

**CMOS 16-BIT SINGLE CHIP MICROCONTROLLER
(C Compiler Package for S1C17 Family) (Ver. 3.2)**

S5U1C17001C

Manual

Evaluation board/kit and Development tool important notice

1. This evaluation board/kit or development tool is designed for use for engineering evaluation, demonstration, or development purposes only. Do not use it for other purposes. It is not intended to meet the requirements of design for finished products.
2. This evaluation board/kit or development tool is intended for use by an electronics engineer and is not a consumer product. The user should use it properly and in a safe manner. Seiko Epson does not assume any responsibility or liability of any kind of damage and/or fire caused by the use of it. The user should cease to use it when any abnormal issue occurs even during proper and safe use.
3. The part used for this evaluation board/kit or development tool may be changed without any notice.

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. When exporting the products or technology described in this material, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You are requested not to use, to resell, to export and/or to otherwise dispose of the products (and any technical information furnished, if any) for the development and/or manufacture of weapon of mass destruction or for other military purposes.

All brands or product names mentioned herein are trademarks and/or registered trademarks of their respective companies.

S5U1C17001C Manual

1	General
2	Source Files
3	GNU17 IDE
4	C Compiler
5	Library
6	Assembler
7	Linker
8	Debugger
9	Creating Data to Be Submitted
10	Other Tools
11	Quick Reference

Introduction

This document describes the development procedure from compiling C source files to debugging and creating the PA file (Data to be submitted) which is finally submitted to Seiko Epson. It also explains how to use each development tool of the S1C17 Family C Compiler Package common to all the models of the S1C17 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 operating procedures for Eclipse IDE for C/C++ Developers Package
- Knowledge about the gnu C, binutils, and the linker script for the gnu linker (ld)
- 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

Please refer to the general documents that describe ANSI C and Eclipse IDE for C/C++ Developers Package and the manuals that describe gnu tools and Windows, for the above contents.

■ Before installation

See readmeVxxx.txt. This file describes the composition of this package, and provides a general outline of each tool.

■ Installation

Install the tools as described in readmeVxxx.txt.

■ For coding

See the necessary parts in Chapter 2. This chapter describes notes on creating source files and the grammar for the assembler language. Also refer to the following manuals when coding:

S1C17xxx Technical Manual

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

S1C17 Core Manual

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

■ For debugging

Chapter 8 explains details of the debugger.

Also refer to the following manuals to understand operations of the debugging tools:

S1C17 Family In-Circuit Debugger Manual

Explains the functions and handling methods of the ICDmini (S5U1C17001H).

■ For details of each tool

Refer to Chapters 3 to 10 and gnu tool manuals for details.

Manual Notations

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

■ Samples

The sample screens shown in the manual may appear slightly different, depending on the specific system or system fonts.

■ 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, [Stop] button, [q] key, etc.

■ 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. A *fixed-width* font is used to describe these words.

■ 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.

■ 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".

■ 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 [].

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.

■ 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, <filename> needs to be replaced with an actual file name.

■ Development tool name

ICD: Indicates the ICDmini (S5U1C17001H).

— Contents —

1 General.....	1-1
1.1 Features	1-1
1.2 Outline of Software Tools.....	1-2
2 Source Files.....	2-1
2.1 File Format and File Name	2-1
2.2 Grammar of C Source	2-2
2.2.1 Data Type	2-2
2.2.2 Library Functions and Header Files	2-3
2.2.3 In-line Assemble	2-4
2.2.4 Prototype Declarations.....	2-4
2.3 Grammar of Assembly Source	2-5
2.3.1 Statements	2-5
2.3.2 Notations of Operands.....	2-9
2.3.3 Extended Instructions	2-11
2.3.4 Preprocessor Directives.....	2-12
2.4 Precautions for Creation of Sources	2-13
3 GNU17 IDE	3-1
3.1 Overview	3-1
3.1.1 Features	3-1
3.1.2 Some Notes on Use of the IDE	3-1
3.2 Starting and Quitting the IDE	3-2
3.2.1 Starting the IDE.....	3-2
3.2.2 Quitting the IDE.....	3-2
3.3 Projects	3-3
3.3.1 What Is a Project?.....	3-3
3.3.2 Creating a New Project.....	3-3
3.3.3 Creating and Adding a Source File	3-4
3.3.4 Interrupt Vector and Boot Processing Descriptions.....	3-5
3.3.5 Importing an Existing Project	3-6
3.3.6 Importing a GNU17 Version 2 Project.....	3-7
3.4 Setting Project Properties	3-8
3.4.1 Setting GNU17 Project Properties	3-8
3.4.2 Setting Environment Variables.....	3-9
3.4.3 Setting Compiler Path	3-11
3.4.4 Setting Compiler Options	3-11
3.4.5 Setting Linker Options.....	3-12
3.4.6 Setting Assembler Options	3-12
3.5 Building a Program	3-14
3.5.1 Editing a Linker Script.....	3-14
3.5.2 Executing a Build Process	3-15
3.5.3 Clean and Rebuild	3-15
3.5.4 Static Stack Usage Analysis Function.....	3-16
3.6 Debugging the Program.....	3-17
3.6.1 GDB Command File.....	3-17
3.6.2 Setting Standard Input/Output	3-18
3.6.3 Using the Debugger	3-19
3.6.4 Setting the Debug Configuration.....	3-20
3.7 Files Generated in a Project by the IDE.....	3-21
4 C Compiler	4-1

4.1	Functions	4-1
4.2	Input/Output File	4-1
4.2.1	Input File	4-1
4.2.2	Output Files	4-1
4.3	Starting Method	4-2
4.3.1	Startup Format	4-2
4.3.2	Command-line Options	4-2
4.4	Compiler Output	4-7
4.4.1	Output Contents	4-7
4.4.2	Data Representation	4-8
4.4.3	Method of Using Registers	4-10
4.4.4	Function Call	4-11
4.4.5	Stack Frame	4-12
4.4.6	Grammar of C Source	4-13
4.4.7	Compiler Implementation Definition	4-13
4.5	Correspond to Shift JIS Code	4-14
4.6	Functions of xgcc and Usage Precautions	4-15
5	Library	5-1
5.1	Library Overview	5-1
5.1.1	Library Files	5-1
5.1.2	Precautions to Be Taken When Adding a Library	5-2
5.2	Startup Processing Library	5-3
5.2.1	Overview	5-3
5.2.2	Vector Tables	5-3
5.2.3	Stack Pointer Initial Values	5-3
5.2.4	Startup Processing	5-4
5.3	Emulation Library	5-5
5.3.1	Overview	5-5
5.3.2	Floating-point Calculation Functions	5-6
5.3.3	Floating-point Number Processing Implementation Definition	5-8
5.3.4	Integral Calculation Functions	5-9
5.3.5	long long Type Calculation Functions	5-9
5.3.6	Compatibility with Coprocessor Instructions	5-10
5.4	ANSI Library	5-11
5.4.1	Overview	5-11
5.4.2	ANSI Library Function List	5-11
5.4.3	Declaring and Initializing Global Variables	5-17
5.4.4	Lower-level Functions	5-18
6	Assembler	6-1
6.1	Functions	6-1
6.2	Input/Output Files	6-1
6.2.1	Input File	6-2
6.2.2	Output File	6-2
6.3	Starting Method	6-3
6.3.1	Startup Format	6-3
6.3.2	Command-line Options	6-3
6.4	Scope	6-4
6.5	Assembler Directives	6-5
6.5.1	Text Section Defining Directive (.text)	6-5
6.5.2	Data Section Defining Directives (.rodata, .data)	6-6
6.5.3	Bss Section Defining Directive (.bss)	6-7
6.5.4	Data Defining Directives (.long, .short, .byte, .ascii, .space)	6-8

6.5.5	Area Securing Directive (.zero)	6-9
6.5.6	Alignment Directive (.align)	6-10
6.5.7	Global Declaring Directive (.global)	6-11
6.5.8	Symbol Defining Directive (.set)	6-12
6.6	Extended Instructions	6-13
6.6.1	Arithmetic Operation Instructions	6-13
6.6.2	Comparison Instructions	6-15
6.6.3	Logic Operation Instructions	6-16
6.6.4	Data Transfer Instructions (between Stack and Register)	6-17
6.6.5	Data Transfer Instructions (between Memory and Register)	6-18
6.6.6	Immediate Data Load Instructions	6-19
6.6.7	Branch Instructions	6-21
6.6.8	Coprocessor Instructions	6-25
6.6.9	Xext Instructions	6-26
6.7	Error/Warning Messages	6-27
6.8	Precautions	6-28
7	Linker	7-1
7.1	Functions	7-1
7.2	Input/Output Files	7-1
7.2.1	Input Files	7-2
7.2.2	Output Files	7-2
7.3	Starting Method	7-3
7.3.1	Startup Format	7-3
7.3.2	Command-line Options	7-3
7.4	Linkage	7-4
7.4.1	Default Linker Script	7-4
7.4.2	Examples of Linkage	7-6
7.4.3	Link Maps	7-8
7.5	Error/Warning Messages	7-11
7.6	Linker Script Generation Wizard	7-12
7.6.1	Output File	7-12
7.6.2	Starting and Terminating the Linker Script Generation Wizard	7-12
7.6.3	Menu	7-13
7.7	Precautions	7-15
8	Debugger	8-1
8.1	Features	8-1
8.2	Input/Output Files	8-1
8.2.1	Input File	8-1
8.2.2	Output File	8-2
8.3	Starting the Debugger	8-3
8.3.1	Startup Format	8-3
8.3.2	Startup Options	8-4
8.3.3	Executing Command Files	8-5
8.3.4	Quitting the Debugger	8-6
8.4	Method of Executing Commands	8-7
8.4.1	Entering Commands From the Keyboard	8-7
8.4.2	Parameter Input Format	8-8
8.5	Command Reference	8-9
8.5.1	List of Commands	8-9
8.5.2	Detailed Description of Commands	8-10
	Command name (operation of command) [Supported modes]	8-10
8.5.3	Memory Manipulation Commands	8-11

	c17 fb (fill area, in bytes).....	8-11
	c17 fh (fill area, in 16 bits).....	8-11
	c17 fw (fill area, in 32 bits) [ICD Mini / SIM]	8-11
	X (memory dump) [ICD Mini / SIM].....	8-13
	set { } (data input) [ICD Mini / SIM]	8-15
	c17 mvb (copy area, in bytes).....	8-16
	c17 mvh (copy area, in 16 bits).....	8-16
	c17 mvw (copy area, in 32 bits) [ICD Mini / SIM]	8-16
	c17 df (save memory contents) [ICD Mini / SIM].....	8-17
8.5.4	Register Manipulation Commands	8-19
	info reg (display register) [ICD Mini / SIM].....	8-19
	set \$ (modify register) [ICD Mini / SIM]	8-20
8.5.5	Program Execution Commands	8-21
	continue (execute continuously) [ICD Mini / SIM].....	8-21
	until (execute continuously with temporary break) [ICD Mini / SIM].....	8-22
	step (single-step, every line)	8-24
	steppi (single-step, every mnemonic) [ICD Mini / SIM]	8-24
	next (single-step with skip, every line)	8-25
	nexti (single-step with skip, every mnemonic) [ICD Mini / SIM]	8-25
	finish (finish function) [ICD Mini / SIM]	8-26
8.5.6	CPU Reset Commands.....	8-27
	c17 rst (reset) [ICD Mini / SIM]	8-27
	c17 rstt (reset target) [ICD Mini]	8-28
8.5.7	Interrupt Commands	8-29
	c17 int (interrupt) [SIM]	8-29
	c17 intclear (clear interrupt) [SIM]	8-30
8.5.8	Break Setup Commands.....	8-31
	break (set software PC break).....	8-31
	tbreak (set temporary software PC break) [ICD Mini / SIM].....	8-31
	hbreak (set hardware PC break).....	8-34
	thbreak (set temporary hardware PC break) [ICD Mini / SIM]	8-34
	delete (clear break by break number) [ICD Mini / SIM]	8-36
	clear (clear break by break position) [ICD Mini / SIM].....	8-37
	enable (enable breakpoint)	8-38
	disable (disable breakpoint) [ICD Mini / SIM].....	8-38
	ignore (disable breakpoint with ignore counts) [ICD Mini / SIM]	8-39
	info breakpoints (display breakpoint list) [ICD Mini / SIM].....	8-40
	commands (setting a command to execute after break) [ICD Mini / SIM]	8-41
8.5.9	Symbol Information Display Commands	8-42
	info locals (display local symbol).....	8-42
	info var (display global symbol) [ICD Mini / SIM]	8-42
	print (alter symbol value) [ICD Mini / SIM]	8-43
8.5.10	File Loading Commands	8-44
	file (load debugging information) [ICD Mini / SIM].....	8-44
	load (load program) [ICD Mini / SIM]	8-45
8.5.11	Trace Command	8-46
	c17 tm (set trace mode) [SIM].....	8-46
8.5.12	Other Commands.....	8-49
	set output-radix (change of variable display format) [ICD Mini / SIM]	8-49
	set logging (log output setting) [ICD Mini / SIM].....	8-50
	source (execute command file) [ICD Mini / SIM]	8-51
	target (connect target MCU) [ICD Mini / SIM]	8-52
	detach (disconnect target MCU) [ICD Mini / SIM]	8-53
	pwd (display current directory).....	8-54
	cd (change current directory) [ICD Mini / SIM]	8-54
	c17 ttbr (set TTBR) [SIM]	8-55
	c17 cpu (set CPU type) [SIM]	8-56

