the target board to the ICE33.

When invoking the debugger in ICE mode, make sure that the ICE33 is connected firmly and that its power is turned on.

Each area in the ICE33 is initialized as follows:

Internal ROM, ROM area assigned in the emulation memory (EROM): 0xff

RAM area assigned in the emulation memory (ERAM): 0xaa

Other IO, RAM, ROM areas: Not initialized

Refer to the "E0C33 Family In-Circuit Emulator (ICE33) Manual" for operating the ICE33.

## ICD mode

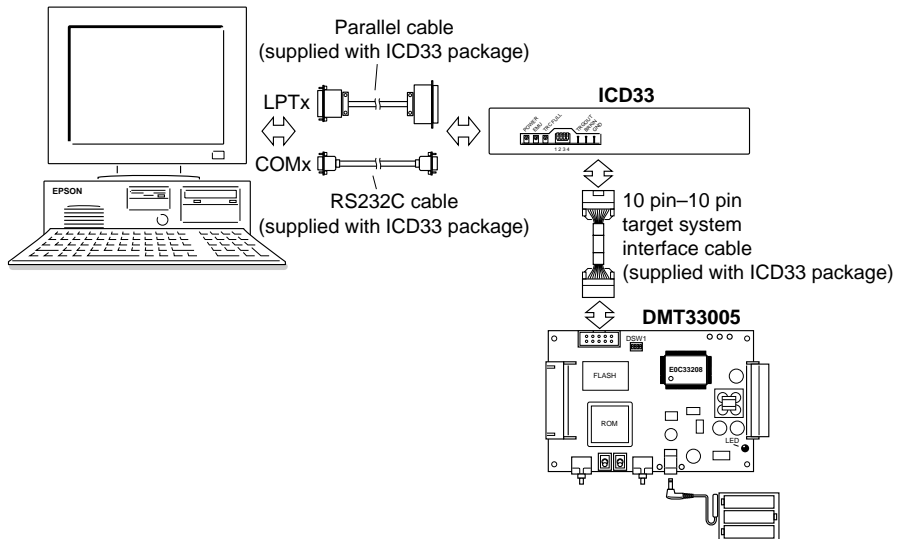


Fig. 16.8.1.2 Debugging system using ICD33 and DMT33005

**Specification at startup**

Startup option: Specify `-icd`

Specification on wb33: Select [ICD] (initial setting)

When the debugger starts up in ICD mode, "ICD" is displayed on the tool bar.

The ICD mode is used to debug a program using the ICD33 in-circuit debugger. In this mode, the program is executed on the target board and trace information is sampled in the ICD33 memory.

Note that the following functions cannot be used in ICD mode:

- Loading/dumping option data
- Sequential break
- Map break

When invoking the debugger in ICD mode, make sure that the ICD33 and the target board are connected firmly and they are turned on.

Furthermore, when using the trace function, the DIP switch SW4 must be set to OPEN.

Refer to the "E0C33 Family In-Circuit Debugger (ICD33) Manual" for operating the ICD33.

## Debug monitor mode

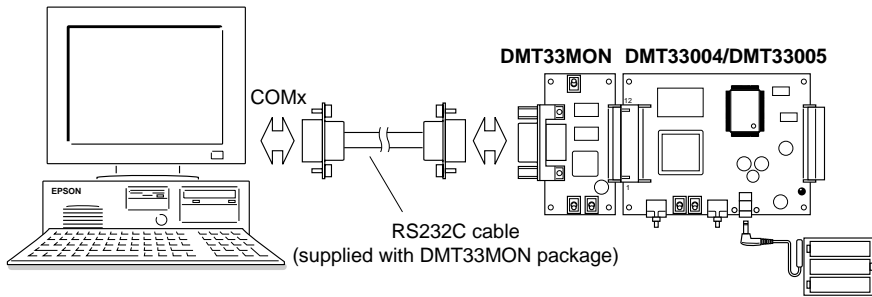


Fig. 16.8.1.3 Debugging system using DMT33004/DMT33005 board

### Specification at startup

Startup option: Specify `-mon`

Specification on wb33: Select `[MON]`

When the debugger starts up in debug monitor mode, "MON" is displayed on the tool bar.

The debug monitor mode is used to debug a program using the target board with the DMT33MON board. The Debug Monitor (MON33) must be implemented to the target board.

Note that the following functions cannot be used in debug monitor mode:

- On-the-fly mode
- Loading/dumping option data
- Sequential break
- Map break
- Tracing
- Execution time/cycle measurement
- Key break

When invoking the debugger in debug monitor mode, make sure that DMT33MON and the target board are connected firmly and they are turned on.

Furthermore, the Debug Monitor on the target board must be activated.

Refer to the "E0C33 Family MON33 Debug Monitor Manual" for details of the Debug Monitor.

## Simulator mode

### Specification at startup

Startup option: Specify `-sim`

Specification on wb33: Select `[SIM]`

When the debugger starts up in simulator mode, "SIM" is displayed on the tool bar.

The simulator mode is used to simulate target program execution in the internal memory of a personal computer; therefore, other debugging tools are not required. In this mode, however, you cannot evaluate the ICE33 dependent functions and the I/O functions of the target system. What is possible in this mode is simulation of the core CPU, memory model, and interrupt.

The sequential break and some other functions available in ICE mode are not supported in simulator mode. The trace method in simulator mode differs from other modes. See the description of each command for details.

Each area in simulator mode is initialized as follows:

RAM: 0xaa  
 ROM: 0xff  
 I/O: 0x00

**Precaution for ICE, ICD and debug monitor mode**

When the program execution is suspended, the ICE33 and ICD33 switch the CPU operating clock to the high-speed (OSC3) clock and halt all the peripheral functions except for the DRAM refresh operation. In the Debug Monitor, the same status occurs instantaneously when a break occurs or program execution starts, however it returns to the previous status immediately.

Therefore, the system that does not use an OSC3 clock cannot be debugged.

When the OSC3 oscillation circuit is stopped and the system is operating with the low-speed clock (OSC1, 32 kHz), the OSC3 oscillation circuit will start oscillating immediately after a break occurs. However, an erroneous operation may result since the oscillation is unstable. Do not suspend the program execution while the OSC3 oscillation is stopped, even in SLEEP status.


## 16.8.2 Loading Files

### File types

The db33 can read a file in srf33 format or Motorola S3 format in the debugging process.

Table 16.8.2.1 lists the files that can be read in by the debugger and the load commands.

Table 16.8.2.1 Files and load commands

File type	Extension	Generation tool	Command	Menu	Button
srf33	.srf	Linker	lf	[File]-[Load File..]	
Motorola S3	.sa	Binary/HEX converter	lh	–	–

### Debugging a program at the source level

To debug a program using the source display and symbols, you must have the object file in srf33 format read into the debugger. If any other program file is read, only the disassemble display is produced.

For the source level debugging of the program written in the target ROM, the ld command is provided. This command reads only the debugging information from the object file in srf33 format.

### Precautions

The lf and lh commands load only the portions that contain code and data. The previous data remains unaffected in all other portions.

If the source display is required, the source files are read into the debugger in addition to the above files according to debugging information in the srf33 object file. For this reason, the source file must be maintained under the same conditions in both content and place of storage (directory) as when the srf33 object file was generated. Up to 32,767 lines one source file can be read in.

### 16.8.3 Source Display and Symbolic Debugging Function

The db33 allows you to debug a program while displaying the C and assembly source statements. Address specification using a symbol and displaying the contents of symbols are also possible.

#### Displaying program code

When the [Source] window is left open, you can display the program to be debugged in that window. The display mode can be changed between the three modes available.

##### (1) Disassemble display

Address	Code	Disassemble	
0080014	8208	pushw	Er0
0080018	8208	id.w	Er11,Er8
008001B	3208	id.w	[Er8],Er11
008001A	8208	id.w	Er0,Er8
008001C	2E0C	id.w	Ir12,Er8
008001E	1206	call	Er5
0080020	16018	add	Er0,Er1
0080022	1EFD	jp	Er1d,Er1
0080024	8248	popw	Er0
0080026	8548	ret	
0080029	781C	and	Ir12,Er1
008002A	1804	ireq	Er4
008002C	388A	id.w	Er16,[Er8]
008002E	891A	add	Er16,Er1
0080030	3D8A	id.w	[Er8],Er10
0080032	8548	ret	
0080034	FFFF	xxx	
0080036	FFFF	xxx	
0080038	FFFF	xxx	
008003A	FFFF	xxx	
008003C	FFFF	xxx	

##### (2) Mixed display




Address	Code	Disassemble	Line	SourceCode
0080014	8208	pushw	00817	{
0080018	8208	id.w	00818	int j;
008001B	3208	id.w	00819	i = 0;
008001A	8208	id.w	00821	for (i=0 ; i <= 10)
008001C	2E0C	id.w	00822	{
008001E	1206	call	00823	sub();
0080020	16018	add	00824	for (i=0 ; i <= 10)
0080022	1EFD	jp	00825	{
0080024	8248	popw	00826	}
0080026	8548	ret	00827	}
0080029	781C	and	00828	sub();
008002A	1804	ireq	00829	int k;
			00830	{
			00831	if (k & Er1)
			00832	{
			00833	}
			00834	}

##### (3) Source display

Address	Line	SourceCode
0080014	00817	{
0080018	00818	int j;
008001B	00819	i = 0;
008001A	00821	for (i=0 ; i <= 10)
008001C	00822	{
008001E	00823	sub();
0080020	00824	}
0080022	00825	}
0080024	00826	}
0080026	00827	sub();
0080028	00828	int k;
0080029	00829	{
008002A	00830	if (k & Er1)
008002B	00831	{
008002C	00832	++i
008002D	00833	}
008002E	00834	}

In the source display mode, only the current source (the one that contains a code corresponding to the current PC) or the source selected in the [Select source] box is displayed.

Table 16.8.3.1 Commands/tool bar buttons to switch display mode

Display mode	Command	Button
Disassemble	u	
Mixed	m	
Source	sc	

When these commands are executed, the [Source] window has its display contents updated so that the current PC address is always displayed in the window. Furthermore, these commands can also be used to specify the display start address.

If the [Source] window is not open, each command displays the above contents in the [Command] window. Each button can only be used when the [Source] window is open. When the display mode is switched using the toolbar button, the [Source] window displays the same part of the currently displayed codes and does not change it to the current PC address.

In the source display mode, you can specify a search character string so that the contents will be displayed beginning with the searched position.

Table 16.8.3.2 Source character string search command

Function	Command
Character string search	ss

## Operating symbols

When debugging a srf33 format object file after reading it into the db33, you can use the symbols defined in the source file to debug the program.

### (1) Address specification by symbol

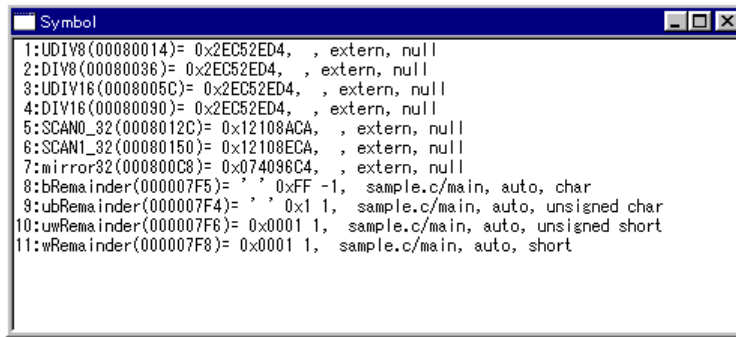
When entering a command that has <address> in its parameter from the [Command] window or entering an address in the dialog box, you can specify the address by using a symbol. For details on how to specify, refer to Section 16.7.2, "Parameter Input Formats".

### (2) Displaying symbol information

The symbol information (e.g., address, content, scope, class, and type) that is used in the program under debug can be displayed in the [Command] window. Commands are available that allow you to display a condition-specified list or verify variables after program execution. For details about display contents, refer to the explanation of the sy command.




Display examples: B00T, 00080004, boot.s/, static, null  
i, 00000000, , extern, int  
j, R0, main.c/main, register, int  
k, R12, main.c/sub, reg parameter, int  
main, 00080014, , extern, int ()  
sub, 00080028, , extern, int ()

**Note:** The debugger reads the contents of symbols from the target in byte units and re-arranges the read data to byte, half-word or word data according to the symbol size before displaying. Note that data cannot be displayed if the big/little endian settings are different between the BCU on the target and the parameter file read in the debugger.

**(3) Monitoring symbols in [Symbol] window**

The [Symbol] window can have up to 99 symbols registered (in watch symbol table). This facility allows you to monitor, for example, the contents of variables that are modified by program execution.

Table16.8.3.3 Commands/menu commands/tool bar buttons to display symbol list

Function	Command	Menu	Button
Displaying symbols	sw	[Symbol]-[Watch]	
Displaying symbol list	sy	-	-
Registering monitor symbols	sa	[Symbol]-[Add]	
Deleting monitor symbols	sd	[Symbol]-[Delete]	



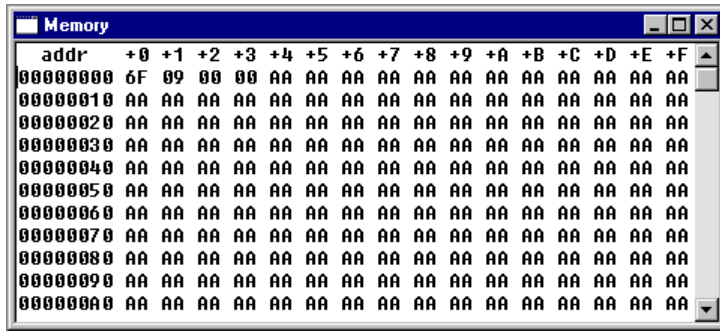
### 16.8.4 Displaying and Modifying Memory Data and Register

The db33 has functions to operate on the memory and registers. Each memory area is set to the debugger according to the map information that is given in a parameter file. Memory access and data display in half word or word units are performed in little-endian format by default. It can be changed so that the specified area will be accessed in big-endian format using the parameter file.

#### Operating on memory area

Following operations can be performed on the memory area:

(1) Dumping data memory



The contents of the memory are displayed in hexadecimal dump format. If the [Memory] window is open, the contents of the [Memory] window are updated; if not open, the contents of data memory are displayed in the [Command] window. (db, dh and dw commands)

(2) Entering/modifying data

The data at a specified address is rewritten by entering hexadecimal data. (eb, eh and ew commands)

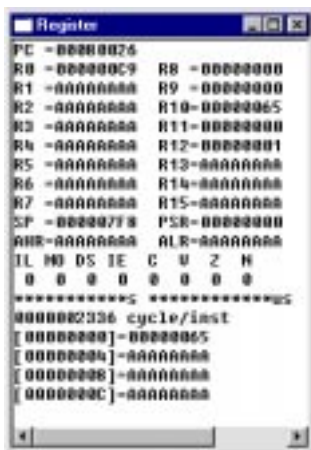
(3) Rewriting a specified area

An entire specified area is rewritten with specified data. (fb, fh and fw commands)

(4) Copying a specified area

The content of a specified area is copied to another area. (mv, mvh and mvw commands)

(5) Monitoring memory



Four memory locations, each with area to store a word (4 bytes), can be registered as watch data addresses. The registered watch data can be verified in the [Register] window. When operating in ICE mode, the content of this window is updated in real time at 1 to 0.1-second intervals by the on-the-fly function. Addresses 0, 4, 8, and C are made the watch data addresses by default.

Note that data of the internal RAM area is not updated in real time since accesses to the internal RAM area cannot be detected from outside the chip. It will be updated after the target program breaks.

← Monitor data

Table 16.8.4.1 Commands to operate on data memory

Function	Command
Dumping memory	db (byte units), dh (half word units), dw (word units)
Entering/modifying data	eb (byte units), eh (half word units), ew (word units)
Rewriting specified area	fb (byte units), fh (half word units), fw (word units)
Copying specified area	mv (byte units), mvh (half word units), mvw (word units)
Setting watch data address	w (word units)

#### • Updating of the [Memory] window

When you open the [Memory] window using the [Memory] command on the [Window] menu, the contents of the memory are displayed in the window. The data at all addresses can be checked by scrolling the window, or you can use the d\* (db, dh, dw) command to specify an address so the window starts displaying the memory contents beginning with the specified address.

When the [Memory] window is open you may modify the address displayed in the window, but the display contents of the [Memory] window will not be updated by that modification. To update the display contents, you need to execute the d\* command or scroll the [Memory] window in the vertical direction.

The [Memory] window is cleared by reading a file. In such a case, redisplay the window using the method described above.

When the program is executed successively, the [Memory] window will be updated immediately after a break occurs. During step execution, the [Memory] window is updated every step. This automatic update function can be disabled using the md command.

#### • Updating the [Source] window

When the [Source] window is open you may modify the content of an address displayed in the window, but the display contents of the [Source] window is not updated by that modification. To update the display contents, you need to temporarily switch the display mode of the [Source] window using a command or scroll the [Source] window in the vertical direction. Note that when code is modified, the disassemble result changes, but the display contents of source do not change.

- Notes:
- When an address in which no registers have been allocated in the internal I/O area is read, CPU-last-read data is displayed.
  - The ICD33 reads memory data 8 bytes at a time. Therefore, data may be read exceeding the range specified using a command (to maximum 7 bytes ahead). Pay attention when reading the I/O memory since some registers change their status by reading.
  - When writing data to the internal ROM emulation memory on the EPOD33XXX through the ICD33 or MON33, byte-access commands (eb, fb, mv) cannot be used. Be sure to use a half-word- or word-access command (eh, ew, fh, fw, mvh, mvw). The file load commands (lf, lh) always write data in half-word units.

## Operating registers

Following operations can be performed on registers:

### (1) Displaying registers

Register contents can be displayed in the [Register] or [Command] window.

General-purpose registers: R0 to R15

Special registers: PC, SP, PSR, AHR, ALR

When operating in ICE mode, the contents of the PC and PSR register are updated in real time every 0.2 seconds (default) by the on-the-fly function. (See Section 16.8.5, "Executing Program".)

### (2) Modifying register values

The contents of the above registers can be set to any desired value.

Table 16.8.4.2 Commands/menu commands to operate registers

Function	Command	Menu
Displaying registers	rd	[Window]-[Register]
Modifying register values	rs	—

### 16.8.5 Executing Program



The debugger can execute the target program successively or execute source lines/instructions one step at a time (single-stepping).

#### Successive execution

##### (1) Successive execution commands

The successive execution command execute the loaded program successively from the current PC address.

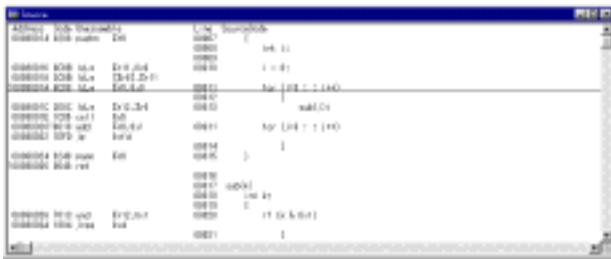
Table 16.8.5.1 Commands/menu commands/tool bar buttons for successive execution

Function	Command	Menu	Button
Successive execution	g	[Run]-[Go]	
Successive execution to the specified address	g <address>	[Run]-[Go to]	

##### (2) Stopping successive execution

Using the successive execution command, you can specify a temporary break addresses that are only effective during program execution.

The temporary break address also can be specified from the [Source] window.



If you click on the address line 0x8001A shown in the [Source] window (after moving the cursor to that line) and click on the [Go to] button, for example, the program starts executing from the current PC and breaks before executing the instruction at 0x8001A.

Except being stopped by this temporary break, the program continues execution until it is stopped by one of the following causes:

- Break conditions set by a break setup command are met.
- You click on the [Key break] button. (not available in debug monitor mode)
- Some other break factor occurs.



[Key break] button

\* When the program does not stop, use this button to forcibly stop it.

##### (3) On-the-fly function

When operating in ICE mode, you can use the on-the-fly function to display the PC, PSR register, and watch data values every 0.2 seconds (default) during successive execution. These contents are displayed in the relevant positions of the [Register] window. If the [Register] window is closed, they are displayed in the [Command] window. In the initial debugger settings, the display update interval of the on-the-fly function is set to 5 times per second. It can be modified to 0 (OFF)–10 (times) per second using the md command. This function provides a complete real-time display that is implemented using the ICE33 hardware.

The on-the-fly function in ICD mode displays a message that indicates trace-memory-full in the [Register] window (or the [Command] window) and real-time trace data. The md command is used to set the update interval similar to ICE mode.

The on-the-fly function is available only in ICE and ICD modes. In other modes, the display of all windows except the [Register] window remains unchanged; changes are cleared during successive execution.

## Single-stepping

### (1) Types of single-stepping

There are two types of single-stepping available:

#### • Stepping through all codes (STEP)

In this single-stepping, the program is executed in units of addresses or source codes – i.e., one address or source code at a time – depending on the [Source] window's display mode as shown below:

Disassemble display mode: Address units

Mixed display mode: Address units



Source display mode: Source code units

#### • Stepping through codes except functions and subroutines (NEXT)

When a C source function call, assembly source subroutine call, or software interrupt is encountered, each called function, subroutine, or interrupt routine is executed as one step. All codes in the current function or subroutine except calls are executed in the same way as in STEP.

In either case, the program starts executing from the current PC.

Table 16.8.5.2 Commands/menu commands/tool bar buttons for single-stepping

Function	Command	Menu	Button
Stepping through all codes	s	[Run]-[Step]	
Stepping through all codes except functions and subroutines	n	[Run]-[Next]	

When executing single-stepping by command input, you can specify the number of steps to be executed, up to 65,535 steps. When using menu commands or tool bar buttons, the program is executed one step at a time.

In the following cases, single-stepping is terminated before a specified number of steps is executed:

- When you click on the [Key break] button (not available in debug monitor mode)
- When a break factor except for user set break occurs

Single-stepping is not halted by breaks set by the user such as a PC breakpoint or data break.



[Key break] button

\* When the program does not stop, use this button to forcibly stop it.

### (2) Display during single-stepping

In the initial debugger settings, the display is updated every step as follows:

When the [Source] window is open, the underline designating the next address to be executed moves every step as the program is stepped through.

The display contents of the [Register] and [Memory] windows are also updated every step.

The display mode can be switched over by the md command so that the display contents of the [Register] window are updated at only the last step in a specified number of steps and the [Memory] window is not updated automatically.

### (3) HALT and SLEEP states and interrupts

In the ICE33, interrupts are disabled during single-stepping.

The halt and slp instructions are executed even during single-stepping, in which case the CPU is placed in a standby mode. The CPU can be released from the standby mode by generating an external interrupt or by pressing the [Key break] button.

## Measuring execution cycles/execution time

### (1) Execution counter and measurement mode

The ICE33 contains three 31-bit execution counters allowing you to measure the program execution time (2 systems) and the number of bus cycles executed (1 system).

The ICD33 contains a 29-bit execution counter that can be set for measuring execution time (sec or  $\mu\text{sec}$ ) or number of cycles using the `md` command.

The execution counter for simulator mode counts only the number of instructions executed.

Note that the execution counter is not available in debug monitor mode.

Table 16.8.5.3 Measurement units and accuracy of the execution counter

Execution counter	ICE mode	ICD mode	Simulator mode
Execution time 1	1 $\pm$ 1 $\mu\text{sec}$	1 $\pm$ 1 $\mu\text{sec}$	–
Execution time 2	50 $\pm$ 50 nsec	50 $\pm$ 50 nsec	–
Bus cycle	1 $\pm$ 1 cycle	1 $\pm$ 16 cycle	–
Instruction	–	–	1 $\pm$ 0 instruction

The following lists the maximum values that can be measured by the execution counter:

	ICE mode	ICD mode
Execution time 1:	2147483647 $\mu\text{sec}$ = approx. 36 min.	536870911 $\mu\text{sec}$ = approx. 9 min.
Execution time 2:	2147483647 x 50 nsec = approx. 107 sec.	536870911 x 50 nsec = approx. 27 sec.
Bus cycle:	2147483647 cycles	536870911 x 4 = 2147483644 cycles

### (2) Displaying measurement results

The measurement result is displayed in the [Register] window. This display is cleared during program execution and is updated after completion of execution. If the [Register] window is closed, the measurement result can be displayed in the [Command] window using the `rd` command. The execution results of single-stepping are also displayed here.

If the count exceeds the counter size, the system indicates "over flow".

### (3) Integrating mode and reset mode

In the initial debugger settings, the execution counter is set to an integrating mode. In this mode, the measured values are combined until the counter is reset.

The reset mode can be set by the `md` command. In this mode, the counter is reset each time the program is executed. In successive execution, the counter is reset when the program is made to start executing by entering the `g` command and measurement is taken until the execution is terminated (break occurs).

In single-stepping, the counter is reset when the program is made to start executing by entering the `s` or `n` command and measurement is taken until execution of a specified number of steps is completed. The counter is reset every step if execution of only one step is specified or execution is initiated by a tool bar button or menu command.

### (4) Resetting execution counter

The execution counter is reset in the following cases:

- When the execution counter mode is switched over by the `md` command (from integrating mode to reset mode)
- When program execution is started in reset mode

## Resetting the CPU

The CPU is cold-reset when the rstc command ([Reset Cold] command on the [Run] menu, or the [Reset cold] button) is executed, or is hot-reset when the rsth command (or [Reset Hot] command on the [Run] menu, or the [Reset hot] button) is executed.

When the CPU is reset, the internal circuits are initialized as follows:

### (1) Internal registers of the CPU

R0–R15: 0xaaaaaaaa  
 PC: Boot address (\*)  
 SP: 0x0aaaaaaaa8  
 PSR: 0x00000000  
 AHR, ALR: 0xaaaaaaaa

\* The boot address is the 4-byte value stored from the beginning of the vector table that is specified by the TTBR register. At cold-reset, the TTBR register is initialized to 0x80000 or 0xc00000. At hot-reset, the TTBR register retains the set value.

### (2) The execution counter is reset to 0.

### (3) The [Source] and [Register] windows are redisplayed.

Because the PC is set to the boot address, the [Source] window is redisplayed beginning with that address. The [Register] window is redisplayed with the internal registers initialized as described above.

The memory contents are not modified.

Note: The function of the rstc command changes according to the debugger mode.

#### ICE mode

The process above is executed and the E0C33 chip is also reset. The target board is not reset.

#### ICD mode

The process above is executed and the E0C33 chip is also reset. The target board is not reset. Furthermore, when the target system is in a free-run state, the rstc command suspends the program execution forcibly before resetting. The target system connected to the ICD33 enters a free-run state when the target board is reset. The rstc command can be used to suspend the program execution in this case.

#### Debug monitor mode

The rstc command functions the same as the rsth command. It does not reset the E0C33 chip and does not initialize the TTBR register.

#### Simulator mode

The boot address is determined by the MCU/MPU specification in the parameter file.

### 16.8.6 Break Functions

The target program is made to stop executing by one of the following causes:

- Break command conditions are satisfied.
- The [Key break] button is activated. (not available in debug monitor mode)
- The ICE33/ICD33 BRKIN pin is pulled low.
- A map break or similar break occurs.

#### Break by command

The db33 has four types of break functions that allow the break conditions to be set by a command. When the set conditions in one of these break functions are met, the program under execution is made to break.

##### (1) Software PC break


This function causes the program to break when the PC matches the address set by a command. The program is made to break before executing the instruction at that address. Up to 16 addresses can be set as the breakpoints.

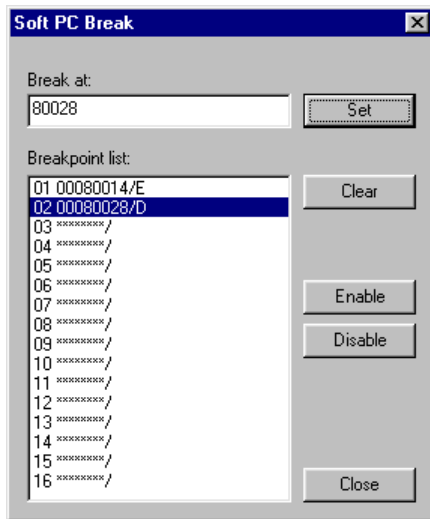
When this break occurs, the db33 displays the following message in the [Command] window and stands by waiting for command input.

Break by software PC break.

>

Table 16.8.6.1 Command/menu command/tool bar button to set software PC breakpoint

Function	Command	Menu	Button
Setting/canceling breakpoints	bp	[Break]-[Soft PC...]	



This dialog box appears on the screen when you select the [Soft PC...] command from the [Break] menu.

Up to 16 addresses can be registered in the breakpoint list.

#### Registering break addresses

Enter an address in the [Break at] text box, then press [Enter] or click the [Set] button. Addresses can be entered using the symbols.

#### Clearing the break point

Select the address to be cleared from the [Break list] box, then click the [clear] button.

#### Enabling/disabling the break point

When a break address is registered, it is configured as an enabled break point. The enabled break point is indicated with "/E" in the list. It can be disabled without clearing the registered address. To disable a break point, select the address from the list, then click the [Disable] button. The "/E" symbol changes to "/D" indicating that the break point is disabled. The [Enable] button switches the disabled break point (/D) to be enabled. (/E).

When using the bp command, follow the guidance displayed in the [Command] window as you enter addresses. The addresses which have a valid (enabled) breakpoint set are marked with a prefix "!" or "?" as they are displayed in the [Source] window.

!": When a breakpoint is set at the displayed address

?: When a breakpoint is set somewhere other than the beginning address of the source code in the source display mode

Using the [Soft PC Break] button allows you to set and cancel breakpoints easily.



Click on the address line in the [Source] window at which you want the program to break (after moving the cursor to that position) and then click on the [Soft PC Break] button. A "!" symbol will be placed at the beginning of the line indicating that a breakpoint has been set there, and the address is registered in the breakpoint list. Clicking on the line that begins with a "!" and then the [Soft PC Break] button cancels the breakpoint you have set, in which case the address is deleted from the breakpoint list.

- \* Software PC breaks are implemented by embedding the BRK instruction. Therefore, software PC breaks cannot be used for the ROM on the target board where instructions cannot be embedded. In this case, use a hardware PC break.

Note: When setting a software PC break point or hardware PC break point to extended instructions with ext or delayed branch instructions, only the first address can be specified.

ext xxxx ... Can be set. jr\*.d xxxx ... Can be set.  
 ext xxxx ... Cannot be set. Delayed instruction ... Cannot be set.  
 Extended instruction ... Cannot be set.

**(2) Hardware PC break**

Hardware PC break is implemented by using the debug mode of the E0C33000 core CPU. This break operation can be simulated even in the simulator mode. This function causes the program to break when the PC matches the address set by a command. The program is made to break before executing the instruction at that address. Up to two addresses can be set as hardware breakpoints.

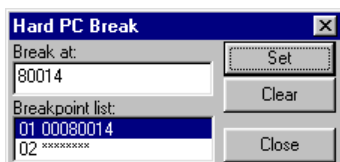
When this break occurs, the db33 displays the following message in the [Command] window and stands by waiting for command input.

Break by hardware PC break. or Break by hardware PC break2.

>

Table 16.8.6.2 Commands to set hardware PC breakpoint

Function	Command	Menu	Button
Setting breakpoint	bh, bh2	[Break]-[Hard PC...]	
Canceling breakpoint	bhc, bhc2		



This dialog box appears on the screen when you select the [Hard PC...] command from the [Break] menu.

Up to two addresses are allowed for hardware PC breakpoints.

To set a hardware PC breakpoint, enter an address in the [Break at] text box, then press [Enter] or click the [Set] button. Addresses can be entered using the symbols.

Clicking the [Clear] button clears the breakpoint.

- \* The [Hard PC Break] button is used to set a breakpoint in the [Source] window similar to the [Soft PC Break] button. The address set as a hardware PC breakpoint is marked with a suffix "!" or "?" as it is displayed in the [Source] window (see "Software PC Break").

Note: The hardware PC break function is disabled when the area trace function is set in ICD mode.



**(3) Data break**

This break function allows you to cause a break when a location in the specified memory address is accessed. In addition to specifying a memory address, you can specify whether you want a break to be caused by a read or write as the break condition. Both the read/write operations can also be specified, so that a break will be generated for whichever operation, read or write, is attempted.

A break occurs after completing the cycle in which an operation to satisfy the above specified condition is performed.

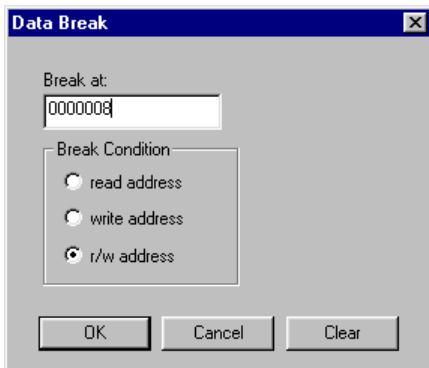
When this break occurs, the db33 displays the following message in the [Command] window and stands by waiting for command input.

Break by data break.

>

Table 16.8.6.3 Command/menu command to set data break

Function	Command	Menu
Setting/canceling data break conditions	bd	[Break]-[Data...]



This dialog box appears on the screen when you select the [Data...] command from the [Break] menu.

Enter an address in the [Break at] text box, and select an access condition from the radio buttons.

In this example, a break occurs when data is read or written from/to memory addresses 0x8.

When using the bd command, follow the guidance shown in the [Command] window as you enter the break conditions.

The address can also be specified using a symbol.

**(4) Sequential break (only in the ICE mode)**

For sequential breaks, you can specify one to three addresses, data patterns, data masks, and bus operation types. A break occurs when the program performs each specified type of bus operation in the order of specified addresses.

Specify data patterns and masks in a 16-bit hexadecimal number.

Choose a bus operation type from the nine types listed below:

- 0. All**            All bus operations
- 1. Inst**        Instruction fetch
- 2. VecR**        Vector fetch
- 3. DatR**        Data read
- 4. DatW**        Data write
- 5. StkR**        Read from stack
- 6. StkW**        Write to stack
- 7. DmaR**        Ready by DMA
- 8. DmaW**        Write by DMA

The sequential break function can only be used in ICE mode.

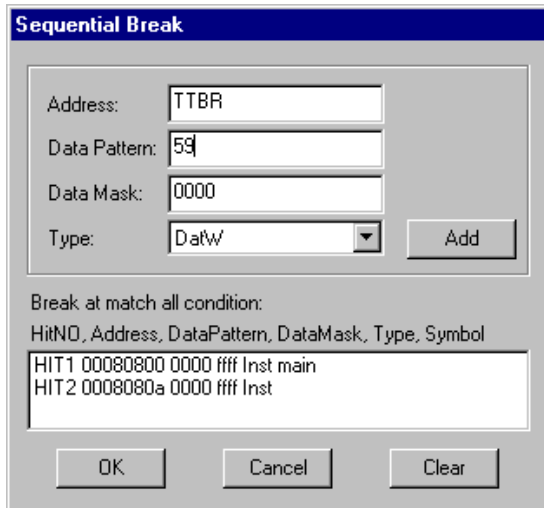
When this break occurs, the db33 displays the following message in the [Command] window and stands by waiting for command input.

Break by sequential break.

>

Table 16.8.6.4 Command/menu command to set sequential break

Function	Command	Menu
Setting/canceling sequential break conditions	bsq	[Break]-[Sequential]



This dialog box appears on the screen when you select the [Sequential...] command from the [Break] menu.

Enter an address, data pattern and data mask in each text box and select a bus operation type from the combo box, then click the [Add] button. The entered break condition is set in order from Hit No.1 to 3.

The address can also be specified using a symbol.

When using the `bsq` command, follow the guidance shown in the [Command] window as you enter the break conditions.

Example:

No.	Address	Data pattern	Data mask	Bus operation
1	0x00c80000	0x0000	0xffff	Inst
2	0x00e00001	0x0001	0xff00	DatW

In this example, a break occurs when the CPU writes 1 to address 0xe00001 after executing the instruction at address 0xc80000. The data mask 0xffff in No.1 specifies the mask in all the data pattern bits (the data pattern is omitted from the break condition). The data mask 0xff00 in No.2 specifies that the low-order 8 bits of the data pattern is compared with the low-order 8 bits of the actual access data.

Note: The sequential break function is not available in ICD, debug monitor, and simulator modes.

### Forced break by [Key break] button



[Key break] button

The [Key break] button can be used to forcibly terminate the program under execution when the program has fallen into an endless loop or cannot exit a standby (HALT or SLEEP) state.

Note: This break function is not available in debug monitor mode.

### Pulling ICE33 BRKIN pin low (only in ICE or ICD mode)

The program is made to break by entering a low pulse to the ICE33/ICD33 BRKIN pin when operating in ICE/ICD mode.

When this break occurs, the db33 displays the following message in the [Command] window and stands by waiting for command input.

Break by external break.

>

Notes: • This function is not available in debug monitor and simulator modes.

- In the ICD33, there is a delay time of approximately 1.5  $\mu$ sec between a pulse input to the BRKIN pin and the actual break generation.

## Map break and break by executing illegal instruction

The program also breaks when one of the following errors is encountered during program execution:

Note: The following break functions are not available in ICD and debug monitor modes.

### (1) Write to data ROM area

A break occurs when the program writes data to the ROM area set by the parameter file.

When this break occurs, the db33 displays the following message in the [Command] window and stands by waiting for command input.

Break by writing ROM area.

>

### (2) Access to no-map area

A break occurs when the program accesses a no-map area that has not be defined in the parameter file.

When this break occurs, the db33 displays the following message in the [Command] window and stands by waiting for command input.

Break by accessing no map area.

>

### (3) Accessing outside stack area (only in ICE mode)

A break occurs when the program accesses an area outside the stack area using the SP.

This break will occur only in ICE mode.

When this break occurs, the db33 displays the following message in the [Command] window and stands by waiting for command input.

Break by out of SP area.

>

### (4) Execution of an illegal instruction (only in the simulator mode)

A break occurs when an illegal instruction (code not generated by the Assembler as33 in which case the instruction is marked by "\*" in disassemble display) is executed in simulator mode.

When this break occurs, the db33 displays the following message in the [Command] window and stands by waiting for command input.

Break by illegal instruction.

>

Notes: • In the ICE33, a bus access to the internal RAM area does not generate a map break or sequential break, since it cannot be detected from outside the chip. However, a no-map area break can occur when instructions are executed in the internal RAM.

- If the CPU is cold-reset while it is executing the program in ICE, ICD or debug monitor mode, the on-chip-supported hardware PC break point (including temporary break used in the go command or internal next operation) and the data break condition are cleared. When the program execution breaks by another break factor, the break conditions are set again. Be aware that no hardware PC or data breaks will occur until the conditions are reset.

## 16.8.7 Trace Functions

The db33 has a function to trace program execution.

Note that the method of operation and functionality differ depending on the debugger mode..

### Trace function in ICE mode

#### (1) Trace memory and trace information

The ICE33 contains trace memory. When the program executes instructions in the trace range according to the trace mode, the trace information on each bus cycle is taken into this memory. The trace memory has the capacity to store information for 32768 cycles. When the trace information exceeds this capacity, the data is overwritten, the oldest data first, unless operating in single-delay trigger mode. Consequently, the trace information stored in the trace memory is always within 32768 cycles. The trace memory is cleared when a program is executed, starting to trace the new execution data.

Cycle	Address	Code	Unassemble	Address	Data	Clk	Type	TRC	File	Line	SourceCode
00001	000012E	5900	2	000012E	5900	2	DatM W I/O				
00004	-----	-----	-----	000012E	0000	2	DatM W I/O				
00009	0000064	E024	ext	0000064	0024	1	Inst # SR0H		(area.s)	00060	x16.h [TTR],#0
00050	0000066	E134	ext	0000066	0134	1	Inst # SR0H				
00057	0000068	0000	ld.h	0000068	[Tr0],#0	1	Inst # SR0H				
00056	000006A	0000	lst	000006A	0000	1	Inst # SR0H		(area.s)	00061	lst #
00055	-----	-----	-----	0000134	0000	2	DatM W I/O				
00054	000006C	0000	nop	000006C	0000	1	Inst # SR0H		(area.s)	00062	nop
00053	-----	-----	-----	0000020	0A20	7	VecR W SR0H				
00052	-----	-----	-----	0000022	0000	2	VecR W SR0H				
00051	0000024	0000	nop	0000024	0000	5	Inst # SR0H		(area.s)	00097	nop
00050	000002C	00C0	ret1	000002C	00C0	3	Inst # SR0H		(area.s)	00098	ret1
00049	000002E	0000	nop	000002E	0000	3	Inst # SR0H		(area.s)	00080	nop
00048	0000060	0000	nop	0000060	0000	4	Inst # SR0H		(area.s)	00062	nop
00047	0000060	0000	ld.w	0000060	[Tr0],#0	1	Inst # SR0H		(area.s)	00053	x16.w #0,(AREA_SEG
00046	0000070	E024	ext	0000070	0024	1	Inst # SR0H				
00045	0000072	E134	ext	0000072	0134	1	Inst # SR0H				
00044	0000074	0000	ld.b	0000074	[Tr0],#0	1	Inst # SR0H		(area.s)	00054	x16.b [TTR+2],#0

The following lists the trace information that is taken into the trace memory in every bus cycle. This list is corresponded to display in the [Trace] window.

**Cycle:** Trace cycle (decimal) The last information taken into the trace memory becomes 00000.

**Address:** CPU-instruction-fetch address (hexadecimal)  
"-----" is displayed for a non instruction-fetch access.

**Code:** Instruction code fetched by the CPU (hexadecimal)  
"----" is displayed for a non instruction-fetch access.

**Unassemble:** Disassembled content of the fetched instruction  
"-----" is displayed for a non instruction-fetch access.