c17 chgclkmd (DCLK change mode) [ICD Mini]	8-57
c17 pwul (unlock flash security password) [ICD Mini]	8-58
c17 help (help) [ICD Mini / SIM]	8-59
c17 model_path (model-specific information file directory setting) [ICD Mini / SIM]	8-61
c17 model (MCU model name setting) [ICD Mini / SIM]	8-62
c17 flv (flash programming power setting) [ICD Mini]	8-64
c17 flvs (flash programming power setting cancellation) [ICD Mini]	8-65
c17 stdin (input of data using input/output functions) [ICD Mini / SIM]	8-66
c17 stdout (output of data using input/output functions) [ICD Mini / SIM]	8-67
c17 lcdsim (LCD panel simulator setting/cancellation) [ICD Mini]	8-68
quit (quit debugger) [ICD Mini / SIM]	8-69
8.6 Status and Error Messages	8-70
8.6.1 Status Messages	8-70
8.6.2 Error Messages	8-70
8.7 Run Time Measurement	8-71
8.7.1 Display Method	8-71
8.7.2 Restrictions	8-71
8.8 Peripheral Circuit Simulator (ES-Sim17)	8-72
8.8.1 Input/Output files	8-73
8.8.2 Starting and Terminating ES-Sim17	8-74
8.8.3 Menus	8-75
8.8.4 Simulating I/O Ports	8-76
8.8.5 Simulating SVD	8-78
8.8.6 Simulating an LCD Driver	8-79
8.8.7 ES-Sim17 Error Messages	8-80
8.8.8 Restrictions	8-80
8.9 LCD Panel Simulator	8-81
8.9.1 Input Files	8-82
8.9.2 Starting and Terminating the LCD Panel Simulator	8-83
8.9.3 Procedure for Modifying the Program	8-84
8.9.4 Restrictions	8-85
8.10 Profiler Coverage	8-86
8.10.1 Input/Output Files	8-86
8.10.2 Starting and Terminating the Profiler Coverage	8-87
8.10.3 Preparation	8-88
8.10.4 Coverage Function	8-89
8.10.5 Profiler Function	8-90
8.10.6 Restrictions	8-90
9 Creating Data to Be Submitted	9-1
9.1 Outline of Tools for Creating Data to Be Submitted	9-1
9.2 Procedure for Creating Data to Be Submitted	9-2
9.2.1 Creating FDC Files (Function Option Documents) Using winfog17	9-2
9.2.2 Creating PSA Files (ROM Data)	9-5
9.2.3 Creating PA Files (Data to Be Submitted) Using windmc17	9-5
9.2.4 PA File (Data to Be Submitted) Separation Procedure	9-6
9.3 Error Messages for Submitted Data Creation Tools	9-9
9.3.1 winfog17 Error Messages	9-9
9.3.2 winmdc17 Error Messages	9-10
9.4 Sample Output for Submitted Data Creation Tools	9-11
10 Other Tools	10-1
10.1 objdump.exe	10-1
10.1.1 Function	10-1
10.1.2 Input Files	10-1

10.1.3 Method for Using objdump	10-1
10.1.4 Error Message	10-2
10.1.5 Precautions	10-2
10.2 objcopy.exe	10-3
10.2.1 Function	10-3
10.2.2 Input/Output Files	10-3
10.2.3 Method for Using objcopy	10-4
10.2.4 Creating SA Files (ROM Data)	10-5
10.3 ar.exe	10-6
10.3.1 Function	10-6
10.3.2 Input/Output Files	10-6
10.3.3 Method for Using ar	10-7
10.4 moto2ff.exe	10-9
10.4.1 Function	10-9
10.4.2 Input/Output Files	10-9
10.4.3 Startup Format	10-9
10.4.4 Error/Warning Messages	10-10
10.4.5 Creating SAF File (ROM Data)	10-10
10.5 sconv32.exe	10-11
10.5.1 Function	10-11
10.5.2 Input/Output Files	10-11
10.5.3 Startup Format	10-11
10.5.4 Error Messages	10-12
10.6 gpdata.exe	10-13
10.6.1 Function	10-13
10.6.2 Input/Output Files	10-13
10.6.3 Method for Using gpdata	10-13
10.7 ptd.exe	10-14
10.7.1 Function	10-14
10.7.2 Input/Output Files	10-14
10.7.3 Method for Using ptd.exe	10-14
10.7.4 Error Messages	10-15
10.7.5 Method for Setting Flash Protection	10-15
10.8 LCDUtil17 (LCD Panel Customizing Tool)	10-16
10.8.1 Overview	10-16
10.8.2 Input/Output files	10-16
10.8.3 Starting and Closing LCDUtil17	10-17
10.8.4 Window	10-17
10.8.5 Menus and Toolbar	10-18
10.8.6 Producing an LCD file	10-21
10.8.7 Shortcut Key list	10-28
10.8.8 Warning Messages and Error Messages	10-29
11 Quick Reference	11-1

1 General

1.1 Features

The S1C17 Family C Compiler Package contains software development tools and utilities for compiling C source programs and assembling and debugging assembly source programs, as well as creating PSA files (ROM data) and PA files (Data to be submitted).

The tools are common to all the models of the S1C17 Family.

Its principal features are as follows:

- **Powerful optimizing function**

The C Compiler is designed to suit to the S1C17 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.

- **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 ICDmini (S5U1C17001H) emulator, the program can be debugged while the target board is running. It also allows use of the S1C17MCU core simulator.

- **Integrated development environment for Windows**

Designed to run under Microsoft Windows, the **GNU17 IDE** is a seamless integrated development environment suitable for a wide range of development tasks, from source creation to debugging.

1.2 Outline of Software Tools

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

(1) C Compiler (**xgcc.exe**)

This tool is made based on GNU C Compiler and is compatible with ANSI C. This tool invokes **cpp.exe** and **cc1.exe** sequentially to compile C source files to the assembly source files for the S1C17 Family. It has a powerful optimizing ability that can generate minimized assembly codes. The **xgcc.exe** can also invoke the **as.exe** assembler to generate object files.

(2) Assembler (**as.exe**)

This tool assembles assembly source files output by the C compiler and converts the mnemonics of the source files into object codes (machine language) of the S1C17 Core. The **as.exe** allows the user to invoke the assembler through **xgcc.exe**, this makes it possible to include preprocessor directives into assembly source files. The results are output in an object file that can be linked or added to a library.

(3) Linker (**ld.exe**)

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

(4) Debugger (**gdb.exe**)

This debugger serves to perform source-level debugging by controlling an ICDmini. It also comes with a simulator function that allows debugging on a personal computer.

(5) Librarian (**ar.exe**)

This tool is used to edit libraries. The **ar.exe** can register object modules created by the C compiler and assembler to libraries, delete object modules in libraries and restore library modules to the original object files.

(6) GNU17 IDE (**eclipse.exe**)

The development workbench provides an integrated development environment for a wide range of development tasks, from source creation to debugging.

This package contains other gnu tools, sample programs and several utility programs. For details on those programs, please refer to "readmeV_{xxx}.txt" (xxx indicates version) on the disk.

Note: Only the command options for each tool described in the respective section are guaranteed to work. If other options are required, they should only be used at the user's own risk.

2 Source Files

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

2.1 File Format and File Name

Use the **GNU17 IDE** editor or a general-purpose editor to create source files.

● File format

Save data in a standard text file.

● File name

C source file <filename>.c

Assembly source file <filename>.s

Specify the <filename> 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 S1C17 tools.

● 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.

● Global variables/static variables

Up to 200 characters can be used to name global and static variables.

A total of 32,000 global and static variables can be accepted.

● File size

The following shows the guide about the upper limit of the C source file size:

- In the case of a source file that contains only variables, constants and arrays, up to 100,000 lines can be accepted.
- In the case of a source file that contains only executable codes (not including arrays and variables), up to 20,000 lines can be accepted. However, the number of acceptable lines varies depending on the source density.
- Consider these two conditions above as reference for sources in which variables, constants, arrays and executable codes are mixed.
- The number of lines shown above varies depending on compile environment conditions. Moreover, the compiler may be forcibly terminated due to insufficient resources. In this case, build the program under a resource-rich environment or divide the source file into multiple files before compiling. (Resources described here depend on the OS used rather than the RAM capacity of the PC.)
- Up to 512 characters can be used per line in C source files.
- In the case of assembly source files, up to 30,000 lines can be accepted.

● Tab setting

The recommended tab stop is every 4 characters. This is the default tab setting when the IDE displays sources.

● 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).

2.2 Grammar of C Source

The **xgcc** C compiler included in this package is the GNU C Compiler under ANSI C standards.

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.

2.2.1 Data Type

The **xgcc** C compiler 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 2.2.1.1.

Table 2.2.1.1 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	2	-32768 to 32767
unsigned int	2	0 to 65535
long	4	-2147483648 to 2147483647
unsigned long	4	0 to 4294967295
pointer	4	0 to 16777215
float	4	1.175e-38 to 3.403e+38 (normalized number)
double	8	2.225e-308 to 1.798e+308 (normalized number)
long long	8	-9223372036854775808 to 9223372036854775807
unsigned long long	8	0 to 18446744073709551615
wchar_t	2	0 to 65535

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

Handling of long long-type constants requires the suffix `LL` or `ll` (long long type) or `ULL` or `ull` (unsigned long long type). If this suffix is not present, a warning is generated, since the compiler may not be able to recognize long long-type constants as such.

```
Example: long long ll_val;
         ll_val = 0x1234567812345678;
           → warning:integer constant is too large for "long" type
         Ll_val = 0x1234567812345678LL;
           → OK
```

Type `wchar_t` is the data type needed to handle wide characters. This data type is defined in `stdlib.h/stddef.h` as the type `unsigned short`.

2.2.2 Library Functions and Header Files

This package contains an ANSI 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.

Certain ANSI library functions not supported by this package are not included in the ANSI library. The client assumes responsibility for function implementation and prototype declarations when using ANSI library functions not supported by this package. For some ANSI library functions not supported by this package, the header files include only prototype declarations. In these cases, include the pertinent header file rather than declaring a prototype before implementing the function.

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

Table 2.2.2.1 List of library files and functions

ANSI library

File name	Functions/macros	Corresponding header file
libc.a	perror, 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, malloc, calloc, realloc, free, atoi, atol, atof, strtol, strtoul, strtod, abs, labs, div, ldiv, rand, srand, bsearch, qsort	stdlib.h
	setjmp, longjmp	setjmp.h
	time, mktime, gmtime	time.h
	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
	memchr, memmove, strchr, strcspn, strncat, strpbrk, strstr, memcmp, memset, strcmp, strerror, strncmp, strchr, strtok, memcpy, strcat, strcpy, strlen, strncpy, strspn	string.h
	isalnum, iscntrl, isgraph, isprint, isspace, isxdigit, toupper, isalpha, isdigit, islower, ispunct, isupper, tolower	ctype.h
	va_start, va_arg, va_end	stdarg.h

Emulation library

File name	Functions
libgcc.a	__subdf3, __adddf3, __addsf3, __ashldi3, __ashlhi3, __ashlsi3, __ashrdi3, __ashrhi3, __ashrsi3, __cmpdi2, __divdf3, __divdi3, __divhi3, __divsf3, __divsi3, __eqdf2, __eqsf2, __extendsfdf2, __fixdfdi, __fixdfsi, __fixsfdi, __fixsfsi, __fixunsdfdi, __fixunsdfsi, __fixunssfdi, __fixunssfsi, __floatdidf, __floatdisf, __floatsidf, __floatsisf, __gedf2, __gesf2, __gtdf2, __gtsf2, __ledf2, __lesf2, __lshrdi3, __lshrhi3, __lshrsi3, __ltdf2, __ltsf2, __moddi3, __modhi3, __modsi3, __muldf3, __muldi3, __mulhi3, __mulsf3, __mulsi3, __nedf2, __negdf2, __negdi2, __negsf2, __nesf2, __subsf3, __truncdfsf2, __ucmpdi2, __udivdi3, __udivhi3, __udivsi3, __umoddi3, __umodhi3, __umodsi3, __cmpsi2, __ucmpsi2

Functions with prototype declarations only

Functions	Corresponding header file
freopen, tmpfile, tmpnam, remove, rename, fopen, fclose, setbuf, setvbuf, fflush, clearerr, feof, ferror, fseek, fgetpos, fsetpos, ftell, rewind	stdio.h
atexit, getenv, system	stdlib.h
difftime, clock, localtime, asctime, ctime	time.h

For details about the functions included in the libraries, refer to Chapter 5, "Library". When using a library function, be sure to specify the library file that contains the function used when linking. The linker extracts only the necessary object modules from the specified library file as it links them.

2.2.3 In-line Assemble

The **xgcc** C compiler 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(" .long BOOT\n\
 .long ADDR_ERR\n\
 .long NMI\n\
 .space 4\n\
 .long EINT0\n\
 .long EINT1");

Example 3: BOOT() {
 asm("xld.a %sp,0x3f00"); /* set SP */
 :
 }

For details on how to write an assembly source, refer to Section 2.3, "Grammar of Assembly Source".

2.2.4 Prototype Declarations

● Declaring interrupt handler functions

Interrupt handler functions should be declared in the following format:

*<type>**<function name>* **__attribute__ ((interrupt_handler))** ;

Example: void foo(void) **__attribute__ ((interrupt_handler))** ;

```
int int_num;
void foo()
{
    int_num = 5;
}
```

Assembler code

```
foo:
    ld.a -[%sp],%r2
    ld %r2,5
    xld [int_num],%r2
    ld.a %r2,[%sp]+
    reti
```

2.3 Grammar of Assembly Source

2.3.1 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>	(<Operands>)	(;<Comment>)
2	<Assembler directive>	(<Parameters>)	(;<Comment>)
3	<Label>:	(;<Comment>)	
4	;<Comment>		
5	<Extended instruction>	<Operands>	(;<Comment>)
6	<Preprocessor directive>	(<Parameters>)	(;<Comment>)

Example:

Statement	Syntax pattern
<code>; boot.s</code>	4
<code>; boot program</code>	4
<code>#define SP_INI,0x3f00 ; Stack pointer value</code>	6
	2
<code>.text</code>	
<code>.long BOOT ; BOOT VECTOR</code>	2
<code>BOOT:</code>	3
<code>xld.a %sp,SP_INI ; set SP</code>	5
<code>xcall main ; goto main</code>	5
<code>jpr BOOT ; infinity loop</code>	1

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

● Restrictions

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

Example: `;OK`
`BOOT: ld %r1,%r2`
`ld %r0,%r1`
`;Error`
`BOOT: ld %r1,%r2 ld %r0,%r1`

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

Example: `;OK`
`ld %r1,%r2`
`;Error`
`ld %r1,`
`%r2`

- 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. When using non-ASCII characters (such as Chinese characters) for comments, use `/* . . . */` as the comment symbol.