**Address:** Address accessed by the CPU (hexadecimal)  
"-----" is displayed for an instruction-fetch access.

**Data:** Read/write data (hexadecimal)  
"----" is displayed for an instruction-fetch access.

**Clk:** Number of clocks used in the bus operation (1 to 7)  
"V" is displayed when 8 or more clocks are used.

**Type:** Bus operation type:  
Inst: Instruction fetch, VecR: Vector read, DatR: Data read, DatW: Data write  
StkR: Stack read, StkW: Stack write, DmaR: DMA read, DmaW: DMA write

Access size:

B: Byte access, H: Half word access, W: Word access

Memory type:

SRAM, DRAM, BROM (burst ROM), IRAM (internal RAM), I/O (internal I/O)  
DEBUG (for ICE development), ERR (others)

**TRC:** Input to TRCIN pin (denoted by L when low-level signal is input)

**T:** Trace trigger point (placed at the beginning of the line)  
Displayed only for the bus cycle that meets trace trigger conditions.

**File:** Source file name (displayed only when source display is selected by the tm command)

**Line:** Source line number (displayed only when source display is selected by the tm command)

**SourceCode:** Source code (displayed only when source display is selected by the tm command)

(2) Trace modes

Two trace modes are available, depending on the method for sampling trace information.

Table 16.8.7.1 Trace mode setup command

Function	Command
Setting trace mode and condition	tm

1. Normal trace mode

In this mode, the trace information on all bus cycles is taken into the trace memory during program execution. Therefore, until a break occurs, the trace memory always contains the latest information on bus cycles up to the one that is executed immediately beforehand.

2. Single delay trigger trace mode

In this mode as in other modes, trace is initiated by a start of program execution. When the trace trigger condition that is set by a command is met, trace is performed beginning from that point (trace trigger point) before being halted according to the next setting, which is also set by a command.

• If the trace trigger point is set to "start"

Trace is halted after sampling trace information for 32768 cycles beginning from the trace trigger point. In this case, the trace information at the trace trigger point is the oldest information stored in the trace memory. If the program stops before tracing all 32768 cycles, trace information on some cycles preceding the trace trigger point may be left in the trace memory within its capacity.

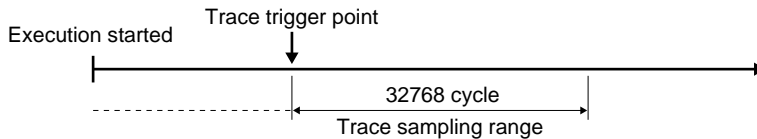


Fig. 16.8.7.1 Trace range when "start" is selected

• If the trace trigger point is set to "middle"

Trace is halted after sampling trace information for 16384 cycles beginning from the trace trigger point. In this case, the trace information of 16384 cycles before and after the trace trigger point are sampled into the trace memory.

If the program stops before tracing about 16384 cycles, trace information for the location 16384 cycles before the trace trigger point may be left in the trace memory, according to its capacity.

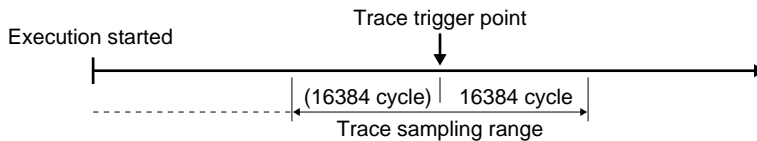


Fig. 16.8.7.2 Trace range when "middle" is selected

• If the trace trigger point is set to "end"

Trace is halted after sampling trace information at the trace trigger point. In this case, the trace information at the trace trigger point is the latest information stored in the trace memory.

If the program stops before tracing the trace trigger point, the system operates in the same way as in normal mode.

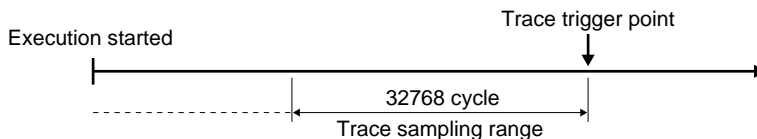


Fig. 16.8.7.3 Trace range when "end" is selected

If the program is halted in the middle of single delay trigger trace, bus cycles are traced from the beginning when trace is executed next.

In addition to the above mode settings, the tm command allows you to set a trace trigger condition (address, data pattern, or bus operation type).

**(3) Displaying and searching trace information**

The sampled trace information can be displayed in the [Trace] window by a command. If the [Trace] window is closed, the information is displayed in the [Command] window. In the [Trace] window, you can see the entire trace memory data by scrolling the window. The trace information can be displayed beginning from a specified cycle. The display contents are as described above.

Table 16.8.7.2 Command to display trace information

Function	Command
Displaying trace information	td

You also can specify a search condition and display the trace information that matches your specified condition. The search condition can be selected from the following:

1. Accessed memory address (or the entire memory space)
2. Bus operation type

When the above condition is specified, the db33 starts searching. When the trace information that matches the specified condition is found, the db33 displays the number of occurrences in the [Command] window. The search data is displayed in the [Trace] window (or in the [Command] window if the [Trace] window is closed).

Table 16.8.7.3 Command to search trace information

Function	Command
Search trace information	ts

The [Trace] window is cleared by executing a program. After a program terminates executing, use the above command to redisplay the trace information.

**(4) Saving trace information**

After the trace information is displayed in the [Trace] window using the td or ts commands, the trace information within the specified range can be saved to a file.

Table 16.8.7.4 Command to save trace information

Function	Command
Saving trace information	tf

**Precautions on trace in ICE mode**

- (1) After a single-step execution or a break occurs, information of the pre-fetched instructions that have not been executed are displayed. When the target program execution is suspended by a software PC break, the fetch cycle information of the brk instruction that was inserted for the software PC break is also displayed. (See example below.)
- (2) When the program starts a successive execution from an address set as a software PC break point, the ICE33 executes single-stepping before starting the successive execution. Therefore, redundant trace information pre-fetched by the single-stepping may be displayed. (See example below.)
- (3) For source-level step execution, the ICE33 repeats single-stepping internally. Therefore, a lot of pre-fetch information of all the steps will be displayed.
- (4) Because of the reason stated above, the execution time measured by the execution counter increases by the number of pre-fetch cycles.
- (5) Trace data for read/write of the internal RAM cannot be referred since the bus access is undetectable.
- (6) During data transfer by the high-speed DMA, data cannot be traced properly.

**Example of pre-fetch data display during step execution**

```

>m Sample execution program (software PC breaks are set at the addresses with "!")
!00080004 C020 ext 0x20 00010 xld.w %r8, SP_INI
00080006 6C08 ld.w %r8, 0x0
00080008 A081 ld.w %sp, %r8 00011 ld.w %sp, %r8 ; set SP
0008000A 6C08 ld.w %r8, 0x0 00012 ld.w %r8, GP_INI ; set gp
!0008000C C000 ext 0x0 00013 xcall main ; goto main
0008000E C000 ext 0x0
00080010 1C02 call 0x2
00080012 1EF9 jp 0xf9 00014 xjp BOOT ; infinity loop
--- main.c ---
00001 /* tst_main.c 1997.2.13 */
00002 /* C main program */
00003
00004 int i;
00005
00006 main()
00007 {
00080014 0200 pushn %r0
>g ... Successive execution from 0x80004
Break by software PC break ... Broken at 0x8000
>td ... Displays trace data
Cycle Address Code Unassemble Address Data Clk Type TRC
00007 0080004 C020 ext 0x20 ----- 1 Inst H SRAM
00006 0080006 6C08 ld.w %r8, 0x0 ----- 1 Inst H SRAM
00005 0080008 A081 ld.w %sp, %r8 ----- 1 Inst H SRAM ... Pre-fetch cycles
00004 008000A 6C08 ld.w %r8, 0x0 ----- 1 Inst H SRAM ... by single-stepping (2)
00003 0080008 A081 ld.w %sp, %r8 ----- 1 Inst H SRAM
00002 008000A 6C08 ld.w %r8, 0x0 ----- 1 Inst H SRAM
00001 008000C C400 brk ----- 1 Inst H SRAM ... Software break inst. (1)
00000 008000E C000 ext 0x0 ----- 1 Inst H SRAM
>s
>td
Cycle Address Code Unassemble Address Data Clk Type TRC
00005 008000C C000 ext 0x0 ----- 1 Inst H SRAM
00004 008000E C000 ext 0x0 ----- 1 Inst H SRAM
00003 0080010 1C02 call 0x2 ----- 1 Inst H SRAM
00002 0080012 1EF9 jp 0xf9 ----- 1 Inst H SRAM
00001 0080014 0200 pushn %r0 ----- 2 Inst H SRAM
00000 0080016 6C0B ld.w %r11, 0x0 ----- 1 Inst H SRAM
>s
>td
Cycle Address Code Unassemble Address Data Clk Type TRC
00002 0080014 0200 pushn %r0 ----- 1 Inst H SRAM ... Executed
00001 0080016 6C0B ld.w %r11, 0x0 ----- 1 Inst H SRAM ... Pre-fetch cycle (1)
00000 0080018 C000 ext 0x0 ----- 1 Inst H SRAM ... Pre-fetch cycle (1)
>

```

## Trace function in ICD mode

### (1) Trace memory and trace information

The ICD33 contains a trace memory that has the capacity to store information for 131072 cycles. The ICD33 stores the information of instruction execution cycles in the trace memory using the debugging signals output from the E0C33 chip and other methods.

Cycle	Address	Code	Unassemble	clk	Method	File	Line	SourceCode
000017	0002000	0FF8	est	0c1f98	002002	DPC		
000018	0002000	0FF9	est	0c1f99	002050	DPC		
000019	0002000	1C44	call	0c44	002075	DPC		
000014	0002010	6C0E	ld.u	7c12,000	002912	DPC (sys.c)	00001	lBytes = 0; /* no read now */
000013	0002010	C80E	est	0c	002920	DPC (sys.c)	00000	for (;;) {
000012	0002010	C401	est	0c401	002940	DPC		
000011	0002010	6C0F	ld.u	7c15,000	002960	DPC		
000010	0002010	6C14	ld.u	7c14,0c1	002970	DPC		
000009	0002010	680E	cmp	7c14,000	002992	DPC (sys.c)	00001	if (lReadBytes == 0) /* if require
000008	0002020	1000	freq	0c	003000	DPC		
000007	0002020	C300	est	0c300	003020	DPC (sys.c)	00000	if (READ_EOF == 1)
000006	0002020	C809	est	0c809	003040	DPC		
000005	0002020	2405	ld.u	7c5,[7c0]	003060	DPC		
000004	0002020	6815	cmp	7c5,0c1	003080	DPC		
000003	0002020	1000	freq	0c3	003100	DPC		
000002	0002030	2408	ld.u	7c11,[7c15]	003170	DPC (sys.c)	00101	ssize = READ_BUF[0];
000001	0002032	6800	cmp	7c11,000	003175	DPC (sys.c)	00102	if (ssize > 0)
000000	0002030	0c14	jrle	0c14	003190	DPC		

The following lists the trace information that is taken into the trace memory in every cycle. This list is corresponded to display in the [Trace] window.

- Cycle:** Trace cycle (decimal)  
The last information taken into the trace memory becomes 000000.
- Address:** CPU-instruction-execution address (hexadecimal)
- Code:** Instruction code executed by the CPU (hexadecimal)
- Unassemble:** Disassembled content of the instruction code
- Clk:** Number of clocks used for executing the instruction  
By default, the cumulative clock count from start of tracing is displayed. It can be changed so that the number of clocks for each executed instruction is displayed.
- Method:** Trace analytical method (to get the executed PC address)  
SPC: Analyzed with the start PC address  
TRG: Analyzed with the trigger address  
DPC: Analyzed with the DPCO signal  
RET: Analyzed with the call/ret statement  
MAP: Analyzed with the map information  
RTI: Analyzed with the reti statement  
---: Cannot be analyzed
- File:** Source file name (which includes the executed instruction)
- Line:** Source line number
- SourceCode:** Source code

### (2) Trace mode and trace condition

Two trace modes are available, depending on the trace range.

Table 16.8.7.5 Trace mode setup command

Function	Command
Setting trace mode and condition	tm

#### 1. All trace mode

In this mode, trace is initiated by a start of program execution. It continues until a break occurs when "with overwriting" is selected as the trace condition. If the trace memory becomes full, the oldest data will be overwritten with the new trace data. If the trace condition is set to "without overwriting", trace is terminated when the trace memory is full.



**2. Area trace mode**

Trace information is taken into the trace memory only when the program within the specified area is executed. The program execution can be suspended at the trace area end address. In this mode, the time measurement condition (all or area) can also be specified.

In addition to the trace mode above, the clock (Clk in the trace information) count method can be selected (accumulating or instruction units).

**(3) Displaying and searching trace information**

The sampled trace information can be displayed in the [Trace] window by a command. If the [Trace] window is closed, the information is displayed in the [Command] window.

Table 16.8.7.6 Command to display trace information

Function	Command
Displaying trace information	td

Furthermore, a search command is provided to display the trace information of the cycle that executes the specified address and the previous and subsequent cycles. The search data is displayed in the [Trace] window (or in the [Command] window if the [Trace] window is closed).

Table 16.8.7.7 Command to search trace information

Function	Command
Search trace information	ts

The [Trace] window is cleared by executing a program. After a program terminates executing, use the above command to redisplay the trace information.

**(4) Displaying and searching trace information**

The ICD33 allows trace data display without suspending the program execution. By clicking the [Display trace] button, the ICD33 suspends tracing and displays the sampled trace memory data to the [Trace] window. The trace operation can be resumed by clicking the [Resume trace] button.



The ts and tf commands cannot be used while the program is being executed. The [Display trace] button functions similar to the td command while the program execution is in break status.

**(5) Saving trace information**

After the trace information is displayed in the [Trace] window using the td or ts commands, the trace information within the specified range can be saved to a file.

Table 16.8.7.8 Command to save trace information

Function	Command
Saving trace information	tf

## ICD trace operation and precautions

The trace function in ICD mode is implemented using the method below.

The following four signals should be input to the ICE33 from the target CPU.

- DST0, DST1, DST2.....Signals that indicate the CPU execution status, such as sequential instruction execution, relative branch operation, absolute branch operation and idle status.
- DPCO.....Serial data signal that indicates the branch destination PC address. This signal is output when an absolute branch operation is performed.

The ICD33 reads this 4-bit information for up to 128K clocks in synchronization with the CPU clock.

The db33 gets the PC value by performing the following flow analysis using the above information and the disassemble information in the db33.

- DST0-2 = sequential instruction execution: +1 instruction
- DST0-2 = relative branch: The number of instructions to the branch destination is calculated from the disassemble information.
- DST0-2 = absolute branch: The branch destination is determined from the DPCO information.

However, this analysis cannot be done if the trace-start point and the corresponding PC value are not determined.

The db33 determines the PC value using the method below. The symbols in the Method column in the trace information represent the method used .

- Method: SPC Determined from the PC value at the start of program execution if it is fixed.  
(All trace mode without overwriting)
- Method: DPC Determined from the complete DPCO information of an absolute branch operation.
- Method: MAP Determined from the incomplete DPCO information of an absolute branch operation and the complement map information. DPCO information is output when the following absolute branch instruction is executed or by an interrupt vector jump operation.  
call %rb, call.d %rb, jp %rb, jp.d %rb, ret, ret.d, reti, int
- Method: TRG Determined by using the trigger address in area trace mode.
- Method: RET Determined from the correspondence between a call statement and a ret statement.
- Method: RTI Determined from the correspondence between an interrupt and a reti statement.

As a result, there are some restrictions as listed below.

### (1) Restriction in overwrite mode

When tracing a looped routine that repeats a relative branch, it will not be able to analyze until the PC value is determined.

As a solution for tracing such routines, there is a way to output DPCO information by generating an interrupt in several ms cycles using the 8-bit timer (see sample in "cc33\sample\icdtrc").

### (2) Restriction in area trace mode

Usually the hardware PC break function enables two break addresses. The area trace mode uses them as the trigger addresses, so they cannot be used for the hardware PC break function until area trace mode is cancelled.

### (3) Restriction in all trace mode without overwriting

The ICD firmware executes the following process when the program execution is started from a software PC break point.

1. Clears the software PC break point set at the execution start address.
2. Executes only the first instruction step.
3. Sets the start address as a software PC break again.
4. Executes the following instructions successively.

Therefore, the db33 cannot use the PC value at the start of program execution for analyzing.

When resuming execution of a program that has been suspended at a software PC break point, perform step execution to skip from the software PC break point before executing the program successively.

The following shows restrictions common to all modes:

(4) Restriction on absolute branch

If two or more absolute branches occur within a 27-clock period, the complete PC values cannot be determined. Although the db33 tries a recovery process using the map information and the call-ret nesting information, the PC values may not be analyzed.

(5) Restriction on the execution program

When a program is loaded by the lf or lh command, the db33 keeps the program information and uses the disassemble information for the PC value analysis. Therefore, there may be differences between the internal analysis information and the target program in the following cases:

- when the program to be executed has been stored in a ROM
- when the target program copies/moves the execution routine dynamically
- when the program area is modified using a db33 command

In those cases, load the necessary part of the program from the target to the db33 using the rm command.

(6) Simulated I/O

The simulated I/O function uses the software PC break, step execution and source step execution functions, so do not use it with the trace function simultaneously.

The following shows the precautions regarding to ICD33 hardware:

(7) Setting the ICD33

To use the trace function in ICD mode, the DIP switch SW4 on the ICD33 must be set to OPEN (upper position). Furthermore, the debugging signals required for tracing (DST0, DST1, DST2, DPCO) must be connected between the target board and the ICD33 using the 10-pin interface cable.

(8) Upper limit clock frequency for ICD trace function

The operating clock frequency is limited to 50 MHz when the ICD33 trace function is used. A higher frequency causes data error.

When using a 50 MHz or higher CPU operating clock, disable the ICD trace function using the DIP switch (SW4) in the ICD33. Furthermore, the speed of the E0C33 BCU (bus) should be set to 1/2 or less of the CPU core operating speed (using the #X2SPD pin).

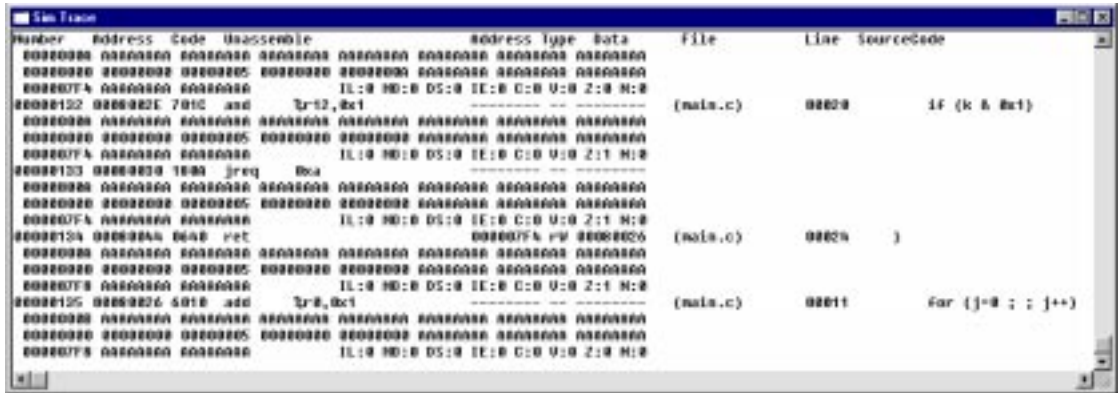
The ICD functions other than the trace function operate at the same speed as the bus and the maximum frequency is 40 MHz.

## Trace function in simulator mode

In the simulator mode, you can specify trace function ON/OFF, display method, and write to a file. When the trace function is turned on, the trace result is displayed on the screen or saved to a file every time an instruction is executed.

Table 16.8.7.9 Command to set trace mode

Function	Command
Turning trace mode on or off	tm



The following lists the trace information that is displayed on the screen in simulator mode:

### <1st line of each trace information>

- Number:** Executed instruction number (decimal).  
This is the executed instruction number after the CPU is reset or trace is turned on.
- Address:** Executed instruction address (hexadecimal).
- Code:** Instruction code (hexadecimal).
- Unassemble:** Disassembled content.
- Address:** Accessed memory address (hexadecimal).
- Type:** Bus operation type.  
rB: Byte data read, rH: Half word data read, rW: Word data read  
wB: Byte data write, wH: Half word data write, wW: Word data write
- Data:** Read/write data (hexadecimal).
- File:** Source file name (displayed only when source display is selected by the tm command).
- Line:** Source line number (displayed only when source display is selected by the tm command).
- SourceCode:** Source code (displayed only when source display is selected by the tm command).

### <Lines 2–4 of each trace information>

These lines are displayed when register option is selected with the tm command.

The register values appear in the order shown below.

```
R0  R1  R2  R3  R4  R5  R6  R7
R8  R9  R10 R11 R12 R13 R14 R15
SP  AHR  ALR          PSR (displayed in flag units)
```

Trace information is displayed in the [Trace] window when display to the window is selected. If the [Trace] window is closed, the information is displayed in the [Command] window.

When saving to a file is selected, the information is output to the file and is not displayed in the window.

Unlike in ICE mode, there is no need to input any specific command to display trace information. Trace information is displayed automatically according to the successive execution or single-stepping of a program. The [Trace] window allows you to see trace information for the last 255 instructions. The trace information for instructions beyond that are deleted.

## 16.8.8 Simulated I/O

The db33's simulated I/O function allows you to evaluate external input/output functions such as a serial interface by means of a standard input/output (stdin, stdout) or file input/output.

Table 16.8.8.1 Commands to set simulated I/O

Function	Command
Input setting	stdin
Output setting	stdout

### Input by stdin

Use the stdin command to set the following conditions:

- Break address
- Input buffer address (buffer size fixed to 65 bytes)
- Input device – [Simulated I/O] window or a file

After setting these conditions, execute the program in continuous mode.

#### When [Simulated I/O] window is selected

When a set break address is reached, the db33 opens the [Simulated I/O] window and waits for data to be input from the keyboard. When input data (up to 64 bytes) and hit the [Enter] key, the db33 writes the input data to a specified buffer, then restarts program execution at the address where it left off.

#### When a file is selected

If a file is selected, the db33 inputs data from the specified file to a specified buffer when it breaks. Then the db33 restarts program execution at the address where it left off. In this case, the [Simulated I/O] window is not opened.

### Output by stdout

Use the stdout command to set the following conditions:

- Break address
- Output buffer address (buffer size fixed to 65 bytes)
- Output device – [Simulated I/O] window or a file or both

After setting these conditions, execute the program in continuous mode.

#### When [Simulated I/O] window is selected

When a set break address is reached, the db33 opens the [Simulated I/O] window and displays the contents set in the buffer in the [Simulated I/O] window. Then the db33 restarts program execution at the address where it left off.

#### When a file is selected

If a file is selected, the db33 outputs the buffer contents to a specified file. Then the db33 restarts program execution at the address where it left off. In this case, the [Simulated I/O] window is not opened.

Data can be output to both the [Simulated I/O] window and the file.

## Program definitions for simulated I/O

Before the simulated I/O function described above can be used, you must write the following definitions in the program.

### Input/output buffer definition

Define the global buffers used by the db33 to input or output data in the following format:

Input buffer definition: `unsigned char READ_BUF[65]` (.comm READ\_BUF 65)

Output buffer definition: `unsigned char WRITE_BUF[65]` (.comm WRITE\_BUF 65)

For these buffer names, you can use any desired name that conforms to symbol name designations. Fix the buffer size to 65 bytes. When executing the `stdin` and `stdout` commands, use this symbol name to specify a buffer address.

When data is input, the size (1 to 64) of the actually input data is placed in `READ_BUF[0]`. If EOF is input, `READ_BUF[0]` is set to 0. The input data is stored in `READ_BUF[1]` and following elements.

When outputting data, write the size of the output data (1 to 64) to `WRITE_BUF[0]`, and the output data to `WRITE_BUF[1]` and following elements. To output EOF, write 0 to `WRITE_BUF[0]`.

Thus, a data stream of up to 64 bytes can be input and output between the db33 and the program.

### Data updating global label definition

Define the global labels shown below at a position where data is fed into the input buffer by the db33 and a position where data is output from the output buffer.

Input position: `.global READ_FLASH`  
`READ_FLASH:`

Output position: `.global WRITE_FLASH`  
`WRITE_FLASH:`

For these labels, you can use any desired name. When executing the `stdin` and `stdout` commands, use this symbol name to specify the break address.

In the C source, define these labels in the lower-level functions "write" and "read" (see Section 8.4) of the standard I/O library function.

For actual examples, refer to the sample programs and debugger command files installed in the `sample\simio\` directory.

When a break occurs at the `READ_FLASH` label, the db33 reads data that input to the [Simulated I/O] window or the file and load it to the defined input buffer. Then the db33 resume executing the program.

When a break occurs at the `WRITE_FLASH` label, the db33 output data that stored in the output buffer to the [Simulated I/O] window or the file, then resume executing the program.

## Precautions

Make sure the break addresses specified by the `stdin` and `stdout` commands do not overlap the software break addresses.

Since software breaks are used for this purpose inside the chip, the ROM area of the target board cannot be specified.

Use only ASCII characters for input and output. If binary data (0x0 and 0x1a in particular) is used, the db33 may operate erratically.

The part of the program to input/output data by `stdin` and/or `stdout` should be successively executed using the `go` command. Do not execute it by single-stepping and make sure that no break occurs in or around the part.

## 16.8.9 Operation of Flash Memory

The db33 supports flash memory on the target board and the ICE33 flash memory for free-run.

### Operation of the flash memory on the target board

The db33 comes provided with the utility and commands that write/erase the flash memory in the E0C33 chip or on the target board. They can be used in ICE (note), ICD and debug monitor mode.

Table 16.8.9.1 Flash memory operation commands

Function	Command
Setting flash memory	fls
Erasing flash memory	fle

Note: To use the commands with the ICE33, the ICE firmware must be Ver. 2.0 or higher.

Since the ICE33 is shipped with the firmware Ver. 1.x, update the ICE firmware using the program located in the "cc33\utility\ice33v20\" directory. For the update procedure, refer to the "readme.txt" of the updater.

Data should be written to the flash memory by the procedure shown below.

The examples in this section are extracted from "cc33\sample\dmt33004\led2.cmd".

For more information, refer to the "readme.txt" for the flash support utility fls33. ("fls33" and "readme.txt" can be installed using "cc33\utility\fls33\fls33vXX.exe".)

#### (1) Loading the flash routine

Load the flash routine (erase and write routines) into a memory such as the internal RAM using the If command.

Example:

```
If flsh\am29f800.srf ;load flash erase and write routine to I RAM area
```

Actual erasing/writing will be done by this routine.

The flash routine provided by Seiko Epson uses 0x40 to 0x7ff (2KB) of the internal RAM.

Note: Use the flash routine provided by Seiko Epson or create an original routine.

The Seiko Epson routines mainly support AMD type flash memories and can be installed by executing "cc33\utility\fls33\fls33.exe". The source and required files are included, so the routine can be modified if necessary.

#### (2) Setting the flash condition

Set the flash memory start and end addresses, and the entry addresses of the erase and write routines loaded in Step (1) into the db33.

Example:

```
fls ;flash set command
1 ;1:set 2:clear
200000 ;flash area start address is 0x200000
2fffff ;flash area end address is 0x2fffff
FLASH_ERASE ;flash erase routine top address
FLASH_LOAD ;flash load routine top address
```

### (3) Erasing the flash memory

Erase all or the specified sector range of the flash memory.

The contents of the flash memory change to 0xff.

Example:

```

fle          ;flash erase command
200000      ;flash control register is 0x200000
0           ;erase start block 0:all area 1-19:section
0           ;erase end block 1-19:sector if start block is 0 ,then this parameter is ignored.

```

First set the flash memory control register address. Normally it is the flash memory start address.

Then, specify the sector range to be erased. When the start and end numbers are specified as 0 and 0, the flash memory will be all erased. If 1 and 3 are specified, only sectors 1 to 3 will be erased. The number of sectors and sector size are different according to the device.

Be sure to execute the fle command after the fls command. To maintain the contents of the flash memory, specify -1 and 0 as the sector range. The process except for erasing will be performed.

### (4) Writing to the flash memory

The lf or lh command is used to write data to the flash memory.

Example:

```

lf led2.srf          ;load to 0x200000(flash)

```

Data for the start and end addresses set by the fls command in Step (1) is sent to the write routine to perform flash writing. Other data is written similar to writing to RAM. An error will result if a time-out occurs during writing or the flash memory has not been erased (not 0xff).

The ew and eh commands can be used for writing as well as the lf and lh commands. For the flash memory with 8-bit data width, the eb command can also be used.



## Operation of the ICE33 flash memory for free-run

The ICE33 in-circuit emulator contains flash memory. This memory is designed to allow data to be transferred to and from the ICE33 internal ROM emulation memory by a command.

The flash memory retains data even when the ICE33 is turned off. By writing the program, data, option data and map information under debug into the flash memory before turning off the power, you can call it up and continue debugging next time. Also, even when operating the ICE33 in free-run mode (in which a program is executed using only the ICE33), you may need to write the program into the flash memory.

The following operations can be performed on the flash memory:

### (1) Read from flash memory

Data is loaded from the flash memory into the internal ROM emulation memory.

### (2) Write to flash memory

Data in the internal ROM emulation memory is saved to the flash memory. Also, the contents of the parameter file can be written to the flash memory as necessary. After writing to the flash memory in this way, you can protect it against read and write.

### (3) Erasing flash memory

All contents of the flash memory are erased.

### (4) Displaying flash memory map information

The flash memory map, chip name, version of the parameter file used and other information are displayed.

The flash memory can only be altered in ICE mode.

Table 16.8.9.2 Commands to operate on flash memory

Function	Command
Reading from flash memory	lfl
Writing to flash memory	sfl
Erasing flash memory	efl
Displaying flash memory map information	maf

## \* Free-run of ICE33

When operating the ICE33 in free-run mode (with the program executed using only the ICE33), the ICE33 uses the data written in the flash memory. Therefore, before the ICE33 can be used in free-run mode, the entire program, data, and option data must be written into the flash memory. However, data not in the internal ROM cannot be saved.

To operate the ICE33 in free-run mode, set the ICE/RUN switch to the RUN position and turn on the power. During free-run, map breaks caused by operation in the program and data areas set by a parameter file are effective. When a map break occurs, the PC LED on the ICE33 stops and the EMU LED turns off. All other break settings are invalid because they cannot be written into the flash memory.

### 16.8.10 Other Functions

In addition to the primary functions described hitherto, the db33 supports several other useful functions as listed below. For details, refer to sections where each command is explained.

#### **Map display function (ma command)**

Displays map information, chip name, and parameter file version.

#### **Type conversion function (ct command)**

Returns input numeric values or character strings after converting them into different formats.

#### **Reverse conversion into an extended instruction (ext command)**

Specifying the address of an immediate-extended instruction with the ext instruction converts the instruction into an extended instruction format of the instruction extender including the extended immediate data and displays the results.

## 16.8.11 Big-Endian Support

The tools from the C compiler to the linker and the libraries support only the little-endian format. Be aware that the C compiler cannot create srf files that can be loaded to big-endian areas. However, data can be processed in big-endian format with the debugger.

### To specify big-endian area

The map information in the parameter file is used to set endian information to the debugger. To set the area format to big-endian, describe letter "B" after the <end address>. However, the E0C33 chip to be developed must be a model that supports big-endian format. Furthermore, the internal memory (ROM, RAM and I/O) cannot be set to big-endian. In addition to specify this parameter file at invocation of the db33, the endian control register in the E0C33 chip must be set correctly (refer to the "Technical Manual").

In simulator mode, the endian format is determined by the parameter file only.

Refer to Section 16.10 for details of the parameter file.

### Operations of debugging commands

(1) db, dh, dw, fb, fh, fw, mv, mvh, mvw, eb, eh, ew commands

These commands read/write data in byte, half word and word units according to the data type, so data is processed and displayed with the endian format of the area to be accessed.

(2) sy, sa, sw (@) commands

These commands read data in byte units regardless of the data type, and then configures the read data according to the specified data type. Data is displayed after swapped if the endian format and data type need it. Therefore, data is not displayed correctly if the endian settings of the BCU and the parameter file are different.

(3) lh, lf commands

These commands swap data according to the endian format and write in half word units. Therefore, a program created by the C compiler cannot be loaded to a big-endian area properly.

(4) sfl, lfl comands

The sfl command does not save the endian information. The lfl command makes the map information by adding the endian information in the parameter file to the information read from the ICE33. Therefore, the parameter file used when data was saved by the sfl command must be specified when invoking the debugger.

(5) Watched data

Data in the watched address set by the w command is handled in word units, so it is displayed according to the endian format of the area.

### Difference in simulator mode

In simulator mode, the address including the TTBR register can be set to big-endian. In this case, the trap table base address should be set as follows:

The contents to be written to 0x48134 in little-endian must be written to 0x48137.

The contents to be written to 0x48135 in little-endian must be written to 0x48136.

The contents to be written to 0x48136 in little-endian must be written to 0x48135.

The contents to be written to 0x48137 in little-endian must be written to 0x48134.

## 16.9 Command Reference

### 16.9.1 Command List

Table 16.9.1.1 Command list

Classification	Command	Function	Mode support				P No.
			ICD	ICE	SYM	MON	
Memory operation	<b>fb</b>	Fills memory area (byte units).	●	●	●	●	288
	<b>fh</b>	Fills memory area (half word units).	●	●	●	●	289
	<b>fw</b>	Fills memory area (word units).	●	●	●	●	290
	<b>db</b>	Dumps memory data (byte units).	●	●	●	●	291
	<b>dh</b>	Dumps memory data (half word units).	●	●	●	●	293
	<b>dw</b>	Dumps memory data (word units).	●	●	●	●	295
	<b>df</b>	Dumps memory data to file.	●	●	●	●	297
	<b>eb</b>	Enters memory data (byte units).	●	●	●	●	298
	<b>eh</b>	Enters memory data (half word units).	●	●	●	●	299
	<b>ew</b>	Enters memory data (word units).	●	●	●	●	300
	<b>mv</b>	Copies memory area (byte units).	●	●	●	●	301
	<b>mvh</b>	Copies memory area (half word units).	●	▲*1	●	●	302
	<b>mvw</b>	Copies memory area (word units).	●	▲*1	●	●	303
	<b>w</b>	Sets watch data address.	●	●	●	●	304
	<b>rm</b>	Reads target memory data.	●	–	–	–	305
	Register operation	<b>rd</b>	Displays register contents.	○	○	○	○
<b>rs</b>		Modifies register contents.	●	●	●	●	307
Program execution	<b>g</b>	Executes program successively.	●	●	●	●	308
	<b>s</b>	Executes program step.	●	●	●	●	310
	<b>n</b>	Executes program step with skip.	●	●	●	●	312
CPU reset	<b>rstc</b>	Cold-resets CPU.	●	●	●	●	313
	<b>rsth</b>	Hot-resets CPU.	●	●	●	●	314
Interrupt	<b>int</b>	Produces interrupt (simulator mode only).	–	–	●	–	315
Break	<b>bp</b>	Sets/cancels software PC breakpoint.	●	●	●	●	316
	<b>bs</b>	Sets software PC breakpoint.	●	●	●	●	320
	<b>bc</b>	Cancels software PC breakpoint.	●	●	●	●	321
	<b>bh</b>	Sets hardware PC breakpoint 1.	●	●	●	●	322
	<b>bhc</b>	Cancels hardware PC breakpoint 1.	●	●	●	●	323
	<b>bh2</b>	Sets hardware PC breakpoint 2.	●	▲	●	●	324
	<b>bhc2</b>	Cancels hardware PC breakpoint 2.	●	▲	●	●	325
	<b>bd</b>	Sets data break condition.	●	●	●	●	326
	<b>bsq</b>	Sets sequential break condition.	–	●	–	–	328
	<b>bl</b>	Displays all break conditions.	○	○	○	○	331
	<b>bac</b>	Clears all break conditions.	○	○	○	○	332
Program display	<b>u</b>	Sets disassemble display mode.	●	●	●	●	333
	<b>sc</b>	Sets source display mode.	●	●	●	●	335
	<b>m</b>	Sets mixed display mode.	●	●	●	●	337
	<b>ss</b>	Searches character string.	●	●	●	●	339
Symbol information	<b>sy</b>	Lists symbol information.	●	●	●	●	340
	<b>sa</b>	Registers symbol to [Symbol] window.	●	●	●	●	345
	<b>sd</b>	Deletes symbol from [Symbol] window.	●	●	●	●	348
	<b>sw</b>	Displays symbol information.	●	●	●	●	349
Load file	<b>lf</b>	Loads srf33 format file.	●	●	●	●	352
	<b>lh</b>	Loads Motorola S3 format file.	●	●	●	●	354
	<b>ld</b>	Loads debug information.	●	●	●	●	355
Flash memory operation	<b>fls</b>	Sets up target flash memory.	●	▲	–	●	356
	<b>fle</b>	Erases target flash memory.	●	▲	–	●	357
	<b>lfl</b>	Reads from ICE33 flash memory.	–	●	–	–	358
	<b>sfl</b>	Writes to ICE33 flash memory.	–	●	–	–	359
	<b>efl</b>	Erases ICE33 flash memory.	–	●	–	–	360
	<b>maf</b>	Displays ICE33 flash memory map.	–	○	–	–	361
Trace	<b>tm</b>	Sets trace mode.	○	○	○	–	362
	<b>td</b>	Displays trace information.	○	○	–	–	368
	<b>ts</b>	Searches trace information.	○	○	–	–	373
	<b>tf</b>	Saves trace information.	○	○	–	–	375
Simulated I/O	<b>stdin</b>	Simulates data input.	●	●	●	●	376
	<b>stdout</b>	Simulates data output.	●	●	●	●	377
Others	<b>com</b>	Executes command file.	●	●	●	●	378
	<b>cmw</b>	Executes command file with interval.	●	●	●	●	379
	<b>log</b>	Turns log output on or off.	●	●	●	●	380
	<b>od</b>	Dumps option data.	–	●	–	–	381
	<b>ct</b>	Converts/display data.	●	●	●	●	382
	<b>ext</b>	Converts into extended instruction format.	●	●	●	●	384
	<b>ma</b>	Displays map information.	●	●	●	●	386
	<b>md</b>	Sets debugger mode.	○	○	○	○	387
	<b>q</b>	Terminates debugger.	●	●	●	●	389
	<b>?</b>	Displays command usage.	●	●	●	●	390

Mode support: ● = Can be used (same function/display in all modes) ▲ = Supported by ICE firmware Ver. 2.0 or higher

○ = Can be used (function/display differ depending on the mode) – = Cannot be used

\*1: Data is copied in byte units if the ICE firmware version is lower than 2.0.

## 16.9.2 Commands to Operate Memory

### fb (fill byte)

[ICD / ICE / SIM / MON]

#### ■ Function

This command rewrites the entire contents of a specified memory area with the specified byte data.

#### ■ Formats

- (1) **fb** (guidance mode)
- (2) **fb <address1> <address2> <data>** (direct input mode)
  - <address1>: Start address of specified range (hexadecimal or symbol)
  - <address2>: End address of specified range (hexadecimal or symbol)
  - <data>: Write data (hexadecimal)
  - Conditions:  $0x0 \leq \text{address1} \leq \text{address2} \leq 0\text{xfffffff}$ ,  $0x0 \leq \text{data} \leq 0\text{xff}$

#### ■ Input examples

```
Format 1) >fb␣
Start address ? : 0000000␣ ...Start address is input. (symbol can be used)
End address ? : 000000f␣ ...End address is input. (symbol can be used)
Data pattern ? : 1␣ ...Write data is input.
>
```

\* Command execution can be canceled by entering the [Enter] key only.

```
Format 2) >fb 0000000 000000f 1␣
>
```

In both of these examples, the entire memory area from 0x0 to 0xf is rewritten with data 0x1.

```
addr +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +A +B +C +D +E +F
00000000 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
```

Using symbols)

```
>fb LABEL1 LABEL2 0
```

When rewriting the codes generated from an assembly source, line numbers can be used for specifying the addresses.

#### ■ Notes

- The addresses specified here must be within the range of 0 to 0xfffffff. An error results if this limit is exceeded. In Format 1, a guidance is displayed prompting you to input an address again. In Format 2, the command input is canceled.
  - Error: Address range (0-0xFFFFFFFF).
- An error results if the start address is larger than the end address.
  - Error: address1 > address2
- This command does not issue an error even if the address range specified for write includes an unused area. All valid locations except the unused area are rewritten with the specified data.
- Data must be input within a range of 8 bits (0 to 0xff). An error results if this limit is exceeded. In Format 1, a guidance is displayed prompting you to input data again. In Format 2, command input is canceled.
  - Error: Data range (0-0xFF).
- The fb command does not update the display contents of the [Memory] and [Source] windows. To update the display contents, redisplay the window with the display command or scroll the window in the vertical direction. The source displayed in the [Source] window remain unchanged even if the program area is rewritten.
- If a large memory area is rewritten at one time in ICE mode, a time-out error may occur, because such operation takes a long time.

**fh (fill half)****[ICD / ICE / SIM / MON]****■ Function**

This command rewrites the entire contents of a specified memory area with the specified half word data. The memory area is rewritten in the endian format specified with the parameter file (default: little endian).

**■ Formats**

- (1) **fh** (guidance mode)  
 (2) **fh <address1> <address2> <data>** (direct input mode)
- <address1>: Start address of specified range (hexadecimal or symbol)  
 <address2>: End address of specified range (hexadecimal or symbol)  
 <data>: Write data (hexadecimal)  
 Conditions:  $0x0 \leq \text{address1} \leq \text{address2} \leq 0\text{xfffffe}$  (half word boundary),  $0x0 \leq \text{data} \leq 0\text{xffff}$

**■ Input examples**

Format 1) &gt;fh␣

```
Start address ? : 0000000␣ ...Start address is input. (symbol can be used)
End address ? : 000000e␣ ...End address is input. (symbol can be used)
Data pattern ? : 1␣ ...Write data is input.
>
```

\* Command execution can be canceled by entering the [Enter] key only.