(1) Instructions (Mnemonics and Operands)

An instruction to the S1C17 Core 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 *<Operands>*

<Mnemonic> tab or space *<Operand 1>*, *<Operand 2>*

Example: `nop`
 `call SUB1`
 `ld%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 S1C17 Core instructions can be used in the S1C17 Family:

ld.b	ld.ub	ld	ld.a		
add	add/c	add/nc	add.a	add.a/c	add.a/nc
adc	adc/c	adc/nc	sub	sub/c	sub/nc
sub.a	sub.a/c	sub.a/nc	sbc	sbc/c	sbc/nc
cmp	cmp/c	cmp/nc	cmp.a	cmp.a/c	cmp.a/nc
cmc	cmc/c	cmc/nc			
and	and/c	and/nc	or	or/c	or/nc
xor	xor/c	xor/nc	not	not/c	not/nc
sr	sa	sl	swap		
cv.ab	cv.as	cv.al	cv.la	cv.ls	
jpr	jpr.d	jpa	ipa.d	jrgt	jrgt.d
jрге	jрге.d	jrlt	jrlt.d	jrle	jrle.d
jrugt	jrugt.d	jruge	jruge.d	jruIt	jruIt.d
jrule	jrule.d	jreq	jreq.d	jrne	jrne.d
call	call.d	calla	calla.d	ret	ret.d
int	intl	reti	reti.d	brk	ret.d
ext	nop	halt	slp	ei	di
ld.cw	ld.ca	ld.cf			

Refer to the "S1C17 Core 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", "LD", and "Ld" are all accepted as "ld" instructions.

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

More will be said about operands later.

(2) Assembler Directives

The **as** assembler supports the standard directives provided in the **gnu** assembler.

Refer to the **gnu** assembler manual for the standard directives. Each directive begins with a period (.). The following lists often-utilized directives.

.text		Declares a <code>.text</code> section.
.section .data		Declares a <code>.data</code> section.
.section .rodata		Declares a <code>.rodata</code> section.
.section .bss		Declares a <code>.bss</code> section.
.long	<data>	Defines a 4-byte data.
.short	<data>	Defines a 2-byte data.
.byte	<data>	Defines a byte data.
.ascii	<string>	Defines an ASCII character strings.
.space	<length>	Defines a blank (0x0) space.
.zero	<length>	Defines a blank (0x0) space.
.align	<value>	Alignment to a specified boundary address.
.global	<symbol>	Defines a global symbol.
.set	<symbol> , <address>	Defines a symbol with an absolute address.

(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 `.text/.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

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.

```
Example: ;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.

Strings from "/*" through the next "*/" are also regarded 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.

Example: ;This line is a comment line.

```

    LABEL:                ;Comment for LABEL.
        ld %a,%b          ;Comment for the instruction on the left.
/*
    This type of comment can include
    newline characters.
*/

```

● Restrictions

When a comment extends to several lines, each line must begin with a semicolon or use "/*" and "*/".

Example: ;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.

```

/*

```

```

    These are

```

comment lines. Both lines will be regarded as a comment.

```

*/

```

(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, for example, when used as a break in a series of routines.

2.3.2 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 S1C17 Core all contain a percentage symbol (%). Register names may be written in either uppercase or lowercase letters.

General-purpose register (%rd, %rs, %rb)	Notation
General-purpose register R0–R7	%r0–%r7 or %R0–%R7
Special register	Notation
Stack pointer SP	%sp or %SP
Program counter PC	%pc or %PC

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

Example: [%r7] [%r1] + [%sp+imm7]

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 **as** 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.

Example: 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.

Example: 0x1a 0xff00

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

● Binary notations of values

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

Example: 0b1001 0b01001100

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

● 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 2.3.2.1 Types of immediate data and their specifiable ranges

Symbol	Type	Decimal	Hexadecimal	Binary
imm3	3-bit immediate data	0 to 7	0x0 to 0x7	0b0 to 0b111
imm5	5-bit immediate data	0 to 31	0x0 to 0x1f	0b0 to 0b1 1111
imm7	7-bit immediate data	0 to 127	0x0 to 0x7f	0b0 to 0b111 1111
sign7	Signed 7-bit immediate data	-64 to 63	0x0 to 0x7f	0b0 to 0b111 1111
sign8	Signed 8-bit immediate data	-128 to 127	0x0 to 0xff	0b0 to 0b1111 1111
sign10	Signed 10-bit immediate data	-512 to 511	0x0 to 0x3ff	0b0 to 0b11 1111 1111
imm13	13-bit immediate data	0 to 8,191	0x0 to 0x1fff	0b0 to 0b1 1111 1111 1111
imm16	16-bit immediate data	0 to 65,535	0x0 to 0xffff	0b0 to 0b1111 1111 1111 1111
sign16	Signed 16-bit immediate data	-32,768 to 32,767	0x0 to 0xffff	0b0 to 0b1111 1111 1111 1111
imm20	20-bit immediate data	0 to 1,048,575	0x0 to 0xfffff	0b0 to 0b1111 1111 1111 1111 1111
sign21	Signed 21-bit immediate data	-1,048,576 to 1,048,575	0x0 to 0x1fffff	0b0 to 0b1 1111 1111 1111 1111 1111
sign23	Signed 23-bit immediate data	-4194304 to 4194303	0x0 to 0x7fffff	0b0 to 0b111 1111 1111 1111 1111 1111
imm24	24-bit immediate data	0 to 16,777,215	0x0 to 0xffffff	0b0 to 0b1111 1111 1111 1111 1111 1111
sign24	Signed 24-bit immediate data	-8,388,608 to 8,388,607	0x0 to 0xffffff	0b0 to 0b1111 1111 1111 1111 1111 1111

(3) Symbols

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

● Definition of symbols

Usable symbols are defined as 24-bit values by any of the following methods:

1. It is described as a label (in text, data or bss section)

Example: LABEL1 :

LABEL1 is a symbol that indicates the address of a described location in the .text, .data, or .bss section.

2. It is defined with the .set directive

Example: .set ADDR1,0xff00

ADDR1 is a symbol that represents absolute address 0x00ff00.

● Restrictions on characters

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 directive.

● 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.

Example: `call LABEL ← LABEL = sign10`
`ld,a %rd,LABEL ← LABEL = sign7`

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

`LABEL + imm24 LABEL + sign24`

Example: `xcall LABEL+0x10`

2.3.3 Extended Instructions

The extended instructions are 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 `as` assembler.

● Types of extended instructions

<code>xadd</code>	<code>xadd.a</code>	<code>xadc</code>	<code>xsub</code>	<code>xsub.a</code>	<code>xsbc</code>	<code>xcmp</code>
<code>xcmp.a</code>	<code>xcmc</code>					
<code>sadd</code>	<code>sadd.a</code>	<code>sadc</code>	<code>ssub</code>	<code>ssub.a</code>	<code>ssbc</code>	<code>scmp</code>
<code>scmp.a</code>	<code>scmc</code>					
<code>xand</code>	<code>xoor</code>	<code>xxor</code>				
<code>sand</code>	<code>soor</code>	<code>sxor</code>				
<code>xld</code>	<code>xld.a</code>	<code>xld.b</code>	<code>xld.ub</code>			
<code>sld</code>	<code>sld.a</code>	<code>sld.b</code>	<code>sld.ub</code>			
<code>xjpr</code>	<code>xjpr.d</code>	<code>xjpa</code>	<code>xjpa.d</code>	<code>xjreq</code>	<code>xjreq.d</code>	<code>xjrne</code>
<code>xjrne.d</code>	<code>xjrgt</code>	<code>xjrgt.d</code>	<code>xjrge</code>	<code>xjrge.d</code>	<code>xjrslt</code>	<code>xjrslt.d</code>
<code>xjrle</code>	<code>xjrle.d</code>	<code>xjrugt</code>	<code>xjrugt.d</code>	<code>xjruge</code>	<code>xjruge.d</code>	<code>xjrult</code>
<code>xjrult.d</code>	<code>xjrle</code>	<code>xjrle.d</code>	<code>xcall</code>	<code>xcall.d</code>	<code>xcalla</code>	<code>xcalla.d</code>
<code>sjpr</code>	<code>sjpr.d</code>	<code>sjpa</code>	<code>sjpa.d</code>	<code>sjreq</code>	<code>sjreq.d</code>	<code>sjrne</code>
<code>sjrne.d</code>	<code>sjrgt</code>	<code>sjrgt.d</code>	<code>sjrge</code>	<code>sjrge.d</code>	<code>sjrslt</code>	<code>sjrslt.d</code>
<code>sjrle</code>	<code>sjrle.d</code>	<code>sjrugt</code>	<code>sjrugt.d</code>	<code>sjruge</code>	<code>sjruge.d</code>	<code>sjrult</code>
<code>sjrult.d</code>	<code>sjrule</code>	<code>sjrule.d</code>	<code>scall</code>	<code>scall.d</code>	<code>scalla</code>	<code>scalla.d</code>
<code>xld.cw</code>	<code>xld.ca</code>	<code>xld.cf</code>				
<code>sld.cw</code>	<code>sld.ca</code>	<code>sld.cf</code>				

● Method for using extended instructions

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

```
Example: xcall LABEL          ; ext LABEL[23:10]
          ; call LABEL[9:0]

          sld.a %r1,imm16     ; ext imm16[15:7]
          ; ld.a %r1,imm16[6:0]

          xld.a %r1,imm24     ; ext imm24[23:20]
          ; ext imm24[19:7]
          ; ld.a %r1,imm24[6:0]
```

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.

```
Example: xld.a %r0,symbol + 0x10 ; R0 ← symbol + 0x10
          xjpa LABEL + 5         ; Jumps to address LABEL + 5.
```

For details about the extended instructions that include operands, refer to Section 6.6, "Extended Instructions".

2.3.4 Preprocessor Directives

The **cpp** C preprocessor directives can be used in assembly source files.
The principal directives are as follows:

#include	Insertion of file
#define	Definition of character strings and numbers
#if-#else-#endif	Conditional assembly

```
Example: #include    "define.h"
         #define     NULL          0
         #ifdef      TYPE1
         ld          %r0,0
         #else
         ld          %r0,-1
         #endif
```

Refer to the gnu C preprocessor manual for details of the preprocessor directives.

Note: The sources that contain preprocessor directives need to be processed by the preprocessor (use the **xgcc** options **-c** and **-xassembler-with-cpp**), and cannot be entered directly into the **as** assembler. (Direct entry into the assembler will result an error.)

2.4 Precautions for Creation of Sources

- (1) Place a tab stop every 4 characters wherever possible. Source display/mixed display with the **gdb** debugger of a source set at a tab interval other than 4 characters may 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 the **gdb** debugger operation error. This does not apply to ordinary header files that do not contain sources.
- (3) 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.
- (4) The C compiler assumes that the address size is 24 bits by default. Therefore, be aware that the expected results may not be obtained from an operation using an `unsigned int/unsigned short` type variable and a pointer, as `int` type variables are 16-bit size, as shown below.

```
int* ip_Pt;
unsigned int i = 1;

ip_Pt += (-1)*i;
```

The code above is written to expect `"ip_Pt += (-1);"`, however it will be processed as `"ip_Pt += 0xffff;"`. Although it will be processed normally when the address space is 16-bit size, an invalid address will result if the address space is 24-bit size.

To perform a pointer operation when the address space is 24-bit size, avoid using `unsigned int/unsigned short` type variables, or add the suffix `'L'` to the constant as shown below so that it will be handled as a `long` type constant.

```
ip_Pt += (-1L)*i;
```

- (5) In C sources, function names can be used as the pointer to the function, note, however, that the pointer values cannot be assigned to real type (`float/double`) variables and arrays using the function names. They can be assigned to integer type variables and arrays. However, if assigning a value to a global variable/array using a function name at the same time the variable/array is declared, the types of variables/arrays are limited depending on the address space size.

In 24-bit address space (default condition)

The `long/unsigned long` type variables/arrays only allow substitution with a function name.

In 16-bit address space (when `-mpointer16` is specified)

The `short/unsigned short/int/unsigned int` type variables/arrays only allow substitution with a function name.

If it is not at declaration, global variables/arrays in any integer type can be substituted with a function name.

Integer type local variables/arrays always allow substitution with a function name regardless of whether it is at declaration or not.

Example: In 16-bit address space (when `-mpointer16` is specified)

- 1) `short s_Global_Val = (short)boot;`
→ A function name can be used to assign the pointer to the `short` type global variable `s_Global_Val` when it is declared.
- 2) `long l_Global_Val = (long)boot;`
→ An error occurs if a function name is assigned to the `long` type global variable `l_Global_Val` when it is declared
error: initializer element is not constant
- 3) `short s_local_val = (short)boot;`
→ A function name can be used to assign the pointer to the `short` type local variable `s_local_val`.
- 4) `long l_local_val = (long)boot;`
→ A function name can be used to assign the pointer to the `long` type local variable `l_local_val`.

```

5) char c_Global_Val;
void sub()
{
    c_Global_Val = (char)boot;
}

```

→ A function name can be used to assign the pointer to the `char` type global variable `c_Global_Val` except when it is declared.

- (6) Function pointers can be used in C sources, note, however, that function pointers cannot be assigned to real type (`float/double`) variables and arrays similar to (5) above.

They can be assigned to integer type variables and arrays.

However, when a global variable/array is declared, a function pointer cannot be assigned at the same time.

If it is not at declaration, a function pointer can be assigned to integer type global variables/arrays.

Integer type local variables/arrays always allow substitution with a function pointer regardless of whether it is at declaration or not.

However, a warning occurs depending on a combination of the address space size and the type of global/local variable or global/local array.

In 24-bit address space (default condition)

A warning occurs if a function pointer is assigned to a variable/array other than `long/unsigned long` data types.

In 16-bit address space (when `-mpointer16` is specified)

A warning occurs if a function pointer is assigned to a variable/array other than `short/unsigned short/int/unsigned int` data types.

Example: In 16-bit address space (when `-mpointer16` is specified)

```
void (* fp_Pt) (void); // Declaration of a function pointer with void type return value and argument
```

```
1) short s_Global_Val = (short)fp_Pt;
```

→ An error occurs if a function pointer is assigned to the global variable `s_Global_Val` when it is declared.
error: initializer element is not constant

```
2) short s_local_val = (short)fp_Pt;
```

→ A function pointer can be assigned to the `short` type local variable `s_local_val`.