Format 2) >fh 0000000 000000e 1␣  
>

In both of these examples, the entire memory area from 0x0 to 0xf (0xe+1) is rewritten with data 0x0001 (when the area is set to little endian).

```
addr +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +A +B +C +D +E +F
00000000 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01 00
```

Using symbols)

&gt;fh LABEL1 LABEL2 0

When rewriting the codes generated from an assembly source, line numbers can be used for specifying the addresses.

**■ Notes**

- Since data is rewritten in units of 16 bits, specify half word boundary addresses (even addresses) for the area start and end addresses. If odd addresses are specified, a warning is generated and the LSBs of the specified addresses are rewritten to 0 as the area is set.  
Warning: Round down to multiple of 2.
- The addresses specified here must be within the range of 0 to 0xffffff. An error results if this limit is exceeded. In Format 1, a guidance is displayed prompting you to input an address again. In Format 2, the command input is canceled.  
Error: Address range (0-0xFFFFFFFF).
- An error results if the start address is larger than the end address.  
Error: address1 > address2
- This command does not issue an error even if the address range specified for write includes an unused area. All valid locations except the unused area are rewritten with the specified data.
- Data must be input within a range of 16 bits (0 to 0xffff). An error results if this limit is exceeded. In Format 1, a guidance is displayed prompting you to input data again. In Format 2, command input is canceled.  
Error: Data range (0-0xFFFF).
- The fh command does not update the display contents of the [Memory] and [Source] windows. To update the display contents, redisplay the window with the display command or scroll the window in the vertical direction. The source displayed in the [Source] window remain unchanged even if the program area is rewritten.
- If a large memory area is rewritten at one time in ICE mode, a time-out error may occur, because such operation takes a long time.

**fw (fill word)**

[ICD / ICE / SIM / MON]

**■ Function**

This command rewrites the entire contents of a specified memory area with the specified word data. The memory area is rewritten in the endian format specified with the parameter file (default: little endian).

**■ Formats**

- (1) **fw** (guidance mode)
- (2) **fw <address1> <address2> <data>** (direct input mode)
  - <address1>: Start address of specified range (hexadecimal or symbol)
  - <address2>: End address of specified range (hexadecimal or symbol)
  - <data>: Write data (hexadecimal)
  - Conditions:  $0x0 \leq \text{address1} \leq \text{address2} \leq 0\text{xfffffc}$  (word boundary),  $0x0 \leq \text{data} \leq 0\text{xffffffff}$

**■ Input examples**

```
Format 1) >fw␣
Start address ? : 000000␣ ...Start address is input. (symbol can be used)
End address ? : 00000c␣ ...End address is input. (symbol can be used)
Data pattern ? : 1␣ ...Write data is input.
>
```

\* Command execution can be canceled by entering the [Enter] key only.

```
Format 2) >fw 000000 00000c 1␣
>
```

In both of these examples, the entire memory area from 0x0 to 0xf (0xc+3) is rewritten with data 0x0001 (when the area is set to little endian).

```
addr +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +A +B +C +D +E +F
00000000 01 00 00 00 01 00 00 00 01 00 00 00 01 00 00 00
```

Using symbols)

```
>fw LABEL1 LABEL2 0
```

When rewriting the codes generated from an assembly source, line numbers can be used for specifying the addresses.

**■ Notes**

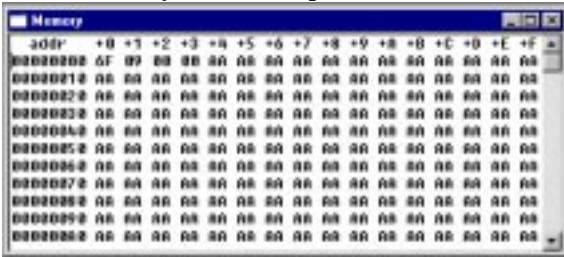
- Since data is rewritten in units of 32 bits, specify word boundary addresses for the area start and end addresses. If invalid addresses are specified, a warning is generated and the two least significant bits of the specified addresses are rewritten to 0 as the area is set.
  - Warning: Round down to multiple of 4.
- The addresses specified here must be within the range of 0 to 0xffffffff. An error results if this limit is exceeded. In Format 1, a guidance is displayed prompting you to input an address again. In Format 2, the command input is canceled.
  - Error: Address range (0-0xFFFFFFFF).
- An error results if the start address is larger than the end address.
  - Error: address1 > address2
- This command does not issue an error even if the address range specified for write includes an unused area. All valid locations except the unused area are rewritten with the specified data.
- Data must be input within a range of 32 bits (0 to 0xffffffff). An error results if this limit is exceeded. In Format 1, a guidance is displayed prompting you to input data again. In Format 2, command input is canceled.
  - Error: Data range (0-0xFFFFFFFF).
- The fw command does not update the display contents of the [Memory] and [Source] windows. To update the display contents, redisplay the window with the display command or scroll the window in the vertical direction. The source displayed in the [Source] window remain unchanged even if the program area is rewritten.
- If a large memory area is rewritten at one time in ICE mode, a time-out error may occur, because such operation takes a long time.

**db (dump byte)****[ICD / ICE / SIM / MON]****■ Function**

This command displays the contents of the memory in a 16 bytes/line hexadecimal dump format.

**■ Formats**

- (1) **db** (direct input mode)
  - (2) **db <address1>** (direct input mode)
  - (3) **db <address1> <address2>** (direct input mode)
- <address1>: Start address to display (hexadecimal or symbol)  
 <address2>: End address to display (hexadecimal or symbol)  
 Condition:  $0x0 \leq \text{address1} \leq \text{address2} \leq 0xfffffff$

**■ Display****(1) When [Memory] window is open**

In Format 1, the [Memory] window is redisplayed beginning with address 0x0.

In Formats 2 and 3, the [Memory] window is redisplayed in such a way that <address1> is displayed at the uppermost line.

Even when <address1> specifies somewhere in 16 addresses/line, data is displayed beginning with the top of that line. For example, even though you may have specified address 0x8 for <address1>, data is displayed beginning with address 0x0 as shown in the diagram. However, if an address near the uppermost part of the memory, such as 0xfffff0, is specified for <address1>, the last line displayed in the window in this case is 0xfffff0, that is, the specified address is not at the top of the window.

Since the [Memory] window can be scrolled to show the entire memory, specification of <address2> in Format 3 does not have any specific effect. In both Formats 2 and 3, the display you get is entirely the same.

**(2) When [Memory] window is closed**

Data is displayed in the [Command] window.

In Format 1, the db33 displays data for 16 lines (default) from address 0x0 before it stands by, waiting for a command input.

```
>db␣
  addr  +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +A +B +C +D +E +F
00000000 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA
00000010 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA
      :
000000F0 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA
>
```

In Format 2, the db33 displays data for 16 lines (default) from <address1> before it stands by, waiting for command input. If the line at address 0xfffff0 is displayed, the db33 waits for command input regardless of whether it has displayed all 16 lines.

If some midway address of a line is specified, columns preceding that address are left blank.

```
>db 7fff8␣
  addr  +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +A +B +C +D +E +F
0007FFF0
00080000 04 00 08 00 20 C0 08 6C 81 A0 08 6C 00 C0 03 1C
      :
000800F0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
>
```

"\*\*" indicates that the address is not mapped.



In Format 3, the db33 displays data from <address1> to <address2> before it waits for command input. Even when you specify a display range of more than 16 lines (default), display is halted at the 16th line (same as in Format 2).

```
>db 0 17.␣
  addr  +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +A +B +C +D +E +F
00000000 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA
00000010 AA AA AA AA AA AA AA AA
>
```

### (3) Number of lines displayed in the [Command] window

The number of lines displayed in the [Command] window by the db command every time it is executed is set to 16 by default. This default setting can be changed to any value within a range of 1 to 1,000 [lines] by the md command.

### (4) Logging

To save the command execution results to a log file, close the [Memory] window and display the results in the [Command] window. If the [Memory] window is opened, the display contents will not be saved in the file because the [Command] window does not display the results.

### (5) Successive display

Once you execute the db command, data can be displayed successively with the [Enter] key only until some other command is executed.

When you hit the [Enter] key, the [Memory] window is scrolled one full screen.

When displaying data in the [Command] window, data is displayed for the 16 lines (default) following the previously displayed address.

```
>db.␣
  addr  +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +A +B +C +D +E +F
00000000 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA
00000010 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA
:
:
000000F0 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA
>␣
  addr  +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +A +B +C +D +E +F
00000100 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA
00000110 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA
:
:
000001F0 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA
>␣
```

### (6) Using symbols

```
>db LABEL1.␣
```

#### ■ Notes

- Both the start and end addresses specified here must be within the range of 0 to 0xffffffff. An error results if this limit is exceeded.  
Error: Address range (0-0xFFFFFFFF).
- An error results if the start address is larger than the end address.  
Error: address1 > address2

**dh (dump half)****[ICD / ICE / SIM / MON]****■ Function**

This command displays the contents of the memory in a 8 half words/line hexadecimal dump format. Data is displayed in the endian format specified with the parameter file (default: little endian).

**■ Formats**

- (1) **dh** (direct input mode)
  - (2) **dh <address1>** (direct input mode)
  - (3) **dh <address1> <address2>** (direct input mode)
- <address1>: Start address to display (hexadecimal or symbol)  
 <address2>: End address to display (hexadecimal or symbol)  
 Condition:  $0x0 \leq \text{address1} \leq \text{address2} \leq 0\text{xfffffff}$

**■ Display****(1) When [Memory] window is open**

In Format 1, the [Memory] window is redisplayed beginning with address 0x0.

In Formats 2 and 3, the [Memory] window is redisplayed in such a way that <address1> is displayed at the uppermost line.

Even when <address1> specifies somewhere in 16 addresses/line, data is displayed beginning with the top of that line. For example, even though you may have specified address 0x8 for <address1>, data is displayed beginning with address 0x0 as shown in the diagram. However, if an address near the uppermost part of the memory, such as 0xfffff0, is specified for <address1>, the last line displayed in the window in this case is 0xfffff0, that is, the specified address is not at the top of the window.

Since the [Memory] window can be scrolled to show the entire memory, specification of <address2> in Format 3 does not have any specific effect. In both Formats 2 and 3, the display you get is entirely the same.

**(2) When [Memory] window is closed**

Data is displayed in the [Command] window.

In Format 1, the db33 displays data for 16 lines (default) from address 0x0 before it stands by, waiting for a command input.

```
>dh.
  addr  +0  +2  +4  +6  +8  +A  +C  +E
00000000 AAAA AAAA AAAA AAAA AAAA AAAA AAAA AAAA
00000010 AAAA AAAA AAAA AAAA AAAA AAAA AAAA AAAA
      :
000000F0 AAAA AAAA AAAA AAAA AAAA AAAA AAAA AAAA
>
```

In Format 2, the db33 displays data for 16 lines (default) from <address1> before it stands by, waiting for command input. If the line at address 0xfffff0 is displayed, the db33 waits for command input regardless of whether it has displayed all 16 lines.

If some midway address of a line is specified, columns preceding that address are left blank.

```
>dh 7fff8.
  addr  +0  +2  +4  +6  +8  +A  +C  +E
0007FFF0          **** *
00080000 0004 0008 C020 6C08 A081 6C08 C000 1C03
      :
000800F0 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
>
```

"\*\*\*\*" indicates that the address is not mapped.

In Format 3, the db33 displays data from <address1> to <address2> before it waits for command input. Even when you specify a display range of more than 16 lines (default), display is halted at the 16th line (same as in Format 2).

```
>dh 0 17.␣
  addr  +0  +2  +4  +6  +8  +A  +C  +E
00000000 AAAA AAAA AAAA AAAA AAAA AAAA AAAA AAAA
00000010 AAAA AAAA AAAA AAAA
>
```

### (3) Number of lines displayed in the [Command] window

The number of lines displayed in the [Command] window by the dh command every time it is executed is set to 16 by default. This default setting can be changed to any value within a range of 1 to 1,000 [lines] by the md command.

### (4) Logging

To save the command execution results to a log file, close the [Memory] window and display the results in the [Command] window. If the [Memory] window is opened, the display contents will not be saved in the file because the [Command] window does not display the results.

### (5) Successive display

Once you execute the dh command, data can be displayed successively with the [Enter] key only until some other command is executed.

When you hit the [Enter] key, the [Memory] window is scrolled one full screen.

When displaying data in the [Command] window, data is displayed for the 16 lines (default) following the previously displayed address.

```
>dh.␣
  addr  +0  +2  +4  +6  +8  +A  +C  +E
00000000 AAAA AAAA AAAA AAAA AAAA AAAA AAAA AAAA
00000010 AAAA AAAA AAAA AAAA AAAA AAAA AAAA AAAA
      :
000000F0 AAAA AAAA AAAA AAAA AAAA AAAA AAAA AAAA
>␣
  addr  +0  +2  +4  +6  +8  +A  +C  +E
00000100 AAAA AAAA AAAA AAAA AAAA AAAA AAAA AAAA
00000110 AAAA AAAA AAAA AAAA AAAA AAAA AAAA AAAA
      :
000001F0 AAAA AAAA AAAA AAAA AAAA AAAA AAAA AAAA
>␣
```

### (6) Using symbols

```
>dh LABEL1.␣
```

#### ■ Notes

- If any address is specified that is not aligned to half word boundaries, the LSB of the specified address is set to 0 as the display range is set.
- Both the start and end addresses specified here must be within the range of 0 to 0xffffffff. An error results if this limit is exceeded.  
Error: Address range (0-0xFFFFFFFF).
- An error results if the start address is larger than the end address.  
Error: address1 > address2

**dw (dump word)****[ICD / ICE / SIM / MON]****■ Function**

This command displays the contents of the memory in a 4 words/line hexadecimal dump format. Data is displayed in the endian format specified with the parameter file (default: little endian).

**■ Formats**

- (1) **dw** (direct input mode)
  - (2) **dw <address1>** (direct input mode)
  - (3) **dw <address1> <address2>** (direct input mode)
- <address1>: Start address to display (hexadecimal or symbol)  
 <address2>: End address to display (hexadecimal or symbol)  
 Condition:  $0x0 \leq \text{address1} \leq \text{address2} \leq 0xffffffff$

**■ Display****(1) When [Memory] window is open**

In Format 1, the [Memory] window is redisplayed beginning with address 0x0.

In Formats 2 and 3, the [Memory] window is redisplayed in such a way that <address1> is displayed at the uppermost line.

Even when <address1> specifies somewhere in 16 addresses/line, data is displayed beginning with the top of that line. For example, even though you may have specified address 0x8 for <address1>, data is displayed beginning with address 0x0 as shown in the diagram. However, if an address near the uppermost part of the memory, such as 0xfffff0, is specified for <address1>, the last line displayed in the window in this case is 0xfffff0, that is, the specified address is not at the top of the window.

Since the [Memory] window can be scrolled to show the entire memory, specification of <address2> in Format 3 does not have any specific effect. In both Formats 2 and 3, the display you get is entirely the same.

**(2) When [Memory] window is closed**

Data is displayed in the [Command] window.

In Format 1, the db33 displays data for 16 lines (default) from address 0x0 before it stands by, waiting for a command input.

```
>dw_↓
  addr      +0      +4      +8      +C
00000000 AAAAAAAAA AAAAAAAAA AAAAAAAAA AAAAAAAAA
00000010 AAAAAAAAA AAAAAAAAA AAAAAAAAA AAAAAAAAA
      :
000000F0 AAAAAAAAA AAAAAAAAA AAAAAAAAA AAAAAAAAA
>
```

In Format 2, the db33 displays data for 16 lines (default) from <address1> before it stands by, waiting for command input. If the line at address 0xfffff0 is displayed, the db33 waits for command input regardless of whether it has displayed all 16 lines.

If some midway address of a line is specified, columns preceding that address are left blank.

```
>dw 7fff8_↓
  addr      +0      +4      +8      +C
0007FFF0                ***** *****
00080000 00080004 6C08C020 6C08A081 1C03C000
      :
000800F0 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
>
```

"\*\*\*\*\*" indicates that the address is not mapped.

In Format 3, the db33 displays data from <address1> to <address2> before it waits for command input. Even when you specify a display range of more than 16 lines (default), display is halted at the 16th line (same as in Format 2).

```
>dw 0 17.␣
  addr      +0      +4      +8      +C
00000000 AAAAAAAAA AAAAAAAAA AAAAAAAAA AAAAAAAAA
00000010 AAAAAAAAA AAAAAAAAA
>
```

### (3) Number of lines displayed in the [Command] window

The number of lines displayed in the [Command] window by the dw command every time it is executed is set to 16 by default. This default setting can be changed to any value within a range of 1 to 1,000 [lines] by the md command.

### (4) Logging

To save the command execution results to a log file, close the [Memory] window and display the results in the [Command] window. If the [Memory] window is opened, the display contents will not be saved in the file because the [Command] window does not display the results.

### (5) Successive display

Once you execute the dw command, data can be displayed successively with the [Enter] key only until some other command is executed.

When you hit the [Enter] key, the [Memory] window is scrolled one full screen.

When displaying data in the [Command] window, data is displayed for the 16 lines (default) following the previously displayed address.

```
>dw.␣
  addr      +0      +4      +8      +C
00000000 AAAAAAAAA AAAAAAAAA AAAAAAAAA AAAAAAAAA
00000010 AAAAAAAAA AAAAAAAAA AAAAAAAAA AAAAAAAAA
:
:
000000F0 AAAAAAAAA AAAAAAAAA AAAAAAAAA AAAAAAAAA
>␣
  addr      +0      +4      +8      +C
00000100 AAAAAAAAA AAAAAAAAA AAAAAAAAA AAAAAAAAA
00000110 AAAAAAAAA AAAAAAAAA AAAAAAAAA AAAAAAAAA
:
:
00000200 AAAAAAAAA AAAAAAAAA AAAAAAAAA AAAAAAAAA
>␣
```

### (6) Using symbols

```
>dw LABEL1.␣
```

#### ■ Notes

- If any address is specified that is not aligned to word boundaries, the two least significant bits of the specified address are set to 0 as the display range is set.
- Both the start and end addresses specified here must be within the range of 0 to 0xfffffff. An error results if this limit is exceeded.  
Error: Address range (0-0xFFFFFFFF).
- An error results if the start address is larger than the end address.  
Error: address1 > address2

**df (dump file)****[ICD / ICE / SIM / MON]****■ Function**

This command outputs the contents of the memory in a 16 byte/line hexadecimal dump format as a text or binary file.

**■ Formats**

**df** (guidance mode)

**■ Input examples**

Following the guidance prompt, enter the address range to be written to a file, file format and the file name.

```
>df␣
Start address :00600000␣           ... Start address is input.
End   address :0061ffff␣           ... End address is input.
File type (1.Binary 2.Text) ... ? 2␣ ... File format is selected.
File name ? : df.txt␣             ... File name is input.
Processing 00600000-00607FFF address.
Processing 00608000-0060FFFF address.
Processing 00610000-00617FFF address.
Processing 00618000-0061FFFF address.
>
```

The specified file is created as follows (in case of text format):

```
addr  +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +A +B +C +D +E +F
00600000 30 31 32 33 34 35 36 37 38 39 41 42 43 44 45 46
00600010 30 31 32 33 34 35 36 37 38 39 41 42 43 44 45 46
00600020 30 31 32 33 34 35 36 37 38 39 41 42 43 44 45 46
:                                     :
```

**■ Notes**

- Both the start and end addresses specified here must be within the range of 0 to 0xffffffff. An error results if this limit is exceeded.  
Error: Address range (0-0xFFFFFFFF).
- An error results if the start address is larger than the end address.  
Error: address1 > address2

**eb (enter byte)**

[ICD / ICE / SIM / MON]

**■ Function**

This command rewrites the contents of the memory with the entered byte data (hexadecimal).

Data can be written to continuous memory locations beginning with a specified address, according to guidance mode.

**■ Formats**

- (1) **eb** (guidance mode)
- (2) **eb <address>** (guidance mode)  
 <address>: Start address from which to write data (hexadecimal or symbol)  
 Conditions:  $0x0 \leq \text{address} \leq 0xffffffff$ ,  $0x0 \leq \text{data} \leq 0xff$

**■ Input examples**

Addresses and current data are displayed as guidance.

```
Format 1) >eb␣
Enter address ? :0␣      ...Start address is input.
00000000 AA :00␣      ...Data is input in hexadecimal.
00000001 AA :q␣      ...Command is terminated.
>
```

```
Format 2) >eb 0␣
00000000 AA :00␣
00000001 AA :^␣      ...Returned to the previous address.
00000000 00 :␣      ...Input is skipped.
00000001 AA :q␣
>
```

In both of these examples, the content of address 0x0 is rewritten with data 0x00.

```
addr +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +A +B +C +D +E +F
00000000 00 AA AA AA AA AA AA AA AA AA AA AA AA AA AA
```

Using symbols)

```
>eb CAHR1␣
```

**■ Notes**

- The start address specified here must be within the range of 0 to 0xffffffff. An error results if this limit is exceeded. In Format 1, guidance is displayed to prompt you to input an address again. In Format 2, your command input is canceled.  
 Error: Address range (0-0xFFFFFFFF).
- For unused addresses in the memory map, data is marked by "\*\*\*" as displayed on the screen. Although it is possible to specify an unused address or display guidance, entering data in this case results in an error. If you encounter any address marked by "\*\*\*", hit the [Enter] key to skip that address or terminate the command.  
 Error: No map area.
- Data must be input by using a hexadecimal number in the range of 8 bits (0 to 0xff). An error results if this limit is exceeded. Guidance is displayed to prompt you to input data at the same address again.  
 Error: Data range (0-0xFF).
- During guidance-assisted input, the addresses are not changed even when you perform an operation to move to an address ahead of address 0 or an operation that results in exceeding address 0xffffffff.
- The eb command does not update the display contents of the [Memory] and [Source] windows. To update the display contents, redisplay the window with the display command or scroll the window in the vertical direction. The source displayed in the [Source] window remain unchanged even if the program area is rewritten.

**eh (enter half)****[ICD / ICE / SIM / MON]****■ Function**

This command rewrites the contents of the memory with the entered half word data (hexadecimal). Data can be written to continuous memory locations beginning with a specified address, according to guidance mode. The memory is rewritten in the endian format specified with the parameter file (default: little endian).

**■ Formats**

- (1) **eh** (guidance mode)
- (2) **eh <address>** (guidance mode)  
 <address>: Start address from which to write data (hexadecimal or symbol)  
 Conditions:  $0x0 \leq \text{address} \leq 0xfffffe$  (half word boundary),  $0x0 \leq \text{data} \leq 0xffff$

**■ Input examples**

Addresses and current data are displayed as guidance.

```
Format 1) >eh␣
Enter address ? :0␣           ...Start address is input.
00000000 AAAA :1234␣         ...Data is input in hexadecimal.
00000002 AAAA :q␣           ...Command is terminated.
>
```

```
Format 2) >eh 0␣
00000000 AAAA :1234␣
00000002 AAAA :^␣           ...Returned to the previous address.
00000000 1234 :␣           ...Input is skipped.
00000002 AAAA :q␣
>
```

In both of these examples, the content of address 0x0 (to 0x1) is rewritten with data 0x1234 (in case of little endian format).

```
addr  +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +A +B +C +D +E +F
00000000 34 12 AA AA AA AA AA AA AA AA AA AA AA AA AA
```

Using symbols)

```
>eh VAR1␣
```

**■ Notes**

- Since data is rewritten in units of 16 bits, specify a half word boundary address (even address) for the start address. If odd address is specified, a warning is generated and the LSB of the specified address is rewritten to 0.  
 Warning: Round down to multiple of 2.
- The start address specified here must be within the range of 0 to 0xfffff. An error results if this limit is exceeded. In Format 1, guidance is displayed to prompt you to input an address again. In Format 2, your command input is canceled.  
 Error: Address range (0-0xFFFFF).
- For unused addresses in the memory map, data is marked by "\*\*\*\*" as displayed on the screen. Although it is possible to specify an unused address or display guidance, entering data in this case results in an error. If you encounter any address marked by "\*\*\*\*", hit the [Enter] key to skip that address or terminate the command.  
 Error: No map area.
- Data must be input by using a hexadecimal number in the range of 16 bits (0 to 0xffff). An error results if this limit is exceeded. Guidance is displayed to prompt you to input data at the same address again.  
 Error: Data range (0-0xFFFF).
- During guidance-assisted input, the addresses are not changed even when you perform an operation to move to an address ahead of address 0 or an operation that results in exceeding address 0xfffff.
- The eh command does not update the display contents of the [Memory] and [Source] windows. To update the display contents, redisplay the window with the display command or scroll the window in the vertical direction. The source displayed in the [Source] window remain unchanged even if the program area is rewritten.



**ew (enter word)**

[ICD / ICE / SIM / MON]

**■ Function**

This command rewrites the contents of the memory with the entered word data (hexadecimal).

Data can be written to continuous memory locations beginning with a specified address, according to guidance mode. The memory is rewritten in the endian format specified with the parameter file (default: little endian).

**■ Formats**

- (1) **ew** (guidance mode)
- (2) **ew <address>** (guidance mode)  
 <address>: Start address from which to write data (hexadecimal or symbol)  
 Conditions:  $0x0 \leq \text{address} \leq 0xfffffc$  (word boundary),  $0x0 \leq \text{data} \leq 0xffffffff$

**■ Input examples**

Addresses and current data are displayed as guidance.

```
Format 1) >ew.␣
Enter address ? :0.␣          ...Start address is input.
00000000 AAAAAAAAA :12345678.␣ ...Data is input in hexadecimal.
00000002 AAAAAAAAA :q.␣      ...Command is terminated.
>
```

```
Format 2) >ew 0.␣
00000000 AAAAAAAAA :12345678.␣
00000002 AAAAAAAAA :.␣      ...Returned to the previous address.
00000000 12345678 :.␣      ...Input is skipped.
00000002 AAAAAAAAA :q.␣
>
```

In both of these examples, the content of address 0x0 (to 0x3) is rewritten with data 0x12345678 (in case of little endian format).

```
addr  +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +A +B +C +D +E +F
00000000 78 56 34 12 AA AA AA AA AA AA AA AA AA AA
```

Using symbols)

```
>ew i.␣
```

**■ Notes**

- Since data is rewritten in units of 32 bits, specify a word boundary address for the start addresses. If an invalid address is specified, a warning is generated and the two least significant bits of the specified address is rewritten to 0.  
 Warning: Round down to multiple of 4.
- The start address specified here must be within the range of 0 to 0xffffffff. An error results if this limit is exceeded. In Format 1, guidance is displayed to prompt you to input an address again. In Format 2, your command input is canceled.  
 Error: Address range (0-0xFFFFFFFF).
- For unused addresses in the memory map, data is marked by "\*\*\*\*\*" as displayed on the screen. Although it is possible to specify an unused address or display guidance, entering data in this case results in an error. If you encounter any address marked by "\*\*\*\*\*", hit the [Enter] key to skip that address or terminate the command.  
 Error: No map area.
- Data must be input by using a hexadecimal number in the range of 16 bits (0 to 0xffffffff). An error results if this limit is exceeded. Guidance is displayed to prompt you to input data at the same address again.  
 Error: Data range (0-0xFFFFFFFF).
- During guidance-assisted input, the addresses are not changed even when you perform an operation to move to an address ahead of address 0 or an operation that results in exceeding address 0xffffffff.
- The ew command does not update the display contents of the [Memory] and [Source] windows. To update the display contents, redisplay the window with the display command or scroll the window in the vertical direction. The source displayed in the [Source] window remain unchanged even if the program area is rewritten.

**mv (move)****[ICD / ICE / SIM / MON]****■ Function**

This command copies the contents of a specified memory area to another area in byte units.

**■ Formats**

- (1) **mv** (guidance mode)  
 (2) **mv <address1> <address2> <address3>** (direct input mode)
- <address1>: Start address of source area to be copied from (hexadecimal or symbol)  
 <address2>: End address of source area to be copied from (hexadecimal or symbol)  
 <address3>: Address of destination area to be copied to (hexadecimal or symbol)  
 Conditions:  $0x0 \leq \text{address1} \leq \text{address2} \leq 0\text{xffffffff}$ ,  $0x0 \leq \text{address3} \leq 0\text{xffffffff}$

**■ Input examples**

Format 1) `>mv␣`  
 Start address ? :000␣ ...Start address of the source area is input.  
 End address ? :0ff␣ ...End address of the source area is input.  
 Destination address ? :300␣ ...Destination address is input.  
 >

\* Command execution can be canceled by entering the [Enter] key only.

Format 2) `>mv 0 ff 300␣`  
 >

In both of these examples, the contents of the memory area from address 0x0 to 0xff are copied to locations following 0x300.

Using symbols)

`>mv LABEL1 LABEL2 LABEL3␣`

**■ Notes**

- The addresses specified here must be within the range of 0 to 0xffffffff. An error results if this limit is exceeded. In Format 1, guidance is displayed prompting you to input an address again. In Format 2, the command input is canceled.  
 Error: Address range (0-0xFFFFFFFF).
- An error results if the start address of the source area to be copied from is larger than its end address.  
 Error: address1 > address2
- If an unmapped area is included in the specified range of source addresses, the data in that area is assumed to be 0xf0 as data is copied from the source to the destination.
- If an unmapped area is included in the specified range of destination addresses, data is copied to only the effective locations, not including the unmapped area.
- If the destination address is smaller than the start address of the source area, data is first copied sequentially from the start address. Conversely, if the destination address is larger than the start address of the source area, data is first copied sequentially from the end address. Consequently, data is copied normally even when the destination address is set within the source area to be copied from.
- If the end address of the destination area exceeds 0xffffffff, the move operation is terminated when data is copied up to that address location.
- The mv command does not update the display contents of the [Memory] and [Source] windows. To update the display contents, redisplay the window with the display command or scroll the window in the vertical direction. The source displayed in the [Source] window remain unchanged even if the program area is rewritten.
- If a large memory area is copied all at once in ICE mode, a time-out error may occur due to the size of the operation.

**mvh (move half)**

[ICD / (ICE) / SIM / MON]

**■ Function**

This command copies the contents of a specified memory area to another area in half word units. Data is copied after converting into the set endian format if it is different between the source area and destination area.

**■ Formats**

- (1) **mvh** (guidance mode)
- (2) **mvh <address1> <address2> <address3>** (direct input mode)
  - <address1>: Start address of source area to be copied from (hexadecimal or symbol)
  - <address2>: End address of source area to be copied from (hexadecimal or symbol)
  - <address3>: Address of destination area to be copied to (hexadecimal or symbol)
  - Conditions:  $0x0 \leq \text{address1} \leq \text{address2} \leq 0\text{xffffffff}$ ,  $0x0 \leq \text{address3} \leq 0\text{xffffffff}$

**■ Input examples**

```
Format 1) >mvh␣
Start address ? :000␣ ...Start address of the source area is input.
End address ? :0ff␣ ...End address of the source area is input.
Destination address ? :300␣ ...Destination address is input.
>
```

\* Command execution can be canceled by entering the [Enter] key only.

```
Format 2) >mvh 0 ff 300␣
>
```

In both of these examples, the contents of the memory area from address 0x0 to 0xff are copied to locations following 0x300.

Using symbols)

```
>mvh LABEL1 LABEL2 LABEL3␣
```

**■ Notes**

- If any address is specified that is not aligned to half word boundaries, the LSB of the specified address is corrected to 0.
- The addresses specified here must be within the range of 0 to 0xffffffff. An error results if this limit is exceeded. In Format 1, guidance is displayed prompting you to input an address again. In Format 2, the command input is canceled.
  - Error: Address range (0-0xFFFFFFFF).
- An error results if the start address of the source area to be copied from is larger than its end address.
  - Error: address1 > address2
- If an unmapped area is included in the specified range of source addresses, the data in that area is assumed to be 0xf0 as data is copied from the source to the destination.
- If an unmapped area is included in the specified range of destination addresses, data is copied to only the effective locations, not including the unmapped area.
- If the destination address is smaller than the start address of the source area, data is first copied sequentially from the start address. Conversely, if the destination address is larger than the start address of the source area, data is first copied sequentially from the end address. Consequently, data is copied normally even when the destination address is set within the source area to be copied from.
- If the end address of the destination area exceeds 0xffffffff, the move operation is terminated when data is copied up to that address location.
- The mvh command does not update the display contents of the [Memory] and [Source] windows. To update the display contents, redisplay the window with the display command or scroll the window in the vertical direction. The source displayed in the [Source] window remain unchanged even if the program area is rewritten.
- If a large memory area is copied all at once in ICE mode, a time-out error may occur due to the size of the operation.
- When using this command in ICE mode, the ICE firmware must be Ver. 2.0 or higher.
  - If the version is less than 2.0, data will be copied in byte units.

**mvw (move word)****[ICD / (ICE) / SIM / MON]****■ Function**

This command copies the contents of a specified memory area to another area in word units. Data is copied after converting into the set endian format if it is different between the source area and destination area.

**■ Formats**

- (1) **mvw** (guidance mode)  
 (2) **mvw <address1> <address2> <address3>** (direct input mode)  
 <address1>: Start address of source area to be copied from (hexadecimal or symbol)  
 <address2>: End address of source area to be copied from (hexadecimal or symbol)  
 <address3>: Address of destination area to be copied to (hexadecimal or symbol)  
 Conditions:  $0x0 \leq \text{address1} \leq \text{address2} \leq 0\text{xffffffff}$ ,  $0x0 \leq \text{address3} \leq 0\text{xffffffff}$

**■ Input examples**

Format 1) `>mvw␣`  
 Start address ? :000␣ ...Start address of the source area is input.  
 End address ? :0ff␣ ...End address of the source area is input.  
 Destination address ? :300␣ ...Destination address is input.  
 >

\* Command execution can be canceled by entering the [Enter] key only.

Format 2) `>mvw 0 ff 300␣`  
 >

In both of these examples, the contents of the memory area from address 0x0 to 0xff are copied to locations following 0x300.

Using symbols)

`>mvw LABEL1 LABEL2 LABEL3␣`

**■ Notes**

- If any address is specified that is not aligned to word boundaries, the low-order 2 bits of the specified address are corrected to 0.
- The addresses specified here must be within the range of 0 to 0xffffffff. An error results if this limit is exceeded. In Format 1, guidance is displayed prompting you to input an address again. In Format 2, the command input is canceled.  
 Error: Address range (0-0xFFFFFFFF).
- An error results if the start address of the source area to be copied from is larger than its end address.  
 Error: address1 > address2
- If an unmapped area is included in the specified range of source addresses, the data in that area is assumed to be 0xf0 as data is copied from the source to the destination.
- If an unmapped area is included in the specified range of destination addresses, data is copied to only the effective locations, not including the unmapped area.
- If the destination address is smaller than the start address of the source area, data is first copied sequentially from the start address. Conversely, if the destination address is larger than the start address of the source area, data is first copied sequentially from the end address. Consequently, data is copied normally even when the destination address is set within the source area to be copied from.
- If the end address of the destination area exceeds 0xffffffff, the move operation is terminated when data is copied up to that address location.
- The mvw command does not update the display contents of the [Memory] and [Source] windows. To update the display contents, redisplay the window with the display command or scroll the window in the vertical direction.  
 The source displayed in the [Source] window remain unchanged even if the program area is rewritten.
- If a large memory area is copied all at once in ICE mode, a time-out error may occur due to the size of the operation.
- When using this command in ICE mode, the ICE firmware must be Ver. 2.0 or higher.  
 If the version is less than 2.0, data will be copied in byte units.

**w (watch)****[ICD / ICE / SIM / MON]****■ Function**

This command registers four memory locations as the watch data addresses. Memory contents equivalent to 4 bytes at each watch address are displayed in the [Register] window.

**■ Format**

**w** (guidance mode)

**■ Input example**

As guidance, the watch data addresses currently set at four locations are displayed sequentially beginning with the lowest address. Skip the watch address that you do not want to be modified by entering the [Enter] key only. Enter a new address for the watch address that you want to be modified. You can also use the [^] key (returns to the previous address) and the [q] key (quit).

```
>w.␣
Address1 00000000      :100.␣ ...Watch address is input.
Address2 00000004      :104.␣
Address3 00000008      :i.␣ ...Variable i is specified.
Address4 0000000C      :.␣ ...Not changed.
>
```

**■ Notes**

- When the db33 starts up, four locations at addresses 0, 4, 8, and 0xc are initially set as the watch data addresses.
- The addresses specified here must be within the range of 0 to 0xffffffff. An error results if this limit is exceeded. Guidance is displayed prompting you to input an address again.  
Error: Address range (0-0xFFFFFFFF).
- The watch data addresses are set in units of 4 bytes. A warning results if you specify an address that is outside the word boundary, with your specified address rounded down to a multiple of 4 (lower 2 bits are 0).  
Warning: Round down to multiple of 4.
- Be aware that a value is displayed as the watch data even if an address that is not mapped is registered. The value in this case is 0xf0 in simulator mode and indeterminate in ICE mode.
- The watch data is displayed in the endian format specified by the parameter file.

**rm (read memory)****[ICD]****■ Function**

This command reads the program on the target board to the debugger. The read data is used for trace analysis to reflect the change. This command should be used after the program on the target memory is modified using a memory operation command or another method.

**■ Formats**

- (1) **rm** (guidance mode)
- (2) **rm <address1> <address2>** (direct input mode)
  - <address1>: Start address of the area to be read (hexadecimal or symbol)
  - <address2>: End address of the area to be read (hexadecimal or symbol)
  - Condition:  $0x0 \leq \text{address1} \leq \text{address2} \leq 0\text{xffffffff}$

**■ Input examples**

```
Format 1) >rm↵
          Start address ? : 0↵      ...Start address is input.
          End   address ? : 7fe↵    ...End address is input.
          >
```

\* Command execution can be canceled by entering the [Enter] key only.

```
Format 2) >rm 0 7fe↵
          >
```

In both of these examples, the contents of the target memory from 0x0 to 0x7fe is read in the debugger.

Using symbols)

```
>rm LABEL1 LABEL2↵
```

**■ Notes**

- Both the start and end addresses specified here must be within the range of 0 to 0xffffffff. An error results if this limit is exceeded.
  - Error: Address range (0-0xFFFFFFFF).
- An error results if the start address is larger than the end address.
  - Error: address1 > address2

## 16.9.3 Commands to Operate on Register

### rd (register display)

[ICD / ICE / SIM / MON]

#### ■ Function

This command displays the contents of the registers, execution counter, and watch data.

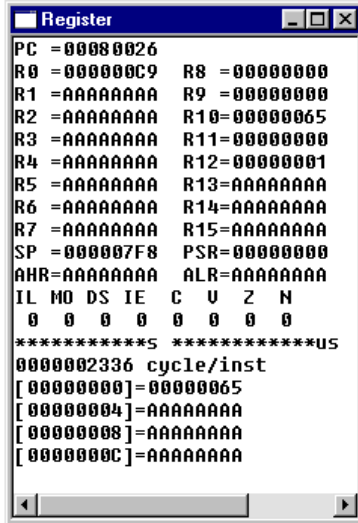
#### ■ Format

rd

#### ■ Display

##### (1) Contents of display

The following lists the contents displayed by this command.



```

Register
PC = 00000026
R0 = 000000C9  R8 = 00000000
R1 = AAAAAAAAAA  R9 = 00000000
R2 = AAAAAAAAAA  R10 = 00000065
R3 = AAAAAAAAAA  R11 = 00000000
R4 = AAAAAAAAAA  R12 = 00000001
R5 = AAAAAAAAAA  R13 = AAAAAAAAAA
R6 = AAAAAAAAAA  R14 = AAAAAAAAAA
R7 = AAAAAAAAAA  R15 = AAAAAAAAAA
SP = 000007F8  PSR = 00000000
AHR = AAAAAAAAAA  ALR = AAAAAAAAAA
IL MO DS IE C U Z N
0 0 0 0 0 0 0 0
*****S *****US
0000002336 cycle/inst
[00000000]=00000065
[00000004]=AAAAAAAA
[00000008]=AAAAAAAA
[0000000C]=AAAAAAAA

```

PC: Program counter  
R0–R15: General-purpose register  
SP: Stack pointer  
PSR: Processor status register  
AHR: Arithmetic operation high register  
ALR: Arithmetic operation low register  
IL: Interrupt level  
MO: MAC overflow flag  
DS: Dividend sign flag  
IE: Interrupt enable flag  
C: Carry flag  
V: Overflow flag  
Z: Zero flag  
N: Negative flag  
s, us: Execution time (effective only in ICE/ICD mode)  
cycle/inst: Execution cycle/instruction count  
[xxxxxxx]: Watch data at four locations

\* Watch data is always displayed even if it resides in an unused area, so be careful.

##### (2) When the [Register] window is open

When the [Register] window is open, all of the above contents are displayed in the [Register] window according to the program execution. When you use the rd command, the display of the [Register] window is updated.

##### (3) When [Register] window is closed

Data is displayed in the [Command] window in the same format as the [Register] window.

##### (4) Logging

To save the command execution results to a log file, close the [Register] window and display the results in the [Command] window. If the [Register] window is opened, the display contents will not be saved in the file because the [Command] window does not display the results.

**rs (register set)**

[ICD /ICE / SIM / MON]

**■ Function**

This command modifies the register values.

**■ Formats**

- (1) **rs** (guidance mode)
- (2) **rs <name> <data>** (direct input mode)
  - <name>: Register name or flag name
  - <data>: Data to be set (hexadecimal)

**■ Input examples**

Format 1) The name of each register and its current value are displayed as guidance. Skip the register that you do not want to be modified by hitting the [Enter] key only. Input a new value using a hexadecimal number for the register that you do want to be modified. You also can use the [^] key (returns to the previous address) and the [q] key (quit).

```
>rs␣
PC =00080004:␣
R0 =AAAAAAAA:0␣
R1 =AAAAAAAA:0␣
R2 =AAAAAAAA:0␣
:
R14=AAAAAAAA:0␣
R15=AAAAAAAA:0␣
IL = 0:␣
MO = 0:␣
DS = 0:␣
IE = 0:␣
C = 0:␣
V = 0:␣
Z = 0:␣
N = 0:␣
SP =AAAAAA8:1f00␣
AHR=AAAAAAAA:0␣
ALR=AAAAAAAA:0␣
>
```

After you execute the command, the [Register] window is updated to show the contents you have input.

If you used the [q] key to stop entering in the middle, the contents input up to that time are updated.

Format 2) >rs r0 0␣ ...R0 register is modified.  
>rs c 1␣ ...C flag is set.

**■ Notes**

- An error results if you input a value exceeding the effective bit size of the register/flag. In Format 1, a guidance is displayed prompting you to input data again. In Format 2, command input is canceled.  
Error: Invalid value.
- The set value of the PC is forcibly rounded down to 16-bit boundaries (LSB = 0). Even when an odd address is specified, no error is assumed.
- The set value of the SP is forcibly rounded down to 32-bit boundaries (low-order 2 bits = 0).



## 16.9.4 Commands to Execute Program

### g (go)

[ICD / ICE / SIM / MON]

#### ■ Function

This command executes the target program from the current PC address.

#### ■ Formats

- (1) **g** (direct input mode)
- (2) **g <address>** (direct input mode)  
 <address>: Temporary break address (hexadecimal, symbol or source line number)  
 Condition:  $0x0 \leq \text{address} \leq 0\text{xffffffe}$

#### ■ Operation

##### (1) Operation of format 1

>g.↓

- \* The same function as this command input can be performed by selecting the [Go] command on the [Run] menu or the [Go] button on the tool bar.



[Go] button

The target program is executed from the address indicated by the PC. Program execution is continued until it is made to break for one of the following causes:

- The set break condition is met
- The [Key break] button is clicked (not supported in debug monitor mode)
- A map break, etc., occurs

##### (2) Operation of format 2

In Format 2, a temporary break address can be specified. The break addresses set here remain effective until some other command is executed.

```
>g 80100.↓
>g main.c#20.↓      ...Specification with a line number
>g SUB1.↓          ...Specification with a symbol
```

- \* The same function as this command input can be performed by selecting the [Go to] command on the [Run] menu or the [Go to] button on the tool bar. In this case, the temporary break address must be specified by clicking on the desired address line in the [Source] window before the command can be executed. The address on the line where the cursor is located during execution is the temporary break address.



[Go to] button

The target program is executed from the address indicated by the PC. Program execution is made to break by one of the causes listed in (1) above or when an instruction at the specified temporary break address is fetched (the break occurs before executing that instruction).

##### (3) Entering the [Enter] key after break

When program execution breaks, the db33 stands by waiting for a command input after displaying a break status message (see Section 16.11.1).

Example: Break by temporary break.

>

When you hit the [Enter] key here, program execution is resumed from the PC address (break address). Temporary break address settings are also valid.

**(4) Window display by program execution****<ICE mode>**

In the initial debugger settings, the on-the-fly function is turned on.

During program execution, the PC, flag, and watch data contents in the [Register] window are updated in real time every 1–0.1 seconds by the on-the-fly function. All other contents are left blank.

If the [Register] window is closed, the above contents are displayed in the [Command] window. The on-the-fly function can be turned off by the md command. In this case, all numeric values in the [Register] window are left blank during program execution. The [Register] window is updated after a break.

The [Source] window is updated after a break in such a way that the break address is displayed within the window.

If the [Trace] window is open, the display contents are cleared as the program is executed. To update this display, use the td or the ts command after a break.

If the [Memory] window is open, the display contents are cleared as the program is executed. It is updated after a break.

**<Other modes>**

In other modes, the on-the-fly display above is disabled. For this reason, all numeric values in the [Register] window are left blank during program execution. The [Register] window is updated after a break.

The [Source] window is updated after a break in such a way that the break address is displayed within the window.

If the [Trace] window is open with the trace mode turned on, the trace results are successively displayed as the program is executed. If the [Trace] window is closed, the trace results are displayed in the [Command] window. No trace results are displayed if the trace mode is turned off or when the trace information is being saved to a file.

If the [Memory] window is open, the display contents are cleared as the program is executed. It is updated after a break.

**(5) Saving on-the-fly information to log file**

To save the on-the-fly information to a file, close the [Register] window and display the information in the [Command] window.

**(6) Execution counter**

The execution counter displayed in the [Register] window indicates the number of cycles/instructions executed or the execution time (only in ICE/ICD mode) of the target program. (See Section 16.8.5 for details.)

In the initial debugger settings, the execution counter is set to an integration mode. If this mode is changed to a reset mode by the md command, the execution counter is cleared to 0 each time the g command is executed. The counter is also reset simultaneously when execution is restarted by hitting the [Enter] key.

**■ Notes**

- The temporary break address must be specified within the range of the program memory area available for each microcomputer model. An error results if this limit is exceeded.
 

Error: Address range (0-0xFFFFFFFF).	...If an address exceeding 0xffffffff is specified.
Error: No map area.	...If an unused address is specified.
- If for a temporary breakpoint you specify a source line that does not have a real code by a line number or in the [Source] window, the temporary breakpoint is set at the address of the code that exists immediately after the specified line.
- If the current PC is a boot address (0x80000 or 0xc00000), the CPU is cold reset immediately before the db33 starts executing the program.

**s (step)**

[ICD / ICE / SIM / MON]

**■ Function**

This command single-steps the target program by executing one instruction at a time, beginning with the current PC position.

**■ Formats**

- (1) **s** (direct input mode)
- (2) **s <step>** (direct input mode)  
     <step>: Number of steps to be executed (decimal)  
     Condition:  $0 \leq \text{step} \leq 65535$

**■ Operation****(1) Units of single-stepping operation**

In this single-stepping operation, the program is executed in units of addresses or source codes – i.e., one address or source code at a time – depending on the [Source] window's display mode as shown below:

Disassemble display mode:	Address units
Mixed display mode:	Address units
Source display mode:	Source code units

**(2) Operation of Format 1**

>s␣

- \* The same function as this command input can be obtained by selecting the [Step] command on the [Run] menu or the [Step] button on the tool bar.



[Step] button

The program at the address indicated by the PC executes one step.

**(3) Operation of Format 2**

>s 10␣

The program executes a specified number of steps from the address indicated by the PC.

Program execution is terminated due to one of the break factors even before the specified number of steps is completed.

**(4) Hitting the [Enter] key after the end of execution**

When program execution is completed by stepping through instructions, the db33 stands by waiting for command input. If you hit the [Enter] key here, the db33 single-steps the program in the same way again.

**(5) Window display during single-stepping**

In the initial debugger settings, the display is updated every step as follows:

When the [Source] window is open, the underline designating the next address to be executed moves every step as the program is stepped through. The display contents of the [Register] window are also updated every step. This default display mode (all steps display mode) can be switched over by the md command so that the display contents are updated at only the last step in a specified number of steps (last step display mode).

Unlike in successive executions (g command), the [Register] window is not blanked even if the execution is not terminated immediately.

If the [Memory] window is open, the display contents are updated every step.

If the [Trace] window is open in ICE/ICD mode, the display contents are cleared as the program is executed. To update this display, use the td or the ts command after the specified steps are executed.

If the [Trace] window is open with the trace mode turned on in simulator mode, the trace results are successively displayed as the program is executed. If the [Trace] window is closed, the trace results are displayed in the [Command] window. No trace results are displayed if the trace mode is turned off or when the trace information is being saved to a file.

**(6) HALT and SLEEP states and interrupts**

In the ICE33, interrupts are disabled during single-stepping.

The halt and slp instructions are executed even during single-stepping, in which case the CPU is placed in a standby mode. The CPU can be released from the standby mode by generating an external interrupt or by pressing the [Key break] button.

**(7) Execution counter**

The execution counter displayed in the [Register] window indicates the number of cycles/instructions executed or the execution time (only in ICE/ICD mode) of the target program. (See Section 16.8.5 for details.)

In the initial debugger settings, the execution counter is set to an integration mode. If this mode is changed to a reset mode by the md command, the execution counter is cleared to 0 each time the s command is executed.

The counter is also reset simultaneously when execution is restarted by hitting the [Enter] key.

**■ Notes**

- The number of steps in Format 2 must be specified within the range of 0 to 65535. An error results if this limit is exceeded.  
Error: Step range (0-65535).
- If the current PC is a boot address (0x80000 or 0xc00000), the CPU is cold reset immediately before the db33 starts executing the program.
- When an infinity-loop such as "jp 0x0" is executed in source-level stepping, the step operation will not be terminated. In this case, forcibly terminate the execution using the [Key break] button.

**n (next)****[ICD / ICE / SIM / MON]****■ Function**

This command single-steps the target program by executing one instruction at a time beginning with the current PC position.

**■ Formats**

- (1) **n** (direct input mode)
- (2) **n <step>** (direct input mode)  
     <step>: Number of steps executed (decimal)  
     Condition:  $0 \leq \text{step} \leq 65535$

\* The same function as the command input in Format 1 can be obtained by selecting the [Next] command on the [Run] menu or the [Next] button on the tool bar.



[Next] button

**■ Operation**

This command basically operates in the same way as the s command.

However, the difference is that if a C source function call or assembly source subroutine call is encountered, each called function or subroutine is executed as one step. For other functions, refer to its explanation of s command.

**■ Note**

The number of steps in Format 2 must be specified within the range of 0 to 65535. An error results if this limit is exceeded.

Error: Step range (0-65535).

## 16.9.5 Commands to Reset CPU

### **rstc** (cold reset CPU)

[ICD / ICE / SIM / MON]

#### ■ Function

This command cold-resets the CPU.

#### ■ Format

**rstc** (direct input mode)

- \* The same function as this command input can be obtained by selecting the [Reset cold] command on the [Run] menu or the [Reset cold] button on the tool bar.



[Reset cold] button

#### ■ Contents of reset

When the CPU is reset, the internal circuits are initialized as follows:

##### (1) Internal registers of the CPU

R0–R15:	0xaaaaaaaa
PC:	Boot address (address pointed by the content of 0x80000 or 0xc0000)
SP:	0x0aaaaaaaa8
PSR:	0x00000000
AHR, ALR:	0xaaaaaaaa

##### (2) The execution counter is reset to 0.

##### (3) The [Source] and [Register] windows are redisplayed.

Because the PC is set to the boot address, the [Source] window is redisplayed beginning with that address.

The [Register] window is redisplayed with the internal registers initialized as described above.

The memory contents and the debugging status such as break and trace conditions are not modified.

Refer to the "Technical manual" of each model for the bus and I/O initial statuses when using in a mode other than simulator mode.

#### ■ Note

The function of the rstc command changes according to the debugger mode.

##### ICE mode

The process above is executed and the E0C33 chip is also reset. The target board is not reset.

##### ICD mode

The process above is executed and the E0C33 chip is also reset. The target board is not reset.

Furthermore, when the target system is in a free-run state, the rstc command suspends the program execution forcibly before resetting. The target system connected to the ICD33 enters a free-run state when the target board is reset. The rstc command can be used to suspend the program execution in this case.

##### Debug monitor mode

The rstc command functions the same as the rsth command. It does not reset the E0C33 chip and does not initialize the TTBR register.

##### Simulator mode

The boot address is determined by the MCU/MPU specification in the parameter file.

**rsth (hot reset CPU)**

[ICD / ICE / SIM / MON]

**■ Function**

This command hot-resets the CPU.

**■ Format**

**rsth** (direct input mode)

- \* The same function as this command input can be obtained by selecting the [Reset hot] command on the [Run] menu or the [Reset hot] button on the tool bar.



[Reset hot] button

**■ Contents of reset**

The registers and execution counter are initialized and the windows are redisplayed in the same way as for the rstc command.

The PC value (boot address) is specified by the TTBR register.

The memory contents and the debugging status such as break and trace conditions are not modified.

When using in ICE mode, the bus and I/O statuses are maintained.

## 16.9.6 Interrupt Command

### int (interrupt)

[SIM]

#### ■ Function

This command simulates the generation of interrupts.

When you specify an interrupt type with this command, your specified interrupt is generated the next time the db33 starts executing the program.

#### ■ Formats

(1) **int** (direct input mode)

(2) **int <type> <level>** (direct input mode)

<type>: Interrupt type (decimal)

<level>: Interrupt level (decimal)

Conditions:  $0 \leq \text{type} \leq 215$ ,  $0 \leq \text{level} \leq 15$

#### ■ Input examples

Format 1) >int␣

When the parameters are omitted, the db33 will issuer a NMI.

Format 2) >int 3 6␣

In Format 2, a number and level of a maskable interrupt can be set.

#### ■ Notes

- The int command can only be used in simulator mode.
- The interrupt type must be specified within the range of 0 to 215. An error results if this limit is exceeded.  
Error: Interrupt type (0-215).
- The interrupt level must be specified within the range of 0 to 15. An error results if this limit is exceeded.  
Error: Interrupt level (0-15).
- TTBR is effective even in simulator mode.



## 16.9.7 Commands to Set Breaks

### bp (break point set)

[ICD / ICE / SIM / MON]

#### ■ Function

This command sets and clears software PC breakpoints and displays the breakpoints set. When the PC matches the set address as the program is executed, the program breaks before executing the instruction at that address. Up to 16 addresses can be set as the breakpoints. Each breakpoint can be enabled and disabled as necessary.

#### ■ Formats

- (1) **bp** (guidance mode)
- (2) **bp <No.> <status> [<address>]** (direct input mode)
  - <No.>: Breakpoint No. (decimal)
  - <status>: Status to be set (1=set, 2=enable, 3=disable, 4=clear)
  - <address>: Break address (hexadecimal, symbol or line number; can be set when <status>=1)
  - Conditions:  $0 \leq \text{No.} \leq 15$ ,  $0x0 \leq \text{address} \leq 0xfffffff$  (16-bit boundary address)

#### ■ Input examples

##### (1) Displaying the breakpoints that have been set

Format 1 displays the contents of current settings.

```
>bp␣
 1 :00080014/E main          9 :*****/
 2 :00080028/E sub         10 :*****/
 3 :0008002C/D             11 :*****/
 4 :*****/                12 :*****/
 5 :*****/                13 :*****/
 6 :*****/                14 :*****/
 7 :*****/                15 :*****/
 8 :*****/                16 :*****/
Number ? :␣              ...Terminated by [Enter] key.
>
```

The contents of 16 breakpoints set are displayed.

"\*\*\*\*\*" indicates the breakpoints that have not been set yet.

The mark "/D" added at a break address denotes Disabled; the mark "/E" denotes Enabled.

A break occurs at the address marked by "/E".

The breakpoints whose addresses are set and which are enabled (/E) are prefixed by "!" when displayed in the [Source] window. However, if in the source display mode a breakpoint is set somewhere other than the beginning address of the source, it is marked with "?" instead of "!".

##### (2) Setting new break addresses

Format 1) First display the current settings as (1) above, then enter setting items as follows:

```
Number ? :4␣              ...Input a breakpoint No. to be set.
 1.set 2.enable 3.disable 4.clear ...? 1␣      ...Choose "1.set".
Break address ? :80030␣   ...Input the break address in hexadecimal.
>
```

Break addresses can be specified using line numbers or symbols.

```
Number ? :5␣
 1.set 2.enable 3.disable 4.clear ...? 1␣
Break address ? :main.c#24␣ ...Break address is input with line number.
>
```

To quit in the middle of guidance, input only the [Enter] key and nothing else.

Format 2) Input a command as follows:

```
>bp 4 1 80030␣           ...Breakpoint No. 4 is set at address 0x80030.
>bp 5 1 main.c#24␣      ...Breakpoint No. 5 is set at line number 24 of main.c.
```

An already set breakpoint number can be specified. In this case, the breakpoint is changed to a newly input address. All set breakpoints are enabled (/E).

A break address that is set for some other breakpoint number cannot be specified. If duplicate break addresses are specified, an error results.

When the above example is executed, the breakpoint list is modified as shown below:

```
>bp␣
 1 :00080014/E main          9 :*****/
 2 :00080028/E sub          10 :*****/
 3 :0008002C/D              11 :*****/
 4 :00080030/E              12 :*****/
 5 :00080032/E main.c#24    13 :*****/
 6 :*****/                  14 :*****/
 7 :*****/                  15 :*****/
 8 :*****/                  16 :*****/
Number ? :
```

### (3) Re-enabling a disabled breakpoint

No break occurs at breakpoints whose addresses are marked with /D (No. 3 in the above example). To re-enable such a disabled break, execute one of the commands shown below:

Format 1) After displaying the current settings, input the following command:

```
Number ? :3                ...Input a breakpoint No. to be re-enabled.
 1.set 2.enable 3.disable 4.clear ...? 2    ...Choose "2. enable".
>
```

An error results if you specify a breakpoint number that has no address set.

To quit in the middle of guidance, input only the [Enter] key and nothing else.

Format 2) Input the following command:

```
>bp 3 2␣                  ...Enables breakpoint No. 3.
>bp
 1 :00080014/E main          9 :*****/
 2 :00080028/E sub          10 :*****/
 3 :0008002C/E              11 :*****/
 4 :00080030/E              12 :*****/
 5 :00080032/E main.c#24    13 :*****/
 6 :*****/                  14 :*****/
 7 :*****/                  15 :*****/
 8 :*****/                  16 :*****/
Number ? :                ...Breakpoint No. 3 is marked with /E, indicating that it has been enabled.
```

### (4) Disabling a valid breakpoint

A break occurs at breakpoints whose addresses are marked with /E. To disable one of these breakpoints while leaving its set address intact, execute a command as shown below:

Format 1) After displaying the current settings, input the following command:

```
Number ? :2                ...Input the breakpoint number to be disabled.
 1.set 2.enable 3.disable 4.clear ...? 3    ...Choose "3. disable".
>
```

An error results if you specify a breakpoint number that has no address set.

To quit in the middle of guidance, input only the [Enter] key and nothing else.

Format 2) Input the following command:

```
>bp 2 3␣                  ...Disables breakpoint No. 2.
>bp
 1 :00080014/E main          9 :*****/
 2 :00080028/D sub          10 :*****/
 3 :0008002C/E              11 :*****/
 4 :00080030/E              12 :*****/
 5 :00080032/E main.c#24    13 :*****/
 6 :*****/                  14 :*****/
 7 :*****/                  15 :*****/
 8 :*****/                  16 :*****/
Number ? :                ...Breakpoint No. 2 is marked with /D, indicating that it has been disabled.
```

**(5) Clearing a breakpoint**

Format 1) After displaying the current settings, input the following command:

```
Number ? :4 ...Input a breakpoint number to be cleared.
1.set 2.enable 3.disable 4.clear ...? 4 ...Choose "4. clear".
>
```

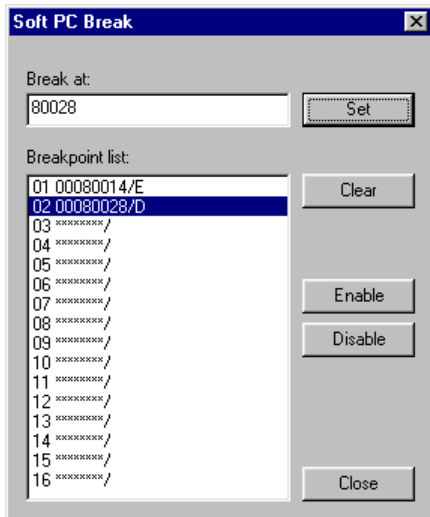
An error results if you specify a breakpoint number that has no address set.  
To quit in the middle of guidance, input only the [Enter] key and nothing else.

Format 2) Input the following command:

```
>bp 4 4 ...Clears breakpoint No. 4.
>bp
1 :00080014/E main          9 :*****/
2 :00080028/D sub          10 :*****/
3 :0008002C/E              11 :*****/
4 :*****/                  12 :*****/
5 :00080032/E main.c#24    13 :*****/
6 :*****/                  14 :*****/
7 :*****/                  15 :*****/
8 :*****/                  16 :*****/
Number ? : ...Breakpoint No. 4 has had its setting cleared.
```

**■ Setting breakpoints from menu**

Choose the [Soft PC...] command from the [Break] menu. The following dialog box will appear.



**Registering break addresses**

Enter an address in the [Break at] text box using a hexadecimal number, symbol or source line number, then press [Enter] or click the [Set] button. The entered address is registered to the break point list in ascending order from No.1.

**Clearing the break point**

Select the address to be cleared from the [Break list] box, then click the [clear] button.

**Enabling/disabling the break point**

To disable a break point, select the address from the list, then click the [Disable] button. The "/E" symbol changes to "/D" indicating that the break point is disabled. The [Enable] button switches the disabled break point (/D) to be enabled. (/E).

### ■ Notes

- Valid breakpoint No. are 1 to 16. Specifying a number greater than 16 results in an error.  
Error: Invalid value.
- Addresses can only be input if you choose "1. set". An error results if you input an address in the direct input mode (Format 2) and the selected item is not "1. set".  
Error: Invalid value.
- Since PC addresses constitute break conditions, breakpoints must always be set at 16-bit boundary addresses. If an odd address is specified, the LSB of the specified address is forcibly set to 0.
- If a source line that does not have real code is specified by a line number or in a window, a warning is issued. In this case, the breakpoint is set at the address of the code that exists immediately after the specified line.  
Warning: Invalid line, move to next valid line.
- The addresses must be specified within the range of the memory area available for each microcomputer model. An error results if this limit is exceeded. In Format 1, guidance is displayed prompting you to input an address again. In Format 2, the command input is canceled.  
Error: Address range (0-0xFFFFFFFF).                   ...If an address exceeding 0xffffffff is specified.  
Error: No map area.   ...If an unused address is specified.
- An error results if you set an address that has already been set.  
Error: Already exist input address.
- An error results if you attempt to enable/disable or cancel a break address that has not been set. In Format 1, guidance is displayed prompting you to input an address again.  
Error: Invalid break number.  
  
In Format 2, the error message is not displayed but the command input is canceled.
- Software PC breaks are implemented by embedding the BRK instruction. Therefore, software PC breaks cannot be used for the ROM on the target board where instructions cannot be embedded. In this case, use a hardware PC break.
- When setting a software PC break point to extended instructions with ext or delayed branch instructions, only the first address can be specified.  
ext   xxxx                   ... Can be set.                   jr\*.d   xxxx                   ... Can be set.  
ext   xxxx                   ... Cannot be set.                   Delayed instruction   ... Cannot be set.  
Extended instruction   ... Cannot be set.

**bs (break software)**

[ICD / ICE / SIM / MON]

**■ Function**

This command sets a software PC break address at a breakpoint number that has not yet been set.

**■ Formats**

- (1) **bs** (guidance mode)
- (2) **bs <address>** (direct input mode)  
 <address>: Break address (hexadecimal, symbol or line number)  
 Condition:  $0x0 \leq \text{address} \leq 0xfffffff$  (16-bit boundary address)

**■ Input examples**

```
Format 1) >bs␣
          Break address ? :80016␣      ...Specify an address in hexadecimal form.
          >bs
          Break address ? :main.c#24␣   ...Specify an address by a line number.
          >

Format 2) >bs 80016␣                  ...Specify an address in hexadecimal form.
          >bs main.c#24␣                ...Specify an address by a line number.
          >
```

The specified addresses are assigned to breakpoints that has not yet been set sequentially beginning with the smallest breakpoint number. The breakpoints set in this way are enabled (marked /E).

**■ Setting by using the tool bar button**

Software PC breakpoints can be set by using the [Soft PC break] button of the [Source] window in the same way as with the bs command explained above.



[Soft PC break] button

Click on an address line that you want set (by moving the cursor) in the [Source] window, then press the [Soft PC break] button. The selected line will be prefixed by "!" or "?" indicating that a breakpoint (enabled) is set there.

**■ Notes**

- The above operation results in an error if 16 breakpoints have already been set.  
 Error: Cannot set address any more.
- Since PC addresses constitute break conditions, breakpoints must always be set at 16-bit boundary addresses. If an odd address is specified, the LSB of the specified address is forcibly handled as 0.
- If a source line that does not have real code is specified by a line number or in a window, a warning is issued (when the [Soft PC break] button is used, warning message is not displayed). In this case, the breakpoint is set at the address of the code that exists immediately after the specified line.  
 Warning: Invalid line, move to next valid line.
- The addresses must be specified within the range of the memory area available for each microcomputer model. An error results if this limit is exceeded. In Format 1, guidance is displayed prompting you to input an address again. In Format 2, the command input is canceled.  
 Error: Address range (0-0xFFFFFFFF). ...If an address exceeding 0xffffffff is specified.  
 Error: No map area. ...If an unused address is specified.
- An error results if you set an address that has already been set.  
 Error: Already exist input address.
- Software PC breaks are implemented by embedding the BRK instruction. Therefore, software PC breaks cannot be used for the ROM on the target board where instructions cannot be embedded. In this case, use a hardware PC break.
- When setting a software PC break point to extended instructions with ext or delayed branch instructions, only the first address can be specified.  
 ext xxxx ... Can be set. jr\*.d xxxx ... Can be set.  
 ext xxxx ... Cannot be set. Delayed instruction ... Cannot be set.  
 Extended instruction ... Cannot be set.

**bc (break clear)****[ICD / ICE / SIM / MON]****■ Function**

This command cancels the software PC breakpoints set by using the bp command, bs command, the [Soft PC...] command on the [Break] menu, or the [Soft PC break] button.

**■ Format**

**bc <address>** (direct input mode)  
 <address>: Break address (hexadecimal, symbol or line number)  
 Condition: Only the addresses set as breakpoints can be specified for <address>.

**■ Input examples**

```
>bc 80016␣           ...Specify an address in hexadecimal form.
>bc main.c#24␣       ...Specify an address by a line number.
>
```

**■ Clearing by using the tool bar button**

Software PC breakpoints can be cleared by using the [Soft PC break] button of the [Source] window in the same way as explained above.



[Soft PC break] button

Click on an address line that has had a breakpoint set (one that is prefixed by "!" or "?") in the [Source] window, then press the [Soft PC break] button. The selected break address will be cleared.

**■ Notes**

- Breakpoints can be cleared by using the bp command or the [Soft PC] command on the [Break] menu. (Refer to the bp command.)
- If an odd address is specified, its LSB is forcibly handled as 0.
- If a source line that does not have real code is specified by a line number or in the [Source] window, a warning is issued (when the [Soft PC break] button is used, warning message is not displayed). In this case, the breakpoint that exists immediately after the specified line is cleared.  
 Warning: Invalid line, move to next valid line.
- An error results if you specify a break address that has not been set.  
 Error: Invalid break number.

**bh (break hardware)**

[ICD / ICE / SIM / MON]

**■ Function**

This command sets hardware PC breakpoint 1 and displays the breakpoint set.

**■ Formats**

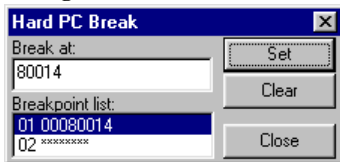
- (1) **bh** ...Display (direct input mode)
- (2) **bh <address>** ...Setting (direct input mode)
  - <address>: Break address (hexadecimal, symbol or line number)
  - Condition:  $0x0 \leq \text{address} \leq 0xfffffff$  (16-bit boundary address)

**■ Input examples**

- Format 1) >bh␣  
 Address :\*\*\*\*\*␣ ...If no breakpoint is set  
 >bh␣  
 Address :80032 main.c#24␣ ...If a breakpoint is set by a line number
- Format 2) >bh 80016␣ ...Specify an address in hexadecimal form.  
 >bh main.c#24␣ ...Specify an address by a line number.

This breakpoint can only be set at one address location. The last address specified is valid.

If the [Source] window is open, the address which has had a hardware PC breakpoint set is marked with "!" immediately after it. If in the source display mode a breakpoint is set somewhere other than the beginning address of the source, the address is marked with "?" instead of "!".

**■ Setting from the menu**

This dialog box appears on the screen when you select the [Hard PC...] command from the [Break] menu. This dialog box is used for both hardware PC breaks 1 and 2 set by the bh and bh2 commands. To set a hardware PC breakpoint, enter the desired address in the text box using a hexadecimal number, symbol or source line number, then press [Enter] or click the [Set] button.

**■ Setting by using the tool bar button**

A hardware PC breakpoint can be set using the [Hard PC break] button of the [Source] window in the same way as with the bh command explained above.



[Hard PC break] button

Click on an address line that you want set (by moving the cursor) in the [Source] window, then press the [Hard PC break] button. The selected line is set as a hardware PC breakpoint.

**■ Notes**

- The addresses must be specified within the range of the memory area available for each microcomputer model. An error results if this limit is exceeded. In Format 1, guidance is displayed prompting you to input an address again. In Format 2, the command input is canceled.
  - Error: Address range (0-0xFFFFFFFF). ...If an address exceeding 0xffffffff is specified.
  - Error: No map area. ...If an unused address is specified.
- When setting a hardware PC break point to extended instructions with ext or delayed branch instructions, only the first address can be specified.
 

ext xxxx	... Can be set.	jr*.d xxxx	... Can be set.
ext xxxx	... Cannot be set.	Delayed instruction	... Cannot be set.
Extended instruction	... Cannot be set.		
- The hardware PC break function is disabled when the area trace function is set in ICD mode. However, the set address is maintained and it will be enabled when the area trace function is cancelled.

**bhc (break hardware clear)****[ICD / ICE / SIM / MON]****■ Function**

This command cancels the hardware PC breakpoint 1 set using the bh command or the [Hard PC...] command on the [Break] menu.

**■ Format**

**bhc** (direct input mode)

**■ Input example**

```
>bhc␣           ...Cancels the hardware PC breakpoint 1 set.
>
```

**■ Clearing from the menu**

When you choose the [Hard PC...] command from the [Break] menu, a dialog box appears (see the bh command). To clear the set hardware PC breakpoint 1, select break address 01 and then click the [Clear] button.

**■ Clearing by using the tool bar button**

Hardware PC breakpoints can be cleared using the [Hard PC break] button of the [Source] window in the same way as explained above.



[Hard PC break] button

Click on an address line that has had a breakpoint set (indicated by "!" or "?" after the address) in the [Source] window, then press the [Hard PC break] button. The selected break address will be cleared.



**bh2 (break hardware 2)**

[ICD / (ICE) / SIM / MON]

**■ Function**

This command sets hardware PC breakpoint 2 and displays the breakpoint set.

**■ Formats**

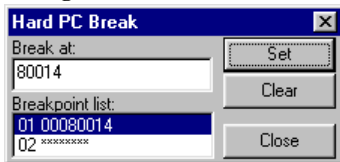
- (1) **bh2** ...Display (direct input mode)
- (2) **bh2 <address>** ...Setting (direct input mode)  
 <address>: Break address (hexadecimal, symbol or line number)  
 Condition:  $0x0 \leq \text{address} \leq 0\text{xfffffff}$  (16-bit boundary address)

**■ Input examples**

- Format 1) >bh2␣  
 Address :\*\*\*\*\*␣ ...If no breakpoint is set  
 >bh2␣  
 Address :80032 main.c#24␣ ...If a breakpoint is set by a line number
- Format 2) >bh2 80016␣ ...Specify an address in hexadecimal form.  
 >bh2 main.c#24␣ ...Specify an address by a line number.

This breakpoint can only be set at one address location. The last address specified is valid.

If the [Source] window is open, the address which has had a hardware PC breakpoint set is marked with "!" immediately after it. If in the source display mode a breakpoint is set somewhere other than the beginning address of the source, the address is marked with "?" instead of "!".

**■ Setting from the menu**

This dialog box appears on the screen when you select the [Hard PC...] command from the [Break] menu. This dialog box is used for both hardware PC breaks 1 and 2 set by the bh and bh2 commands. To set a hardware PC breakpoint, enter the desired address in the text box using a hexadecimal number, symbol or source line number, then press [Enter] or click the [Set] button.

**■ Setting by using the tool bar button**

A hardware PC breakpoint can be set using the [Hard PC break] button of the [Source] window in the same way as with the bh2 command explained above.



[Hard PC break] button

Click on an address line that you want set (by moving the cursor) in the [Source] window, then press the [Hard PC break] button. The selected line is set as a hardware PC breakpoint.

**■ Notes**

- The addresses must be specified within the range of the memory area available for each microcomputer model. An error results if this limit is exceeded. In Format 1, guidance is displayed prompting you to input an address again. In Format 2, the command input is canceled.
 

Error: Address range (0-0xFFFFFFF).	...If an address exceeding 0xfffffff is specified.
Error: No map area.	...If an unused address is specified.
- When setting a hardware PC break point to extended instructions with ext or delayed branch instructions, only the first address can be specified.
 

ext xxxx	... Can be set.	jr*.d xxxx	... Can be set.
ext xxxx	... Cannot be set.	Delayed instruction	... Cannot be set.
Extended instruction	... Cannot be set.		
- The hardware PC break function is disabled when the area trace function is set in ICD mode. However, the set address is maintained and it will be enabled when the area trace function is cancelled.
- When using this command in ICE mode, the ICE firmware must be Ver. 2.0 or higher.

**bhc2 (break hardware 2 clear)****[ICD / (ICE) / SIM / MON]****■ Function**

This command cancels the hardware PC breakpoint 2 set using the bh2 command or the [Hard PC...] command on the [Break] menu.

**■ Format**

**bhc2** (direct input mode)

**■ Input example**

```
>bhc2␣          ...Cancels the hardware PC breakpoint 2 set.
>
```

**■ Clearing from the menu**

When you choose the [Hard PC...] command from the [Break] menu, a dialog box appears (see the bh2 command). To clear the set hardware PC breakpoint 2, select break address 02 and then click the [Clear] button.

**■ Clearing by using the tool bar button**

Hardware PC breakpoints can be cleared using the [Hard PC break] button of the [Source] window in the same way as explained above.



[Hard PC break] button

Click on an address line that has had a breakpoint set (indicated by "!" or "?" after the address) in the [Source] window, then press the [Hard PC break] button. The selected break address will be cleared.

**■ Notes**

When using this command in ICE mode, the ICE firmware must be Ver. 2.0 or higher.

**bd (data break)**

[ICD / ICE / SIM / MON]

**■ Function**

This command sets and clears data break conditions and displays the conditions set.

This command allows you to specify the following break conditions:

1. Memory address to be accessed (one location)
2. Memory read/write (three conditions: read, write, or read or write)

The program breaks after completing a memory access that satisfies the above conditions.

**■ Formats**

- (1) **bd** (guidance mode)
- (2) **bd <mode> <address> {r | w | \*}** (direct input mode)

<mode>: Set/clear specification (1=set, 2=clear)

<address>: address (hexadecimal or symbol)

r, w, \*: Access condition (enter either one)

r: Read

w: Write

\*: Read or write

Condition:  $0x0 \leq \text{address} \leq 0xfffffff$

**■ Input examples****(1) Displaying data break conditions**

Format 1 displays the contents of current settings.

```
>bd␣
Address   : *****
R/W/*    : -
1. set   2. clear ... ? ␣      ...Terminated by [Enter] key.
>
```

Shown above is an example in which data break conditions have not been set yet.

**(2) Setting data break conditions**

Format 1) After displaying the current settings as described in (1), input the following command:

```
1. set   2. clear ... ? 1␣      ...Choose "1. set".
Address   : 100␣                ...Input an address in hexadecimal form.
R/W (R, W, *) : *␣              ...Input access conditions (* for R/W).
>
```

Addresses can be specified using a symbol.

```
1. set   2. clear ... ? 1␣
Address   : i␣                  ...Input an address using a symbol.
R/W (R, W, *) : w␣
>
```

To quit in the middle of guidance, input only the [Enter] key and nothing else.

Format 2) Input the following command:

```
>bd 1 100 *␣                    ...Set address to 0x100 and access condition to R/W.
>bd 1 i w␣                      ...Set address to variable i and access condition to W.
```

If conditions have already been set, the previous conditions are changed to the newly input conditions.

**(3) Clearing data break conditions**

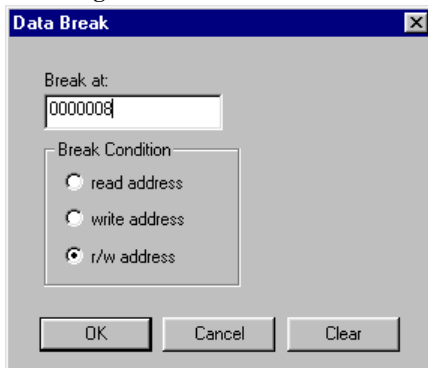
Format 1) After displaying the current settings, input the command shown below:

```
>bd
Address   : 00000000 i
R/W/*    : W
1. set   2. clear ... ? 2          ...Choose "2. clear".
>
```

To quit in the middle of guidance, input only the [Enter] key and nothing else.

Format 2) Input the following command:

```
>bd 2↵
```

**■ Setting from the menu**

This dialog box appears on the screen when you select the [Data...] command from the [Break] menu.

Enter an address in the [Address] text box using a hexadecimal number or a symbol.

Use one of the radio buttons to choose an access condition.

To clear, click on the [Clear] button.

**■ Notes**

- The addresses must be specified within the range of the memory area available for each microcomputer model. An error results if this limit is exceeded. In Format 1, guidance is displayed prompting you to input an address again. In Format 2, the command input is canceled.
  - Error: Address range (0-0xFFFFFFFF). ...If an address exceeding 0xffffffff is specified.
  - Error: No map area. ...If an unused address is specified.
- The program stops one to several instructions after the break condition is satisfied.

**bsq (break sequential)****[ICE]****■ Function**

This command sets, clears, and displays a sequential break.

Up to three combinations of an address, data pattern, data mask and bus operation type can be set. A break occurs when the CPU executes a bus operation that satisfies the set conditions in the order these conditions are set.

**■ Formats**

- (1) **bsq** (guidance mode)
- (2) **bsq <mode> <hit> [<condition1> [<condition2> [<condition3>]]]** (direct input mode)
- <mode>: Set/clear specification (1=set, 2=clear)
- <hit>: Sequence specification (1=Hit 1 only, 2=Hits 1 and 2, 3: Hits 1 to 3)
- <condition>: Break condition (<address> <data> <mask> <bus type>)
- <address>: Address (hexadecimal, symbol or line number)
- <data>: Data pattern (16-bit hexadecimal number to be compared.)
- <mask>: Data mask (16-bit hexadecimal number for masking the data bits.)  
Bits set at "0" specify the data bit to be compared, and bits set at "1" specify the data bits to be masked.
- <bus type>: Bus operation type (decimal number from 0 to 8)
- |         |                   |
|---------|-------------------|
| 0: All  | All bus operation |
| 1: Inst | Instruction fetch |
| 2: VecR | Vector fetch      |
| 3: DatR | Data read         |
| 4: DatW | Data write        |
| 5: StkR | Stack read        |
| 6: StkW | Stack write       |
| 7: DmaR | DMA read          |
| 8: DmaW | DMA write         |

Condition: 0x0 ≤ address ≤ 0xfffffff

**■ Input examples****(1) Displaying sequential break conditions**

Format 1 displays the contents of current settings.

```
>bsq␣
Hit1
Address      : 00C80000
Data pattern : 6C00
Data mask    : 0000
Type         : Inst
Hit2
Address      : 00C80002
Data pattern : 6C11
Data mask    : 0000
Type         : Inst
Hit3
Address      : *****
Data pattern : ****
Data mask    : ****
Type         : ****
1. set  2. clear ... ? ␣          ...Terminated by [Enter] key.
>
```

Shown above is an example where conditions 1 and 2 are set. The asterisks "\*\*\*\*\*" in condition 3 indicate that no condition has been set. The data mask "0000" indicates that the specified data pattern bits will all be compared. In this example, a break occurs when the CPU fetches instruction code 0x6c00 from address 0xc80000 and then instruction code 0x6c11 from address 0xc80002. When the CPU accesses the addresses in the retrograde order or only condition 2 is met, no break occurs.

**(2) Setting sequential break conditions**

Format 1) After displaying the current settings as described in (1), input a command as shown below:

```

1. set 2. clear ... ? 1.␣ ...Choose "1. set".
Number of sequential address (1-3) ? :3␣ ...Choose a sequence. (3=Hits 1 to 3)
Address 00C80000 :␣ ...Address set in condition 1
Data pattern 6C00 :␣ ...Press [Enter] alone to skip guidance. (not changed)
Data mask 0000 :␣
Bus type 0: All 1:Inst 2:VecR 3:DatR 4:DatW 5:StkR 6:StkW 7:DmaR 8:DmaW... ? Inst :␣
Address 00C80002 :␣ ...Address set in condition 2
Data pattern 6C11 :␣
Data mask 0000 :␣
Bus type 0: All 1:Inst 2:VecR 3:DatR 4:DatW 5:StkR 6:StkW 7:DmaR 8:DmaW... ? Inst :␣
Address ***** :d000000␣ ...Input an address for condition 3.
Data pattern **** :0012␣ ...Input a data pattern for condition 3.
Data mask **** :^␣ ...[^] key returns to the previous guidance.
Data pattern 0012 :1200␣ ...Input a data pattern for condition 3 again.
Data mask **** :00ff␣ ...Input a data mask for condition 3.
Bus type 0: All 1:Inst 2:VecR 3:DatR 4:DatW 5:StkR 6:StkW 7:DmaR 8:DmaW... ? Inst :4␣
> ...Choose a bus operation type for condition 3.

```

In this example, conditions 1 and 2 are left as previous settings, and condition 3 is newly set.

Condition 3 specifies that program execution breaks when byte data 0x12 is written to address 0xd000000. Since the data pattern must be specified in 16 bits, a data mask is required for setting a byte access condition. In a byte access, the high-order 8 bits of the data bus are used when an even address is accessed and the low-order 8 bits of the data bus are used for an odd address. Therefore, to set a condition as a byte access with a write data 0x12 for example, specify 0x1200 for the data pattern and mask the low-order 8-bits using the data mask 0x00ff. For an odd address, specify 0x0012 for the data pattern and 0xff00 for the data mask.

A symbol or source line number can be used to specify an address.

```

Address ***** :i␣ ...Sample entry of a symbol
Address ***** :main.c#24␣ ...Sample entry of a line number

```

To quit in the middle of guidance, press the [q] key and then the [Enter] key. When the command is suspended, already specified contents are validated.

When setting two or three conditions, input the conditions in order of the sequence number. The conditions cannot be input from condition 2 or 3. A sequential break can occur when all the set conditions are met in the set sequence.