```
3) long l_local_val = (long)fp_Pt;
```

→ Although a function pointer can be assigned to the `long` type local variable `l_local_val`, a warning will occur.
warning: cast from pointer to integer of different size

```
4) short s_Global_Val;
void sub()
{
    s_Global_Val = (short)fp_Pt;
}

```

→ A function pointer can be assigned to the `short` type global variable `s_Global_Val` except when it is declared.

- (7) Be sure to include the prototype declaration or the `extern` declaration of the functions.

If there is no prototype or `extern` declaration and if a function without its definition part in an earlier part of the same file is called, the type assumed in the file calling the function may differ from the function type actually called, resulting in a potential malfunction. Even so, the function will compile without errors.

However, a warning is generated if the definition part of the called function is present in the same file. If the definition part of the called function is located in another file, no warning is generated unless the `-Wall` option is attached.

Since the return value is implicitly assumed to be of the `int` type, the correct value will not be returned if the return value has a data type larger than `int`.

Example:

```
long l_Val=0x12345678,l_Val_2;

int main()
{
    l_Val_2 = sub();          // l_Val_2 is substituted with 0x5678.
    return 0;
}

long sub()
{
    long l_wk;

    l_wk = l_Val;
    return l_wk;
}
```

- (8) Do not use a pointer other than "char" to perform a read/write operations to an odd-number memory. Failure to observe this warning will result in an address error exception.

Example:

```
int *ip_Pt;

int sub()
{
    ip_Pt = (int *)0x3;
    (*ip_Pt) = 0x2;

    return 0;                // Address error exception occurs here.
}
```

- (9) Due to the specifications of the C language, note that processing an undefined action can result in different calculation results due to differences in optimization options and local/external variables. Undefined processing includes the following cases:
- When overflow is occurring during conversion from floating decimal to integer
 - When shift calculation is performed with a negative value or a value equal to or greater than the bit length of the calculation target after a type promotion.

- (10) Due to C language specifications, an attempt to access a variable using an incompatible pointer may result in the following warning message if the `-Wall` option is specified. In this case, reference or assignment of variables via pointers may not be performed correctly.

warning: dereferencing type-punned pointer will break strict-aliasing rules

Example: When the `-O3` option is specified.

```
int sub()
{
    int p1 = 0 ;
    short *p2 = (short *)&p1 ;
}
```

Because `p2` and `&p1` are incompatible pointers, a warning message will appear.

In that case, variable `p1` may not be referenced by means of pointer `p2` or assignment may not be performed correctly.

3 GNU17 IDE

This chapter describes the facilities available with the **GNU17 IDE** and describes how to use the **GNU17 IDE**.

3.1 Overview

3.1.1 Features

The **GNU17 IDE** (hereafter simply the **IDE**) provides an integrated development environment that makes it user to develop software using the S1C17 Family C Compiler Package (S5U1C17001C). This **IDE** combines the Eclipse IDE for C/C++ Developers package with the functions required for S1C17 program development (simply “GNU17-specific plug-ins” hereinafter). For detailed information on Eclipse standard functions, refer to the general publications that describe the Eclipse IDE for C/C++ Developers Package. The main features of the **IDE** are outlined below.

- **Project creation and management**
The **IDE** lets users create GNU17 projects (GNU17-specific plug-in function) and collectively manage all source files needed to create an application as a single project.
- **Importing GNU17 version 2 projects** (GNU17-specific plug-in function)
The **IDE** lets users import projects created using GNU17 version 2 (e.g., GNU17 V2.3.0).
- **Project settings**
The **IDE** lets users set the properties for projects required to build S1C17 programs (GNU17-specific plug-in function).
- **Supports GNU-compliant C and assembler**
The **IDE** lets users create and edit sources in GNU-compliant C or assembly language. User can also load source code written in other editors into the **IDE**.
- **Program building function**
The **IDE** lets users build applications (*.elf/*.psa) by implementing a sequence from compiling to linker in accordance with the project settings.
- **Launcher for calling the gdb debugger**
After a build process, the user can call the **gdb** debugger to debug a built application.

3.1.2 Some Notes on Use of the IDE

● About the guaranteed operation of the IDE

The **IDE** is designed to run on the Eclipse development platform and uses Eclipse facilities during development work. Note that the facilities not described in this manual lie beyond the scope of guarantee for the **IDE**.

● Eclipse plug-in versions

Listed below are the Eclipse plug-ins and versions that form the base of the **IDE**:

Table 3.1.2.1 Eclipse plug-in versions

Plug-in	Version
Eclipse Platform	4.4.1
Eclipse CDT	8.5.0

The GNU17-specific plug-ins are provided on the assumption that they will operate on these Eclipse plug-ins.

● About the use of Japanese language in the IDE

Although the **IDE** permits Japanese (using Shift-JIS/MS-932 character code) file and directory names and strings, the GNU17 tools used to build projects do not support the Japanese language. Do not use the Japanese language for file and directory names or in executable source code.

(Comments in the source code may be written in Japanese.)

3.2 Starting and Quitting the IDE

3.2.1 Starting the IDE


The method for starting the **IDE** is described below.

- (1) Double-click the **eclipse.exe** icon in the C:\EPSON\GNU17V3\eclipse directory to start the **IDE**.
You can also start the **IDE** by selecting [EPSON MCU] > [GNU17V3] > [GNU17V3 IDE] from the Windows Start menu, or from the command line without parameters.
- (2) After the Eclipse splash screen, the [Workspace Launcher] dialog box shown below is displayed. Here, specify the working directory (workspace) in which you want to store projects and associated files. You can select any other directory or create a new directory and set it as the workspace. Select [Switch Workspace] from the [File] menu to change the workspace. This can be done even after launching.

* Do not specify the project directory (directory containing .project file) as a workspace directory. Doing so may result in failures with project imports (when [Copy projects into workspace] is selected). The current workspace directory can be checked by selecting [File] > [Switch workspace] > [other...] and opening the [Workspace Launcher] dialog box.
- (3) Click the [OK] button.
Launches the **IDE**.

3.2.2 Quitting the IDE

Select [Exit] from the [File] menu to close the **IDE**.

If any open files in the editor have not been saved, you will be prompted to save or discard your changes. Select [Yes] or [No] before quitting the **IDE**. You also can use the  (Close) button to quit the **IDE**. Click the [OK] button at the quit confirmation dialog to quit or [Cancel] to continue working.

3.3 Projects

3.3.1 What Is a Project?

The **IDE** manages individual applications being developed under a project name, creating a directory with the name you specified before beginning to develop an application, managing resources such as source files and files generated by the compiler and other tools in it. In addition, project management files (`.cproject` and `.project`) are generated in a project directory and are updated from time to time by the **IDE**.

Note: These project management files which reside in the project directory must not be edited, moved, or deleted except when you manipulate them in the IDE. Attempting to do so will prevent you from restarting the project.

3.3.2 Creating a New Project

Application development by the **IDE** starts with creating a new project. The procedure is given below.

- (1) Launch the [GNU17 Project] wizard by one of the following methods.
 - Select [New] > [GNU17 Project] from the [File] menu.
 - Select [GNU17 Project] from the [New] shortcut in the toolbar.
 - Select [GNU17 Project] from the [New C/C++ Project] shortcut in the toolbar.
 - Select [New] > [GNU17 Project] from the context menu for the [Projects Explorer] view.
- (2) After launching the wizard, enter a project name in the [Project name:] text box.
 - Only single-byte alphanumeric characters and underscores may be used for project names.
- (3) Specify the location at which you want to create a project directory. (This is necessary if you want to specify a specific location.)
 With default settings, the [Use default location] check box is selected, and a project directory is generated in the workspace directory specified when you started the **IDE**. Normally, go to the next step directly.
- (4) On the GNU17 setup screen, enter the information required to build an S1C17 program.
- (5) From the [Program Type] combo box, select the program to be generated.
 Application(.elf/.psa): Executable program
 Library(.a): Library file
- (6) From the [Target CPU] combo box, select the target CPU, also known as the target processor model. (If Program Type is Application).
 If you do not find the intended target CPU in the list, obtain the model-specific information file (gnu17_mcu_model_xxx.zip) by visiting the Seiko Epson website or contacting the Seiko Epson sales operations.
- (7) From the [Memory Model] combo box, select the memory model of the target.
 REGULAR: 24 bits (Up to 16M-byte space can be used.)
 SMALL: 16 bits (Up to 64K-byte space can be used.)
- (8) Select the GCC version from the [GCC Version] combo box.
 4.9: GCC4.9
 6.4: GCC6.4
- (9) Click the [OK] button.
 A project is created under the name specified.
 Creating a new project creates a directory with the same name as the project in the current workspace or the directory specified in (3). If a directory with this project name already exists, the **IDE** uses it as the project directory.

3.3.3 Creating and Adding a Source File

The **IDE** supports C and assemblers and lets users create objects from source files in these languages. Place the source files required to create objects to the src folder. The files in the src folder will be used for building.

● Creating a source file

The procedure for creating a source file is given below.

(1) Perform one of the following operations:

- Select [New] > [SourceFile] (to create a source file) or [HeaderFile] (to create a header file) from the [File] menu.
- Select [New] > [SourceFile] or [HeaderFile] from the [Project Explorer] view context menu.
- Select [SourceFile] or [HeaderFile] from the [New] shortcut in the toolbar.
- Select [SourceFile] or [HeaderFile] from the [New C/C++ SourceFile] shortcut in the toolbar.
- Click the [New C/C++ SourceFile] button in the toolbar (to create a source file).

The [New Source File] (or [New Header File]) dialog box is displayed.

(2) Enter the name of the file to be created in the [SourceFile:] (or [HeaderFile:]) text box.

Enter the extension “.c” to create a C source file and “.s” to create an assembler source file. A warning message will appear if no corresponding source file extension is entered. However, the file can still be created with the file name entered.

The name of the project folder currently in use appears in the [SourceFolder:] text box. To create a file in another directory, enter the appropriate path or select the desired directory using the [Browse...] button.

(3) Click the [Finish] button or click the [Cancel] button to cancel.

The file created appears in the view, and the editor opens.

If [HeaderFile] is selected from the menu to create a new header file, a macro definition (<file name>_H_) is automatically described within the file.

● Adding a source file

You can add source files to a project by copying them in Windows Explorer and pasting to the project in the [Project Explorer] view.

You can also add source files to a project by importing them as described below:

(1) Click and select the project or folder to which the file is to be imported in the [Project Explorer] view.

(2) Perform one of the following actions:

- Select [import...] from the [File] menu.
- Select [import...] from the [Project Explorer] view context menu and launch the [import] wizard.

(3) Select [File System] from the list. Click the [Next >] button.

(4) Click the [Browse...] button for the [From directory:] text box and select the folder containing the file to be imported from the folder selection dialog box. The path for the folder selected is input in the [From directory:] text box. To select folders from which you previously imported files, use the [From directory:] button to display a history list.

(5) Select the file to be imported.

(6) You can also select and import folders. Importing folders also imports the directory configuration within the folder. (Only files within the selected folder are imported.)

(7) Click [Finish].

(8) The imported file will appear in [Project Explorer] view.

3.3.4 Interrupt Vector and Boot Processing Descriptions

The library crt0.o provides an interrupt vector table and processing from when the target MCU is reset until the main function is called (boot processing).

For detailed information on crt0.o, refer to Section 5.2, “Startup Processing Library”.

● Adding boot processing

Specific processing can be executed on booting by defining the following functions. If they are not defined, the details defined by crt0.o will be enabled.

void _init_device (void)	Initializes the device. This is called before _init_section, _init_lib, _init_sys. In crt0.o, this is defined as a function that does nothing.
void _start_device (void)	Starts device operation. This is called after _init_section, _init_lib, _init_sys, and before main. In crt0.o, this is defined as a function that executes interrupt enable ei.
void _stop_device (void)	Stops device operation. This is called after main. In crt0.o, this is defined as a function that executes interrupt disable di.

● Setting the stack pointer initial value

crt0.o sets the stack pointer initial value in accordance with the __START_stack value. Set the __START_stack value to suit the amount of RAM installed on the target model. If set by a C source file, the __START_stack value should be described as follows somewhere in the C source file:

```
asm(".global __START_stack");
asm(".set __START_stack, 0x7c0"); /* initial value of SP register */
```

If set by a linker symbol file (ldsyms.ini), the __START_stack value should be described as follows:

```
__START_stack = 0x7c0; /* initial value of SP register */
```

In the above example, the value is set to 0x7c0. If no value is set, the value defined by the default linker script will be used.

● Registering an interrupt vector

With crt0.o, the name of the interrupt handler function is fixed at _vectorXX_handler (where XX is a two-digit base-10 vector number). Providing a _vector08_handler function as shown below registers it in the interrupt vector table as an interrupt handler function for vector number 8.