Format 2) Input a command as follows:

```
>bsq 1 1 c80000 6c00 0000 1␣
```

... This is the same specification as condition 1 described in (1). The parameters must be separated with a space. Conditions 2 and 3 are cleared if they have been set.

Even when setting multiple conditions, input all conditions in one line.

```
>bsq 1 2 c80000 6c00 0000 1 c80002 6c11 0000 1␣
          Condition 1          Condition 2
```

**(3) Clearing sequential break conditions**

Format 1) After displaying the current settings, input a command as shown below:

```

>bsq␣
(current settings)
1. set 2. clear ... ? 2.␣ ...Choose "2. clear".
>

```

All conditions set will be cleared.

Format 2) Input a command as follows:

```
>bsq 2␣
```

■ Notes

- The bsq command can only be used in ICE mode.
- A sequential break occurs only when all the set conditions are met in the set sequence. The break does not occur if only part of condition is met.
- The break conditions must continuously be specified from condition 1. The bsq command does not allow settings such as conditions 1 and 3, or conditions 2 and 3. However, it is unnecessary to set three conditions.
- The addresses must be specified within the range of the memory area available for each microcomputer model. An error will result if this limit is exceeded.
  - Error: Address range (0-0xFFFFFFFF). ...If an address exceeding 0xfffff is specified.
  - Error: No map area. ...If an unused address is specified.
- If the number of parameters entered in direct input mode is incorrect, an error will result.
  - Error: Number of parameter.
- If the entered number is illegal or it cannot be recognized as a symbol, an error will result.
  - Error: Invalid value.
- The sequential break does not occur in the internal RAM area, since the bus operation cannot be detected from outside the chip.

**bl (break list)****[ICD / ICE / SIM / MON]****■ Function**

This command lists all break conditions.

**■ Format**

**bl** (direct input mode)

**■ Display**

This command displays all contents that are displayed individually by each break setting command.

The following list is an example in ICE mode.

```
>bl↵
Soft PC break:
 1 :00F00000/E qa_cmain.c#1      9 :*****/
 2 :*****/                    10 :*****/
 3 :*****/                    11 :*****/
 4 :*****/                    12 :*****/
 5 :*****/                    13 :*****/
 6 :*****/                    14 :*****/
 7 :*****/                    15 :*****/
 8 :*****/                    16 :*****/
Hard PC break1:
Address : 00F00000 qa_cmain.c#100
Hard PC break2:
Address : *****
Data break:
Address : 00000200
R/W/* : R
Sequential break:
Hit1
  Address : 00000000
  Data pattern : 0000
  Data mask : 0000
  Type : Inst
Hit2
  Address : 00C00000
  Data pattern : 2323
  Data mask : 0000
  Type : VecR
Hit3
  Address : *****
  Data pattern : ****
  Data mask : ****
  Type : ****
>
```

**■ Note**

The display contents are changed depending on the debugger mode.



**bac (break all clear)**

[ICD / ICE / SIM / MON]

**■ Function**

This command clears all break conditions set by commands (software PC breakpoint, hardware PC break point, data break and sequential break).

**■ Format**

**bac** (direct input mode)

## 16.9.8 Commands to Display Program

### u (unassemble)

[ICD / ICE / SIM / MON]

#### ■ Function

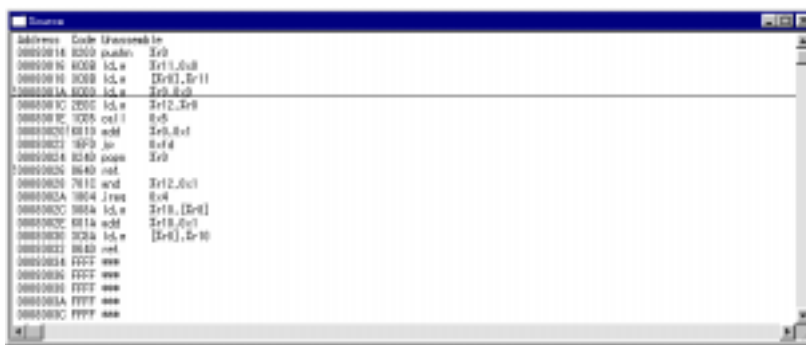
This command displays a program in disassembly format.

#### ■ Formats

- (1) **u** (direct input mode)
- (2) **u <address>** (direct input mode)
  - <address>: Display start address (hexadecimal, symbol or line number)
  - Condition:  $0x0 \leq \text{address} \leq 0xffffffff$

#### ■ Display

##### (1) When [Source] window is open



**Address:** Memory address

**Code:** Object code

**Unassemble:** Disassembled contents of program

- \* The address line of the current PC is underlined.

In Format 1, display in the [Source] window is changed to the disassemble display mode. At the same time, code is displayed beginning with the current PC address.

The [Unassemble] button performs the same function as you input the command in Format 1. However, the display location is not changed.



[Unassemble] button

In Format 2, display in the [Source] window is changed to the disassemble display mode. At the same time, code is displayed beginning with <address>.

##### (2) When [Source] window is closed

The 16 lines (default) of disassembled result are displayed in the [Command] window. The db33 then waits for a command input.

In Format 1, this display begins with the current PC (displayed in the [Register] window); in Format 2, the display begins with <address>.

```
>u 80014↓ ...Address can be specified using a symbol or line number.
00080014 0200 pushn %r0
00080016 6C0B ld.w %r11, 0x0
      :
00080030 3C8A ld.w [%r8], %r10
00080032 0640 ret
>
```

The number of display lines in the [Command] window can be changed using the md command.

**(3) Logging**

To save the command execution results to a log file, close the [Source] window and display the results in the [Command] window. If the [Source] window is open, the display contents will not be saved in the file because the [Command] window does not display the results.

**(4) Successive display**

If you execute the u command after entering it from the keyboard, code can be displayed successively by entering the [Enter] key only until some other command is executed.

When you hit the [Enter] key, the [Source] window is scrolled one full screen.

When displaying data in the [Command] window, data is displayed for the 16 lines (default) following the previously displayed address.

**■ Notes**

- Specify the display start address within the range of 0 to 0xffffffff. An error results if this limit is exceeded.  
Error: Address range (0-0xFFFFFFFF).
- "no map" is displayed for address locations outside the mapped memory area.

**sc (source code)****[ICD / ICE / SIM / MON]****■ Function**

This command displays the source file contents of the program along with addresses and line numbers.

**■ Formats**

- (1) **sc** (direct input mode)
- (2) **sc <address>** (direct input mode)  
 <address>: Display start address (hexadecimal, symbol or line number)  
 Condition:  $0x0 \leq \text{address} \leq 0xffff$

**■ Display****(1) When [Source] window is open**

Address	Line	SourceCode
0080014	0087	{
0080018	0088	int j;
0080019	0089	
008001B	0090	j = 0;
008001C	0091	for (i = 0; i < n; i++)
008001E	0092	{
008001F	0093	sub(j);
0080020	0094	}
0080024	0095	}
0080025	0096	sub(k);
0080026	0097	int k;
0080027	0098	{
0080028	0099	if (k & 0x1)
0080029	0100	{
008002A	0101	++i;
008002B	0102	}
008002C	0103	}

**Address:** Memory address

**Line:** Line numbers in source file

**SourceCode:** Source code

- \* The address line of the current PC is underlined.

In Format 1, display in the [Source] window is changed to the source display mode. At the same time, code is displayed beginning with the current PC address.

The [Source] button performs the same function as you input the command in Format 1. However, the display location is not changed.



[Source] button

In Format 2, display in the [Source] window is changed to the source display mode. At the same time, code is displayed beginning with <address>.

**(2) When [Source] window is closed**

The 16 lines (default) of source code are displayed in the [Command] window. The db33 then waits for a command input.

In Format 1, this display begins with the current PC (displayed in the [Register] window); in Format 2, the display begins with <address>.

```
>sc main.c#1_          ...Address can be specified using a hexadecimal number or symbol.
      00001 /* tst_main.c 1997.2.13 */
      00002 /* C main program */
      00003
      00004 int i;
      00005
      00006 main()
00080014 00007 {
      00008     int j;
      00009
00080016 00010     i = 0;
0008001A 00011     for (j=0 ; ; j++)
      00012         {
0008001C 00013             sub(j);
      00014         }
00080024 00015     }
      00016
>
```

The number of display lines in the [Command] window can be changed using the md command.

**(3) Logging**

To save the command execution results to a log file, close the [Source] window and display the results in the [Command] window. If the [Source] window is open, the display contents will not be saved in the file because the [Command] window does not display the results.

**(4) Successive display**

If you execute the sc command after entering it from the keyboard, code can be displayed successively by entering the [Enter] key only until some other command is executed.

When you hit the [Enter] key, the [Source] window is scrolled one full screen.

When displaying data in the [Command] window, data is displayed for the 16 lines (default) following the previously displayed address.

**■ Notes**

- Source code and line numbers can only be displayed if the srf33 object file that contains the source information has been read.
- In the source display mode, only one source file can be displayed at a time. Even when addresses are contiguous, you cannot display multiple source files in succession.
- Specify the display start address within the range of 0 to 0xfffffff. An error results if this limit is exceeded.  
Error: Address range (0-0xFFFFFFF).
- If you specify an unmapped memory address or an address that does not have source information, the display mode is changed to "Mixed". In this case, the source display part is blank.

**m (mix)****[ICD / ICE / SIM / MON]****■ Function**

This command displays the disassembled result of a program and the contents of the program source file.

**■ Formats**

- (1) **m** (direct input mode)
- (2) **m <address>** (direct input mode)
  - <address>: Display start address (hexadecimal, symbol or line number)
  - Condition:  $0x0 \leq \text{address} \leq 0xfffffff$

**■ Display****(1) When [Source] window is open**

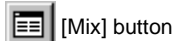
Address	Code	Unassemble	Line	SourceCode
0080014	8208	pushl %r0	00817	{
			00818	int j;
0080019	820B	ldw %r11,%r8	00819	int k;
0080019	820B	ldw [%r6],%r11	00819	int l;
008001A	8208	ldw %r0,%r0	00821	for (int i = 1; i++)
			00822	{
008001C	200C	ldw %r12,%r8	00823	sub(i);
008001E	1006	call %r5	00824	sub(i);
0080020	1018	add %r0,%r1	00821	for (int i = 1; i++)
0080022	10FD	jmp %rfd	00824	}
0080024	8248	popl %r0	00825	}
0080025	8248	ret	00825	}
			00826	sub(i);
			00827	int k;
			00828	{
0080029	781C	and %r12,%r1	00829	if (k & k)
008002A	1804	jmp %r4	00829	{
			00821	}

**Address:** Memory address  
**Code:** Object code  
**Unassemble:** Disassembled contents  
**Line:** Line numbers in source file  
**SourceCode:** Source code

- \* The address line of the current PC is underlined.

In Format 1, display in the [Source] window is changed to the mixed display mode. At the same time, code is displayed beginning with the current PC address.

The [Mix] button performs the same function as you input the command in Format 1. However, the display location is not changed.



[Mix] button

In Format 2, display in the [Source] window is changed to the mixed display mode. At the same time, the disassembled contents and the source is displayed beginning with <address>.

In the mixed display mode, multiple source files can be displayed in succession. A source file name is displayed at a position where files change.

**(2) When [Source] window is closed**

The 16 lines (default) of mixed display are produced in the [Command] window. The db33 then waits for a command input.

In Format 1, this display begins with the current PC (displayed in the [Register] window); in Format 2, the display begins with <address>.

```

>m main          ...Address can be specified using a hexadecimal number or line number.
00080014 0200 pushn %r0          00007      {
                                00008      int j;
                                00009
                                00010      i = 0;
00080016 6C0B ld.w   %r11, 0x0    00011      for (j=0 ; ; j++)
00080018 3C8B ld.w   [%r8], %r11  00012      {
0008001A 6C00 ld.w   %r0, 0x0      00013          sub(j);
                                00014      }
0008001C 2E0C ld.w   %r12, %r0    00015      }
0008001E 1C05 call  0x5
00080020 6010 add   %r0, 0x1      00016
00080022 1EFD jp    0xfd          00017  sub(k)
>

```

The number of display lines in the [Command] window can be changed using the md command.

**(3) Logging**

To save the command execution results to a log file, close the [Source] window and display the results in the [Command] window. If the [Source] window is open, the display contents will not be saved in the file because the [Command] window does not display the results.

**(4) Successive display**

If you execute the m command after entering it from the keyboard, code can be displayed successively by entering the [Enter] key only until some other command is executed.

When you hit the [Enter] key, the [Source] window is scrolled one full screen.

When displaying data in the [Command] window, data is displayed for the 16 lines (default) following the previously displayed address.

**■ Notes**

- Source code and line numbers can only be displayed if the srf33 object file that contains the source information has been read.
- Specify the display start address within the range of 0 to 0xffffffff. An error results if this limit is exceeded.  
Error: Address range (0-0xFFFFFFFF).
- "no map" is displayed for address locations outside the mapped memory area.
- If, although source line information is included, the source file cannot be read, a "no source" message is displayed.

**ss (search strings)****[ICD / ICE / SIM / MON]****■ Function**

This command searches the source file for a specified character string and starts displaying the source contents from the position where the character string is found. This command is valid only when the [Source] window is open in the source display mode.

**■ Format**

**ss <string>** (direct input mode)  
<string>: Search character string

**■ Input example**

```
>ss main↵  
>
```

The source contents are displayed in the [Source] window beginning with a position where the first instance of main is found in the current source (one that contains the code corresponding to the current PC address). Then when you press the [Enter] key, the next instance of main is searched.

If the specified character string is not found, the following message is displayed:

Error: Not found input string.

**■ Notes**

- An error results if the ss command is executed when the [Source] window is closed.  
Error: Source window not opened.
- Also, if the [Source] window is not in the source display mode when the command is executed, an error results.  
Error: Current mode is not source mode.



## 16.9.9 Commands to Display Symbol Information

### sy (symbol list)

[ICD / ICE / SIM / MON]

#### ■ Function

This command displays symbol information, source file, functions, or tag list in the [Command] window. Symbols can be conditionally searched after specifying a search range or a search character string.

#### ■ Format

**sy** [**<option>**] (direct input mode)

**<option>**: Specification of display item and search condition

When omitted: List of all symbols

#: File list

/: Function list

@: Tag list

@<string>: Tag list beginning with character string <string>

[<file>]/[<function>]: List of symbols in a specified file or function

[<file>]/[<function>]/<string>: List of symbols in a specified file or function beginning with character string <string>

<file>: Source file name

<function>: Function name

<string>: Search character string

Condition:  $1 \leq \text{number of characters in string} \leq 255$

Symbols searched by [<file>]/[function]/

// Global symbols

./ Auto/static symbols in current function

.. Static symbols in current source file

file// Static symbols in specified source file

/function/ Symbols in specified function

./function/ Auto/static symbols in specified function of current source file

file/function/ Symbols in specified function of specified source file

- \* The current source files and current functions refer to those that contain the code corresponding to the current PC.

## ■ Display

A list is displayed in the [Command] window in alphabetical order. By default, up to 16 lines are displayed at a time (can be changed by the md command). If there are more than 16 lines of display items, the window is placed in a command input state after displaying 16 lines. Therefore, input only the [Enter] key and nothing else to display the remaining other lines.

### (1) Symbol list

Symbol information is displayed in the following format:

<symbol>, <address>, <scope>, <class>, <type>

<symbol>: Indicates a symbol name.

<address>: Indicates the address of a symbol. The display content varies with the symbol's storage class.

Storage class	Display content
null, extern, static, label	8-digit hexadecimal address
auto, argument	SP + offset (0xXX)
register, reg parameter	Register number (R0 to R15)
bit field	Bit field size

<scope>: Indicates a file name/function name.

Extern symbols are left blank. For static symbols, only a file name is displayed. For the variables defined within a block, the start and the end addresses of the block are displayed. (<start addr> ... <end addr>)

<class>: Indicates a storage class.

null, auto, extern, static, register, label, argument, reg parameter

<type>: Indicates the type of symbol.

null, void, char, short, int, long, float, double, struct tag, union tag, enum tag, unsigned char, unsigned short, unsigned int, unsigned long

If the symbol is a pointer, array, or function, these types of symbols are followed by "\*", "[ ]" (including declaration content), or "(" ).

### Displaying all symbols

To display all symbols, input a command as follows:

```
>sy↵ ...Execute without an option.
BOOT, 00080004, boot.s/, static, null
LOOP, 00C00000, , label, null
READ_BUF, 00800048, , extern, unsigned char [65]
READ_EOF, 00800089, sys.c/, static, unsigned char
:
_init_sys, 00C00004, , extern, void ()
_iob, 0080008C, , extern, struct __T2 [4]
>↵ ...Display the next list when [Enter] alone is input.
dfIn, R0, ansilib.c/main, register, double
dfOut, R10, ansilib.c/main, register, double
:
iSize, R12, sys.c/write, register, int
>
```

### Listing names after specifying search range

A file name and function name ([<file>]/[function]/) can be specified using an option.

```
>sy //↵ ...Specify a list of global symbols.
READ_BUF, 00800048, , extern, unsigned char [65]
READ_FLASH, 00C0005C, , extern, null
WRITE_BUF, 00800004, , extern, unsigned char [65]
:
read, 00C00014, , extern, int ()
seed, 008000D0, , extern, unsigned int
>
```

**Listing symbols after specifying the search range and search character string**

The symbols in a search range that begin with a specified character string are searched and displayed.

```
>sy sys.c//R
READ_EOF, 00800089, sys.c/, static, unsigned char
>
```

**(2) Source file list**

A file list is displayed in the following format:

```
<file>(<line>), <start addr> .. <end addr> [, <start addr> .. <end addr>]
```

<file>: Indicates a source file name.

<line>: Indicates the number of lines in the source file.

If no source file is read into the debugger, "not read" is displayed.

<start addr>: Indicates the start address of the area where code is located in hexadecimal form.

<end addr>: Indicates the end address of the area where code is located in hexadecimal form.

If the code is located in multiple areas, multiple instances of <start addr> and <end addr> are displayed.

If the code exists in an include file, <end addr> is followed by "<include file name>".

To display a list of source files, input the following command:

```
>sy #
boot.s(29 lines) 80004..80023
sys.c(180 lines) C00000..C000AF
lib.c(83 lines) C000B0..C00133
ansilib.c(not read) C00342..C00509
..\src\strlen.s(not read) C02B94..C02B9F
..\src\memcpy.s(not read) C02BA0..C02BAF
>
```

**(3) Function list**

A function list is displayed in the following format:

```
<file>
```

```
<function>() <start addr> .. <end addr>
```

<file>: Indicates a source file name.

File names are displayed one at a time for each source file. Even a file that does not have functions (assembly source) is displayed.

<function>: Indicates a function name.

Functions are displayed for each source file.

<start addr>: Indicates the start address of the area where the function is located in hexadecimal form.

<end addr>: Indicates the end address of the area where the function is located in hexadecimal form.

To display a list of functions, input the following command:

```
>sy /
boot.s
sys.c
_exit() C00000..C00003
_init_sys() C00004..C00013
read() C00014..C00071
write() C00072..C000AF
lib.c
_init_lib() C000B0..C00133
ansilib.c
main() C00342..C00509
>
```

**(4) Tag list**

A tag list is displayed in the following format:

<file>

```
offset  <type> <tag> {
  <offset> <m type> <member>
    :
    } (total <size>)
```

<file>: Indicates a source file name.

File names are indicated one at a time for each source file. Files where no tag is declared are not displayed.

<type>: Indicates a type of tag.

<tag>: Indicates a name of tag.

<offset>: Indicates a member offset in hexadecimal form.

<m type>: Indicates a member type.

<member>: Indicates a member name.

<size>: Indicates a size (bytes) in decimal form.

**Displaying all tags**

To display all tags, input the following command:

```
>sy @,
lib.c
  offset struct tm {
00000000      int tm_sec
00000004      int tm_min
    :
00000020      int tm_isdst
    } (total 36 byte)
  offset struct __T2 {
00000000      short _flg
00000002      unsigned char _buf
00000004      int _fd
>,
    } (total 8 byte)
  offset union __T1 {
00000000      struct __T0 st
00000000      double _D
    } (total 8 byte)
>
```

...Display the next list when [Enter] alone is input.

**Listing tags after specifying the search character string**

The tags that begin with the specified character string are searched and displayed.

```
>sy @t,
lib.c
  offset struct tm {
00000000      int tm_sec
00000004      int tm_min
00000008      int tm_hour
0000000C      int tm_mday
00000010      int tm_mon
00000014      int tm_year
00000018      int tm_wday
0000001C      int tm_yday
00000020      int tm_isdst
    } (total 36 byte)
ansilib.c
  offset struct tm {
00000000      int tm_sec
00000004      int tm_min
>
```

...Specify a list of tags that begin with t.

Bit fields are displayed as follows.

"offset" indicates each bit offset value from the beginning address.

```

offset struct bitSt {
00000000    unsigned int  b1   : 1
00000001    unsigned int  b11  : 11
0000000C    unsigned int  b3   : 3
0000000F    unsigned int  b17  : 17
00000020    unsigned int  b21  : 21
} (total 8 byte)

```

"enum" type is displayed as follows.

"value" indicates the value of each member.

```

value enum iEnum {
00000000    enum member  iE1
00000001    enum member  iE2
00000002    enum member  iE3
0000000A    enum member  iE10
}

```

#### ■ Notes

- Symbol information can only be displayed if the srf33 format object file that contains debugging information is read into the debugger.
- Search character strings <string> that are upper case are distinguished from these that are lower case. Up to 255 characters can be specified.

**sa (symbol add)****[ICD / ICE / SIM / MON]****■ Function**

This command registers specified symbols (variables) in a symbol watch table. Up to 99 symbols can be registered. The contents of registered symbols can be monitored in the [Symbol] window. A display format can be specified along with the symbols to be registered.

**■ Formats**

- (1) **sa** Display (direct input mode)  
 (2) **sa <symbol> [-<switch>]** Register (direct input mode)

<symbol>: Symbol name

-<switch>: Specification of display format

- b<size> Binary
- d<size> Signed decimal
- u<size> Unsigned decimal
- h<size> Hexadecimal
- c 8-bit integer
- f 32-bit real number
- df 64-bit real number

<size> specifies the number of bits; specify 8, 16, 32, or 64 (e.g., -b8, -h32). If this specification is omitted, symbols are displayed in a size that suits the symbol type.

You cannot specify 64 bits for -d and -u.

Default: Applied when -<switch> is omitted

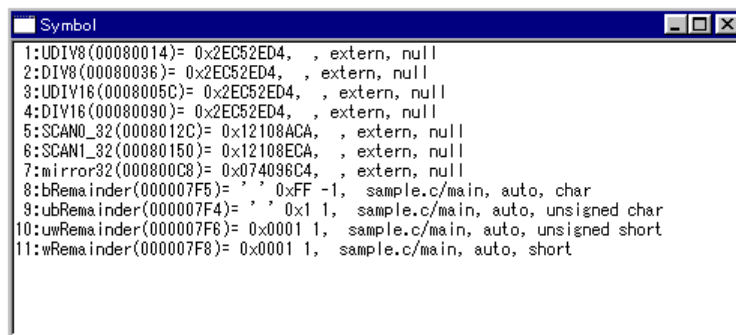
- Symbol with an unknown type: Displayed in 32-bit hex.
- int, short, long: Displayed in both decimal and hex.
- char: ASCII code displayed in decimal and hex.

**■ Input examples****(1) Display**

If you input only sa and nothing else (Format 1), the contents of the registered symbols are displayed.

**When [Symbol] window is open**

The contents displayed in the [Symbol] window are updated.



```

Symbol
1:UDIv8(00080014)= 0x2EC52ED4, , extern, null
2:DIv8(00080036)= 0x2EC52ED4, , extern, null
3:UDIv16(0008005C)= 0x2EC52ED4, , extern, null
4:DIv16(00080090)= 0x2EC52ED4, , extern, null
5:SCAN0_32(0008012C)= 0x12108ACA, , extern, null
6:SCAN1_32(00080150)= 0x12108ECA, , extern, null
7:mirror32(000800C8)= 0x074096C4, , extern, null
8:bRemainder(000007F5)= ' ' 0xFF -1, sample.c/main, auto, char
9:ubRemainder(000007F4)= ' ' 0x1 1, sample.c/main, auto, unsigned char
10:uwRemainder(000007F6)= 0x0001 1, sample.c/main, auto, unsigned short
11:wRemainder(000007F8)= 0x0001 1, sample.c/main, auto, short
  
```

**When [Symbol] window is closed**

The contents are displayed in the [Command] window. By default, 16 lines are displayed at a time. To display the next lines, press the [Enter] key. The number of lines displayed at a time can be changed by the md command.

```
>sa␣
1:UDIV8(00080014)= 0x2EC52ED4, , extern, null
2:DIV8(00080036)= 0x2EC52ED4, , extern, null
3:UDIV16(0008005C)= 0x2EC52ED4, , extern, null
4:DIV16(00080090)= 0x2EC52ED4, , extern, null
5:SCAN0_32(0008012C)= 0x12108ACA, , extern, null
6:SCAN1_32(00080150)= 0x12108ECA, , extern, null
7:mirror32(000800C8)= 0x074096C4, , extern, null
8:bRemainder(000007F5)= ' ' 0xFF -1, sample.c/main, auto, char
9:ubRemainder(000007F4)= ' ' 0x1 1, sample.c/main, auto, unsigned char
10:uwRemainder(000007F6)= 0x0001 1, sample.c/main, auto, unsigned short
11:wRemainder(000007F8)= 0x0001 1, sample.c/main, auto, short
>
```

**Display contents**

The contents of each symbol are displayed in the following formats:

<No.> <symbol>(<address>) = <value>, <scope>, <class>, <type>

<No.>: Indicates a registered number in the watch symbol table.

<symbol>: Indicates a symbol name.

<address>: Indicates a symbol's address in 8-digit hexadecimal form. Register variables are indicated by R0 to R15.

<value>: Indicates a stored value according the <switch> specification made when registered. A hexadecimal number is prefixed by "0x" and a binary number is prefixed by "0b". A negative number is prefixed by "-", but nothing is added in the case of positive numbers. If the stored value is out of scope, "out of scope" is displayed.

<scope>: Indicates a file name/function name.

Extern symbols are left blank. For static symbols, only a function name is displayed.

<class>: Indicates a storage class.

null, auto, extern, static, register, label, argument, reg parameter

<type>: Indicates a type of symbol.

null, void, char, short, int, long, float, double, struct tag, union tag, enum tag, unsigned char, unsigned short, unsigned int, unsigned long

If the symbol is a pointer, array or function, these types of symbols are followed by "\*", "[ ]" (including declaration content) or "()".

**(2) Registering symbols**

The following shows an example in which symbols are registered by specifying <switch>.

```
>sa chChar -b8␣ ...Register chChar as 8-bit binary representation.
>sa stA.ulCount -u32␣ ...Register stA member "ulCount" as a 32-bit unsigned decimal representation.
>sa iLoop -h␣ ...Register a size conforming to iLoop type as a hexadecimal representation.
>sa dDecimal -df␣ ...Register dDecimal as a double-type representation.
>sa *plCount -u32␣ ...Register pointer "plCount" as a 32-bit signed decimal representation.
```

When the command is executed, information on the registered symbols is displayed in the [Symbol] window. If the [Symbol] window is closed, the information is displayed in the [Command] window. The registered symbols are assigned a registration number (1 to 99).





**sd (symbol delete)**

[ICD / ICE / SIM / MON]

**■ Function**

This command deletes a symbol displayed in the [Symbol] window (registered in the watch symbol table) from the window (table).

**■ Format**

**sd <number>** (direct input mode)  
 <number>: Registration number of the symbol (decimal)  
 Condition:  $1 \leq \text{number} \leq \text{number of registered symbols (max. 99)}$

**■ Input example**

```
>sd 2↵      ...Delete symbol No. 2.
>
```

When a symbol is deleted, the symbol numbers following it are each decreased by one.

- \* Symbols can also be deleted using the [Delete] command on the [Symbol] menu or the [Symbol delete] button.



[Symbol delete] button

Place the cursor at the symbol information line in the [Symbol] window that you want deleted, then click on the [Symbol delete] button or choose the [Delete] command from the [Symbol] menu. The symbol will be deleted.

These menu commands and buttons can be selected only when the [Symbol] window is active.

**■ Note**

An error results if a number greater than 99 or an unregistered number is specified.

Error: No symbol at the number.

**sw (symbol watch)****[ICD / ICE / SIM / MON]****■ Function**

This command displays the current value of a specified symbol in the [Command] window. Each symbol individually or a range of symbols collectively can be specified. You also can specify a display format.

**■ Formats**

- (1) **sw** <symbol> [-<switch>] (direct input mode)
- (2) **sw** @<symbol> [-<switch>] (direct input mode)
- (3) **sw** <scope> (direct input mode)

<symbol>: Symbol name

-<switch>: Specification of display format

- b<size> Binary
- d<size> Signed decimal
- u<size> Unsigned decimal
- h<size> Hexadecimal
- c 8-bit integer
- f 32-bit real number
- df 64-bit real number

<size> specifies the number of bits; specify 8, 16, 32, or 64 (e.g., -b8, -h32). If this specification is omitted, symbols are displayed in a size that suits the symbol type.

You cannot specify 64 bits for -d and -u.

Default: Applied when -<switch> is omitted

- Symbol with an unknown type: Displayed in 32-bit hex.
- int, short, long: Displayed in both decimal and hex.
- char: ASCII code displayed in decimal and hex.

<scope>: Specification of a range of symbols to be listed ([<file>]/[function]/)

- // Global symbols
- ./ Auto/static symbols in the current function
- ./ Static symbols in the current source file
- file// Static symbols in the specified source file
- /function/ Symbols in the specified function
- ./function/ Auto/static symbols in the specified function of the current source file
- file/function/ Symbols in the specified function of the specified source file

**■ Display****(1) Displaying each symbol individually**

In Format 1, symbol information is displayed by specifying one symbol at a time.

```
>sw READ_BUF [0] ␣ ...Specify the default display format.
READ_BUF[0] (00800048)= ' ' 0xAA 170, , extern, unsigned char [65]
>
>sw WRITE_BUF -u8␣ ...Specify the display in 8-bit unsigned decimal form.
WRITE_BUF (00800004)= 170, , extern, unsigned char [65]
>
```

For details on the display contents, refer to the explanation of the sa command.

- \* Symbol display in Format 1 can also be performed using the [Watch] command on the [Symbol] menu or the [Symbol watch] button.



[Symbol watch] button

Place the [Source] window in the mixed or source display mode, and place the cursor in or immediately before or after the symbol name that you want displayed. Then click on the [Symbol watch] button or choose the [Watch] command from the [Symbol] menu to display the symbol in the [Command] window.

These menu commands and buttons can be selected only when the [Source] window is active. No display format can be set. Information is displayed in the default format that conforms to the selected symbol type.

## (2) Displaying structure/union members and array elements

In Format 2, members of a structure/union or elements of an array can be specified to display the contents. For details on the display contents, refer to the explanation of the sa command.

By default, 16 lines of symbol information are displayed. To display the following information, press the [Enter] key. The number of lines displayed at a time can be changed by the md command.

### Displaying array elements

Example: char buf[2][2][2]

```
>sw @buf.␣
buf[0][0][0] (000006B0)= '#' 0x23 35, qa_cmain.c/main, auto, char [2][2][2]
buf[0][0][1] (000006B1)= ' ' 0xF9 -7, qa_cmain.c/main, auto, char [2][2][2]
      :
buf[1][1][1] (000006B7)= ' ' 0xE0 -32, qa_cmain.c/main, auto, char [2][2][2]
>
>sw @buf[0][1][0].␣
buf[0][1][0] (000006B2)= ' ' 0x8F -113, qa_cmain.c/main, auto, char [2][2][2]
buf[0][1][1] (000006B3)= ' ' 0x1C 28, qa_cmain.c/main, auto, char [2][2][2]
      :
buf[1][1][1] (000006B7)= ' ' 0xE0 -32, qa_cmain.c/main, auto, char [2][2][2]
>
```

### Displaying members of a structure

Example: struct STRUCT\_UNION{

```
    int iAddr
    char cFlag;
    char *pcFlag;
    char cAry[3][3]
    union ExchangeType stSize;
};
>sw @stStructTest1.cAry.␣
stStructTest1.cAry[0][0] (00000414)= ' ' 0x10 16, qa_cmain.c/main, struct, char [3][3]
stStructTest1.cAry[0][1] (00000415)= ' ' 0x04 4, qa_cmain.c/main, struct, char [3][3]
      :
stStructTest1.cAry[2][2] (0000041C)= ' ' 0x0C 12, qa_cmain.c/main, struct, char [3][3]
>
>sw @stStructTest1.stSize.␣
stStructTest1.stSize (00000420)= 0xF4D18517, qa_cmain.c/main, struct, union ExchangeType
stStructTest1.stSize.ucChar (00000420)= ' ' 0x17 23, qa_cmain.c/main, union unsigned char [4]
stStructTest1.stSize.usShort (00000420)= 0x8517 34071, qa_cmain.c/main, union unsigned short [2]
stStructTest1.stSize.ulLong (00000420)= 0xF4D18517 4107371799, qa_cmain.c/main, union unsigned int [1]
>
```

For structures, the sw command displays all members at 1-level lower than the specified member. Two or more lower-level members are not displayed.

**(3) Listing the specified range of symbols**

In Format 2, you can specify the range of files and functions, and display a list of symbols included in the specified range.

```
>sw //␣                               ...Specify a list of global symbols.
READ_BUF (00800048)= ' ' 0xAA 170, , extern, unsigned char [65]
READ_FLASH (00C0005C)= 0x680A24FA, , extern, null
WRITE_BUF (00800004)= ' ' 0xAA 170, , extern, unsigned char [65]
:
read (00C00014)= 0xC0106C0C -1072665588, , extern, int ()
seed (008000D0)= 0xAAAAAAAA 2863311530, , extern, unsigned int
>
```

For details about display contents, refer to the explanation of the sa command.

By default, 16 lines of symbols are displayed. To display the next symbols, press the [Enter] key. The number of lines displayed at a time can be changed by the md command.

**■ Notes**

- Symbol information can only be displayed if the srf33 format object file that contains debugging information is read into the debugger.
- An error results if the specified symbol cannot be found.  
Error: Symbol not found. ...Symbol cannot be found.
- If the specified symbol is out of the scope or it points to a no-map area, the symbol information is displayed as below.  
<symbol> = out of the scope, .... ...Symbol is out of the scope.  
<symbol> = symbol points to no map area, .... ...No-map area is pointed.
- "->" and "." are not distinguished.
- Pointers (\*) can be specified up to three nest levels and array elements can be specified up to the fourth dimension.
- Structures, unions and bit fields can be specified up to 10 levels. However, it is limited to 9 levels for the "sw @" command that adds a member.
- Array elements (number in [ ]) can only be specified in a decimal number.
- Symbol length including a scope is limited to a maximum of 127 characters.

## 16.9.10 Commands to Load Files

### If (load file)

[ICD / ICE / SIM / MON]

#### ■ Function

This command loads the object file in srf33 format into the debugger.

#### ■ Formats

- (1) **If** (guidance mode)
- (2) **If <file name>** (direct input mode)  
     <file name>: File name to be loaded

#### ■ Input examples

Format 1) >|f␣  
 File name ? :test.srf␣      ...Input a file name including the extension.  
 >

Format 2) >|f test.srf␣  
 >

- \* The [Load File..] command on the [File] menu or the [Load file] button on the tool bar can also be used to load a file. Use the dialog box that appears on the screen to select a file.



[Load file] button



After selecting a directory and file, click on the [Open] button (or double-click the file name).

The directory selected here becomes the current directory.

#### ■ Notes

- Only the srf33 object file in the executable format (generated by the linker) can be loaded by the If command.  
     Error: Cannot load data, please check SRF33 file.
- If you want to use source display and symbols when debugging a program, the object file must be in the srf33 format that contains debug information loaded into the debugger. A warning results if the loaded file does not contain debug information. The actual data is useful, however.  
     Warning: No debug information, <file name>.
- The object file in the srf33 format contains the source file information, including the directory structure. Therefore, the source file cannot be loaded unless it resides in a specified directory within the object file as viewed from the current directory.  
     When loading a file using a menu command or tool bar button, the directory you select in the dialog box becomes the current directory.  
     When using the If command, the current directory is not modified.  
     Seiko Epson recommends that you basically perform a series of operations from the C Compiler gcc33/Preprocessor pp33 to the Debugger db33 in the same directory (after making it the current directory).
- Up to 32767 lines in one source file can be loaded.
- If the [Source] window is open when loading a file, its contents are updated. The program contents are displayed in the currently set display mode beginning with the current PC. The PC is not modified by loading a file.

- When a file is read in, the current debugging information and the symbol registration information in the [Symbol] window are invalidated. However, break information is left intact, so clear all break information using the bac command before loading a file. Furthermore, if the [Memory] window is open, all of its contents will be cleared. In this case, redisplay the [Memory] window by executing a dump (db, dh or dw) command or clicking on the vertical scroll bar.
- If an error occurs when loading a file, portions of the file that have already been read will remain in the emulation memory. However, in this case, you cannot use the source display or symbols to debug the program. Nor can you see to what extent the file has been loaded. Furthermore, the db33 forcibly switches the [Source] window in disassemble display mode.

**lh (load hex)**

[ICD / ICE / SIM / MON]

**■ Function**

This command loads the Motorola S3 format file output from the hex33 into the debugger.

**■ Formats**

- (1) **lh** (guidance mode)  
 (2) **lh <file name> <offset>** (direct input mode)  
 <file name>: File name to be loaded (path can also be specified)  
 <offset>: Offset address

**■ Input examples**

```
Format 1) >|h␣
           File name ? :test.sa␣    ...Input a file name including the extension.
           offset      : 80000␣    ...Specify the offset address.
           >
```

Specify offset = 0 for the Motorola S3 format that has absolute addresses attached when generated.

```
Format 2) >|h test.sa 80000␣
           >
```

**■ Notes**

- With the Motorola S3 format program file loaded, you cannot use the source display or symbols to debug a program.
- If the [Source] window is open when loading a file, its contents will be updated. The program contents are displayed in disassemble mode beginning with the current PC. The PC is not modified by loading a file.
- If an error occurs when loading a file, portions of the file that have already been read are left as they were loaded.
- An error results if you specify an offset address that will cause the file to be loaded into an unmapped memory area.

Error: No map area.

**ld (load file)****[ICD / ICE / SIM / MON]****■ Function**

This command loads the debugging information included in the specified srf33 object file into the debugger.

**■ Formats**

- (1) **ld** (guidance mode)
- (2) **ld <file name>** (direct input mode)  
 <file name>: File name to be loaded (path can also be specified)

**■ Input examples**

Format 1) >|d␣  
 File name ? :test.sa␣ ...Input a file name including the extension.  
 >

Format 2) >|d test.sa␣  
 >

**■ Notes**

- The ld command does not change data except for the debugging information. This command should be used only when the program has been loaded, such as debugging for the program written to the ROM.
- Only the srf33 object file in the executable format (generated by the linker) can be loaded by the ld command.  
 Error: Cannot load data, please check SRF33 file.
- A warning results if the loaded file does not contain debug information.  
 Warning: No debug information, <file name>.
- The object file in the srf33 format contains the source file information, including the directory structure. Therefore, the source file cannot be loaded unless it resides in a specified directory within the object file as viewed from the current directory.  
 Seiko Epson recommends that you basically perform a series of operations from the C Compiler gcc33/Preprocessor pp33 to the Debugger db33 in the same directory (after making it the current directory).



## 16.9.11 Commands to Operate Flash Memory

### fls (flash memory set)

[ICD / (ICE) / MON]

#### ■ Function

This command initializes the flash memory information used to write data to the flash memory on the target system.

#### ■ Format

**fls** (guidance mode)

#### ■ Input example

```
>fls␣
Flash start address : *****           ...Undefined address
Flash end address   : *****
Erase routine address : *****
Write routine address : *****
1. set 2. clear ...? clear :1␣           ...Choose "1. set".
Flash start address ? ***** : 200000␣ ...Flash start address
Flash end address   ? ***** : 2ffffff␣ ...Flash end address
Erase routine address ? ***** : FLASH_ERASE␣ ...Erase-routine start address
Write routine address ? ***** : FLASH_LOAD␣ ...Write-routine start address
>
```

\* Press the [q] key to cancel the command.

#### ■ Notes

- The fls command can be used only for the flash memory on the target system and does not affect the ICE33 flash memory. Furthermore, simulator mode does not support this command.
- To erase and write data of the flash memory on the target system, a data write/erase routine must be loaded to the specified address before using this command. Refer to the "readme.txt" of the flash support utility fls33 for the flash write/erase routine.  
("fls33" and "readme.txt" can be installed using "cc33\utility\fls33\fls33vXX.exe".)
- When using this command in ICE mode, the ICE firmware must be Ver. 2.0 or higher.

**file (flash memory erase)****[ICD / (ICE) / MON]****■ Function**

This command erases the contents of the flash memory on the target system.

**■ Format**

**file** (guidance mode)

**■ Input example**

```
>file␣
Control Register : 00200000      ...Flash start address set by fls
Start block      : *****      ...Undefined address
End block        : *****
Control Register ? 00200000 : 200000␣ ...Flash start address
Start block      ? ***** : 0␣    ...Specify the range to be erased.
End block        ? ***** : 0␣    Start = End = 0: erase all
Finish with 0x00000000      ...The return value from the erase routine is displayed.
>
```

\* Press the [q] key to cancel the command.

**■ Notes**

- The file command can be used only for the flash memory on the target system and cannot be used for the ICE33 flash memory. Furthermore, simulator mode does not support this command.
- To erase the flash memory on the target system, a data write/erase routine must be loaded to the target memory and the fls command must be executed before using this command. If the erase routine has not been loaded, an error will result when the file command is executed.  
Error: Erase routine is not set.
- This command must be executed before writing data to the flash memory on the target system.
- When using this command in ICE mode, the ICE firmware must be Ver. 2.0 or higher.

**lfl (load from flash memory)****[ICE]****■ Function**

This command loads the memory contents from the flash memory of the ICE33 into the target memory.

It therefore allows you to debug the program beginning from the contents previously saved to the flash memory up to latest one.

**■ Format**

**lfl** (guidance mode)

**■ Input example**

```
>lfl␣
Are you sure to load      1.yes 2.no ...? 1␣      ...Confirmation of whether or not to load
Loading from flash memory ..... done
>
```

Choose 2 if you want to stop loading memory contents.

**■ Notes**

- The lfl command is designed specifically for the ICE33 flash memory and does not support the flash memory on the target system. Therefore, this command can be used only in ICE mode.
- If the flash memory is protected against read/write or has been erased, an error will result and memory contents will not be loaded into the target memory.  
Error: Flash ROM is protected. ...If the flash memory is protected.
- Even if the flash memory and target memory are mapped differently, memory contents are loaded and the map is rewritten.
- If an error occurs when loading data, portions of the data that have already been read into the target memory are left as they were loaded.

**sfl (save to flash memory)****[ICE]****■ Function**

This command writes the contents of the target memory in the ICE33 into the ICE33 flash memory.

Writing to the flash memory allows the ICE33 to be operated in free-run mode. Furthermore, the next debug session can be continued immediately from the current contents in the flash memory.

The flash memory can be write-protected.

**■ Format**

**sfl** (guidance mode)

**■ Input example**

According to the guidance, protect the flash memory and confirm whether you want contents to be written to the flash memory.

```
>sfl␣
Protect flash memory      1.yes 2.no ...? 2␣    ...write-protect specification
Are you sure to save     1.yes 2.no ...? 1␣    ...Confirmation of whether or not to write
Saving to flash memory ..... done
>
```

Choose 2 when prompted for confirmation if you want to stop writing memory contents.

**■ Notes**

- The sfl command is designed specifically for the ICE33 flash memory and does not support the flash memory on the target system. Therefore, this command can be used only in ICE mode.
- If the flash memory is write-protected, an error results and memory contents are not written to the flash memory.

Error: Flash ROM is protected.

The write-protect can be removed by erasing the flash memory with the efl command.

**efl (erase flash memory)****[ICE]****■ Function**

This command erases the contents of the ICE33 flash memory (including map information) and removes its protect function.

**■ Format**

**efl** (direct input mode)

**■ Input example**

```
>efl␣  
Are you sure to erase      1.yes 2.no ...? 1␣      ...Confirmation of erasing  
Erasing flash memory ..... done  
>
```

**■ Note**

The efl command is designed specifically for the ICE33 flash memory and does not support the flash memory on the target system. Therefore, this command can be used only in ICE mode.

**maf (map flash memory)**

[ICE]

**■ Function**

This command displays the ICE33 flash memory map, chip name, version of the parameter file used and other information.

**■ Format**

**maf** (direct input mode)

**■ Display examples****(1) When the flash memory is not write-protected:**

```
>maf↵
Chip name           : 33104
Parameter file version : 01
Internal ROM area   : 80000 - 80FFF
Size of F0 area     : 1000
Emulation memory area
  00C00000 - 00CFFFFF ROM
  00800000 - 008FFFFF RAM
Map break information
  00000000 - 000007FF RAM
  00040000 - 000402FF I/O
  00048000 - 000482FF I/O
  00C00000 - 00CFFFFF ROM emulation ...Emulation memory settings
  00800000 - 008F7FFF RAM emulation
  00000000 - 000007FF stack area ...Stack area settings
  00800000 - 008FFFFF stack area
>
```

**(2) When the flash memory is write-protected:**

```
>maf↵
Flash memory is protected. ...Protection status
Chip name           : 33104
Parameter file version : 01
Internal ROM area   : 80000 - 80FFF
Size of F0 area     : 1000
Emulation memory area
  00C00000 - 00CFFFFF ROM
  00800000 - 008FFFFF RAM
Map break information
  00000000 - 000007FF RAM
  00040000 - 000402FF I/O
  00048000 - 000482FF I/O
  00C00000 - 00CFFFFF ROM emulation
  00800000 - 008F7FFF RAM emulation
  00000000 - 000007FF Stack area
  00800000 - 008FFFFF Stack area
>
```

**(3) When the flash memory is initialized:**

```
>maf↵
Flash memory is not mapped.
>
```

**■ Note**

The maf command is designed specifically for the ICE33 flash memory and does not support the flash memory on the target system. Therefore, this command can be used only in ICE mode.

## 16.9.12 Trace Commands

### tm (trace mode)

[ICD / ICE / SIM]

#### ■ Function

##### (1) ICE mode

In ICE mode, this command displays and sets a trace mode and trace trigger conditions.

##### Trace mode

The following two trace modes can be set:

##### 1) Normal trace mode

The data written to the trace memory is always the latest trace information.

##### 2) Single-delay trigger trace mode

The following three types of trace sampling areas can be specified with respect to the trace trigger point (establishment of trace trigger condition):

1. start ...Trace information is collected beginning with the trace trigger point.
2. middle ...Trace information before and after the trace trigger point is collected.
3. end ...Trace information is collected until the trace trigger point is reached.

##### Trace trigger conditions

The following three types of trace trigger conditions can be set:

##### 1) Address

One memory address can be specified. The trace trigger is generated on condition that the CPU accesses this address.

##### 2) Data pattern

Specify the data pattern that the CPU reads or writes to the above address. You can specify a 16-bit pattern, setting each bit as desired. Selected bits or all bits can be masked out for exclusion from trace trigger conditions.

##### 3) Bus operation type

Specify a bus operation type in which operation the CPU accesses the above address. One of the following bus operation types can be selected:

0. All ...All bus operations
1. Inst ...Instruction fetch
2. VecR ...Vector fetch
3. DatR ...Data read
4. DatW ...Data write
5. StkR ...Read from stack
6. StkW ...Write to stack
7. DmaR ...Read by DMA
8. DmaW ...Write by DMA

When one of these conditions is satisfied, a point in time (trace trigger point) at which single-delay trigger trace or pulse output from the ICE33's TRGOUT pin is controlled.

##### (2) ICD mode

In ICD mode, this command displays and sets a trace mode and trace trigger addresses.

##### Trace mode

The following two trace modes can be set:

##### 1) All trace mode

Trace is initiated by a start of program execution. The trace information is written to the trace memory regardless of the address executed.

##### 2) Area trace mode

Trace information is taken into the trace memory only when the program within the range from trigger address 1 to trigger address 2 is executed.

**Trace condition in all trace mode**

In all trace mode, a trace memory write condition can be specified.

## 1) Overwrite enabled

The trace operation does not stop even if the trace memory (131072 cycles) is full and new data is overwritten to the oldest data. Consequently, the data written to the trace memory is always the latest trace information.

## 2) Overwrite disabled

Trace information is written to the trace memory until the memory becomes full, and then the trace operation stops.

**Trace conditions in area trace mode**

In area trace mode, the following two conditions can be set:

## 1) Break at trigger address 2

The program execution can be suspended or continued at trace trigger address 2 after tracing the specified area.

## 2) Time measurement mode

A measurement range of the program execution time can be selected from the following two types:

All measurement mode: The execution time is measured from start to break of the program execution.

Area measurement mode: Only the time while the program is executed within the range between trigger addresses 1 and 2 is measured.

In addition to the conditions above, the clock counter (Clk in the trace information) display mode can be set either to count accumulating from start of tracing or to count in instruction units.

Example:

Counter display mode = "accumulate"

Cycle	Address	Code	Unassemble	Clk	...
000009	0610FFE	0000	nop	000000	...
000008	0611000	C000	ext 0x0	000008	...
000007	0611002	D00A	ext 0x100a	000016	...
000006	0611004	6DF0	ld.w %r0, 0x1f	000024	...

Counter display mode = "each instruction"

Cycle	Address	Code	Unassemble	Clk	...
000009	0610FFE	0000	nop	000008	...
000008	0611000	C000	ext 0x0	000008	...
000007	0611002	D00A	ext 0x100a	000008	...
000006	0611004	6DF0	ld.w %r0, 0x1f	000008	...

**(3) Simulator mode**

In the simulator mode, only the following operation can be selected:

## 1) Trace function ON/OFF

When the trace function is turned ON, trace information is collected according to program execution.

## 2) Display of register value

You can choose to collect register contents, in addition to basic trace information.

## 3) Output destination of trace information

An output destination for the collected trace information can be selected from a window or file. If you choose a window, the trace information is displayed in the [Trace] window or (if the [Trace] window is closed) in the [Command] window. When selecting a file, specify the desired file name too.



## ■ Format

**tm** (guidance mode)

## ■ Input examples for ICE mode

### (1) Displaying current setup contents

When the tm command is executed, the current setup contents are displayed first.

```
>tm␣
Trace mode      : normal          ... or "single delay"
Address        : 00000000
Data pattern   : 0000
Data mask      : ffff
Type          : Inst
Display option: normal
1.normal 2.single delay ...? normal :␣    ...Press [Enter] if you do not change any settings.
>
```

### (2) Changing settings

According to the guidance that appears after the current setup contents are displayed, input or choose the desired new setup contents.

```
1.normal 2.single delay ...? normal :2␣    ...Choose "2.single delay".
Trigger address 00000000 :00C00100␣    ...Input a trigger address.
Data pattern    0000 :␣    ...Press [Enter] to skip the setting (not changed).
Data mask      ffff :␣
Bus type 0:All 1:Inst 2:VecR 3:DatR 4:DatW 5:StkR 6:StkW 7:DmaR 8:DmaW ...? Inst :␣
1.start 2.middle 3.end ...? start :1␣    ...Choose a trigger position.
Display option 1.normal 2.source ...? normal :2␣    ...Choose a display option.
>
```

In this example, single delay trigger mode is selected so that the trace starts when the CPU fetches the instruction at address 0xc00100.

The guidance for selecting a trigger position appears only when single delay trigger mode is selected. It is not displayed when normal mode is selected.

The specified data pattern is 0x0000 but it does not affect the trigger condition since all the data bits are masked by the data mask 0xffff. In this example, the trace trigger condition is satisfied when the CPU fetches an instruction from address 0xc00100 regardless of the fetched instruction code. When including a data read/write in the trigger condition, data must be specified in 16 bits. Therefore, a data mask is required for setting a byte access condition. For example, to set a condition as a byte access with data 0x12, specify 0x1200 for the data pattern and mask the low-order 8-bits using the data mask 0x00ff. For an odd address, specify 0x0012 for the data pattern and 0xff00 for the data mask.

The display option allows selection of the [Trace] window display format. When "1. Normal" is selected, the [Trace] window displays only the information traced in the ICE33. When "2. Source" is selected, the source codes are displayed as well as the trace information.

A symbol or source line number can be used to specify an address.

```
Trigger address 00000000 :i␣    ...Sample entry of a symbol
Trigger Address 00000000 :main.c#24␣    ...Sample entry of a line number
```

To quit in the middle of guidance, press the [q] key and then the [Enter] key. When the command is suspended, already specified contents are validated.

To return to the immediately preceding guidance, press the [^] key and then the [Enter] key.

## ■ Note for ICE mode

The trigger address must be specified within the range of the memory area available for each microcomputer model. An error results if this limit is exceeded. In this case, the system brings up a guidance for entering addresses again.

```
Error: Address range (0-0xFFFFFFFF).    ...If an address exceeding 0xffffffff is specified.
Error: No map area.                    ...If an unused address is specified.
```

## ■ Input examples for ICD mode

### (1) Displaying current setup contents

When the tm command is executed, the current setup contents are displayed first.

```
>tm␣
Trace mode           : area           ...or "all"
Over write          : no             ...or "yes"
Measure mode        : all            ...or "area"
Trigger address1    : 0000100
Trigger address2    : 0000000
Address2 break      : on             ...or "off"
Counter display mode : accumulate     ...or "each instruction"
Trace mode 1.all 2.area ...? area :q␣ ...Press [q␣] if you do not change any settings.
>
```

### (2) Changing settings

According to the guidance that appears after the current setup contents are displayed, input or choose the desired new setup contents.

Setting all trace mode

```
Trace mode 1.all 2.area ...? area :1␣ ...Choose "1.all".
Over write 1.yes 2.no ...? no :1␣ ...Choose overwrite condition.
Counter display mode 1.accumulate 2.each instruction ...? accumulate :2␣
> ...Choose counter display mode.
```

In this example, all trace mode is selected, overwrite to the trace memory is enabled and the counter display mode is set to instruction units.

Setting area trace mode

```
Trace mode 1.all 2.area ...? all :2␣ ...Choose "2.area".
Measure mode 1.all 2.area ...? all :2␣ ...Choose time measurement condition.
Trigger address1 ? 0000100 :600000␣ ...Enter trigger address 1.
Trigger address2 ? 0000000 :600100␣ ...Enter trigger address 2.
Address2 break 1.on 2.off ...? on :1␣ ...Enable/disable address 2 break function.
Counter display mode 1.accumulate 2.each instruction ...? accumulate :1␣
...Choose counter display mode.
```

Warning: Hard PC break is not stopped at present mode.

>

In this example, the trace mode and the time measurement mode are set to "area". The trace range is set to 0x600000–0x60010 and break at 0x600100 is enabled. The clock count in the trace information will be displayed with the accumulated value.

To quit in the middle of guidance, press the [q] key and then the [Enter] key. When the command is suspended, already specified contents are validated.

To return to the immediately preceding guidance, press the [^] key and then the [Enter] key.

To skip a guidance, press the [Enter] key.

## ■ Note for ICE mode

- The trigger address must be specified within the range of the memory area available for each microcomputer model. An error results if this limit is exceeded. In this case, the system brings up a guidance for entering addresses again.
  - Error: Address range (0-0xFFFFFFFF). ...If an address exceeding 0xffffffff is specified.
  - Error: No map area. ...If an unused address is specified.
- The hardware PC break function is disabled when the area trace function is set. However, the set hardware PC break address is maintained and it will be enabled when the area trace function is cancelled.

■ Input examples for simulator mode

(1) Turning the trace mode on

```
>tm
Trace 1.on 2.off ...? 1┘                               ...Turn the trace mode on.
Display option 1.normal 2.register 3.source 4.register, source ...? 1┘
Display mode 1.window 2.file ... ? 1┘                 ...Set a window for the output destination.
>
```

```
>tm
Trace 1.on 2.off ...? 1┘                               ...Turn the trace mode on.
Display option 1.normal 2.register 3.source 4.register, source ...? 4
Display mode 1.window 2.file ... ? 2┘                 ...Set a file for the output destination.
File name ? : test.trc┘                               ...Specify an output file name.
R0 R1 R2 R3 R4 R5 R6 R7                               ...Choose to display registers.
R8 R9 R10 R11 R12 R13 R14 R15
SP AHR ALR
>
```

When the program is executed after the above is set up, trace information is displayed or output for every instruction executed. Command execution is terminated only when you input the [Enter] key in the middle of guidance. Refer to "Displaying trace information in the simulator mode" for the display option.

(2) Turning the trace mode off

```
>tm┘
Trace 1.on 2.off ...? 2┘                               ...Turn the trace mode off.
>
```

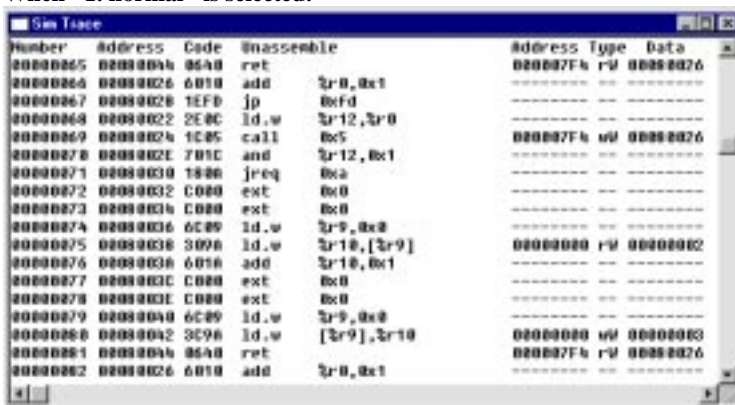
■ Displaying trace information in the simulator mode

If, when turning the trace mode on, a window is set for the output destination, the trace information is displayed in the [Trace] window irrespective of whether the program is run continuously or stepwise. If the [Trace] window is closed, the information is displayed in the [Command] window.

If a file is selected, the information is output to a file, and is not displayed in any window.

The [Trace] window shows the trace information from the latest one to that of maximum 255 instructions before. The following shows display examples according to the display option selected by the tm command.

When "1. normal" is selected:



When "3. source" is selected:

Number	Address	Code	Disassemble	Address	Type	Data	File	Line	SourceCode
0000059	00000028	6810	add	3r0,0x1			(main.c)	00011	for (j=0 ; ; j++)
0000060	00000029	1EFD	jp	0x0					
0000061	00000022	2E0C	ld.w	3r12,3r0			(main.c)	00013	sub(j);
0000062	0000002A	1C05	call	0x5	000007F4	w0 00000025			
0000063	00000025	781C	and	3r12,0x1			(main.c)	00020	if (k & kst)
0000064	00000030	1800	jreq	0x0					
0000065	000000A4	05A0	ret		000007F4	r0 00000025	(main.c)	00024	}
0000066	00000028	6810	add	3r0,0x1			(main.c)	00011	for (j=0 ; ; j++)
0000067	00000029	1EFD	jp	0x0					
0000068	00000022	2E0C	ld.w	3r12,3r0			(main.c)	00013	sub(j);
0000069	0000002A	1C05	call	0x5	000007F4	w0 00000025			
0000070	00000025	781C	and	3r12,0x1			(main.c)	00020	if (k & kst)
0000071	00000030	1800	jreq	0x0					
0000072	00000032	C000	ext	0x0			(main.c)	00022	i++;
0000073	00000034	C000	ext	0x0					
0000074	00000036	6C00	ld.w	3r9,0x0					
0000075	00000038	2000	ld.w	3r10,[3r9]	00000000	r0 00000002			
0000076	0000003A	6810	add	3r10,0x1					
0000077	0000003C	C000	ext	0x0					

When "4. register, source" is selected:

Number	Address	Code	Disassemble	Address	Type	Data	File	Line	SourceCode
0000000	00000000	00000005	00000000	00000000	00000000	00000000			
0000074	00000000	00000005	00000000	00000000	00000000	00000000			
0000152	0000002E	781C	and	3r12,0x1			(main.c)	00020	if (k & kst)
0000000	00000000	00000005	00000000	00000000	00000000	00000000			
0000074	00000000	00000005	00000000	00000000	00000000	00000000			
0000152	0000002E	781C	and	3r12,0x1			(main.c)	00020	if (k & kst)
0000000	00000000	00000005	00000000	00000000	00000000	00000000			
0000074	00000000	00000005	00000000	00000000	00000000	00000000			
0000152	0000002E	781C	and	3r12,0x1			(main.c)	00020	if (k & kst)
0000000	00000000	00000005	00000000	00000000	00000000	00000000			
0000074	00000000	00000005	00000000	00000000	00000000	00000000			
0000152	0000002E	781C	and	3r12,0x1			(main.c)	00020	if (k & kst)
0000000	00000000	00000005	00000000	00000000	00000000	00000000			
0000074	00000000	00000005	00000000	00000000	00000000	00000000			
0000152	0000002E	781C	and	3r12,0x1			(main.c)	00020	if (k & kst)
0000000	00000000	00000005	00000000	00000000	00000000	00000000			
0000074	00000000	00000005	00000000	00000000	00000000	00000000			
0000152	0000002E	781C	and	3r12,0x1			(main.c)	00020	if (k & kst)
0000000	00000000	00000005	00000000	00000000	00000000	00000000			
0000074	00000000	00000005	00000000	00000000	00000000	00000000			
0000152	0000002E	781C	and	3r12,0x1			(main.c)	00020	if (k & kst)
0000000	00000000	00000005	00000000	00000000	00000000	00000000			
0000074	00000000	00000005	00000000	00000000	00000000	00000000			
0000152	0000002E	781C	and	3r12,0x1			(main.c)	00020	if (k & kst)
0000000	00000000	00000005	00000000	00000000	00000000	00000000			
0000074	00000000	00000005	00000000	00000000	00000000	00000000			
0000152	0000002E	781C	and	3r12,0x1			(main.c)	00020	if (k & kst)
0000000	00000000	00000005	00000000	00000000	00000000	00000000			
0000074	00000000	00000005	00000000	00000000	00000000	00000000			
0000152	0000002E	781C	and	3r12,0x1			(main.c)	00020	if (k & kst)
0000000	00000000	00000005	00000000	00000000	00000000	00000000			
0000074	00000000	00000005	00000000	00000000	00000000	00000000			
0000152	0000002E	781C	and	3r12,0x1			(main.c)	00020	if (k & kst)
0000000	00000000	00000005	00000000	00000000	00000000	00000000			
0000074	00000000	00000005	00000000	00000000	00000000	00000000			
0000152	0000002E	781C	and	3r12,0x1			(main.c)	00020	if (k & kst)
0000000	00000000	00000005	00000000	00000000	00000000	00000000			
0000074	00000000	00000005	00000000	00000000	00000000	00000000			
0000152	0000002E	781C	and	3r12,0x1			(main.c)	00020	if (k & kst)

When "2. register" is selected from the display option, the display format is the same as "4" but the source part is not displayed.

Trace information is displayed in the [Command] window by using the same format as shown above.

The following lists the trace information that is displayed on the screen in simulator mode:

<1st line of each trace information>

- Number:** Executed instruction number (decimal).  
This is the executed instruction number after the CPU is reset or trace is turned on.
- Address:** Executed instruction address (hexadecimal).
- Code:** Instruction code (hexadecimal).
- Unassemble:** Disassembled content.
- Address:** Accessed memory address (hexadecimal).
- Type:** Bus operation type.  
rB: Byte data read, rH: Half word data read, rW: Word data read  
wB: Byte data write, wH: Half word data write, wW: Word data write
- Data:** Read/write data (hexadecimal).
- File:** Source file name (displayed only when source display is selected by the tm command).
- Line:** Source line number (displayed only when source display is selected by the tm command).
- SourceCode:** Source code (displayed only when source display is selected by the tm command).

<Lines 2-4 of each trace information>

These lines are displayed when register option is selected with the tm command.

The register values appear in the order shown below.

R0	R1	R2	R3	R4	R5	R6	R7
R8	R9	R10	R11	R12	R13	R14	R15
SP	AHR	ALR		PSR (displayed in flag units)			

## td (trace dump)

[ICD / ICE]

### ■ Function

This command displays the trace information that is sampled into the trace memory in the ICE33/ICD33.

### ■ Format

```
td [<No.>] (direct input mode)
<No.>: Trace cycle No. (decimal)
When omitted, trace data is displayed beginning with the latest data.
Condition: ICE mode 0 (latest data) ≤ No. ≤ 32767 (oldest data)
ICD mode 0 (latest data) ≤ No. ≤ 131071 (oldest data)
```

### ■ Display

#### (1) When the [Trace] window is open

```
>td,↓
>
```

When the td command is executed after breaking the program, the [Trace] window redisplay the latest data. The most recently traced data is shown on the bottom line of the window. All trace data can be displayed by scrolling the window.

When a trace cycle No. is specified, the data of a specified cycle is displayed on the bottom line of the window. In this case too, all trace data can be displayed by scrolling the window.

### Display example in ICE mode

When "normal" is selected from the display option of the tm command:

Cycle	Address	Code	Unassemble	Address	Data	Ck	Type	TRC
00152	000040	6E08	ld.w	3r8,0x26		1	Inst # SR00	
00151	000040	6D29	ld.w	3r9,0x12		1	Inst # SR00	
00150	000040	3889	ld.h	[3r8],3r9		1	Inst # SR00	
00149	000040	6C00	ld.w	3r8,0x0		1	Inst # SR00	
00148				00A0126	0012	2	Data M # I/O	
00147	000042	C000	ext	0x0		1	Inst # SR00	
00146	000044	C001	ext	0x1		1	Inst # SR00	
00145	000046	1E75	call	0x15		1	Inst # SR00	
00144	000048	1E74	jp	0x14		1	Inst # SR00	
00143	000000	6000	ld.w	3r0,0x10		2	Inst # SR00	
00142	000002	0000	ld.w	3rsr,3r0		1	Inst # SR00	
00141	000004	0000	sep			1	Inst # SR00	
00140	000006	0400	int	0x0		1	Inst # SR00	
00139	000008	0000	sep			1	Inst # SR00	
00138				0000030	0420	5	UecR # SR00	
00137				0000032	0000	1	UecR # SR00	
00136	000028	0000	sep			2	Inst # SR00	
00135	00002C	04C0	retl			1	Inst # SR00	

When "source" is selected from the display option of the tm command:

Cycle	Address	Code	Unassemble	Address	Data	Ck	Type	TRC	File	Line	SourceCode
00061				00A012E	5000	2	Data M # I/O				
00060				00A012E	0000	2	Data M # I/O				
00059	00006A	E024	ext	0x24		1	Inst # SR00		[area.s]	00050	xl.d.h [TTR],3r0
00058	000066	E134	ext	0x14		1	Inst # SR00				
00057	000068	3880	ld.h	[3r8],3r0		1	Inst # SR00				
00056	00006A	0400	int	0x0		1	Inst # SR00		[area.s]	00061	int #
00055				00A0134	0000	2	Data M # I/O				
00054	00006C	0000	nap			1	Inst # SR00		[area.s]	00062	nap
00053				0C00030	0420	7	UecR # SR00				
00052				0C00032	00C0	2	UecR # SR00				
00051	0C00A20	0000	nap			5	Inst # SR00		[area.s]	00097	nap
00050	0C00A2C	04C0	retl			3	Inst # SR00		[area.s]	00098	retl
00049	0C00A2C	0000	nap			3	Inst # SR00		[area.s]	00080	nap
00048	00006C	0000	nap			4	Inst # SR00		[area.s]	00062	nap
00047	00006E	6C00	ld.u	3r0,0x0		1	Inst # SR00		[area.s]	00063	xl.d.w 3r0,(00E0_06E
00046	000070	E024	ext	0x24		1	Inst # SR00		[area.s]	00064	xl.d.h [TTR+2],3r0
00045	000072	E136	ext	0x16		1	Inst # SR00				
00044	000074	3400	ld.b	[3r8],3r0		1	Inst # SR00				

The following lists the contents of trace information displayed in ICE mode:

- Cycle:** Trace cycle (decimal). The last information taken into the trace memory becomes 00000.
- Address:** CPU-instruction-fetch address (hexadecimal).  
"-----" is displayed for a non instruction-fetch access.
- Code:** Instruction code fetched by the CPU (hexadecimal).  
"----" is displayed for a non instruction-fetch access.
- Unassemble:** Disassembled content of the fetched instruction.  
"-----" is displayed for a non instruction-fetch access.
- Address:** Address accessed by the CPU (hexadecimal).  
"-----" is displayed for an instruction-fetch access.
- Data:** Read/write data (hexadecimal).  
"----" is displayed for an instruction-fetch access.
- Clk:** Number of clocks used in the bus operation (1 to 7).  
"V" is displayed when 8 or more clocks are used.
- Type:** Bus operation type:  
Inst: Instruction fetch, VecR: Vector read, DatR: Data read, DatW: Data write  
StkR: Stack read, StkW: Stack write, DmaR: DMA read, DmaW: DMA write  
Access size:  
B: Byte access, H: Half word access, W: Word access  
Memory type:  
SRAM, DRAM, BROM (burst ROM), IRAM (internal RAM), I/O (internal I/O)  
DEBUG (for ICE development), ERR (others)
- TRC:** Input to TRCIN pin (denoted by L when low-level signal is input).
- T:** Trace trigger point (placed at the beginning of the line).  
Displayed only for the bus cycle that meets trace trigger conditions.
- File:** Source file name (displayed only when source display is selected by the tm command).
- Line:** Source line number (displayed only when source display is selected by the tm command).
- SourceCode:** Source code (displayed only when source display is selected by the tm command).

## Display example in ICD mode

Cycle	Address	Code	Unassemble	Clk	Method	File	Line	SourceCode
000017	0002008	0FF8	ext	01FFF	000002	DPC		
000018	0002008	0FF8	ext	01FFF	000003	DPC		
000019	000200C	1CA4	call	0144	000004	DPC		
000020	0002014	6C8C	ld.u	0112,0x0	000005	DPC (sgs.c)	00001	ifBytes = 0; /* no read mem */
000021	0002018	080C	ext	0x0	000006	DPC (sgs.c)	00002	for (;;)
000022	0002018	CA81	ext	0x0	000007	DPC		
000023	0002018	6C8F	ld.u	0115,0x0	000008	DPC		
000024	000201C	6C14	ld.u	0114,0x1	000009	DPC		
000025	000201C	688E	cmp	0114,0x0	000010	DPC (sgs.c)	00001	if (iheadBytes == 0) /* if require
000026	0002020	1986	jr.nq	0x0	000011	DPC		
000027	0002022	0388	ext	0x0	000012	DPC (sgs.c)	00002	if (HEAD_EOF == 1)
000028	0002024	0889	ext	0x0	000013	DPC		
000029	0002025	2485	ld.uh	0115,[0x0]	000014	DPC		
000030	0002028	6815	cmp	0115,0x1	000015	DPC		
000031	0002028	1A8D	jr.se	0x0	000016	DPC		
000032	0002030	248E	ld.uh	0111,[0x15]	000017	DPC (sgs.c)	00001	iSize = HEAD_BUF[0];
000033	0002032	688E	cmp	0111,0x0	000018	DPC (sgs.c)	00002	if (iSize > 0)
000034	0002034	0E14	jr.le	0x14	000019	DPC		

The following lists the contents of trace information displayed in ICD mode:

- Cycle:** Trace cycle (decimal)  
The last information taken into the trace memory becomes 000000.
- Address:** CPU-instruction-execution address (hexadecimal)
- Code:** Instruction code executed by the CPU (hexadecimal)
- Unassemble:** Disassembled content of the instruction code
- Clk:** Number of clocks used for executing the instruction  
By default, the cumulative clock count from start of tracing is displayed. It can be changed so that the number of clocks for each executed instruction is displayed.
- Method:** Trace analytical method (to get the executed PC address)
- SPC: Analyzed with the start PC address
  - TRG: Analyzed with the trigger address
  - DPC: Analyzed with the DPCO signal
  - RET: Analyzed with the call/ret statement
  - MAP: Analyzed with the map information
  - RTI: Analyzed with the reti statement
  - : Cannot be analyzed
- File:** Source file name (which includes the executed instruction)
- Line:** Source line number
- SourceCode:** Source code

In ICD mode, the trace information can also be displayed while the program is being executed. By clicking the [Display trace] button, the ICD33 suspends tracing and displays the sampled trace memory data to the [Trace] window. The trace operation can be resumed by clicking the [Resume trace] button.



[Display trace] button



[Resume trace] button

## (2) When the [Trace] window is closed

16 lines (default) of trace data are displayed in the [Command] window. The number of display lines can be changed using the md command.

The latest data is shown on the bottom line of the window if trace cycle No. is omitted. When a trace cycle No. is specified, data of the specified cycle is shown on the bottom line.

## (3) Logging

To save the command execution results to a log file, close the [Trace] window and display the results in the [Command] window. If the [Trace] window is open, the display contents will not be saved in the file because the [Command] window does not display the results.

**(4) Successive display**

Once you execute the `td` command, data can be displayed successively with the [Enter] key only until some other command is executed.

When you hit the [Enter] key, the [Trace] window is scrolled forward one full screen.

When displaying data in the [Command] window, data is displayed for the 16 lines (default) following the previously displayed address.

The direction of display is such that each time you input the [Enter] key, data on older execution cycles is displayed (FORWARD). This direction can be reversed (BACKWARD) by entering the [B] key. To return the display direction to FORWARD, input the [F] key. If the [Trace] window is open, the direction in which the window is scrolled is also changed.

```
>t d 100.␣          ...Started display in FORWARD.
(Data on cycle Nos. 115 to 100 is displayed.)
>b.␣              ...Changed to BACKWARD.
(Data on cycle Nos. 99 to 84 is displayed.)
>.␣              ...Continued display in BACKWARD.
(Data on cycle Nos. 84 to 69 is displayed.)
>f.␣            ...Changed back to FORWARD.
(Data on cycle Nos. 99 to 84 is displayed.)
>
```

**■ Notes for ICE mode**

- Specify the trace cycle No. within the range of 0 to 32767. An error results if this limit is exceeded.  
Error: Trace range (0-32767).
- For reasons of the ICE33 operation timing, the trace data at the boundary of operations, such as in the fetch cycle at which trace starts or the execution cycle at which trace ends, will not always be stored in memory.
- After a single-step execution or a break occurs, information of the pre-fetched instructions that have not been executed are displayed. When the target program execution is suspended by a software PC break, the fetch cycle information of the `brk` instruction that was inserted for the software PC break is also displayed.
- When the program starts a successive execution from an address set as a software PC break point, the ICE33 executes single-stepping before starting the successive execution. Therefore, redundant trace information pre-fetched by the single-stepping may be displayed.
- For source-level step execution, the ICE33 repeats single-stepping internally. Therefore, a lot of pre-fetch information of all the steps will be displayed.
- Trace data for read/write of the internal RAM cannot be referred since the bus access is undetectable.
- During data transfer by the high-speed DMA, data cannot be traced properly.



■ Notes for ICD mode

- Specify the trace cycle No. within the range of 0 to 131071. An error results if this limit is exceeded.  
Error: Trace range (0-131071).
- The program executed address cannot be analyzed in the following cases:
  - Programs in which the call instructions do not correspond to the return instruction one by one
  - Programs in which the interrupts do not correspond to the reti instruction one by one
  - Programs that contain the retf instruction.
  - After a fill or move command is executed (executing the rm command enables analyzing)
  - When the routine that is moved by the user program or is transferred with DMA is executed
  - When a program below is executed from the label "aaa:"

```

ext      ....
aaa:
        ext      ....
        call     ....
    
```

Furthermore, the analyze-accuracy is decreased when the program is executed from the address set as a software PC break point.

**ts (trace search)**

[ICD / ICE]

**■ Function**

This command searches trace information from the trace memory under a specified condition.  
In ICE mode, the search condition can be selected from two available conditions:

**1. Address**

The cycle in which a specified address was accessed is searched. This condition can be specified to search the entire memory area.

**2. Bus operation type**

The cycle in which a specified bus operation was performed is searched. A bus operation type can be selected from the following:

0. All	...All bus operations	5. StkR	...Stack read cycle
1. Inst	...Instruction fetch cycle	6. StkW	...Stack write cycle
2. VecR	...Vector fetch cycle	7. DmaR	...DMA read cycle
3. DatR	...Data read cycle	8. DmaW	...DMA write cycle
4. DatW	...Data write cycle		

In ICD mode, a program execution address can be specified as the search condition.

It is also possible to display the information before and after the searched line in the range of 0 to 256 lines each.

**■ Format**

**ts** (guidance mode)

**■ Input example****ICE mode**

```
>ts␣
Search address(* is all area) ? : 200␣ ...Specify an address (input * for the entire area).
Bus type 0:All 1:Inst 2:VecR 3:DatR 4:DatW 5:StkR 6:StkW 7:DmaR 8:DmaW ...? 2
Number of pre lines (0-256) ? : 2␣ ...*1
Number of post lines (0-256) ? : 3␣ ...*2
Find 0 trace data. (0 lines) ...Displays the number of lines searched.
>
```

**ICD mode**

```
>ts␣
Search address ? : 811038␣ ...Specify an address
Number of pre lines (0-256) ? : 1␣ ...*1
Number of post lines (0-256) ? : 2␣ ...*2
Find 2001 trace data. (8003 lines) ...Displays the number of lines searched.
>
```

\*1 Number of lines to display the data preceding the searched line

\*2 Number of lines to display the data following the searched line

**■ Displaying search results**

The search result (occurrences found) is displayed in the [Command] window.

The trace information is displayed in order of the trace cycle number.

**(1) When the [Trace] window is open**

The searched trace information is displayed in the [Trace] window.

The [Trace] window is switched to the search mode so that the searched data can be displayed successively by scrolling the window in the vertical direction. This display mode remains effective until you input the td command.

**(2) When the [Trace] window is closed**

The 16 lines (default) of searched data are displayed in the [Command] window. The number of display lines can be changed using the md command. The display mode here is the same as with the td command. Also, if the search result includes more than 16 occurrences, data is displayed in the same way as for the td command.

■ Notes

- The ts command can only be used in ICE and ICD modes.
- The address must be specified within the range of the memory area available for each microcomputer model. An error results if this limit is exceeded. In this case, the system brings up a guidance for entering addresses again.

Error: Address range (0-0xFFFFFFFF).	...If an address exceeding 0xffffffff is specified.
Error: No map area.	...If an unused address is specified.

**tf (trace file)****[ICD / ICE]****■ Function**

This command saves a specified range of data in the trace memory or from the search results of the ts command (if immediately after execution of the ts command) to a file.

**■ Format**

**tf** (guidance mode)

**■ Input example****ICE mode**

```
>tf␣
Start cycle number (max 32767) ? : 2000␣ ...Save start cycle number is input.
End cycle number (min 0) ? : 0␣ ...Save end cycle number is input.
File name ? : d:\trace.txt␣ ...File name is input.
Processing 2000-1001 cycle. ...Displays the progress in 1000 cycle units.
Processing 1000-1 cycle.
Processing 0-0 cycle.
>
```

**ICD mode**

```
>tf␣
Start cycle number (max 131071) ? : 1000␣ ...Save start cycle number is input.
End cycle number (min 0) ? : 0␣ ...Save end cycle number is input.
File name ? : d:\trace.txt␣ ...File name is input.
Reading trace data.
Making file.
>
```

**■ Notes**

- The tf command can only be used in ICE and ICD modes.
- When an existing file is specified, the file is overwritten with new data.
- The search results of the ts command are saved in the same order of the numbers displayed beginning with the smallest number.

## 16.9.13 Simulated I/O

### stdin (standard input)

[ICD / ICE / SIM / MON]

#### ■ Function

This command sets up the environment necessary to input data from the [Simulated I/O] window or a file, and pass it on to the program. Use this command to set up the following conditions:

- Break address (a position at which the db33 takes in data)
- Input buffer address (a 65-byte buffer)
- Input device ([Simulated I/O] window or a file)

For preparation on the program side, refer to Section 16.8.8, "Simulated I/O".

#### ■ Format

**stdin** (guidance mode)

#### ■ Input examples

##### (1) Setting

```
>stdin↓
Break address : ***** ...Current setup contents (***** denotes that there are no settings.)
Buffer address : *****
1. set 2. clear ... ? 1↓ ...Choose "1. set".
Break address ? : READ_FLASH↓ ...Set a break address.
Buffer address ? : READ_BUF↓ ...Set an input buffer address.
Input mode 1. window 2. file ... ? 1↓ ...Specify the source from which data is input.
>
```

If the program is run continuously after setting up the above, the db33 stops executing at the position of a label READ\_FLASH in the program, and brings up the [Simulated I/O] window. When you input data in the window and press the [Enter] key, the db33 takes in the input data into the input buffer (READ\_BUF), then restarts executing the program.

If you chose "2. file" in the input mode, input a file name too.

```
Input mode 1. window 2. file ... ? 2↓
File name ? : input.txt↓ ...Specify a file name from which you want to input data.
>
```

If you chose a file for the input source, the db33 takes in one line of data from a specified file at the break position without bringing up the [Simulated I/O] window.

To terminate command execution, input only the [Enter] key in the middle of guidance.

##### (2) Clearing

```
>stdin↓
Break address : 0008017C READ_FLASH ...Current setup contents
Buffer address : 00000048 READ_BUF
1. set 2. clear ... ? 2↓ ...Choose "2. clear".
>
```

The data input function is deactivated.

#### ■ Notes

- The break address you set in the stdin command cannot overlap any software PC breakpoint. In such a case, clear the software PC breakpoint before you execute the stdin command. Overlapping with a hardware PC breakpoint is accepted.
- The break and buffer addresses must be specified within the range of the memory area available for each microcomputer model. An error results if this limit is exceeded. In this case, the system brings up a guidance for entering addresses again.
  - Error: Address range (0-0xFFFFFFFF). ...If an address exceeding 0xffffffff is specified.
  - Error: No map area. ...If an unused address is specified.
- When using the simulated I/O function in ICE mode, the display response time is improved by setting the baud rate to 115200bps and the on-the-fly interval to 0 (md command).

**stdout (standard output)****[ICD / ICE / SIM / MON]****■ Function**

This command sets up the environment necessary to output data from a specified output buffer to the [Simulated I/O] window or a file. Use this command to set up the following conditions:

- Break address (a position at which the db33 outputs data)
- Output buffer address (a 65-byte buffer)
- Output device ([Simulated I/O] window or a file or both)

For preparation on the program side, refer to Section 16.8.8, "Simulated I/O."

**■ Format**

**stdout** (guidance mode)

**■ Input examples****(1) Setting**

```
>stdout␣
Break address : *****      ...Current setup contents (***** denotes that there are no settings.)
Buffer address : *****
1. set 2. clear ...? 1␣      ...Choose "1. set".
Break address ? :WRITE_FLASH␣      ...Set a break address.
Buffer address ? :WRITE_BUF␣      ...Set an output buffer address.
Output mode 1. window 2. file 3. window & file ...? 3␣      ...Specify the destination.
File name ? :output.txt␣      ...Specify a file name to which data is output.
>
```

If the program is run continuously after the above is set up, the db33 stops executing at the position of a label WRITE\_FLASH in the program and brings up the [Simulated I/O] window. Next, the db33 outputs data from a specified buffer (WRITE\_BUF) to the [Simulated I/O] window and a specified file. If you only specified a file for the output destination, the [Simulated I/O] window is not opened.

To terminate command execution, input only the [Enter] key in the middle of guidance.

**(2) Clearing**

```
>stdout␣
Break address : 000801D0 WRITE_FLASH      ...Current setup contents
Buffer address : 00000004 WRITE_BUF
1. set 2. clear ...? 2␣      ...Choose "2. clear".
>
```

The data output function is deactivated.