```
void _vector08_handler(void) __attribute__((interrupt_handler));
```

To register the existing interrupt handler function sampleInterrupt to vector number 8, add the following description to the C source file defining sampleInterrupt:

```
void _vector08_handler(void) __attribute__((alias("sampleInterrupt")));
```

This description defines the _vector08_handler with the alias sampleInterrupt. The sampleInterrupt function is an interrupt handler function and must have a prototype declaration as shown below:

```
void sampleInterrupt(void) __attribute__((interrupt_handler));
```

3.3.5 Importing an Existing Project

This section describes how to import an existing project (for example, a sample project provided with this **IDE**) into the current workspace. For detailed information on how to import projects created using GNU17 version 2, refer to Section 3.3.6, “Importing a GNU17 Version 2 Project.”

The import procedure is described below.

(1) Perform one of the operations described below.

- Select [Import...] from the [File] menu.
- Select [Import...] from the context menu for the [Project Explorer] view.

The [Import] wizard will start.

(2) Select [Existing Projects into Workspace] from the list and click the [Next>] button.

(3) Select the [Select root directory] radio button. Then select the project directory you want to import in the directory select dialog box displayed by clicking the [Browse...] button.

(4) If you wish to copy a project to the current workspace, select the [Copy projects into workspace] checkbox in the Options list. After importing, editing operations will be applied to files located in the workspace. If the [Copy projects into workspace] checkbox is unselected, the specified project folder will be used and edited from the workplace.

(5) Click the [Finish] button.

The imported project will be displayed in the [Project Explorer] view.

(6) Select the Target CPU for the imported project.

View GNU17 Setting in the [Properties] dialog to confirm the Target CPU.

You also can import and execute a build process on a project created in another environment (PC) into the current PC by copying it in its entirety, including the project directory. However, if the project is configured with exclusive include search paths and library paths in the build options, you may have to modify these paths after importing the project.

3.3.6 Importing a GNU17 Version 2 Project

All project properties need to be reset if you build a project created using GNU17 Version 2 (such as GNU17 v2.3.0) with this package. This function eliminates the need to reset all properties.

- (1) Select [Import...] from the [File] menu.
Select "GNU17 v2 Project" in the import dialog box to proceed. The "GNU17 v2 Project Import Wizard" will start.
- (2) Select the existing project folder, then select [Finish].
The source file for the existing project is copied to the src folder of the new project.
- (3) Select the Target CPU for the imported project.
View GNU17 Setting in the [Properties] dialog to confirm the Target CPU.
- (4) Rename the interrupt processing function.
Startup Processing Library crt0.o contains an interrupt vector table and processes from when the target MCU is reset until the main function is invoked. As described in Section 3.3.4, "Interrupt Vector and Boot Processing Descriptions," with crt0.o, the name of the interrupt handler function is fixed at `_vectorXX_handler`. Refer to the interrupt vector table in the imported project. Rename the interrupt handler function registered in the interrupt vector; assign a name that corresponds to the interrupt vector number.
- (5) Delete the processing in the existing source file that duplicates crt0.o.
The following processing that duplicates crt0.o should be deleted from the existing source file:
 - Vector table
 - Copying to data section RAM and bss section initialization (`_init_section`)
 - Standard library initialization (`_init_lib` and `_init_sys`)
 The contents of crt0.o can be checked in the utility/lib_src/crt0/crt0.c source code included in this package.
- (6) Manage boot processing.
Manage the boot processing contents defined in the existing source file in accordance with Section 3.3.4, "Interrupt Vector and Boot Processing Descriptions" and define the stack pointer initial value.

If you are not using the startup processing library crt0.o, you can omit steps (4), (5), and (6). Note that you must do the following instead:

- (a) Add a linker script file to the project.
Copy an existing linker script file and add it to the project. Specify the linker script file added in [Other options] for linker options in the project properties.
- (b) Delink crt0.o.
Select the linker option [Do not use standard start files] in the project properties to delink crt0.o.
- (c) Edit the linker script file in accordance with the existing project definition.
Edit the following settings for the existing linker script:
 - Register the reset interrupt handler function for the entry point (ENTRY).
 - Place the interrupt vector table (often contained in the .rodata section of boot.o or vector.o) in the .vector section.
 - Exclude the interrupt vector table placed in the .vector section from the .rodata section.
- (d) Edit the GDB command file.
If gdbsim.ini includes the `c17 ttbr` command, edit this so that it points to the interrupt vector table. For example, if the interrupt vector table is the vector array, write this as `c17 ttbr &vector`.
- (7) Build the project.
Build the project and check the build results in the [Console] view.

3.4 Setting Project Properties

Each project has various properties that can be referenced and configured in the [Properties] dialog box. Setting the project properties lets you specify the options and linker scripts required for building.

Do the following to open the [Properties] dialog box:

- (1) Select a project in the [Project Explorer] view.
Select [Properties] from the [Project] menu or the context menu in the above view.
- (2) This will open the [Properties] dialog box.

3.4.1 Setting GNU17 Project Properties

When building an S1C17 program, switch the startup command options for tools and libraries to be linked based on the processor type and the memory space size of the application system you are developing. You must select the correct processor type and memory model in the project properties to ensure that the IDE switches automatically correctly.

Set the following by selecting GNU17 Setting in the [Properties] dialog box. The target CPU type and memory model are typically selected when creating a new project, so they do not need to be selected subsequently.

	Change location	Setting method	
Target model selection	"GNU17 Setting" Target CPU	Select the name of the S1C17 MCU. The selected information on this item is reflected in the environment variables GCC17_COPRO and GNU17_MODELXXX.	
Memory model selection	"GNU17 Setting" Memory Model	Select one of the memory models indicated below. The selection on this item is reflected in the environment variable GCC17_POINTER.	
		Value	Meaning
		REGULAR	24-bit address mode
		SMALL	16-bit address mode
C compiler selection	"GNU17 Setting" GCC Version	Select one of the versions indicated below. The selection on this item is reflected in the GCC17_LOC environment variable.	
		Value	Meaning
		4.9	GCC 4.9
		6.4	GCC 6.4
Stack pointer initial value setting	"GNU17 Setting" SP register initial value	Set the stack pointer initial value. The value set for this item is reflected in the symbol file as the symbol __START_stack value.	
Flash security setting	"GNU17 Setting" Flash Security Key	Set the following parameters. The settings for this item are reflected in the GNU17_SECURITY_KEY environment variable.	
		Parameter	Meaning
		Version	Flash security version
		Password	Password preset in IC
Flash protection setting	"GNU17 Setting" Flash Protect Bits	Select the corresponding checkboxes to enable or disable flash protection for each area. The selections for this item are reflected in the GNU17_PTD_FILE and GNU17_PTD_OPTION environment variables.	
		State	Meaning
		Read protect : ON	Data reading is prohibited. However, the CPU can execute commands.
		Read protect : OFF	Data reading is allowed.
		Write protect : ON	Data writing is prohibited.
		Write protect : OFF	Data writing is allowed.

3.4.2 Setting Environment Variables

Values are set for the environment variables corresponding to the settings entered in GNU17 Setting properties. These values can be altered directly.

Select C/C++ Build > Environment from the [Properties] dialog box and set the following values:

	Change location	Setting method	
C compiler selection	[Variable] GCC17_LOC	Set the following values:	
		Value	Meaning
		\$(GNU17_LOC)/gcc4	Use GCC4.
		\$(GNU17_LOC)/gcc6	Use GCC6.
Memory model selection	[Variable] GCC17_POINTER	Set the following values:	
		Value	Meaning
		24	REGULAR model (24-bit address mode)
		16	SMALL model (16-bit address mode)
Target model selection	[Variable] GCC17_COPRO	Set the following values:	
		Value	Meaning
			Models without COPRO (S1C17701, etc.)
		\\M	Models with multiplication coprocessor
		\\MD	Models with COPRO
		\\MD2	Models with COPRO2 (S1C17W Series)
Path	[Variable] GCC17_INC	Sets the path for the ANSI library include file. \$(GCC17_LOC)\include	
	GCC17_LIB	Sets the path for the emulation library and ANSI library. \$(GCC17_LOC)\lib\\\$(GCC17_COPRO)\\\$(GCC17_POINTER)bit	
User library link	[Variable] GCC17_USER_LIBS	Registers the name of the user library file to be linked. The library must be located in the \Project\Debug folder. In the order of links, the library link registered in this variable will come after the link for the object generated within the project.	
	GCC17_STARTUP_LIB	Registers the name of the user library file to be linked. The library must be located in the \Project\Debug folder. In the order of links, the library link registered in this variable will come before the link for the object generated within the project.	
Model information	[Variable] GNU17_MODEL	Reflects the values selected in GNU17 Setting > Target CPU.	
	GNU17_MODEL_LOC	Model information path setting \$(GNU17_LOC)\mcu_model	
	GNU17_MODEL_NAME	A value dependent on GNU17_MODEL is set when the GNU17 Setting property is changed. Normally, you should leave this unchanged.	
	GNU17_MODEL_RS	A value dependent on GNU17_MODEL is set when the GNU17 Setting property is changed. Normally, you should leave this unchanged.	
	GNU17_MODEL_SIZE	A value dependent on GNU17_MODEL is set when the GNU17 Setting property is changed. Normally, you should leave this unchanged.	
	GNU17_MODEL_TOP	A value dependent on GNU17_MODEL is set when the GNU17 Setting property is changed. Normally, you should leave this unchanged.	
Flash security setting	[Variable] GNU17_SECURITY_KEY	To use flash security, set the startup option to be handed to winmdc17.	
		Startup option	Meaning
		-s1	Flash security version A value dependent on GNU17_MODEL is set. Normally, you should leave this unchanged.
		-s2	Sets the flash security password.

Flash protection setting	[Variable] GNU17_PTD_OPTION	To use flash protection, set the startup option to be handed to ptd.exe.
	GNU17_PTD_FILE	When using flash protection, set the character string ("_ptd") to be added to the file name of the ROM data (PSA file) protected.
Cygwin setting	CYGWIN	Set "nodosfilewarning" to hide the file path format warning output by Cygwin.

3.4.3 Setting Compiler Path

The compiler path is set as follows:

Select C/C++ Build > Settings > [Tool Settings] > [Cross Settings] > [path] from the [Properties] dialog box and describe the folder using the full path, as shown below:

When using GCC4:	C:\EPSON\GNU17V3\gcc4
When using GCC6:	C:\EPSON\GNU17V3\gcc6

3.4.4 Setting Compiler Options

Set the compiler command options as follows:

Select C/C++ Build > Settings > [Tool Settings] > [Cross GCC Compiler] from the [Properties] dialog box and set the compiler command options. For detailed information on options, refer to Chapter 4, “C Compiler.”

● Symbols

Set compiler macro-definition options from this page.

[Defined symbols (-D)] (default: none)

Specify a macro name and replacement character.

● Includes

Set compiler search path options from this page.

[Include paths (-I)] (default: "\${GCC17_INC}", ../inc)

Set the include file search path.

● Optimization

Select compiler optimization options from this page.

[Optimization Level] (default: -O1)

Select the optimization level.

- -O0: No optimization. For example, select when checking the C program execution line by line.
- -O1: Optimizes code size and execution speed.
- -O2: Optimizes code size and execution speed.(gcc4 only)
- -O3: Optimizes code execution speed than -O1.(gcc4 only)
- -Os: Optimizes code size more extensively than -O1.

The optimization options available will differ for each C compiler. Refer to Section 4.3.2, “Command-line Options” for more information on optimization.

● Miscellaneous

Select other compiler options from this page.

[Other flags] (default: -c -mpointer\${GCC17_POINTER} -B\${GCC17_LOC})

A compiler flag can be added.

3.4.5 Setting Linker Options

Set the linker command options as follows:

Select C/C++ Build > Settings > [Tool Settings] > [Cross GCC Linker] from the [Properties] dialog box and set the linker command options. For detailed information on options, refer to Chapter 7, “Linker.”

● General

Set default library usage from this page.

[Do not use standard start files] (default: Not selected)

crt0.o is not linked if this item is selected.

Select for specific startup processing—for example, if a GNU17 Version2 project has been imported or when describing a program using an assembler only.

[Do not use default libraries] (default: Not selected)

Default libraries are not linked if this item is selected.

libc.a, libgcc.a, and libg.a will be the default libraries. Select this item if you do not wish to link to these libraries.

● Libraries

Set the libraries to be linked and library search paths from this page.

If using your own library, this should be set to the environment variable GCC17_USER_LIBS.

[Libraries]

Specify the library to be linked.

Setting is not required as libc.a/libgcc.a/libg.a is specified beforehand as default linker library.

[Library search path]

Specify the path for searching libraries.

The following path is set by default:

```
${GCC17_LIB}
```

This indicates the location of the emulation library and ANSI library.

● Miscellaneous

Specify other linker options from this page.

[Linker flags] (default: `-gstabs -B${GCC17_LOC} -mrelax`)

Linker flags can be added.

[Other options]

Specify other linker options.

The MAP file generation option is set by default.

```
-Map=${ProjName}.map
```

To specify your own linker script, specify the linker script file using the `-T` option.

If this is not set, linking will use the default linker script.

Example: To specify the linker script file `elf32c17.x` on the project folder