**■ Notes**

- The break address you set in the stdout command cannot overlap any software PC breakpoint. In such a case, clear the software PC breakpoint before you execute the stdout command. Overlapping with a hardware PC breakpoint is accepted.
- The break and buffer addresses must be specified within the range of the memory area available for each microcomputer model. An error results if this limit is exceeded. In this case, the system brings up a guidance for entering addresses again.
  - Error: Address range (0-0xFFFFFFFF). ...If an address exceeding 0xffffffff is specified.
  - Error: No map area. ...If an unused address is specified.
- When using the simulated I/O function in ICE mode, the display response time is improved by setting the baud rate to 115200bps and the on-the-fly interval to 0 (md command).

## 16.9.14 Other Commands

### com (execute command file)

[ICD / ICE / SIM / MON]

#### ■ Function

This command reads a command file and successively executes the debug commands written in that file.

#### ■ Formats

- (1) **com** (guidance mode)
- (2) **com <file name>** (direct input mode)  
 <file name>: Command file name

#### ■ Input examples

```
File name = startup.cmd
; Load file           ...A description from ";" to the end of the line is regarded as a comment.
lf test.srf
; Cold reset
rstc
; Display mode
m BOOT

>com␣
File name ? :startup.cmd␣    ...Command file name is input.
>lf test.srf
>rstc
>m BOOT
>
```

The commands written in the file are displayed in the [Command] window as they are executed.

#### ■ Notes

- Another command file can be read in from within a command file. However, the nesting of command files is limited to a maximum of five levels. An error will result if a com (or cmw) command at the sixth level is encountered, and the subsequent execution will be halted.

Error: Cannot open file, <file name>.

- By specifying the -c option with the db33 startup command, you can execute one command file simultaneous with the startup of the debugger.

Example: db33 -c startup.cmd -p 88104\_1.par

- Once the commands described in the specified command file are executed by entering the com command, the commands can be executed repeatedly by pressing the [Enter] key until another command is executed similarly to the g, s and n commands.

For example, if the command file "test.cmd" contains the following two commands, they can be repeatedly executed after once the com command is executed.

s	
db 800100	... Contents of test.cmd
>com test.cmd␣	... Executes "s" and "db 800100".
>␣	... Repeats execution of the above commands.
>␣	... Repeats execution of the above commands.

This makes it possible to repeat multiple commands using the [Enter] key only.

- The [Key break] button can be used to suspend the command execution by a command file. When a command that takes a long execution time (fill command for large area, etc.) is executed, keep the mouse button pressed until the operation is accepted.

**cmw (execute command file with wait)****[ICD / ICE / SIM / MON]****■ Function**

This command reads a command file and executes the debug commands written in that file at predetermined time intervals.

The execution interval of each command can be set in a range of 1 to 256 seconds (in 1-second increments) using the md command. In the initial debugger settings, the execution interval is 1 second.

**■ Formats**

- (1) **cmw** (guidance mode)
- (2) **cmw <file name>** (direct input mode)  
     <file name>: Command file name

**■ Input example**

```
>cmw␣
File name ? :infodisp.cmd␣           ...Command file name is input.
:                                       ...Commands are executed.
```

**■ Notes**

- Another command file can be read in from within a command file. However, the nesting of command files is limited to a maximum of five levels. An error will result if a cmw (or com) command at the sixth level is encountered, and the subsequent execution will be halted.  
     Error: Cannot open file, <file name>.
- If the cmw command is written in the command file that you want to be read by the com command, all other commands following that command in the file (even when a com command is included) will be executed at predetermined time intervals.
- The cmw allows repeat execution by the [Enter] key similar to the com command.
- The [Key break] button can be used to suspend the command execution by a command file. When a command that takes a long execution time (fill command for large area, etc.) is executed, keep the mouse button pressed until the operation is accepted.



**log (logging)**

[ICD / ICE / SIM / MON]

**■ Function**

This command saves the input commands and the execution results of the commands that are displayed in the [Command] window to a file.

**■ Formats**

- (1) **log** (guidance mode)
- (2) **log <file name>** (direct input mode)  
     <file name>: Log file name

**■ Saved contents**

The contents displayed in the [Command] window are written as displayed directly to the log file.

The commands executed from a tool bar or menu and the execution results displayed in other windows are not displayed in the [Command] window, so they are not output to a file either. To save a log, close all windows other than the [Command] window before you execute the log command.

**■ Input examples**

```
>log␣
File name ? :log1.log␣      ...Log file name is input.
log on                       ...Starts outputting a log.
>
: Log output remains effective until the log command is executed next.
>log␣
log off                       ...Finishes outputting a log.
>
```

**■ Notes**

- When an existing file is specified, the file is overwritten with new data.
- When outputting a log, close all windows other than the [Command] window and increase the number of lines for the execution results to be displayed in the [Command] window (16 lines by default) by using the md command. This will help you reduce the labor and time required for key operation.

**od (option data dump)****[ICE]****■ Function**

This command displays option data in the [Command] window in a hexadecimal dump format after reading it from the ICE33.

**■ Format**

**od** (guidance mode)

**■ Input example**

Input the start and end addresses of the display range sequentially in the order given by the guidance.

```
>od␣
Start address ? :0␣
End address ? :f␣
   0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
>
```

- The start and end addresses can be omitted by entering the [Enter] key only. If the start address is omitted, data is displayed beginning with address 0. If the end address is omitted, the end address of the option area is assumed.
- The maximum number of lines that can be displayed at once is 16 (default). Even if you specify the end address in an attempt to display more than 16 lines, the db33 will only display data for 16 lines and then stand by waiting for a command input. The following addresses are displayed by entering the [Enter] key. The number of display lines can be changed using the md command.
- Data in unused areas is marked by an "\*" as it is displayed in the window.

**■ Notes**

- The od command cannot be executed in the modes other than ICE mode.  
Error: Command is not supported at present mode.
- Both the start and end addresses must be specified within the setup range of the option. An error results if this limit is exceeded.  
Error: Address range (0-0x3FFF). ...Specified address is outside the range.
- An error results if the start address is larger than the end address.  
Error: address1 > address2
- The default value of option data is 0.

**ct (change type)**

[ICD / ICE / SIM / MON]

**■ Function**

This command converts the input numeric values and character strings to other display formats before displaying them on the screen.

**■ Formats**

- (1) **ct** (guidance mode)
- (2) **ct <value>** (direct input mode)
- (3) **ct <option><value>** (direct input mode)
  - <value>: Numeric value to be converted (decimal, hexadecimal, real number)
  - <option>: : ...Converts binary
  - . ...Converts hexadecimal to double type
  - ' ...Converts hexadecimal to character string
  - " ...Converts character string to hexadecimal

**■ Input examples****(1) Guidance input**

```
>ct␣
Value ? :12345␣          ...Input the numeric value to be converted.
bin   : 000000000000000001100000111001
hex   : 00003039
>
```

**(2) Converting a binary number**

Add a colon (:) at the beginning of a binary number when you input it. The input binary number is converted to a decimal, hexadecimal and single-precision real number representation.

```
>ct :01000001␣
dec   : 65
hex   : 00000041
float : 9.108440018e-044          ..."float" is displayed down to 9 decimal places.
>
```

**(3) Converting a decimal number**

Add a minus sign (-) to a negative number when you input it. Do not add a "+" for any positive number. The input decimal number is converted to a binary and hexadecimal number.

```
>ct 123456␣
bin   : 00000000000000011110001001000000
hex   : 0001E240
>ct -1␣
bin   : 11111111111111111111111111111111
hex   : FFFFFFFF
>
```

**(4) Converting a hexadecimal number****Conversion to binary, decimal or single-precision real number representation**

Normal hexadecimal representation is converted to a binary, decimal and single-precision real number representation.

```
>ct 0x41abcd␣
bin   : 0000000010000011010101111001101
dec   : 4303821
float : 6.030937758e-039
>
```



**ext (extended instruction)****■ Function**

This command calculates the immediate data extended by the ext instruction and returns the result in the extended instruction format of the instruction extender.

**■ Format**

- (1) **ext** (guidance mode)  
 (2) **ext <address>** (direct input mode)  
 <address>: Target instruction address (hexadecimal, symbol or source line number)  
 Condition:  $0x0 \leq \text{address} \leq 0xfffffff$

**■ Input examples****Guidance mode**

```
>ext␣
Address ? :80100␣    ...Input a target instruction address.
:
```

**Direct input mode**

```
>ext 80100␣
:
```

**(1) Branch instruction**

When the ext instruction has not been used:

```
address code instruction
C00000 0000 nop
C00002 1E0F jp      0xf

>ext c00002
xjp      0x1e      (0x00C00020)
>
```

For branch instructions, the immediate data is extended with 0 at the LSB. ( ) indicates the branch destination address.

When the ext instruction has been used:

```
address code instruction
C00000 0000 nop
C00002 C010 ext    0x10
C00004 C000 ext    0x0
C00006 1E0F jp      0xf

>ext c00006
xjp      0x800000  (0x00140006)
>
```

**(2) Other instructions**

```
address code instruction
C00000 0000 nop
C00002 C100 ext    0x100
C00004 3021 ld.w   %r1, [%r2]

>ext c00004
xld.w   %r1. [%r2+0x100]
>
```

**■ Notes**

- An error results if an address of the instruction that cannot be extended by the ext instruction is specified.  
Error: Target instruction cannot be extended.
- Up to two ext instructions immediately preceding the specified address are effective for the calculation.
- The address must be specified within the range of the memory area available for each microcomputer model. An error results if this limit is exceeded.  
Error: Address range (0-0xFFFFFFFF).                   ...If an address exceeding 0xffffffff is specified.  
Error: No map area.   ...If an unused address is specified.
- Specify a half word boundary address (even address) for the address. If odd address is specified, a warning is generated and the LSB of the specified address is rewritten to 0.  
Warning: Round down to multiple of 2.

**ma (map information)**

[ICD / ICE / SIM / MON]

**■ Function**

This command displays the map information that is set by a parameter file.

**■ Format**

**ma** (direct input mode)

**■ Display**

After the command is input, the db33 displays the chip name, version of the parameter file, and map information in each area.

Example: map information when the area from 0x600000 to 0x6ffff is set to big endian.

```
>ma
Chip name           : 33208
Parameter file version : 01
Internal ROM area   : 80000 - 80FFF
Size of F0 area     : 0000
Emulation memory area (not used in simulator mode)
  00C00000 - 00CFFFFF RAM
Memory map informatiopn
  00000000 - 000007FF RAM
  00040000 - 0004FFFF IO
  00200000 - 002FFFFF RAM
  00600000 - 006FFFFF RAM .Big endian
  00C00000 - 00CFFFFF ROM emulation
  00600000 - 006FFFFF Stack area
>
```

**md (mode)****[ICD / ICE / SIM / MON]****■ Function**

This command sets the debugger modes described below.

**1. Step display for single-step execution results and [Memory] window**

When the step display mode is set to on, the execution results of all steps will be displayed during single-step (s, n command) operation. When the step display mode is set to off, the execution result of only the last step will be displayed. The register values are updated when their contents are displayed in the [Register] window; they are displayed in the [Command] window if the [Register] window is closed. If the [Source] window is open, the displayed lines are underlined as they are executed according to the setting of this mode.

The [Memory] window while the step display mode is on updates its display contents every step during single-step operation or updates after a break has occurred during successive execution. When the step display mode is off, the [Memory] window is not updated automatically. To update the window, it is necessary to execute a memory dump command or to scroll the window.

**2. Mode of execution counter**

This can be selected from the integration mode or the reset mode. In reset mode, the counter value is reset to 0 each time you enter a program execution command (including execution by the [Enter] key).

The value of the execution counter is also reset when you switch the integration mode to the reset mode.

**3. ICD execution counter function (only for ICD mode)**

The measurement unit of the ICD33 execution counter can be selected from three types: cycle units, second units and  $\mu$ sec units.

**4. Number of lines for displaying command execution results**

When displaying the execution results of the commands listed below in the [Command] window, you can choose the number of lines that you want displayed at a time from 1 to 1,000 lines.

Applicable commands: db, dh, dw, sc, m, u, sy, sw, sa, sd, od, td, ts

**5. cmw command wait time**

A cmw command wait time can be set in the range of 1 to 256 seconds (in 1-second increments).

**6. TAB stop**

The TAB stops used in the source display can be set every 2, 4, or 8 characters.

**7. Displaying on-the-fly information (only for ICE and ICD modes)**

You can choose the display interval of the on-the-fly information from 0 to 10 (times) per second. When 0 is chosen, the on-the-fly information will not be displayed.

Default values of debugger modes

Mode	ICE mode	ICE mode	SIM mode	MON mode
Step display	On	On	On	On
Execution counter mode	Integrating	Integrating	Integrating	Integrating
ICD execution counter	–	Number of cycles	–	–
Number of display lines	16 lines	16 lines	16 lines	16 lines
cmw wait time	1 second	1 second	1 second	1 second
TAB stop	8	8	8	8
On-the-fly function	5 times per second	5 times per second	OFF (fixed)	OFF (fixed)



### ■ Format

**md** (guidance mode)

### ■ Input example

The db33 displays the contents of current settings to provide guidance that you can follow as you perform the operations below.

```
>md␣
Step and memory window each display mode : on
Counter mode          : hold
Counter function      : cycle
Number of display line: 16 line
Cmw wait time         : 1 s
Tab stop              : 8
On the fly interval   : 5 times/sec

Step and memory window each display mode 1.on 2.off ...? on :1␣
Counter mode          1.reset 2.hold ...? hold :2␣
Counter function 1.cycle 2.time[us] 3.time[s] ? cycle :2␣
Number of display line 1 - 1000 line ...? 16 line :16␣
Cmw wait time         1 - 256 s ...? 1 s :1␣
Tab Stop              2, 4, 8 size ...? 8 :8␣
On the fly interval   0 - 10 times/sec ...? 5 times/sec :2␣
>
```

The above example applies to the ICD mode. In other modes, the set value and guidance for "Counter function" is not displayed. In simulator and debug monitor mode, the set value and guidance for "On the fly interval" is not displayed.

If you enter the [Enter] key only in the middle of a guidance, the previously set data will not be modified.

To quit in the middle, press the [q] key and then the [Enter] key. The contents you have input up until that time will be modified.

The [^] key allows you to return to the immediately preceding guidance.

### ■ Note

The actual interval of the on-the-fly display is obtained from the expression below.

$$(1 [\text{sec}] / \text{Count set}) + (\text{Overhead of the PC, RS232C interface and ICE33} [\text{sec}]) = \text{display interval} [\text{sec}]$$

The overhead varies depending on the performance of the PC and baud rate of the RS232C interface. Be aware that there is a 0.05 sec to 0.1 sec overhead in this system.

The debugger checks a break generation and simulated I/O status in the on-the-fly interval. To improve these responses, set the on-the-fly interval to 10 or 0 (OFF).

**q (quit)****[ICD / ICE / SIM / MON]**

---

**■ Function**

This command quits the debugger.

If the COM port, parallel port, log file, or command file are open, they will close when you execute this command.

**■ Format**

**q** (direct input mode)

- \* The db33 can also be terminated by selecting the [Exit] command from the [File] menu.

**? (help)**

[ICD / ICE / SIM / MON]

**■ Function**

This command displays the input format of each command.

**■ Formats**

- (1) ? (direct input mode)
- (2) ? <n> (direct input mode)
- (3) ? <command> (direct input mode)
  - <n>: Command group number (decimal)
  - <command>: Command name
  - Condition:  $0 \leq n \leq 8$

**■ Display**

When you input the command in Format 1 or 2, the db33 displays a list of commands classified by function. Use the command in Format 3 if you want to display the input format of each individual command.

**Format 1)**

```
>?␣
group 1: memory ..... fb, fh, fw / db, dh, dw, df / eb, eh, ew / mv, mvh, mvw / w / rm
group 2: execution & register ..... g, s, n, rstc, rsth / int / rd, rs
group 3: break ..... bp, bs, bc, bh, bhc, bd, bsq, bl, bac, bh2, bhc2
group 4: source & symbol ..... u, sc, m / ss / sy, sa, sw, sd
group 5: file & flash memory ..... lf, lh / lo / ld / fls, fle / lfl, sfl, efl, maf (only for ICE)
group 6: trace & simulated I/O ..... tm, td, ts, tf / stdin, stdout
group 7: others ..... com, cmw, log / od, ct, ext / ma, md, q, ?
group 8: input number method ..... number, data, address, linenum, symbol
Please type "? 1" to show group 1 or type "? fb" to get usage of command "fb".
>
```

**Format 2)**

```
>? 1␣
group 1: memory
fb (fill byte), fh (fill half), fw (fill word),
db (dump byte), dh (dump half), dw (dump word),
eb (enter byte), eh (enter half), ew (enter word),
mv (move), mvh (move half), mvw (move word), df (dump file),
w (watch data),
rm (read memory)
Please type "? fb" to get usage of command "fb".
>
```

**Format 3)**

```
>? fb␣
fb (fill byte): fill memory with byte data
usage: >fb addr1 addr2 data ... fill data from addr1 to addr2
>fb ... fill memory with guidance
Start address ? :addr1 ... input start address
End address ? :addr2 ... input end address
Data pattern ? :data ... input data pattern
(data:0x0-0xFF)
>
```

**ice (ice)****[ICE]****■ Function**

This command sends specified data directly to the ICE33. After transmitting data, the db33 displays returned data from the ICE33 in hexadecimal form.

**■ Format**

**ice** (guidance mode)

**■ Input examples**

```
>ice␣
1. OK 2. CAN 3. Message ...? 1␣
Control code = 70
>

>ice␣
1. OK 2. CAN 3. Message ...? 2␣
Control code = 40
Error code = 08
Error address= 00000000
>

>ice␣
1. OK 2. CAN 3. Message ...? 3␣
Message ID ? :7␣
Send data ---[CR] : Quit guidance ---q[CR]
Data0001 ? :00␣
Send data ---[CR] : Quit guidance ---q[CR]
Data0002 ? :c0␣
Send data ---[CR] : Quit guidance ---q[CR]
Data0003 ? :00␣
Send data ---[CR] : Quit guidance ---q[CR]
Data0004 ? :30␣
Send data ---[CR] : Quit guidance ---q[CR]
Data0005 ? :00␣
Send data ---[CR] : Quit guidance ---q[CR]
Data0006 ? :21␣
Send data ---[CR] : Quit guidance ---q[CR]
Data0007 ? :␣
Data size = 0022
ID = 07
Data = 90 1C 00 C0 FF D7 E5 1C FD 1E AA AA AA AA AA AA
      AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA
      AA
BCC = D3
>
```

To quit in the middle, press the [q] key and then the [Enter] key. The [^] key allows you to return to the immediately preceding guidance.

The transfer data should be input as a hexadecimal number within the range from 0x00 to 0xff or a decimal number within the range from +0 to +255. Up to 8192 bytes of message are allowed for transmission.

When only the [Enter] key is pressed or data for Data8192 is input, the db33 transmits the input data with the size and BCC.

**■ Notes**

- The ice command is used in-house for the development of the ICE33 by Seiko Epson, and is not provided for use by general users.
- This command does not support a parallel data transfer.
- This command cannot be suspended by the [Key break] button. The time out period is set to 150 seconds.

## 16.10 Parameter File

Before the db33 can be started up, you must have a parameter file that contains a description of memory map and PRC board (a peripheral circuit board provided for each model to be installed in the ICE33) information. The db33 and ICE33 use the contents written in this parameter file to configure a memory map, and they handle errors such as address specifications outside the effective memory area or map breaks.

The basic parameter file can be created using the [Par gen] button of the wb33. This is a text format file, so you can customize it before use by adding a specification for an external memory area, etc. to suit your application needs.

The parameter file name created by the wb33 comes in the following format:

```
<chip name>_<parameter file version>.par
```

Example: 33104\_01.par

The following shows a sample parameter file.

### Sample parameter file

```
CHIP      33104      ; chip name (33XXX)          ... (1)
IROM      1000       ; internal ROM is 80000 to 80FFF          ... (2)
FOPT      0000       ; f option size                          ... (3)
PRC VER   00 ff      ; allow any PRC board                    ... (4)
PRC STATUS ***** ; allow any PRC board status            ... (5)
MPU       ; 0xC00000 external boot address ... (6)
VER       1          ; this file version                      ... (7)

; Emulation memory allocation (max 8 areas, 1MB/area, 1MB boundary) ... (8)

EMRAM     400000 4ffff ; emulation RAM 1MB
EMROM     c800000 cffff ; emulation ROM 1MB

; Map allocation (max 31 areas, 256bytes boundary) ... (9)

RAM       0        7FF      ; internal RAM area 2KB
IO        40000    4ffff     ; internal IO area 64KB
RAM       200000    2ffff     ; external FLASH 1MB (on target board)
ERAM      400000    407fff    ; emulation RAM area 32KB (in emulation memory)
EROM      480000    4bffff    ; emulation ROM area 256KB (in emulation memory)
EIO       4c0000    4cffff    ; emulation IO area 64KB (in emulation memory)
RAM       600000    6ffff    B ; external SRAM 1MB (on target board)
EROM      c00000    cffff     ; emulation ROM area 1MB (in emulation memory)

; Stack area except internal RAM area (max 8 areas, 256bytes boundary) ... (10)

STACK     0        3ff      ; internal stack area 1KB
STACK     600000    6ffff     ; external stack area 1MB

END                                              ... (11)
```

\* A description from ";" to the end of the line is regarded as a comment.

## Explanation of contents

### (1) Chip name

Write a chip name.

CHIP <chip name>

### (2) Internal ROM size

Write the internal ROM size.

IROM <size>

Only a hexadecimal number can be used to write the size. (The "0x" is unnecessary.)

The specified sized area that begins from address 0x80000 is mapped as the internal ROM area. If 0 is specified, size = 0 (internal ROM is not used).

### (3) Option size

Write the size (up to 16KB) of the function options set for each model.

FOPT <size>

Only a hexadecimal number can be used to write the size. (The "0x" is unnecessary.)

### (4) PRC board version

Write a version range of the PRC board matched to the model being developed.

PRC VER <version (1)> <version (2)>

Write a small version number for version (1) and a larger one for version (2) using an 8-bit hexadecimal number. If the model corresponds to only one version of PRC board, write the same value for both.

When the db33 is started up in the ICE mode, it checks the PRC board version, and if the mounted PRC board does not fall within the range of version (1) to version (2), it issues a warning. For PRC board versions matched to each model, refer to the manual of the PRC board. When version (1) is set to 00 and version (2) is set to ff, any version of the PRC board is permitted to use for debugging.

### (5) PRC status

Specify the PRC board's status bits to be checked at startup in the ICE mode.

PRC STATUS <bit 15><bit 14>...<bit 0>

Specify 1, 0, or \* for 16 status bits. If you specify 1 or 0 and the PRC board's corresponding status bit is found to be different from that found in a startup check, the db33 issues a warning. If you specify an asterisk (\*) for a bit, the bit is masked and is not checked. For details on how to set the status bits, refer to the manual of the PRC board. When all the bits are specified with \*, the PRC board is permitted to use for debugging regardless of the status bits.

### (6) MCU/MPU mode

Specify whether you want the CPU to be started up in the MCU mode (booted from 0x80000 of internal ROM) or in the MPU mode (booted from 0xc00000 of external ROM).

- To specify the MCU mode: MCU
- To specify the MPU mode: MPU

This specification is valid in the simulator mode.

### (7) Parameter file version

Write the version of the parameter file.

VER <version>

Use a hexadecimal number 0 to ff for this specification. This is provided for version management by the user.

**(8) Emulation memory allocation**

The ICE33 allows you to use up to eight areas of emulation memory, each area with a size of 1M bytes. External memory areas can be allocated to this emulation memory when debugging them. When using emulation memory, specify whether you want each area to be used as RAM or as ROM.

- To use an area as RAM  
EMRAM <start address> <end address> (Multiple entries accepted; or can be omitted)
- To use an area as ROM  
EMROM <start address> <end address> (Multiple entries accepted; or can be omitted)

Specify each area in 1M-byte units, ranging from start address X00000 to end address Xffff (X = 1 to ff). Areas specified for EMROM are read-only, and no data can be written to the area by a program. Areas specified for EMRAM can be accessed for read and write by a program.

This specification is valid in the ICE mode.

If you do not use emulation memory (i.e., internal memory-only system or evaluated using memory mounted on a target board), omit this specification.

**(9) Setting the memory map**

Specify the memory area to be used. The information set here is used for a map break.

**Mapping of the emulation memory (for ICE33)**

Set the areas used in the emulation memory (1M bytes each) that have been declared in (8) using the formats shown below:

- To set an area used as RAM  
ERAM <start address> <end address> (Multiple entries accepted; or can be omitted)
- To set an area used as ROM  
EROM <start address> <end address> (Multiple entries accepted; or can be omitted)
- To set an I/O area  
EIO <start address> <end address> (Multiple entries accepted; or can be omitted)

The areas specified for ERAM can be accessed for read and write, and are initialized with 0xaa.

The areas specified for EROM are write-only, and are initialized with 0xff. When a write to this area is attempted, a break occurs.

The areas specified for EIO can be accessed for read and write, and are initialized with 0x00.

The address ranges are limited to the emulation memory areas set in (8). Specify a start address that resides on a 256-byte boundary. Specify an end address so that the area size is an integer multiple of 256 bytes.

No error is assumed even when you specify a memory map that does not match the memory attribute (ROM or RAM) specified in (8).

**Mapping of other types of memory**

Set a I/O map of the internal RAM, internal I/O, and the memory or I/O mounted on the target board using the formats shown below:

- To set RAM area (read/writable area)  
RAM <start address> <end address> (Multiple entries accepted; or can be omitted)
- To set ROM area (write-only area)  
ROM <start address> <end address> (Multiple entries accepted; or can be omitted)
- To set I/O area (read/writable area)  
IO <start address> <end address> (Multiple entries accepted; or can be omitted)

The internal ROM does not need to be mapped for ROM here because it is mapped by IROM setting.

Specify a start address that resides on a 256-byte boundary. Specify an end address so that the area size is an integer multiple of 256 bytes.

If the addresses set here overlap the areas used in emulation memory specified by EMRAM and/or EMROM, the RAM, ROM, or IO settings made here have priority. Namely, if an access to the overlapping area is made, it is the target board that is accessed and the emulation memory is not used. Thus the 1M-byte emulation memory can be further divided into areas that use the emulation memory and other areas that do not use the emulation memory.

Up to 31 maps can be set, for all RAM, ROM, IO, ERAM, EROM, and EIO included.

#### **Specification of big-endian areas**

By default, the mapped areas are set as little-endian areas. To change the area format to big-endian, describe letter "B" after the <end address> (select [Big] when creating in the wb33). However, the E0C33 chip to be developed must be a model that supports big-endian format. Furthermore, the internal memory (ROM, RAM and I/O) cannot be set to big-endian.

In addition to specify this parameter file at invocation of the db33, the endian control register in the E0C33 chip must be set correctly (refer to the "Technical Manual").

In simulator mode, the endian format is determined by the parameter file only.

This setting affects memory operation and file loading in half word or word units.

#### **(10) Setting stack area**

Specify an area you want to be used as the stack.

**STACK <start address> <end address>** (Multiple entries accepted; or can be omitted)

Up to 8 stack areas can be set.

Specify a start address that resides on a 256-byte boundary. Specify an end address so that the area size is an integer multiple of 256 bytes.

This setting is valid in ICE mode, so that when a stack operation is performed on a non-specified area, a break occurs. However, a stack operation performed on the internal RAM that starts from address 0 is excluded from break generation and, hence, does not cause a break. Therefore, STACK settings for areas in the internal RAM can be omitted.

This setting does not affect the SP operation by a program.

#### **(11) End mark**

Always be sure to write END at the end of a parameter file.

### **Precautions on creating a parameter file**

- Write each setup item in order of numbers (1) to (11).
- Items (1) to (6) and (11) cannot be omitted.
- Write item names (e.g., CHIP, IROM) with uppercase letters.
- Write each item from the beginning of the line and insert at least one space or tab between parameters. Parameters required for each item cannot be omitted.
- Make sure each memory map is set in units of 256 bytes, and that emulation memory is set in units of 1M bytes. A warning is generated if nonconforming boundary addresses are specified.



### Parameter file created by wb33

When using a parameter file created by using the [Par gen] button of the wb33, pay attention especially to the following when you customize it:

- **Option size (FOPT)**

The option size is set to 0. If you are developing a model which has function options, be sure to set the size.

- **PRC board version (PRC VER)**

The PRC version is set to "00 ff" so that any desired PRC board can be used. If you are using multiple PRC boards, correct this setting to limit the versions.

- **Internal I/O area (RAM, IO)**

The internal I/O area set by IO is 64K bytes from 0x40000 to 0x4ffff. Correct this setting to suit the I/O area of the model.

- **Emulation memory allocation, and external memory and I/O mapping**

If you chose an external memory area in the [Parameter file generator] window, these items are set as follows:

**When the [Emu] button is selected:**

**[RAM] button:** The emulation memory is configured using EMRAM, and ERAM is set so that the entire emulation memory area is used as RAM.

**[ROM] button:** The emulation memory is configured using EMROM, and EROM is set so that the entire emulation memory area is used as ROM.

**[IO] button:** The emulation memory is configured using EMRAM, and EIO is set so that the entire emulation memory area is used as I/O.

**When the [Emu] button is not selected:**

**[RAM] button:** RAM is set so that the entire area is accessed as a target RAM.

**[ROM] button:** ROM is set so that the entire area is accessed as a target ROM.

**[IO] button:** IO is set so that the entire area is accessed as a target I/O (same as RAM).

Correct or add memory maps as required for the system you are developing.

- **Stack area (STACK)**

Only when you choose the [RAM] button to specify the external memory, a stack area of 512K bytes beginning with a specified start address is set. Correct this setting whenever necessary.

## 16.11 Status/Error/Warning Messages

This section describes the messages that are displayed in the [Command] window by the debugger.

### 16.11.1 Status Messages

When the target program breaks, the db33 displays one of the following messages to indicate the cause of the break immediately before it stands by for a command input.

Table 16.11.1.1 Status messages

Message	Content
Break by software PC break.	Break caused by software PC breakpoint
Break by hardware PC break.	Break caused by hardware PC breakpoint
Break by hardware PC break 2.	Break caused by hardware PC breakpoint
Break by temporary break.	Break caused by temporary breakpoint
Break by data break.	Break caused by data break condition
Break by read memory.	Data break caused by reading memory
Break by write memory	Data break caused by writing to memory
Break by sequential break.	Break caused by sequential break condition
Break by key break, xxxx.	Break at address xxxx caused by [Key break] button
Break by accessing no map area	Break caused by accessing no map area
Break by writing ROM area	Break caused by writing to ROM area
Break by out of SP area	Break caused by accessing outside stack area
Break by external break	Break caused by signal input to ICE33/ICD33 BRKIN pin
Break by illegal instruction	Break caused by executing illegal instruction in simulator mode

### 16.11.2 Error Messages

Table 16.11.2.1 Error messages

(alphabetical order)

Error message	Content
address1 > address2	The beginning address is larger than end address.
Address is in no map area.	The specified address (symbol) is out of the mapped area.
Address is not 2 byte boundary.	The program code address is not a 2-byte boundary address.
Address range (0-0xFFFFFFF).	The address is out of the range.
Already exist input address.	The address has been set to a break point.
Aymbol not in scope.	The symbol cannot be found in the scope.
Break number (1-16).	The software PC break point number is out of the range.
Cannot add symbol any more.	99 symbols have been registered.
Cannot allocate memory.	Memory cannot be allocated.
Cannot close file.	The file cannot be closed.
Cannot get file status.	The file information is incorrect.
Cannot get input, please check the system.	An error has occurred during input process.
Cannot get memory.	Memory allocation has failed.
Cannot load data, file open failure.	The srf33 file load has failed; the file cannot be opened.
Cannot load data, file read failure.	The srf33 file load has failed; the file cannot be read.
Cannot load data, memory allocation failure.	The srf33 file load has failed; memory cannot be allocated.
Cannot load data, please check SRF33 file.	The srf33 file load has failed; some file other than srf33 executable format is specified.
Cannot load debug information, debug information is wrong.	The debug information load has failed; the debug information is illegal. (Program/ data is loaded successfully.)
Cannot load debug information, file open failure.	The debug information load has failed; the source file cannot be opened. (Program/ data is loaded successfully.)
Cannot load debug information, file read failure.	The debug information load has failed; the source file cannot be read. (Program/ data is loaded successfully.)
Cannot load debug information, memory allocation failure.	The debug information load has failed; memory cannot be allocated. (Program/ data is loaded successfully.)
Cannot load debug information, please check SRF33 file.	The debug information load has failed; the srf33 format is illegal. (Program/ data is loaded successfully.)
Cannot load debug information, too many lines.	The debug information load has failed; too many source lines are included. (Program/ data is loaded successfully.)
Cannot open com port.	The COM port or baud rate cannot be set.
Cannot open file any more.	The number of files exceeds the limit.

Error message	Content
Cannot open file.	The file cannot be opened.
Cannot open parallel port.	The parallel port cannot be opened.
Cannot open parameter file.	The parameter file cannot be opened.
Cannot open stdio file.	The stdio file cannot be opened.
Cannot read chip name.	Chip name information (.par file) cannot be read.
Cannot read emulation memory map.	Emulation memory map information (.par file) cannot be read.
Cannot read FO area size.	FOPT size information (.par file) cannot be read.
Cannot read IROM size.	Internal ROM size information (.par file) cannot be read.
Cannot read MCU/MPU information.	MCU/MPU mode information (.par file) cannot be read.
Cannot read parameter file version.	Parameter file version information (.par file) cannot be read.
Cannot read PRC board status.	PRC board status information (.par file) cannot be read.
Cannot read PRC board version.	PRC board version information (.par file) cannot be read.
Cannot read stdin file.	The stdin file cannot be read.
Cannot set address any more.	16 software PC break points have been set.
Cannot set hardware PC break.	The hardware PC break point cannot be set.
Cannot set software PC break.	The software PC break point cannot be set.
Cannot set temporary break.	Temporary break point cannot be set.
Cannot write flash memory.	Data cannot be written to the flash memory.
Cannot write log file.	Log data cannot be written to the file.
Cannot write stdout file.	The output data cannot be written to the stdout file.
Cannot write trace file.	Trace data cannot be written to the file.
Chip name should be 5 characters.	The chip name length is not 5 characters.
Chip name should be start with "33".	The chip name must begin with 33.
Command is not supported at present mode.	A command not supported for the current mode (ICE or simulator) is executed.
Communication data size error.	The communication data size is incorrect.
Communication error.	Overrun, framing, or BCC error has occurred during transmission from/to the ICE33.
CPU down.	The PRC board operates erratically.
CPU is not running.	The ICE33 CPU has stopped operating.
CPU is running.	The ICE33 CPU is executing.
Current mode is not source mode.	String search is only available in the source display mode.
Data alignment error.	Alignment in the srf file is incorrect.
Data incomplete.	The file structure is illegal.
Data range (0-0xFF).	The input data is out of the range.
Data range (0-0xFFFF).	The input data is out of the range.
Data range (0-0xFFFFFFFF).	The input data is out of the range.
Debug data failure.	The debugging data is illegal.
Diagnostic test failure.	The ICE33 self-diagnosis resulted in error.
Duplicate input address.	Same break address is set twice.
Duplicate input break number.	Same break point number is set twice.
Empty file.	The file does not contain data.
Erase routine is not set.	A flash memory erase routine has not been defined.
File end during guidance input.	The command file has ended in the middle of the parameters of the guidance format.
File not found.	The file cannot be found.
Flash memory error.	Error in writing or erasing flash memory.
Flash memory is not mapped.	The ICE flash memory is not mapped.
Flash ROM is protected.	Flash memory is protected against access.
Fo address range (0-0x3FFF).	The option dump address is out of the range.
Format error.	The format is illegal.
Function not found.	The function cannot be found.
ICE is busy.	The ICE33 is busy processing a job.
ICE is free run mode.	The ICE33 is operating in free-run mode.
ICE is maintenance mode.	The ICE33 is placed in maintenance mode.
ICE is not mapped.	The ICE built-in memory is not mapped.
ICE system error.	ICE system error has occurred.
Interrupt level (0-15).	The interrupt level is out of the range.
Interrupt type (0-215).	The interrupt type is out of range.
Invalid break address.	The break address has not been set.
Invalid break number.	The break point number has not been set.
Invalid command or parameter.	The specified command or parameter is invalid.
Invalid emulation memory map.	The emulation memory map information (.par file) is invalid.

Error message	Content
Invalid file name.	The file name is invalid.
Invalid group number.	The command group number is invalid.
Invalid map command or invalid sequence.	The map file contains an illegal character or incorrect sequence.
Invalid memory map.	The memory map information (.par file) is invalid.
Invalid parameter.	The parameter is incorrect.
Invalid stack map.	The stack map information (.par file) is invalid.
Invalid value.	The input value is illegal.
IROM size is too long.	The IROM size is too large.
No "END" in parameter file.	There is no end mark (END) in the parameter file.
No map area.	The input address is out of the mapped area.
No symbol at the number.	Symbol is not registered in the specified number.
Not ASCII character.	The string contains some other ASCII character.
Not defined ID.	ICE33's response ID is invalid.
Not found input strings.	The string cannot be found.
Number of emulation memory is wrong.	The number of emulation memory block in the parameter file is invalid.
Number of parameter.	The number of parameters in the command is invalid.
On tracing.	The ICE33 is tracing execution data.
Over max include file number.	The number of include files exceeds the limit.
Parallel interface time out.	The file cannot be loaded through the parallel interface within the predefined time.
Parallel port is busy.	The parallel port is in busy status.
Pointer pointed no map area.	The pointer variable has pointed out of the mapped area.
Post line range (0-256).	The number of post-display line in the trace search is out of the range.
Pre line range (0-256).	The number of pre- display line in the trace search is out of the range.
Register variable cannot be changed to address.	Addresses cannot be specified with a register variables.
Reset timeout.	The ICE33 CPU cannot be reset.
Sequential break format error.	The sequential break condition is invalid.
Shared RAM is busy.	An ICE33 internal error has occurred.
Source window not opened.	The [Source] window is closed.
Start block > End block.	The end block number is greater than the start block number.
Start cycle number > End cycle number.	The end cycle number is greater than the start cycle number.
Stdout data size.	The output data size in the output buffer is illegal.
Step range (1-65535).	The step count is out of the range.
Symbol is too long.	The symbol name is too long.
Symbol not found.	The symbol cannot be found.
Target down.	The PRC board does not operate correctly or remains reset.
Target instruction cannot be extended.	The instruction cannot be extended by ext.
Time out.	Communication time-out. *1
Too many include.	Number of included files exceeds the limit.
Too many source file.	The source file is too large.
Trace range (0-32767).	The trace cycle number is invalid.
Verify error.	Verify error when writing to flash memory.
Wrong data.	Data in the file is incorrect.
Wrong header.	The file header is incorrect.

\*1: A time-out error occurs in the following processing if no response is returned from the ICE33 within a predetermined time:

- Initial connection 2.5 seconds
- Escape Break 1 second
- Map initialization 150 seconds
- f or mv execution 150 seconds
- lfl or sfl execution 150 seconds
- g, s, or n execution No time-out is set.
- Others 6 seconds

### 16.11.3 Warning Messages

Table 16.11.3.1 Warning messages

(alphabetical order)

Warning message	Content
Debugger mode does not match with a target.	The debugger mode speified by the option (-icd, -ice, -mon) does not match with the connected target system.
Emulation memory address is not 1M byte boundary.	The emulation memory map address in the parameter file is not a 1MB boundary address.
FO size (0-0x4000), map as 0x4000.	The FO size is incorrect, so it is mapped as 0x4000.
FO size should be an even number, map as 0XXXXXXXX.	The FO size must be an even number, so it is mapped as 0XXXXXXXX.
Invalid line, move to next valid line.	The source line has no address. The next effective address is used.
IROM size (0-0x80000), map as 0x80000.	The IROM size is incorrect, so it is mapped as 0x80000.
Line number of source file is invalid.	The line number is not included in the source file.
Memory map is not 256 byte boundary.	The memory map (.par file) must be specified in 256-byte units.
No debug information.	The srf33 file does not have the debug information.
No source, display on mix mode.	There is no source information. The program is displayed in mix mode.
Number of source line exceeded 65535.	The line number is out of the range.
PRC status does not match.	The PRC board status is different from the parameter file.
PRC version does not match.	The PRC board version is different from the parameter file.
Round down to multiple of 16.	The input address is adjusted to a 16-byte boundary.
Round down to multiple of 2.	The input address is adjusted to a 2-byte boundary.
Round down to multiple of 4.	The input address is adjusted to a 4-byte boundary.
Stack map is not 256 byte boundary.	The stack map (.par file) must be specified in 256-byte units.

## Chapter 17 Other Tools

This chapter explains the other tools that are included in the E0C33 Family C Compiler package.

### 17.1 Make

The E0C33 Family C Compiler Package contains a make tool (hereafter referred to as the "make") that efficiently processes compilation to linkage.

Based on the dependence relationship between the sources written in a make file and the files output by each tool, the make uses the necessary tools to update the files to the latest version. For example, if only one source file is corrected, the make executes compilation or operation from preprocess to assemble only for that file. Other modules only have object files read in during linkage, and are not processed sufficiently to include assembly.

The make in this package only supports the dependency lists, suffix definitions, and macro definitions necessary to perform the above processing.

It provides the subset functions of make in UNIX.

#### 17.1.1 Starting Method

##### Startup format

**make ^ [<option>] ^ [<target name>]**

^ denotes a space.

[ ] indicates the possibility to omit.

Example: c:\cc33\make -f test.mak opt

##### Operations on work bench

Select options and a make file (.mak), then click the [MAKE] button.

##### Options

The make comes provided with the following three types of startup options:

###### **-f <file name>**

Function: Specifies a make file.