```
-T ../elf32c17.x
```

3.4.6 Setting Assembler Options

Set the assembler command options as follows:

Select C/C++ Build > Settings > [Tool Settings] > [Cross GCC Assembler] from the [Properties] dialog box and set the assembler command options. For more information on options, refer to Chapter 6, “Assembler.”

Note: When the IDE assembles the assembler source, the assembler is launched using `xgcc (-c`

-x assembler-with-cpp option specification) to process using the C preprocessor.

- **General**

[Assembler flags] (default: `-B${GCC17_LOC} -c -mpointer${GCC17_POINTER} -x assembler-with-cpp -Wa,--gstabs`)

Assembler flags can be added.

[Include paths] (default: `../inc`)

Set the search path for the include file.

3.5 Building a Program

Building a program means compiling/assembling the necessary sources and linking the compiled/assembled sources, including libraries, to generate an executable object file. This section describes how to execute a build process.

3.5.1 Editing a Linker Script

A linker script file is used to pass location information on object files comprising the executable file (.elf) to the linker.

This is normally not specified, and the default linker script is used. The default linker script is designed to write a source file in C language and link it to a startup processing library (crt0.o). Thus, the linker script must be created and specified in the following cases that differ from this scenario:

- When altering the basic layout
- When the RAM area is jointly used for multiple variables
- When executing a program on RAM
- When the program entry point (reset exception handler) has been changed from _start
- When the vector table is defined other than crt0.o
- When specifying 3.3 for the GCC Version.

Normally, the linker script must be created and specified if all source files of a project are written in assembly language.

● Creating a new linker script

The following folder contains the default linker script:

\GNU17V3\sample\sample_gcc6\elf32c17.x

Copy this file and save it in the project folder before modifying.

You can also use a text editor to create a new linker script.

● Editing a linker script

Linker script contents can be edited using a text editor. For detailed information on linker script contents, refer to Section 7.4.2, “Examples of Linkage” in Chapter 7, “Linker.”

● Specifying a linker script

Select C/C++ Build > Settings > [Tool Settings] > [Cross GCC Linker] > Miscellaneous from the [Properties] dialog box and specify the linker script.

Specify a linker script using the `-T` option. If no linker script is specified, linking will use the default linker script.

Example: To specify the linker script file elf32c17.x in the project folder

`-T ../elf32c17.x`

3.5.2 Executing a Build Process

After creating source files and setting build options, you can execute a build process. Shown below is the procedure for executing a build process.

● Building all projects in the workspace

Do one of the following to build all projects present in the workspace:

- Select [Build All] from the [Project] menu.
- Click the [Build All] button in the window toolbar.

● Building a selected project

Do the following to build a project individually:

- (1) Select the project you want to build in the [Project Explorer] view.
- (2) Do the following to execute a build process:
 - Select [Build Project] from the [Project] menu.
 - Select [Build Project] from the context menu in the [Project Explorer] view.

● Build process

When you begin building a project, the **IDE** executes the processing described below.

- (1) Save any unsaved files in the editor.
- (2) Execute a build process. The following files will be generated:
 - Object file for each source (*<source name>.o*)
 - Executable format object file (*<project name>.elf*)
 - S-record-format psa file (*<project name>.psa*)
 - PA file as data to be submitted (*<project name>.PA*)
 - Gang Programmer user setting/program data file (*gpdata.bin*)
 - Link map file (*<project name>.map*)

While a build process is underway, the command line in [Console] view shows each tool being executed.

Any errors occurring during a build process can be reviewed in [Problems] view. From there, you can jump the corresponding spot in the editor in error.

3.5.3 Clean and Rebuild

No object files (*.o) are regenerated unless the source or include files have been altered. If all of the generated object files (*.o) are erased, a build (or rebuild) from all sources can be re-executed.

● Clean processing

- (1) Select the project you want to rebuild in the [Project Explorer] view.
- (2) Select [Clean...] from the [Project] menu to open the [Clean] dialog box.
- (3) Select the [Clean projects selected below] radio button and select the project to execute “clean” (rebuild) from the list. Select [Clean all projects] to rebuild all projects in the workspace.
- (4) Click the [OK] button.
This deletes all generated object files in the selected project.

You also can also execute “clean” as described below:

- (1) Select the project you want to execute “clean” in the [Project Explorer] view.
- (2) Select [Clean Project] from the context menu in the [Project Explorer] view.

In this case, no dialog boxes are displayed, and the “clean” process only is executed.

Except when you intend to rebuild a project, you will need to execute a build process after altering certain source files or header files. You must perform a rebuild in the following cases:

- When project properties have been changed
- When the included header file has been changed

3.5.4 Static Stack Usage Analysis Function

C17StackCounter statically analyzes the size of the memory areas used by the stack. The result of the analysis can be displayed in [Console] view when building a project.

● Setting

- (1) Select a project, and select [Properties] from the [Project] menu.
- (2) Select [C/C++ Build] > [Settings] > [Build Steps] tab.
Add the following description to the [Command:] box in the post-build steps:
* Do not delete existing settings when adding the description.

```
;"${GNU17_LOC}\utility\stack\C17StackCounter" ${ProjName}.elf
```

● Output

When you execute the build process after entering the above setting, the execution results will be displayed in [Console] view. The value in [Estimation of Total Stack Size is] indicates the stack size (in bytes).
An example is shown below.

```
"C:\EPSON\GNU17V3\utility\stack\C17StackCounter" sample_gcc6.elf
```

Stack Size of Normal Call Trees:

```
- _start1 : 64 = 8 + 56
  > main : 56 = 4 + 52
  >> puts : 52 = 20 + 32
  >>> putc : 32 = 8 + 24
  >>>> write : 24
- read : 12
- _crt0_start0 : 0
- emu_copro_process : 0
- _crt0_vector_handler : 0
- _crt0_exit : 0
```

Estimation of Total Stack Size is: 64

```
= Normal Call Tree Max: 64
+ Interrupt Disable Call Tree Max: 0
+ Interrupt Enable Call Tree Sum: 0
```

● Restrictions

Analysis cannot be performed in the following cases:

- The assembler was used to describe a function exit process other than gcc.
- Jump addresses are managed using a table or variables (excluding vector tables).

In the event of analysis failure

A group of functions not linked to [Stack Size of Unsolved Functions and its Callees] are displayed. The following message appears:
Note: This program contains some unsolved calls.

3.6 Debugging the Program

The debugger can be started once an execution file (.elf) has been generated using a build operation and the preparations described in the preceding section have been completed.

The debugger is started via the launch configuration window.

The launch configuration window is used for the various settings to start debugging and for launching the debugger (GDB).

3.6.1 GDB Command File

The commands executed when the debugger starts can be described in the GDB command file in the project.

The project contains the following two types of GDB command files:

gdbsim.ini	Simulator GDB command file
gdbmini2.ini	ICDmini2 (S5U1C17001H2) GDB command file
gdbmini3.ini	ICDmini3 (S5U1C17001H3) GDB command file

Select the GDB command file you want to use in the “Debug configuration” window.

Example: GDB command file [gdbmini3.ini]

```
c17 model_path c:/EPSON/GNU17V3/mcu_model
c17 model 17W23
target icd icdmini3
load
# Please uncomment following commented out lines to enable STDOUT while debugging.
# c17 stdout 1 WRITE_FLASH WRITE_BUF
# Please uncomment following commented out lines to enable STDIN while debugging.
# c17 stdin 1 READ_FLASH READ_BUF
# Please uncomment following commented out lines to enable LCD panel simulator while
debugging.
# c17 lcdsim on
```

If you use the ICDmini for debugging, edit the gdbmini3.ini GDB command file corresponding to the ICDmini in the project, and specify the target model.

With the c17 model_path command, specify the MCU model information folder. If GNU17 is installed in c:\EPSON\GNU17V3, specify as follows:

```
c17 model_path c:/EPSON/GNU17V3/mcu_model
```

With the c17 model command, specify the target model. If the target model is S1C17W23, specify as follows:

```
c17 model 17W23
```

The settings above are not required if gdbsim.ini is selected and the simulator used for debugging.

If flash security is set for the target MCU, the flash security password must be unlocked by connecting to the target MCU using the target command and then executing the c17 pwul command before executing the load command. If the flash security version is M03 and the password set is "ABCD1234", specify as follows:

```
C17 pwul M03 ABCD1234
```

Edit the file and add any other commands you wish to execute when the debugger starts.

For detailed information on debugger commands, refer to Chapter 8, “Debugger.”

3.6.2 Setting Standard Input/Output

Character strings can be output to the GDB console window using input/output functions from within the program. Character strings can be input using the dedicated input window.

This requires the settings given below.

- To enable, remove the “#” characters for the lines in red text within the GDB command file.

Example: GDB command file [gdbmini3.ini]

```
# Please uncomment following commented out lines to enable STDOUT while debugging.  
c17 stdout 1 WRITE_FLASH WRITE_BUF  
# Please uncomment following commented out lines to enable STDIN while debugging.  
c17 stdin 1 READ_FLASH READ_BUF
```

Note: Enabling this function occupies one hardware breakpoint.

3.6.3 Using the Debugger

● Starting debugging

The procedure for starting the debugger for the first time with the current project is described below.

- (1) Select [Debug Configurations...] on the [Run] menu to display the launch configuration dialog box. It can also be opened from the [Debug] button menu on the toolbar.
- (2) Select the C/C++ Application type and select the Debug configuration that corresponds to the project name.
- (3) Select the GDB command file within the project using GDB Command file on the Debugger tab.
 gdbsim.ini: When debugging in simulator mode
 gdbmini2.ini: When debugging using the ICDmini2 (S5U1C17001H2)
 gdbmini3.ini: When debugging using the ICDmini3 (S5U1C17001H3)
- (4) Specify the symbol for starting debugging using "Stop on startup at" on the Debugger tab. The main function is specified beforehand to support programs written in C.
- (5) Click the [Debug] button to start debugging.

For details of the Debug configuration, refer to Section 3.6.4, "Setting the Debug Configuration."

● Debugging the program

The program can be executed from the [Run] menu while debugging is in progress.

Operations specific to the GNU17 debugger (gdb c17 commands) can be selected and executed from [Debug Command] on the [C17] menu.

Command	Function
c17 rst	Reset
c17 rstt	Reset target
c17 int	Interrupt
c17 intclear	Clear interrupt
c17 tm	Set trace mode
c17 chgclkmd	DCLK change mode

Note: The c17 ttbr, c17 pwul, c17 model_path, and c17 model commands must be described in the GDB command file executed when the debugger is launched.

The following views can also be used to check the program status during debugging:

View	Function
Breakpoints	Allows the breakpoint settings to be checked and modified for all projects within the workspace.
Console	Allows the following outputs to be checked: <ul style="list-style-type: none"> · Messages output by tools while building is in progress · GDB command file execution results · gdb c17 command execution results · Standard output for programs during execution
Disassembly	Allows reverse assembly of programs to be checked.
EmbSys Registers	Allows peripheral circuit register values to be checked and modified.
Expressions	Allows any monitoring expression (global symbol or register) to be registered and the values checked for all projects within the workspace.
Memory Browser	Allows the memory contents to be checked and modified.
Registers	Allows the CPU register values to be checked and modified.
Variables	Allows the variables to be checked and modified. Local variables are displayed automatically, depending on program execution status. Global variables are displayed if they are registered.

● Quitting the debugger

The debugger can be quit using any of the following methods.

After the debugger terminates, the [Debug] view display changes to the terminated state.

- Select [Terminate] in the [Run] menu.
- Click the [Terminate] button in [Debug] view.
- Click the [Terminate] button in [Console] view.
- Select [Terminate] in the [Debug] view Context menu.

3.6.4 Setting the Debug Configuration

When a GNU17 project of which the Program Type is Application is created, Debug configuration is created at the same time to execute the program generated. To create additional Debug configuration after creating a project, select [New] > [GNU17 Debug Configuration] from the [File] menu and start the [GNU17 Debug Configuration] wizard.

All Debug configurations within the workspace can be set as follows by selecting [Debug configurations] from the [Run] menu and displaying the Debug Configuration dialog.

● Debugger tab

[GDB debugger] (default: `..\gdb`)

Specify the path for the debugger (gdb) used.

[GDB command file] (default: `gdbsim.ini`)

Specify the GDB command file to be executed when the debugger is launched.

● Source tab

[Source Lookup Path]

Specify the search path for the source file used for symbolic debugging. The project folder is the default location of the source file, and the GNU17 library source code is set by default.

3.7 Files Generated in a Project by the IDE

Table 3.7.1 List of files generated by the IDE

Filename	File type	Editing	File management required
.project	IDE project file	×	○
.cproject	IDE project file (CDT)	×	○
<project name> Debug.launch	GDB launch setting file	×	○
gdbmini2.ini	GDB command file (ICDmini2)	○	○
gdbmini3.ini	GDB command file (ICDmini3)	○	○
gdbsim.ini	GDB command file (simulator)	○	○
gpdata.ini	gpdata option setting (Gang Programmer)	○	○
ldsyms.ini	Linker symbol file	○	○
\.settings	Project settings directory	×	○
\Debug \<project name>.elf	Executable file	×	×
\Debug \<project name>.map	Map file	×	×
\Debug \<project name>.psa	ROM data	×	×
\Debug \<project name>_ptd.psa	ROM data with flash protection set	×	×
\Debug \<project name>.sa	S3 format executable file	×	×
\Debug \<project name>.saf	File generated by filling open areas in S3 format executable file with 0xff	×	×
\Debug \<project name>.PA	Data to be submitted	×	×
\Debug\gpdata.bin	Gang Programmer user setting/program data file	×	×
\Debug \<project name>.o	Object file	×	×

The files in the “File management required” column must be managed using a source management application.

4 C Compiler

This chapter explains how to use the **xgcc** C compiler, 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.

4.1 Functions

The **xgcc** C compiler compiles C source files to generate an assembly source file that includes S1C17 Core instruction set mnemonics, extended instructions, and assembler directives. The **xgcc** is a gnu C compiler in conformity with an ANSI standard. Since special syntax is not supported, the programs developed for other types of microcomputers can be transplanted easily to the S1C17 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: **xgcc.exe**, **cpp.exe** and **cc1.exe**.

The **xgcc** is based on the C compiler of Free Software Foundation, Inc. A GCC version 6 C compiler is provided in the gcc6 folder of this package. A GCC version 4 C compiler is located in the gcc4 folder. Details about the license of this compiler are written in the text files "COPYING3" and "COPYING", therefore, be sure to read this file before using the compiler.

4.2 Input/Output File

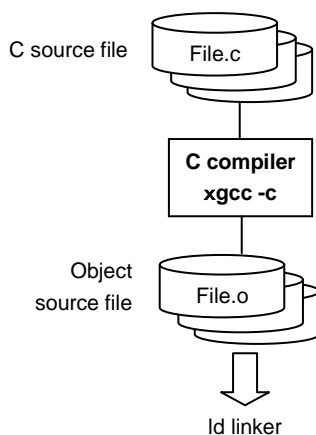


Figure 4.2.1 Flowchart

4.2.1 Input File

● C source file

File format: Text file
 File name: *<filename>.c*
 Description: File in which the C source program is described.

4.2.2 Output Files

● Assembly source file

File format: Text file
 File name: *<filename>.s*
 Description: An assembly source file to be input to the **as** assembler. This file is generated when the **-S** option is specified.

● Object file

File format: Binary file
 File name: `<filename>.o`
 Description: A relocatable object file to be input to the **ld** linker. This file is generated when the `-c` option is specified.

Note: The **xgcc** C compiler generates an elf format executable object file or preprocessed source file according to the option specified.

4.3 Starting Method

4.3.1 Startup Format

To invoke the **xgcc** C compiler, use the command shown below.

xgcc *<options>* *<filename>*

<options> See Section 4.3.2 Command-line Options.
<filename> Specify C source file name(s) including the extension (`.c`).

4.3.2 Command-line Options

The compiler provided in this package formally supports the command line options described below. All other command line options lie beyond the scope of the performance guarantee, and use thereof is solely the user's responsibility.

-c

Function: **Output relocatable object file**
 Description: This option is used to output a relocatable object file (*<input file name>.o*). When this option is specified, the **xgcc** C compiler stops processing after the stage of assembly has finished and does not link. Do not specify the `-S` or `-E` option simultaneously when this option is used.
 Default: The **xgcc** C compiler generates the elf executable object file.

-S

Function: **Output assembly code**
 Description: This option is used to output an assembly source file (*<input file name>.o*). When this option is specified, the **xgcc** C compiler stops processing after the stage of compilation has finished and does not assemble the compiled code. Do not specify the `-c` or `-E` option simultaneously when this option is used.
 Default: The **xgcc** C compiler generates the elf executable object file.

-E

Function: **Execute C preprocessor only**
 Description: When this option is specified, the **xgcc** C compiler stops processing after the stage of preprocessing has finished and does not compile or assemble the preprocessed code. The results are output to the standard output device. Do not specify the `-S` or `-c` option simultaneously when this option is used.
 Default: The **xgcc** C compiler generates the elf executable object file.

-B<directory>

Function: **Specify compiler search path**
 Description: This option is used to add the *<directory>* to the search paths of the **xgcc** C compiler. Input *<directory>* immediately after `-B`. Multiple directories can be specified. In this case, input as many instances of `-B<directory>` as necessary. The sub-programs (**cpp**, **cc1**, etc.) and other data files of the compiler itself are searched in the order they appear in the command line. File search is performed in order of priorities, i.e., current directory, `-B` option, and `PATH` in that order.
 Default: The **xgcc** C compiler searches sub-programs in the current directory and the `PATH` directory.

-I<directory>

Function: **Specify include file directory**

Description: This option is used to specify the directory that contains the files included in the C source.

Input <directory> immediately after -I. Multiple directories can be specified. In this case, input as many instances of -I<directory> 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 option is unnecessary.

File search is performed in order of priorities, i.e., current directory, -I option, and C_INCLUDE_PATH in that order.

Default: The **xgcc** C compiler searches include files in the current directory and the C_INCLUDE_PATH directory.

-D<macro name>[=<replacement character>]

Function: **Define macro name**

Description: 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.

*** About automatic generation of macro names**

The macro names listed below are automatically defined during compilation. These macro names can be referenced from any source file. Note, however, that the same macro names cannot be used for macro definitions in the user program.

Macro name	Contents
__c17	Indicates that the source was compiled for S1C17 processors.
__INT__	Indicates the data size of int type variables (16).
__POINTER24	Indicates that the source was compiled without the -mpointer16 compile option specified.
__POINTER16	Indicates that the source was compiled with the -mpointer16 compile option specified.

-O0, -O1, -O2, -O3, -Os

Function: **Optimization**

Description: Specify one of the switches, then optimize.

The code generated is optimized by prioritizing speed and size. When the -O3 option is specified, optimization prioritizes speed only.

The larger the number following "-O", the stronger the optimization applied. However, keep in mind that large values may generate issues, such as failure to output parts of the debug information.

Reduce this value if the optimization cannot be executed properly. Register interlocks are ignored during optimization. Since the -O3 option is designed to optimize speed, size in certain cases can grow larger than that resulting from the use of the -O option. In ordinary cases, we recommend using -O1 when compiling.

The characteristics of each option are described below.

-O0

No optimization performed.

An area is secured in the stack even if an unused local variable is declared.

Code is compiled unchanged, generating unnecessary code as well, including code that assigns values to local variables that are never referenced. While the values of the variables loaded in registers will not be reused, local variables that are declared as *registered* will be optimized and deleted, as needed.

-O1

Optimizes code size and execution speed.

The optimization performed here includes the following processes:

Unnecessary code is deleted (e.g., code that assigns a value to a never-referenced local variable).

Variable processing is assigned with a register, and the value of this register is reused to reduce memory read/write counts.

However, since this removes guaranteed memory access, variables that require fail-proof read/writes to memory must be declared as `volatile`.

Loop process optimization is performed.

Optimization based on predicting branch conditions prevents repetition of duplicate compare instructions.

-O2 (gcc4 only)

This setting optimizes code size and execution speed more aggressively than the `-O1` option.

-O3 (gcc4 only)

This setting optimizes code execution speed more effectively than the `-O1` option, resulting in the following differences with respect to the `-O1` option:

Common computation processes in the global region are replaced by single computation (common equations in the global region are deleted). Loop process optimization is performed twice.

Register allocation is optimized for operands for simple commands (e.g., `ld`).

Branch condition blocks without attainable destinations are ignored; no code is generated. Functions lacking `inline` declarations are expanded inline. The subroutine of a simple code copies the code of the function itself rather than calling a function, eliminating the overhead associated with a function call.

Depending on the source code, the `-O3` option may not result in the fastest execution speed in certain cases.

-Os

This setting optimizes code size more effectively than the `-O1` option.

Default: Code optimization is performed.

-gstabs/-g

Function: **Add debugging information with relative path to source files**

Description: This option is used to create an output file containing debugging information. The source file location information is output as a relative path.

For `Gcc4`: `-gstabs`

For `Gcc6`: `-g`

Default: Debugging information is output.

-fno-builtin

Function: **Disable built-in functions**

Description: The functions listed below are always called, not compiled as built-in functions. If this option is not specified, the compiler will expand the following functions inline or replace them with other functions to make code generation more efficient, depending on circumstances.

`abort`, `abs`, `cos`, `exit`, `exp`, `fabs`, `fprintf`, `fputs`, `labs`, `log`, `memcpy`, `memmove`, `memset`, `printf`, `putchar`, `puts`, `scanf`, `sin`, `sprint`, `sqrt`, `sscanf`, `strcat`, `strchr`, `strcmp`, `strcpy`, `strcspn`, `strlen`, `strncat`, `strncmp`, `strncpy`, `strpbrk`, `strrchr`, `strspn`, `strstr`, `vprintf`, `vsprintf`

Default: The built-in functions are enabled.

-mpointer16

Function: **Generate code for 16-bit (64KB) data space**

Description: This option is used to generate codes that use 16-bit data pointers (the data space is limited up to 64KB).

This option allows the user program to reduce the RAM size for storing static variable pointers. However, the stack size cannot be reduced by this option.

Default: The C compiler generates the object that allows data to be located in the 24-bit (16MB) space.

-mrelax

Function: **Output code size optimization**

Description: Specifying the `-mrelax` option optimizes output code size by deleting the `ext 0` instruction when linking.

Default: The `ext 0` instruction is not deleted when linking.

-Wall

Function: **Enables warning option**

Description: This function enables all of the following warning options.

These warning options can be individually disabled by adding "-Wno-." For example, to disable just the "-Wcomment" warning, add "-Wno-comment" after "-Wall."

"-Wchar-subscripts"

Outputs a warning when the subscript of an array is of the type "char."

"-Wcomment"

Outputs a warning when "/"* the starting character string for a comment line occurs inside a comment beginning with "/"*.

Also outputs a warning when a comment starting with "/" ends with a backslash.

"-Wformat"

Checks whether the argument is appropriate for a converted character string when the printf or scanf function is invoked. Also checks whether the conversion specified by the converted character string is appropriate.

"-Wimplicit-int"

Outputs a warning if a format is not specified when a variable or function is declared.

"-Wimplicit-function-declaration"

Outputs a warning when a function is used before declaration.

"-Wimplicit"

Same as "-Wimplicit-int" and "-Wimplicit-function-declaration" in enabled state.

"-Wmain"

Outputs a warning when the format of the main function is incorrect.

The main function has an external linkage and the return value is in int format. It should have 0, 2, or 3 arguments of the appropriate format.

"-Wmissing-braces"

Outputs a warning when parentheses are used incorrectly during initialization of arrays. For example, when a multidimensional array is initialized, a warning is output if parentheses are not used correctly for each dimension.

Example: `long l_Array_1[3][3] = { 0, 1, 2, 3, 4, 5, 6, 7, 8 };`

(Warning is output.)

`long l_Array_2[3][3] = { { 0, 1, 2 }, { 3, 4, 5 }, { 6, 7, 8 } };`

(No warning is output.)

"-Wparentheses"

Outputs a warning when omission of parentheses results in ambiguities in the description.

For example, a warning is output if "{ }" are omitted from a nested if statement.

"-Wsequence-point"

Outputs a warning in the case of the standard C language specification if a code described might result in undefined behavior due to the absence of an accurate execution sequence indication.

Example: `i_Array[i_Val++] = i_Val;`

"-Wreturn-type"

Outputs a warning when the return value format is defined as the default "int" format since it is not specified when the function was defined. Also outputs a warning when no value is returned when the return value is a function other than the void type.

"-Wswitch"

Outputs a warning when case statements do not exist for all enum values when the switch statement uses a variable of the enum type for the index. (If a default label exists, this warning is not output.) Also outputs a warning when a case statement specifies a value outside the range of enum type.

"-Wunused-function"

Outputs a warning when a static function is declared but not defined. Also outputs a warning when a static function that is not inline is defined but not used.

"-Wunused-label"

Outputs a warning when a label is declared but not used.

"-Wunused-variable"

Outputs a warning when a static variable other than local variable or const is declared but not used.

"-Wunused-value"

Outputs a warning when a calculation is performed even though the calculation result clearly will not be used.

"-Wunused"

Same as all "-Wunused-xxxx" above in the enabled state.

"-Wuninitialized"

Outputs a warning when a local variable is used without initialization. This warning is not output when `-O0` is selected.

Default: The above warning options are disabled.

-Werror-implicit-function-declaration

Function: **Error output for undeclared functions**

Description: This outputs an error if an undeclared function is used in a C source file.

Default: An error is not output even if an undeclared function is used in a C source file.

-xassembler-with-cpp

Function: **Invoking C preprocessor**

Description: When this option is specified, the **cpp** C preprocessor will be executed before the source is assembled. This allows assembly sources to include preprocessor instructions (`#define`, `#include`, etc.).

Default: The C preprocessor is not invoked.

-Wa, <option>

Function: **Specify an assembler option**

Description: The specified option will be passed to the assembler. To specify two or more options, input as many instances of `-Wa, <option>` as necessary.

Default: No option will be passed to the assembler.

When entering options in the command line, you need to place one or more spaces before and after the option.

Example: `xgcc -c -g test.c`

Note: • Be aware that the compile processing will be unsteady if the same compiler option is specified twice or more with different settings.

- Be sure to specify one of the `-s`, `-E` or `-c` options when invoking `xgcc`. If none is specified, `xgcc` continues processing until the linkage stops. Thus, the necessary linker options must also be specified.

4.4 Compiler Output

This section explains the assembly sources output by the **xgcc** C compiler and the registers used by the **xgcc**.

4.4.1 Output Contents

After compiling C sources, the **xgcc** C compiler outputs the following contents:

- S1C17 Core instruction set mnemonics
- Extended instruction mnemonics
- Assembler directives

All but the basic instructions are output using extended instructions.

Since the system control instructions cannot be expressed in the C source, use in-line assemble by `asm` or an assembly source file to process them.

Example: `asm("halt");`

Assembler directives are output for section and data definitions.

The following describes the sections where instructions and data are set.

Instructions

All instructions are located in the `.text` section.

Constants

Constants are located in the `.rodata` section.

```
Example: const int i=1;      .global    i
                               .section   .rodata
                               .align     2
                               .type      i,@object
                               .size      i,4

                               i:
                               .long      1
```

Global and static variables with initial values

These variables are located in the `.data` section.

```
Example: int i=1;           .global    i
                               .section   .data
                               .align     2
                               .type      i,@object
                               .size      i,4

                               i:
                               .long      1
```

Global and static variables without initial values

These variables are located in the `.bss` section.

```
Example: int i;             .global    i
                               .section   .bss
                               .align     2
                               .type      i,@object
                               .size      i,4

                               i:
                               .zero      4
```

For all symbols including function names and labels, symbol information by the `.stab` assembler directive is inserted (when the `-gstabs/-g` option is specified).

4.4.2 Data Representation

The **xgcc** C compiler supports all data types under ANSI C. Table 4.4.2.1 below lists the size of each type (in bytes) and the effective range of numeric values that can be expressed in each type.

Table 4.4.2.1 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	2	-32768 to 32767
unsigned int	2	0 to 65535
long	4	-2147483648 to 2147483647
unsigned long	4	0 to 4294967295
pointer	4	0 to 16777215
float	4	1.175e-38 to 3.403e+38 (normalized number)
double	8	2.225e-308 to 1.798e+308 (normalized number)
long long	8	-9223372036854775808 to 9223372036854775807
unsigned long long	8	0 to 18446744073709551615
wchar_t	2	0 to 65535

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

Handling of `long long`-type constants requires the suffix `LL` or `ll` (`long long` type) or `ULL` or `ull` (`unsigned long long` type). If this suffix is not present, a warning is generated, since the compiler may not be able to recognize `long long`-type constants as such.