Specification on wb33: Always specify (choose a file name from the list box).

Explanation: The make reads in a make file specified by <file name> (extension included), and processes its contents.

Default: Unless the -f option is specified, a file named "makefile" is input as the make file.

###### **-h**

Function: Outputs usage.

Specification on wb33: Check [usage].

Explanation: Only a message about how to use the standard output device (stdout) is output before terminating.

###### **-n**

Function: Displays commands.

Specification on wb33: Check [no exe cmd].

Explanation: Only the command to be processed by make is output to the standard output device (stdout) and no operation actually is performed on it. This is effective for verifying the dependence relationship of files.

## Target name

Specify the target name for the command to be executed. If this specification is omitted, the first target that appears in the make file is executed.

A make file created by the Make file editor of the wb33 has commands and target names (opt, clean) to implement two functions recorded in it, in addition to the dependency lists used to update files.

**opt:** Target name to execute commands for 2-pass make

**clean:** Target name for commands to erase all but source file

To execute these targets from the wb33, perform the following operation after selecting the make file:

To executes opt, select [2 pass] on the [Other option] window and then click [MAKE].

To executes clean, click [MAKE clean].

## 17.1.2 Messages

The make delivers its messages through the Standard Output (stdout).

If the wb33 is started up by using the wb33's [MAKE] button, the message is output to "wb33.err". When execution is completed, a message is displayed in the output window (default).

### Execution message

During execution, the make outputs the command under execution; when completed, it outputs an end message.

Example:

```
C:\CC33\pp33 -g sample1.s
Pre Processor Completed
C:\CC33\ext33 -gp 0x0 sample1.ps
Extend Completed
C:\CC33\as33 -g sample1.ms
Assembly Completed
C:\CC33\lk33 -g -s -m -c sample.cm
Link Completed
Make Completed
```

- \* When executed in the wb33, the make uses ccap to save messages to a file, "wb33.err", while at the same time counting the error/warnings encountered in each tool and the compiler messages. The count values are displayed after the make's end message.

```
Make Completed
```

```
0 error(s), 0 warning(s), 0 compiler message(s)
```

### Usage output

If no file name was specified or an option was not specified correctly, the make ends after delivering the following message concerning the usage:

```
Make for 33/63 Ver x.x
Copyright (C) SEIKO EPSON CORP. 199x
Usage:
    make [options] [target]
Options:
    -f <file name> : makefile name
    -h : output usage
    -n : no command execution
Example:
    make -f test.mak
    make -f test.mak CLEAN
```

### When error/warning occurs

If an error or a warning is produced, an error/warning message will appear before the end message shows up.

```
Example: Warning : sample.srf is up-to-date
```

```
Make Completed
```

For details on errors and warnings, refer to Section 17.1.6 "Error/Warning Messages".

### 17.1.3 Make File

The make file is a text file that contains a description of the dependence relationship of the files and the commands to be executed. A basic make file can be created using the Make file editor on the wb33, so use it after entering additions or corrections as necessary.

Shown below is an example of a make file.

#### make file example (without suffix definition)

```
# make file made by wb33                                     ...Comment

# macro definitions for tools & dir

TOOL_DIR = C:\CC33                                         ...Macro definition
GCC33 = $(TOOL_DIR)\gcc33
PP33 = $(TOOL_DIR)\pp33
EXT33 = $(TOOL_DIR)\ext33
AS33 = $(TOOL_DIR)\as33
LK33 = $(TOOL_DIR)\lk33
MAKE = $(TOOL_DIR)\make
SRC_DIR =

# macro definitions for tool flags

GCC33_FLAG = -B$(TOOL_DIR)\ -S -g -O
PP33_FLAG = -g
EXT33_FLAG = -gp 0x0
AS33_FLAG = -g
LK33_FLAG = -g -s -m -c
EXT33_CMX_FLAG = -lk test -c

# dependency list

test.srf : test.cm boot.o main.o                           ...Dependency list
              $(LK33) $(LK33_FLAG) test.cm                ...Command line

boot.ms : $(SRC_DIR)boot.s
           $(PP33) $(PP33_FLAG) $(SRC_DIR)boot.s
           $(EXT33) $(EXT33_FLAG) boot.ps
boot.o : boot.ms
         $(AS33) $(AS33_FLAG) boot.ms

main.ms : $(SRC_DIR)main.c
           $(GCC33) $(GCC33_FLAG) $(SRC_DIR)main.c
           $(EXT33) $(EXT33_FLAG) main.ps
main.o : main.ms
         $(AS33) $(AS33_FLAG) main.ms

# optimaization by 2 pass make

opt:
    $(MAKE) -f test.mak
    $(TOOL_DIR)\cwait 2
    $(EXT33) $(EXT33_CMX_FLAG) test.cmx
    $(MAKE) -f test.mak

# clean delete files except source

clean:
    del *.srf
    del *.o
    del *.ms
    del *.ps
```



**Dependency list**

The make is executed according to a dependency list that is written in the following formats:

```
Format 1: <target file name>:<dependent file name 1>[<dependent file name2>...]
      [   TAB      <command 1>
        TAB      <command 2>
          :
        ]
```

```
Format 2: <target name>
      TAB      <command 1>
      [   TAB      <command 2>
        :
      ]
```

^ denotes a space.

[ ] indicates that entries in brackets can be omitted.

The command lines must begin with a TAB (space is not allowed).

**Format 1**

In Format 1, the dependent files necessary to obtain a target file is specified, and in cases when no target file has been created or the dependent file has not been updated, the command that follows is executed.

```
Example: test.srf : test.cm boot.o main.o
          $(LK33) $(LK33_FLAG) test.cm
```

In this example, the target file "test.srf" depends on "test.cm", "boot.o", and "main.o".

If the target file "test.srf" is nonexistent or there is any dependent file that is newer than the target file, the command "\$\$(LK33)\$(LK33\_FLAG)test.cm" (link by lk33) is executed. The \$(<name>) written here is replaced with a macro defined by <name>.

**If the dependent file is some other target**

If the dependent file is specified as the target of some other dependency list, the other dependency list is evaluated first. For example, since the dependent file "boot.o" is associated with the next two dependency lists, the make is performed first in those lists.

```
boot.ms : $(SRC_DIR)boot.s
          $(PP33) $(PP33_FLAG) $(SRC_DIR)boot.s
          $(EXT33) $(EXT33_FLAG) boot.ps
          ... If the ext33's output file "boot.ms" is nonexistent or the source file "boot.s" is
             newer than "boot.ms", the pp33 and ext33 are executed.

boot.o : boot.ms
          $(AS33) $(AS33_FLAG) boot.ms
          ... If the as33's output file "boot.o" is nonexistent or "boot.ms" is newer than
             "boot.o", the as33 is executed.
```

**If the dependent file is nonexistent**

If the described dependent file cannot be found and there is no dependent file specified for other targets, an error is assumed.

**If the command line is nonexistent**

Nothing is executed. However, if a target file and a suffix list (described later) that has the extension of the first dependent file are written, the command associated with it is executed.

**Format 2**

If no dependent file is written, <target name> is used only as a label. By specifying a <target name> with the make's startup command, it is possible to execute the written command.

Example: Commands executed by make -f text.mak clean

```
clean:
    del *.srf
    del *.o
    del *.ms
    del *.ps
```

If no <target name> is specified in the startup command, the first dependency list written in the file is used to execute the make.

**Command line**

The cc33 tool names and DOS prompt commands can be written in a command line. However, the colon (:) to indicate a drive cannot be written directly in the command line. When specifying a path in the command line, prepare a macro definition of the path before using it.

The following two symbols can be inserted at the beginning of a command line:

@ Turns off the echo display of the command line in which this symbol is inserted.

Example: @copy test.s test.sbk

Normally, the command line executed is output to stdout. Command lines that begin with @ are not output.

- Even if the command has resulted in an error (terminated for some reason other than exit(0)), the error is ignored and the command that follows is executed.

Example: -make -f test mak -n

Normally, the make is terminated with a command in error.

A predefined macro can be referenced in the command line. Furthermore, the following two macro symbols can be used.

\$\* This is replaced with the target file name (not including the extension) currently being processed.

Example: test.dis: test.srf

```
$(TOOL_DIR)\dis33 $*.dis
```

\$@ This is replaced with the target file name (including the extension) currently being processed.

Example: 33xxxx.sa: test.sa\_80000\_80fff

```
copy test.sa_80000_80fff $@
```

These macro symbols cannot be used anywhere other than in a command line.

**Macro definition**

You can define a macro in a make file and reference a defined macro from a command line. The following shows the formats in which a macro can be defined and referenced.

Definition: <macro name> = <macro body>

Reference: \$(<macro name>)

Examples:

```
TOOL_DIR = C:\CC33                ...Macro definition
GCC33 = $(TOOL_DIR)\gcc33         ...Macro definition and
GCC33_FLAG = -B$(TOOL_DIR)\-S -g -O    macro reference in macro definition
$(GCC33) $(GCC33_FLAG) $(SRC_DIR)main.c ...Macro reference in a command line
boot.ms : $(SRC_DIR)boot.s         ...Macro reference of a dependence file name
```

The colon (:) used to specify a drive can only be used in macro definition, except when you use it immediately after a target name or in a comment. Therefore, when specifying a path in a command line or dependent file, use a macro that is defined in advance, as shown by the above examples.

Macros cannot be referenced in the following places:

- Lines preceding macro definition
- Target file names
- Lines where a suffix is defined or the first line of a suffix list

• **Precautions about writing "\" at the end of a macro**

If \<CR> is written at the end of a line, it is assumed that this line continues to the next line. Therefore, when defining a path that ends with \ as a macro, write the following:

Example: When defining "c:\e0c33\"

```
SRC_DIR = c:\e0c33\           ...Write two \.
                               ...Insert one blank line (CR only).

<next statement>
```

## Comments

A statement from # to the end of the line is regarded as a comment. Characters other than the ASCII code can be written in a comment.

Example: # make file made by wb33

However, any comment cannot be written in a command line because it will be assumed to be part of the execution command.

## Suffix definition and suffix list

If you write a suffix definition and a suffix list, you can omit commands in a dependency list. When creating a make file in the wb33, check the [suffix type] option. If this option is specified, a suffix definition and a suffix list are included in the make file thus created.

### Dependency list with no suffix defined

```
# dependency list

test.srf : test.cm boot.o main.o
          $(LK33) $(LK33_FLAG) test.cm

boot.ms  : $(SRC_DIR)boot.s
          $(PP33) $(PP33_FLAG) $(SRC_DIR)boot.s
          $(EXT33) $(EXT33_FLAG) boot.ps

boot.o   : boot.ms
          $(AS33) $(AS33_FLAG) boot.ms

main.ms  : $(SRC_DIR)main.c
          $(GCC33) $(GCC33_FLAG) $(SRC_DIR)main.c
          $(EXT33) $(EXT33_FLAG) main.ps

main.o   : main.ms
          $(AS33) $(AS33_FLAG) main.ms
```

### Example with suffix defined

```
# suffix & rule definitions

.SUFFIXES : .c .s .ps .ms .o .srf           ...Suffix definition

.c.ms :                                     ...Suffix list
      $(GCC33) $(GCC33_FLAG) $(SRC_DIR)$*.c
      $(EXT33) $(EXT33_FLAG) $*.ps

.s.ms :
      $(PP33) $(PP33_FLAG) $(SRC_DIR)$*.s
      $(EXT33) $(EXT33_FLAG) $*.ps

.ms.o :
      $(AS33) $(AS33_FLAG) $*.ms

# dependency list

test.srf : test.cm boot.o main.o
          $(LK33) $(LK33_FLAG) test.cm

boot.ms  : $(SRC_DIR)boot.s                 ...Dependency list
boot.o   : boot.ms

main.ms  : $(SRC_DIR)main.c
main.o   : main.ms
```

**Suffix definition**

Before a suffix list can be used, you must first define the file extensions used in the suffix list. The following shows the format of a suffix definition:

Format: `.SUFFIXES : .xxx .yyy .zzz .....`

Example: `.SUFFIXES : .c .s .ps .ms .o .srf`

**Suffix list**

The following shows the format of a suffix list:

Format: `<.extension of dependent file 1><.extension of target file>:  
TAB <command 1>  
[ TAB <command 2>  
: ]`

Example: `main.o : main.ms` ...Dependency list  
`.ms.o :` ...Suffix list  
`$(AS33) $(AS33_FLAG) $*.ms` ...\$\* is a macro symbol that is replaced with a target name (main).

The suffix list in this example corresponds to a dependency list that has a target file whose extension is ".o" and dependent file 1 (first dependent file written) whose extension is ".ms". Thus, when commands in this dependency list are omitted, commands in the suffix list are executed.

Since one suffix list corresponds to multiple dependency lists that have the same combination of extensions, it helps you simplify a description of dependency lists when there is a large number of files.

**Restriction on characters**

The table below lists the characters that can be used in each item of a make file. Do not use any other characters.

Table 17.1.3.1 Usable characters in make files

Item		Usable characters
Dependency list	Target name	a to z A to Z 0 to 9 _ - .
	Dependent file name	a to z A to Z 0 to 9 _ - . / \ \$( )
Macro	Macro name	a to z A to Z 0 to 9 _ -
	Macro body	a to z A to Z 0 to 9 _ - . / \ : \$( )
Suffix		a to z A to Z 0 to 9 _ - .
Command line		cc33 tools and DOS prompt commands (note) \$* @\$ \$( )
Comment		Any character that can be displayed

(note) The colon (:) to indicate a drive cannot be used in a command line. Use a path specification defined as a macro in advance.

- Write a file name (including path), a macro name, and a command line of not more than 100 characters respectively.
- Create a macro body of not more than 1,000 characters.

### 17.1.4 2-pass make

The make file created by the wb33 contains a description of the 2-pass make commands necessary to optimize code generation after reading in the symbol and map files linked by the ext33. These commands are included in a make file so that they can be executed when you execute the make after specifying target name "opt" (choosing [2 pass] in the wb33).

The following lists the commands included in a make file:

```
# optimization by 2 pass make

opt:
    $(MAKE) -f test.mak                ...(1)
    $(TOOL_DIR)\cwait 2                ...(2)
    $(EXT33) $(EXT33_CMX_FLAG) test.cmx ...(3)
    $(MAKE) -f test.mak                ...(4)
```

- (1) The files are processed through to linking.
- (2) A tool called "cwait" is used to provide a 2-second wait time. This wait time is provided to ensure that make in the second pass will be executed without failure.
- (3) The ext33 is executed by entering link map and symbol files and by specifying the optimize option (-lk).
- (4) The make is executed again to create an absolute object file.

For optimization using the link map and symbol files, refer to Section 10.7.3, "Optimization by Symbol Information".

### 17.1.5 clean

The make file created by the wb33 contains a description of the commands to delete intermediate and object files other than the source. These commands are included in a make file so that they can be executed when you execute the make after specifying the target name "clean" (click [MAKE clean] after selecting a make file in the wb33).

The following lists the commands included in a make file:

```
# clean delete files except source

clean:
    del *.srf
    del *.o
    del *.ms
    del *.ps
    del *.map
    del *.sym
```

All files in the current directory that have extensions ".srf", ".o", ".ms", ".ps", ".map", and ".sym" are deleted.

## 17.1.6 Error/Warning Messages

Error and warning messages are displayed/output through the Standard Output (stdout).

If the make is started up using the wb33's [MAKE] button, the message is output to "wb33.err". When execution is completed, a message is displayed in the output window (default).

If an error/warning occurs in the make itself, the make immediately stops processing after displaying a message. If an error occurs in the tool executed by a command that begins with "-" within the make file, the make continues processing. For error messages generated by tools, refer to the chapters where each tool is described.

The table below lists the error and warning messages generated by the make.

Table 17.1.6.1 Error/warning messages

Error/warning message	Content
Error: Cannot open XXXXXXXX	make file cannot be opened.
Error: Cannot open tmp file	Temporary file cannot be opened.
Error: Invalid syntax near line #	Syntactically erroneous. Use line number indicated by # to locate an error. Lines ending with <CR> are assumed to continue to the next line and not included in line counts.
Error: Invalid suffix .XXX	Suffix is not defined yet.
Error: Invalid macro name XXXX	Macro name is not defined yet.
Error: Abnormal termination in XXXX	Error occurred during the processing of module XXXX.
Error: Not enough memory	Memory is insufficient.
Error: Too long string	One character string exceeds 1,000 characters.
Error: No target found	Target cannot be found.
Error: Don't know how to make XXXX	Target file is nonexistent.
Error: Command exit with X	Command is terminated abnormally. (X = exit code)
Warning: XXXX is up-to-date	Last target has already been updated. Terminated without executing a command.

## 17.1.7 Precautions

The make included in the E0C33 Family C Compiler Package does not support any other functions (e.g., default settings of macro and suffix or macro symbols such as \$< and \$?). Only the functions described here are supported. Therefore, be careful if you are regularly using the make in UNIX.

If EXIT code = 0 is returned when executing a command line, the make suspends execution of the commands that follow. However, when a EXE file is executed under Windows95, the EXIT code always returns to 0 regardless of whether any error appears.

Although the make performs special processing on the cc33 tool to determine the status of the EXIT code, it cannot make such a determination in other EXE files, and therefore continues processing. Thus, you should be careful when using the make in Windows95. This problem does not occur in Windows NT 4.0.

The Make file editor of the wb33 can add/delete files to/from a make file. Since this function uses comments and character patterns in the make file, pay attention when editing the make file using an editor. If the necessary comments and character patterns are deleted, the Make file editor will not be able to edit the make file. Refer to "Precautions on editing the make file" in Section 5.2.6 for more information.

## 17.2 *cwait*

---

### 17.2.1 Functions

The `cwait` is used to create a wait time of several seconds. Therefore, this tool is used in the make file created by the `wb33` to provide a time allowance when executing a 2-pass make.

### 17.2.2 Method for Using `cwait`

#### Startup format

`cwait ^ [<number of seconds>]`

^ denotes a space.

[ ] indicates the possibility to omit.

<number of seconds>: Specify a wait time in seconds. This duration can be specified in the range of 0 to TBD seconds.

Example: `c:\cc33\cwait 2` ... Create a 2-second wait time.

#### Usage output

No message is displayed when executing `cwait`. However, the following message is displayed if it is started up without specifying a time in seconds.

```
Cwait ver x.x
Copyright (C) SEIKO EPSON CORP. 199x
Usage:
    cwait <wait_second>
Example:
    cwait 2
```

## 17.3 ccap

### 17.3.1 Functions

This tool produces a file from the messages output to the console (standard output or standard error) by other tools or commands.

When executing a tool using the wb33's execution button, the tool's messages are output to a file called "wb33.err" by the ccap, and when execution of the tool is completed, the contents output to wb33.err are displayed in the output window (or editor).

### 17.3.2 Method for Using ccap

#### Startup format

**ccap ^ [<option>] ^ <output file name> ^ "<execution command>"**

^ denotes a space.

[ ] indicates the possibility to omit.

<output file name>: Specify a file name to which you want the messages to be output.

<execution command>: Input the startup command of the tool to be executed.

#### Options

The ccap comes provided with the following four types of startup options:

##### -a

Function: Adds to an existing file.

Explanation: If this option is specified, the output contents are added at the end of an existing file. If no file exist, the ccap creates a new file.

Default: Unless this option is specified, the contents are overwritten to a specified file (if the file exists) or (if the file does not exist) the ccap creates a new file.

##### -o

Function: Outputs only a file.

Explanation: The messages of the execution command are output to only a file, and not output to the console.

Default: Unless this option is specified, the messages are output to both console and file.

##### -c

Function: Disables outputting execution command line.

Explanation: If this option is specified, the execution command line is output to neither the console nor a file.

Default: Unless this option is specified, the execution command line is output along with messages.

##### -e

Function: Error count

Explanation: If this option is specified, the ccap outputs a count of the error messages output by the execution command. The messages counted are those which begin with the following character strings:

Error Count of the error messages

Warning Count of the warning messages

.c: Count of the gcc33 messages

.h: Count of the gcc33 messages

Default: Error messages are not counted.

When entering an option, you need to place one or more spaces before and after the option.

Example: c:\cc33\ccap -a -o -e wb33.err "gcc33 -S test.c"



**Usage output**

If no file name or execution command was specified or an option was not specified correctly, the ccap ends after delivering the following message concerning the usage:

```
ccap
Console Capture Ver x.xx
Copyright (C) SEIKO EPSON CORP. 199x
Usage:
    ccap [options] <output-file> "command line"
Options:
    -a : append mode
    -o : disable console output
    -c : disable command echo
    -e : display error count
Example:
    ccap -a -o -c console.cap "gcc sample.c"
```

**Error messages**

The following shows the error messages generated by the ccap:

```
Error: Cannot execute           ...The specified "execution command" cannot be executed.
Error: Cannot open output file  ...The output file cannot be opened.
```

# Appendix srf33 File Structure

## A-1 srf33 Object File Structure

The structure of the srf33 format files created by the Assembler as33 and Linker lk33 is explained below. (srf33 is an abbreviation for "Seiko Epson Relocatable File Format" for the E0C33.)

Note: The structure of the srf33 file for the loader created with the -ld command of the lk33 is different from that of the standard srf33 file shown in this section. Refer to the readme.txt (English) or readmeja.txt (Japanese) located in the "utility¥ld33¥" directory of the srf33 file for the loader.

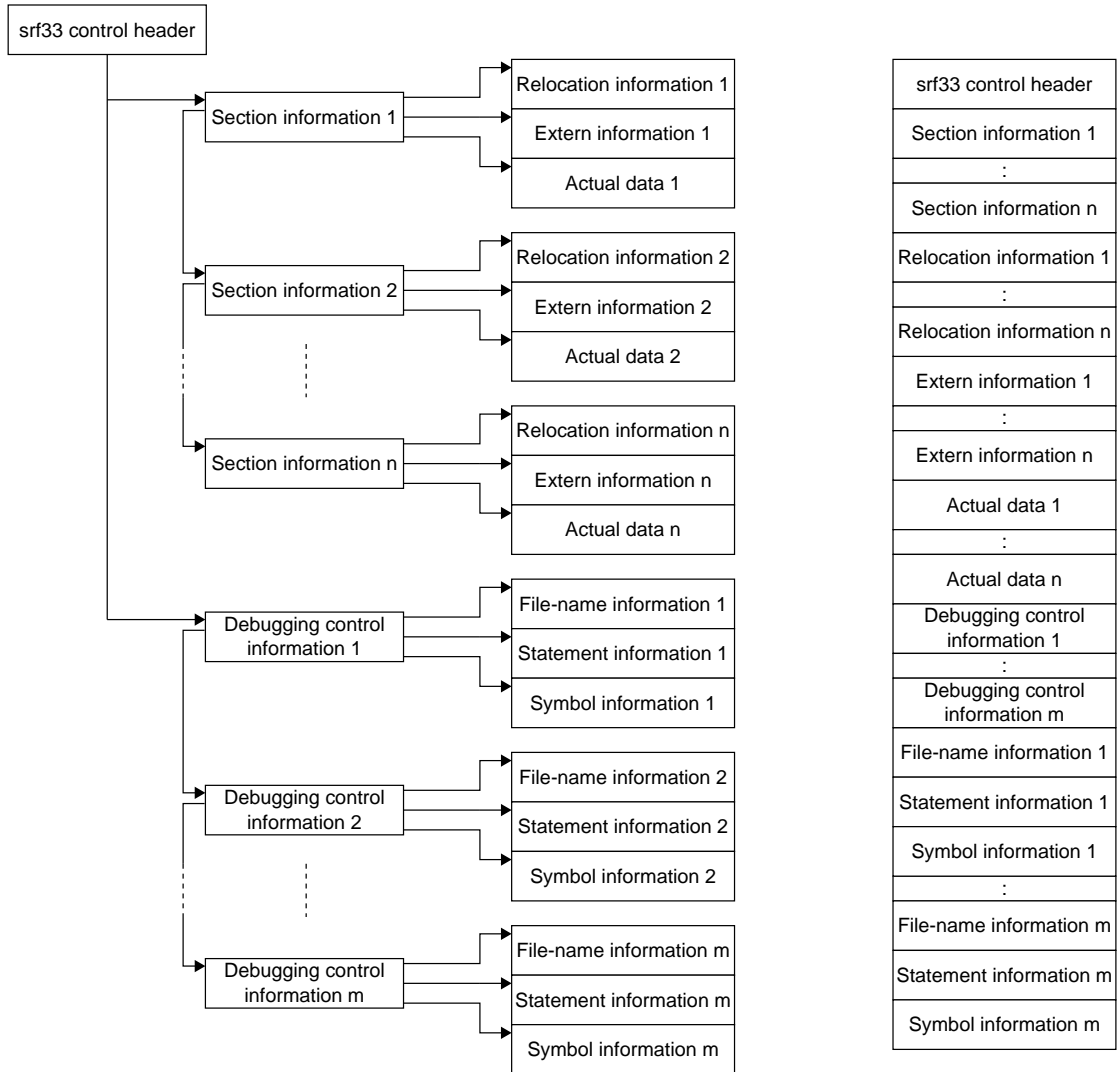


Fig. A.1 Structure of srf33 object file

### srf33 control header

The srf33 file always has one srf33 header at its top. The srf33 control header carries chains (in-file positional information) toward Section Information 1 and Debugging Control Information 1.

**Section information**

An object file created by the assembler has three pieces of section information (one each for CODE, DATA and BSS) in the case of relocatable modules, or has one to three section information (one or zero each for CODE, DATA and BSS) in the case of absolute modules. In the modules after the linking process, continuous relocatable sections are grouped together, but other sections (individually defined section and absolute sections) exist independently of each other. The section information after the linking process is grouped together in the CODE Section, DATA Section, and BSS Section in the order of the addresses in each sections. The section information in the CODE section, DATA section, and the BSS section, which do not have actual data, is also delivered.

Each piece of section information contains the attribute of that section (CODE, DATA, or BSS), mapping addresses, chains to relocation information/extern information/actual information, actual data size, and so on. It also has a chain to the following section information.

**Relocation information**

Necessary information for relocation of the linker. It is contained in the object file created by the assembler. The relocation information is not delivered in the object file after the linking process.

The relocation information contains relocation types for instruction codes in the section, positions in the section, index to the extern information to be referred to, and so on.

**Extern information**

Symbol information necessary for linking. It is contained in the object file created by the assembler. The extern information is not delivered in the object file after the linking process.

The extern information contains the names and types of symbols used in the section, the positions in the section, and so on.

**Actual data**

Actual data of each section (although the BSS section does not hold any data). In the case of the CODE section, two bytes are used for one code (instruction). In the DATA section, each piece of data takes up one, two or four bytes.

**Debugging control information**

The debugging control information is created in a quantity equal to the number of linked object modules, and contains the file name information of each module, statement information, symbol information size, and chains. It also carries a chain that goes to the following debugging control information. The debugging control information is arranged in order of the linked modules.

The debugging control information, as well as the file name information, the statement information, and the symbol information, are delivered when the processing is executed from the C Compiler gcc33/ Preprocessor pp33 through Linker lk33, with the -g option specified. The source display and the use of symbols take place according to this debugging control information. Even when there is no part following the debugging control information (or if such part is available but it cannot be read successfully) the debugging can be executed by disassembling display, as long as the portion up to the actual data can be read correctly.

**File-name information**

File-name information created by the .file pseudo-instruction. It contains information on the file names of each module, included-file names, and their respective directory structures. It is primarily referred to by the debugger when it displays source codes.

**Statement information**

Consists of line-number information and file-switching information created by the .loc and .endfile pseudo-instructions. It is mainly referred to by the debugger to establish correspondence between actual data and source codes.

**Symbol information**

Contains information on all the symbols defined in the module. It is referred to in the symbol display or in the address specification using symbols.

**<Reference: Contents of Information>**

- \* Chain: Denotes the connection to the continuing information by the number of bytes from the top of the file.  
If that number is 0 (zero), there is no continuing information.
- Index: Number to identify a section or symbol. First ID No. is 0 (zero).

**(1) srf33 Control Header**

Information		Byte	Contents
c_fatt	File control flag	2	The following OR values: 0x0001: Relocatable file 0x0002: Absolute file 0x0004: Execution format (Linker output file) 0x0008: Debugging information included 0x0010: Library file
c_pentry	Entry address	2	0x0000: Boot address
c_ver	srf33 version information	2	0x3300 (Version: 33, Revision: 00)
c_scnptr	Section-information chain	4	0x00000000: There is no section information. Other than 0: Chain
c_debptr	Debugging-control information chain	4	0x00000000: There is no debugging information. Other than 0: Chain

**(2) Section Information (Maximum 65535)**

Information		Byte	Contents
s_nxptr	Chain to the following section	4	0x00000000: Terminal end of section information Other than 0: Chain
s_scntyp	Section type	2	0x0001: CODE section 0x0002: DATA section 0x0003: BSS section 0x0004: Dummy section
s_lnktyp	Linking method	2	0x0000 (unused)
s_scnatt	Section attribute	2	0x0001: Absolute, 0x0002: Relocatable
s_off	Section start address	4	0x00000000–0xffffffff
s_rcptr	Relocation information chain	4	0x00000000: There is no relocation information. Other than 0: Chain
s_rcsiz	Relocation information byte size	4	0x00000000: When the relocation information chain is 0. Other than 0: Byte size
s_exptr	Extern information chain	4	0x00000000: There is no extern information. Other than 0: Chain
s_exsiz	Extern information byte size	4	0x00000000: When the extern information chain is 0. Other than 0: Byte size
s_excnt	Number of pieces of extern information	4	0x00000000: When the extern information chain is 0. Other than 0: Number of pieces of information
s_rdptra	Chain to actual data	4	0x00000000: There is no actual data (always 0 in BSS). Other than 0: Chain
s_dsiz	Actual data byte size	4	0x00000000: When the chain to actual data is 0 in CODE/ DATA section, or when there is no BSS area in BSS section. Other than 0: Byte size of actual data in CODE/DATA section. Byte size of BSS area in BSS section.
s_scnndx	Section ID	2	0x0000–0xffff

**(3) Relocation Information**

Information		Byte	Contents
r_rctyp	Relocation type	2	0x0001: 8-bit relative symbol, SYMBOL<0x200 0x0002: 32-bit relative symbol (31:22), SYMBOL@rh 0x0003: 32-bit relative symbol (21:9), SYMBOL@rm 0x0004: 32-bit relative symbol (8:1), SYMBOL@rl 0x0005: 26-bit relative symbol (25:13), SYMBOL+sign32@ah 0x0006: 26-bit relative symbol (12:0), SYMBOL+sign32@al 0x0007: 32-bit absolute symbol (31:19), SYMBOL+imm32@h 0x0008: 32-bit absolute symbol (18:6), SYMBOL+imm32@m 0x0009: 32-bit absolute symbol (5:0), SYMBOL+imm32@l 0x000a: 32-bit absolute symbol (31:0), SYMBOL
r_scnoff	Offset in the section	4	0x00000000–Word size of the section to which this information belongs
r_exndx	Index of the extern information to be referred	4	0x00000000–Number of pieces of extern information in the section to be referred
r_scnndx	Section ID to which the extern information to be referred belongs	2	0x0000–Number of sections in the same file
r_symoff	Offset from the symbol	4	0x00000000 (offset 0)–0xffffffff

**(4) Extern Information**

Information		Byte	Contents
e_scnoff	Offset in the section	4	0x00000000–Word size of the section to which this information belongs
e_size	Symbol size	4	0x00000000–0xffffffff (corresponds to .comm, .lcomm)
e_scnndx	Section ID to which the extern information to be referred belongs	4	0x00000000 (used only inside the linker) Reserved area
e_extyp	Extern type	2	0x0001: Global symbol 0x0002: Local symbol 0x0003: Extern symbol
e_namsiz	Length of symbol name	1	0x00–0x20
e_exnam	Symbol name	*	Symbol name, *: Max. 32 bytes

**(5) Actual Data**

CODE section: One code is output in 2 bytes. (in order of upper to lower)

DATA section: One piece of data is output in 1 byte (byte), 2 bytes (half word) or 4 bytes (word). (in order of upper to lower)

BSS section: Does not hold any actual data.

**(6) Debugging Control Information**

Information		Byte	Contents
d_nxptr	Debugging-control-information chain	4	0x00000000: Terminal end of debugging control information Other than 0: Chain
d_flptr	File-name-information chain	4	0x00000000: There is no file-name information. Other than 0: Chain
d_flsiz	File-name-information byte size	4	0x00000000: When the file-name information chain is 0. Other than 0: Byte size
d_flcnt	Number of pieces of file-name information	4	0x00000000: When the file-name information chain is 0. Other than 0: Number of pieces of information
d_stptr	Statement-information chain	4	0x00000000: There is no statement information. Other than 0: Chain
d_stsiz	Statement-information byte size	4	0x00000000: When the statement-information chain is 0. Other than 0: Byte size
d_syptr	Symbol-information chain	4	0x00000000: There is no symbol-information. Other than 0: Chain
d_sysiz	Symbol-information byte size (symbol info.+special statement info.)	4	0x00000000: When the symbol-information chain is 0. Other than 0: Byte size
d_syscnt	Number of pieces of symbol information (symbol info.+special statement info.)	4	0x00000000: When the symbol-information chain is 0. Other than 0: Number of pieces of information

**(7) File-Name Information**

Information		Byte	Contents
f_ftyp	Type	2	0x0000 (unused)
f_dirsiz	Length of directory name	1	0x00–0xff
f_fnamsiz	Length of source file name	1	0x00–0xff
f_fnam	Source file information	*	Path and file name, *: Max. 510 bytes

**(8) Statement Information**

<General Statement Information> \* Line information of source (Statement information when top is other than 0xff)

Information		Byte	Contents
t_line	Number of lines in the source file	4	0x00000000–Last line of source file
t_stat	Statement	2	0x0000 (unused)
t_scnoff	Offset in the reference section	4	0x00000000–Actual size of reference section
t_scnndx	Reference section ID	2	0x0000–Number of sections in the srf33 file to be debugged

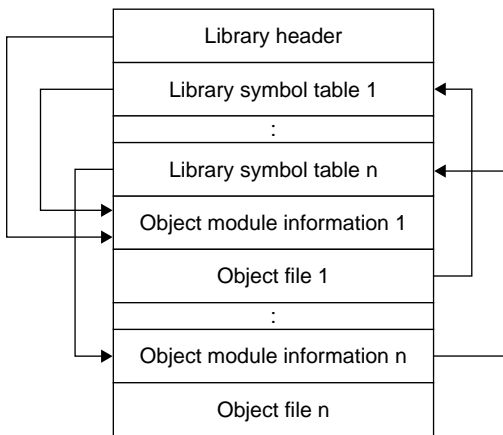
<Special Statement Information> \* Switching information of source file (Top at 0xff)

Information		Byte	Contents
t_type	Type of statement information	2	0xffff: Start of source-file-name reference range 0cffe: End of source-file-name reference range
t_findx	Index to file-name information	4	0x00000000–Number of pieces of file-name information within the same debugging-control information
t_fnoff	Offset in last CODE section of source file	4	0x00000000–Actual size of reference section
t_scnndx	Reference section ID	2	0x0000–Number of sections in the srf33 file to be debugged



## A-2 Library File Structure

The structure of the library files created by the Librarian lib33 is explained below.



Composition of library file  
Fig. A.2 Structure of library file

### Library header

The library file always has one library header at its top. The library header contains the library file name, file size and the pointer (in-file positional information) toward the first object module.

### Object module information

Object module information is created for each object module. It contains the object file name, file size and the pointer toward the corresponding library symbol table.

### Library symbol table

This is the global symbol information table corresponding to each object module. The library symbol table is not created for the object module that has no global symbol. It contains the symbol table size and information of each global symbol (symbol name and pointer toward the corresponding object module information).

## <Reference: Contents of Information>

### (1) Library Header

Information		Byte	Contents
l_att	File control flag	2	0x0010
l_size	Size of entire library	4	Library file size
l_ver	srf33 version information	2	0x3300 (Version: 33, Revision: 00)
l_objptr	Pointer to first object module	4	Offset of the object module information from the beginning of the file
l_namsiz	Length of library file name	1	0x00–0xff
f_fname	Library file name	*	Path and file name, *: Max. 510 bytes

### (2) Object Module Information

Information		Byte	Contents
o_att	File control flag	4	0xffffffff
o_size	Object file size	4	Object file size
o_ismptr	Pointer to library symbol table	4	Offset of the corresponding library symbol table from the beginning of the file.
l_namsiz	Length of object file name	1	0x00–0xff
f_fname	Object file name	*	Path and file name, *: Max. 510 bytes

### (3) Library Symbol Table

Information		Byte	Contents
lst_size	Table size	4	Total size of library symbol table

Individual global symbol information follows the table size information.

Information		Byte	Contents
ls_objptr	Pointer to object module information	4	Offset of the object module information from the beginning of the file
ls_namsiz	Length of global symbol	1	0x00–0x20
ls_glnam	Global symbol name	*	Symbol name, *: Max. 32 bytes

# EPSON International Sales Operations

---

## AMERICA

---

### EPSON ELECTRONICS AMERICA, INC.

#### - HEADQUARTERS -

1960 E. Grand Avenue  
El Segundo, CA 90245, U.S.A.  
Phone: +1-310-955-5300 Fax: +1-310-955-5400

#### - SALES OFFICES -

##### West

150 River Oaks Parkway  
San Jose, CA 95134, U.S.A.  
Phone: +1-408-922-0200 Fax: +1-408-922-0238

##### Central

101 Virginia Street, Suite 290  
Crystal Lake, IL 60014, U.S.A.  
Phone: +1-815-455-7630 Fax: +1-815-455-7633

##### Northeast

301 Edgewater Place, Suite 120  
Wakefield, MA 01880, U.S.A.  
Phone: +1-781-246-3600 Fax: +1-781-246-5443

##### Southeast

3010 Royal Blvd. South, Suite 170  
Alpharetta, GA 30005, U.S.A.  
Phone: +1-877-EEA-0020 Fax: +1-770-777-2637

## EUROPE

---

### EPSON EUROPE ELECTRONICS GmbH

#### - HEADQUARTERS -

Riesstrasse 15  
80992 Muenchen, GERMANY  
Phone: +49-(0)89-14005-0 Fax: +49-(0)89-14005-110

#### - GERMANY -

##### SALES OFFICE

Altstadtstrasse 176  
51379 Leverkusen, GERMANY  
Phone: +49-(0)217-15045-0 Fax: +49-(0)217-15045-10

#### - UNITED KINGDOM -

##### UK BRANCH OFFICE

2.4 Doncastle House, Doncastle Road  
Bracknell, Berkshire RG12 8PE, ENGLAND  
Phone: +44-(0)1344-381700 Fax: +44-(0)1344-381701

#### - FRANCE -

##### FRENCH BRANCH OFFICE

1 Avenue de l'Atlantique, LP 915 Les Conquerants  
Z.A. de Courtaboeuf 2, F-91976 Les Ulis Cedex, FRANCE  
Phone: +33-(0)1-64862350 Fax: +33-(0)1-64862355

## ASIA

---

#### - CHINA -

##### EPSON (CHINA) CO., LTD.

28F, Beijing Silver Tower 2# North RD DongSanHuan  
ChaoYang District, Beijing, CHINA  
Phone: 64106655 Fax: 64107320

##### SHANGHAI BRANCH

4F, Bldg., 27, No. 69, Gui Jing Road  
Caohejing, Shanghai, CHINA  
Phone: 21-6485-5552 Fax: 21-6485-0775

#### - HONG KONG, CHINA -

##### EPSON HONG KONG LTD.

20/F., Harbour Centre, 25 Harbour Road  
Wanchai, HONG KONG  
Phone: +852-2585-4600 Fax: +852-2827-4346  
Telex: 65542 EPSCO HX

#### - TAIWAN, R.O.C. -

##### EPSON TAIWAN TECHNOLOGY & TRADING LTD.

10F, No. 287, Nanking East Road, Sec. 3  
Taipei, TAIWAN, R.O.C.  
Phone: 02-2717-7360 Fax: 02-2712-9164  
Telex: 24444 EPSONTB

##### HSINCHU OFFICE

13F-3, No. 295, Kuang-Fu Road, Sec. 2  
HsinChu 300, TAIWAN, R.O.C.  
Phone: 03-573-9900 Fax: 03-573-9169

#### - SINGAPORE -

##### EPSON SINGAPORE PTE., LTD.

No. 1 Temasek Avenue, #36-00  
Millenia Tower, SINGAPORE 039192  
Phone: +65-337-7911 Fax: +65-334-2716

#### - KOREA -

##### SEIKO EPSON CORPORATION KOREA OFFICE

50F, KLI 63 Bldg., 60 Yoido-Dong  
Youngdeungpo-Ku, Seoul, 150-010, KOREA  
Phone: 02-784-6027 Fax: 02-767-3677

#### - JAPAN -

##### SEIKO EPSON CORPORATION

##### ELECTRONIC DEVICES MARKETING DIVISION

##### Electronic Device Marketing Department

##### IC Marketing & Engineering Group

421-8, Hino, Hino-shi, Tokyo 191-8501, JAPAN  
Phone: +81-(0)42-587-5816 Fax: +81-(0)42-587-5624

##### ED International Marketing Department I (Europe & U.S.A.)

421-8, Hino, Hino-shi, Tokyo 191-8501, JAPAN  
Phone: +81-(0)42-587-5812 Fax: +81-(0)42-587-5564

##### ED International Marketing Department II (Asia)

421-8, Hino, Hino-shi, Tokyo 191-8501, JAPAN  
Phone: +81-(0)42-587-5814 Fax: +81-(0)42-587-5110





In pursuit of **“Saving” Technology**, Epson electronic devices.  
Our lineup of semiconductors, liquid crystal displays and quartz devices  
assists in creating the products of our customers' dreams.  
**Epson IS energy savings.**

# EPSON

---

**SEIKO EPSON CORPORATION**  
**ELECTRONIC DEVICES MARKETING DIVISION**

■ Electronic devices information on Epson WWW server

<http://www.epson.co.jp>

First issue JUNE 1998, Printed AUGUST 1999 in Japan  B