```
Example: long long ll_val;
        ll_val = 0x1234567812345678;
           → warning: integer constant is too large for "long" type
        Ll_val = 0x1234567812345678LL;
           → OK
```

Type `wchar_t` is the data type needed to handle wide characters. This data type is defined in `stdlib.h/stddef.h` as the type `unsigned short`.

● Store positions in memory

The positions in the memory where data is stored depend on the data type and the variable area storing the data. Table 4.4.2.2 below shows the store positions in the variable areas by data type.

Table 4.4.2.2 Data types and store positions

Data type	Global variable area	Local variable area
char	1-byte boundary	
short	2-byte boundary	
int	2-byte boundary	
long	4-byte boundary	
pointer	4-byte boundary	
long long	4-byte boundary	
Structure	When 4-byte boundary type is not contained: 2-byte boundary	When size is 2 bytes or less: 2-byte boundary
	When 4-byte boundary type is contained: 4-byte boundary	When size is 3 bytes or more: 4-byte boundary
Array	When 4-byte boundary type is not contained: 2-byte boundary	When the number of elements is 1: Store position for data type
	When 4-byte boundary type is contained: 4-byte boundary	When the number of elements is 2 or more: 4-byte boundary

4.4.3 Method of Using Registers

The following shows how the **xgcc** C compiler uses general-purpose registers.

Table 4.4.3.1 Method of using general-purpose registers by **xgcc**

Register	Method of use
%r0	Register for passing argument (1st word) Register for storing returned values (8/16-bit data, pointer, 16 low-order bits of 32-bit data)
%r1	Register for passing argument (2nd word) Register for storing returned values (16 high-order bits of 32-bit data)
%r2	Register for passing argument (3rd word)
%r3	Register for passing argument (4th word)
%r4	Registers that need have to their values saved when calling a function
%r5	
%r6	
%r7	

● Registers for passing arguments (%r0 to %r3)

These registers are used to store arguments when calling a function. Arguments exceeding four words are stored in the stack before being passed. They are used as scratch registers before storing arguments.

%r0 ← First argument
 %r1 ← Second argument
 %r2 ← Third argument
 %r3 ← Fourth argument

A pair of the registers is used to store a 32-bit (**long**) argument.

%r1 (high-order 16 bits) and %r0 (low-order 16 bits)

%r3 (high-order 16 bits) and %r2 (low-order 16 bits)

Example:

- First argument: **long**, second argument: **long**
`foo(long lData1, long lData2);`

%r0 ← lData1 (low-order 16 bits)
 %r1 ← lData1 (high-order 16 bits)
 %r2 ← lData2 (low-order 16 bits)
 %r3 ← lData2 (high-order 16 bits)

- First argument: **short**, second argument: **long**
`foo(short sData, long lData);`

%r0	sData (16 bits)
%r1	lData (low-order 16 bits)
%r2	lData (high-order 16 bits)
%r3	Unused

- First argument: **long**, second argument: **short**, third argument: **short**
`foo(long lData, short sData1, short sData2);`

%r0 ← lData (low-order 16 bits)
 %r1 ← lData (high-order 16 bits)
 %r2 ← sData1 (16 bits)
 %r3 ← sData2 (16 bits)

- First argument: long; second argument: pointer; third argument: pointer
`foo(long lData, int *ip_Pt, char *cp_Pt);`

```
%r0 ← lData (lower-order 16 bits)
%r1 ← lData (high-order 16 bits)
%r2 ← ip_Pt (24 bits or 16 bits)
%r3 ← cp_Pt (24 bits or 16 bits)
```

64-bit (long long or double) arguments are stored in the stack before delivery.

If the return value is 64-bit (long long, double) data, a return value area is secured during invocation, and its leading address is placed in %r0 before being passed on to the function.

● Registers for storing returned values (%r0, %r1)

These registers are used to store returned values. They are used as scratch registers before storing a returned value.

- When the returned value is an 8-bit/16-bit data or a pointer (24 bits)

```
%r0 ← Returned value
%r1 ← Unused
```

- When the returned value is a 32-bit data

```
%r0 ← Returned value (low-order 16 bits)
%r1 ← Returned value (high-order 16 bits)
```

● Registers for saving values when calling a function (%r4 to %r7)

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.

4.4.4 Function Call

● The way arguments are passed

When calling a function, up to four arguments are stored in registers for passing argument (%r0 to %r3) while larger arguments are stored in the stack frame of the calling function (explained in the next section) before they are passed.

● Handling of structure arguments

When an argument is a 64-bit or smaller structure, the values of the structure members are stored in the registers to pass through the function if the registers can be used. If the registers for passing argument (%r0 to %r3) cannot be used, the values of the structure members are passed through the stack.

When an argument is a structure larger than 64 bits, the values of the structure members are passed through the stack.

4.4.5 Stack Frame

When calling a function, the **xgcc** C compiler creates the stack frame shown in Figure 4.4.5.1. The start address of the stack frame is always a 32-bit boundary address.

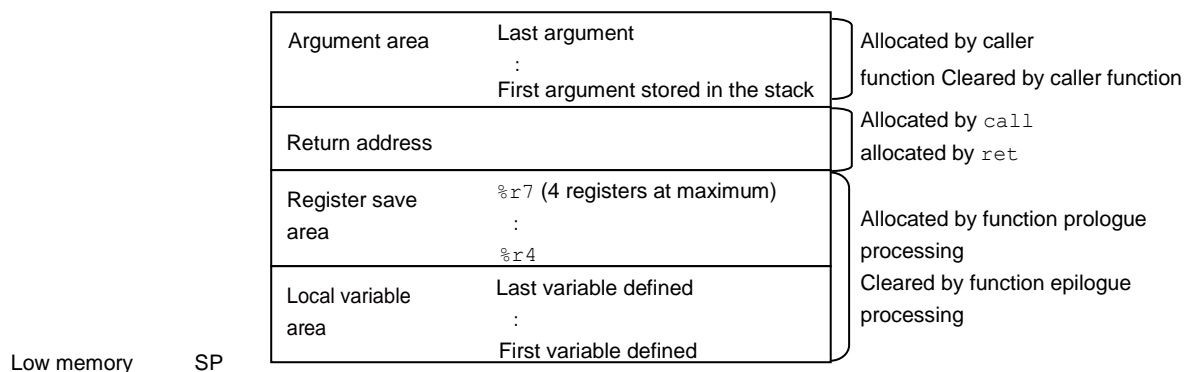


Figure 4.4.5.1 Stack frame

● Argument area

If there are any arguments for function call that cannot be stored in the registers for passing argument, an area is allocated in the stack frame. All arguments are located at 4-byte boundaries.

● Return address

This is the return address to the caller function.

● Register save area

If any registers from %r4 to %r7 are used by the called function, they are saved to this area.

If none of the registers from %r4 to %r7 is used by the called 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.

This area is not allocated if there is no local variable that needs to be saved in the stack.

4.4.6 Grammar of C Source

Refer to Section 2.2, "Grammar of C Source", for data type, library functions and header files, in-line assemble, and prototype declarations (declaring interrupt handler functions).

4.4.7 Compiler Implementation Definition

C language specifications permit implementation-defined adjustment of the method of configuring member variables of a structure or union.

The C compiler in this package is adjusted to yield even-number bytes for the size of a structure or union as an implementation definition feature.

4.5 Correspond to Shift JIS Code

The basic character code set for the GCC is UTF-8. Any SJIS character codes like the one shown below will not be processed correctly.

Example:

If an SJIS character code like “能” (0x945c) exists, the “0x5c” part of the code will be incorrectly interpreted as a line connector (\).

```
i_Val = 0;    // 機能
i_Val = 1;    ← This line is joined to the above line and processed as a comment.
```

To prevent such errors, specify SJIS as character codes handled by the GCC.

- **Method for specifying character code set**

From the [Properties] dialog box, select C/C++ Build > Settings > [Tool Settings] > [Cross GCC Compiler] > [Dialect] > [Other dialect flags] and add **-finput-charset=CP932**.

“-finput-charset” is an option used to specify a character code set. Specify SJIS (CP932).

4.6 Functions of xgcc and Usage Precautions

- For details about the **xgcc** C compiler, refer to the documents for the gnu compiler. The documents can be acquired from the GNU mirror sites located in various places around the world through Internet, etc.
- For detailed information on known issues and limitations concerning the C compiler, refer to: readmeVxxx.txt in this package or the release history (GNU17v3_release_history_e.pdf).
- For information on the C99 standard supported by the C compiler, visit the site below.
<http://gcc.gnu.org/c99status.html>

5 Library

This chapter explains the emulation library and the ANSI library included in the S1C17 Family C Compiler Package.

5.1 Library Overview

Briefly described below are general aspects of the libraries supplied with this package.

5.1.1 Library Files

Libraries are provided for each compiler version.

libc.a	ANSI library Provides ANSI standard functions.
libgcc.a	Emulation library Provides single-precision (32-bit) and double-precision (64-bit) floating-point functions including arithmetic operations, comparison and type conversion, integer multiplication/division/shift functions, and long long-type addition/subtraction/shift functions.
libg.a	Debugging library Provides support functions for the gcc4 debugger.

The libraries for each compiler are installed in one of the following directories based on the coprocessor specifications and memory model:

\EPSON		
\GNU17V3\gccx		
\lib\24bit		24-bit memory model libraries for all models
\lib\16bit		16-bit memory model libraries for all models
\lib\M\24bit		24-bit memory model libraries for models with multiplication coprocessors
\lib\M\16bit		16-bit memory model libraries for models with multiplication coprocessors
\lib\MD\24bit		24-bit memory model libraries for models with COPRO
\lib\MD\16bit		16-bit memory model libraries for models with COPRO
\lib\MD2\24bit		24-bit memory model libraries for models with COPRO2
\lib\MD2\16bit		16-bit memory model libraries for models with COPRO2

Link the 16-bit libraries with the application program when the `-mpointer16` option is specified in the C compiler and assembler.

5.1.2 Precautions to Be Taken When Adding a Library

There is a dependency relationship between the libraries.

When writing a library to a link, specify the libraries in the following sequence:

1. Additional libraries
2. `libc.a`
3. `libgcc.a`
4. `libc.a` (Duplication with 2 does not cause an error. Both files can be referenced normally.)

The object file (or library) can reference only the files present after it, in the order in which they are passed to the linker. If the added library is specified last, none of the external libraries can be used in the added library. Because the basic functions such as `float` and `double` arithmetic and the ANSI library cannot be used, always make sure the added library is located before the emulation and ANSI libraries.

Example: 1. NG

```
ld.exe -T withmylib.x -o withmylib.elf boot.o libc.a libgcc.a libc.a mylib.a
```

If `mylib.a` is using the emulation and ANSI libraries, an error should always occur during linking.

2. OK

```
ld.exe -T withmylib.x -o withmylib.elf boot.o mylib.a libc.a libgcc.a libc.a
```

No errors should occur during linking, allowing `mylib.a` to use the emulation and ANSI libraries normally.

If the added libraries have a dependent relationship, make sure the basic library is located last.

Example: `lib1.a` calls only the emulation and ANSI libraries

`lib2.a` calls `lib1.a` in addition to the emulation and ANSI libraries

`lib3.a` calls `lib1.a` and `lib2.a` in addition to the emulation and ANSI libraries

```
ld.exe -T withmylib.x -o withmylib.elf boot.o lib3.a lib2.a lib1.a libc.a libgcc.a libc.a
```

5.2 Startup Processing Library

5.2.1 Overview

The crt0.o library provides the processing from when the target MCU is reset until the main function is called, together with the vector tables. For information on the crt0.o processing details, refer to the source code utility/lib_src/crt0/crt0.c included in this package.

5.2.2 Vector Tables

The vector tables are registered in crt0.o, as shown below:

```
void * const _vector[] = {
    VECTOR(_start),
    VECTOR(_vector01_handler),
    VECTOR(_vector02_handler),
    VECTOR(emu_copro_process),
    VECTOR(_vector04_handler),
    VECTOR(_vector05_handler),
    VECTOR(_vector06_handler),
    VECTOR(_vector07_handler),
    :
    VECTOR(_vector30_handler),
    VECTOR(_vector31_handler),
};
```

crt0.o always uses the name _vectorXX_handler (where "XX" is a 2-digit base-10 vector number) for the interrupt handler function registered in vector tables. Interrupt handler functions should be declared as shown below in accordance with "2.2 Grammar of C Source".

Example: Declaring an interrupt handler function for vector number 8

```
void _vector08_handler (void) __attribute__((interrupt_handler));
```

Use the following declaration to register an interrupt handler function defined with a different name to the crt0.o vector table:

Example: Registering interrupt handler function sampleInterrupt to vector number 8

```
void sampleInterrupt (void) __attribute__((interrupt_handler));
void _vector08_handler(void) __attribute__((alias("sampleInterrupt")));
```

5.2.3 Stack Pointer Initial Values

crt0.o sets the symbol __START_stack to the stack pointer initial value immediately after the program starts. With this package configuration, the __START_stack value is normally defined by the linker script.

To alter the stack pointer initial value, either change the linker script definition or set using a C source file or linker symbol file as follows:

Example: To set the stack pointer initial value to 0x700 at a location on the C source file

```
asm (".global __START_stack");
asm (".set __START_stack, 0x700"); /* initial value of SP register */
```

Example: To set the stack pointer initial value to 0x700 using a linker symbol file

```
__START_stack = 0x700; /* initial value of SP register */
```

Set the stack pointer initial value to a value appropriate for the target model of the program. The linker script definition will be used if this is not set. The default linker script definition will be used if no linker script is specified.

5.2.4 Startup Processing

crt0.o calls the following functions in sequence. Startup processing can be altered by defining these functions individually. If they are not individually defined, the default operations will be executed by crt0.o. The following table gives an overview of the operations defined. For specific mounting details, refer to the source code utility/lib_src/crt0/crt0.c bundled with this package.

Table 5.2.4.1 Startup processing functions

Execution order	Startup processing function	Default operation
1	void _init_device (void)	Do nothing.
2	void _init_section (void)	Copy data section to RAM and initialize bss section.
3	void _init_lib (void)	Initialize the standard library.
4	void _init_sys (void)	Initialize the standard library (input/output).
5	void _start_device (void)	Permit interrupts using ei instruction.
6	int main (void)	(None)
7	void _stop_device (void)	Prevent interrupts using di instruction.
8	void _exit (int)	The argument is the value returned by the main function. Continue as an infinite loop.

5.3 Emulation Library

5.3.1 Overview

The S1C17 Family C Compiler Package includes the emulation library `libgcc.a` that supports the arithmetic operation, comparison, and type conversion of single-precision (32-bit) and double-precision (64-bit) floating-point numbers that conform to IEEE format, integer multiplication/division/shift operations, and `long long`-type addition/subtraction. The **xgcc** C compiler calls up functions from this library when a floating-point number, `long long` data or integer calculation is performed. Since library functions exchange data via a designated general-purpose register/stack, they can be called from an assembly source. To use emulation library functions, specify `libgcc.a` and `libc.a` when linking.

Registers used in the libraries

The registers `%r0` to `%r7` are used.

The registers `%r4` to `%r7` are protected by saving to the stack before execution of a function and by restoring from the stack after completion of the function.

5.3.2 Floating-point Calculation Functions

● Function list

Table 5.3.2.1 below lists the floating-point calculation functions.

Table 5.3.2.1 Floating-point calculation functions

Classification	Function name	Functionality	
Double-precision floating-point calculation	<code>__adddf3</code>	Addition	$x \leftarrow a + b$
	<code>__subdf3</code>	Subtraction	$x \leftarrow a - b$
	<code>__muldf3</code>	Multiplication	$x \leftarrow a * b$
	<code>__divdf3</code>	Division	$x \leftarrow a / b$
	<code>__negdf2</code>	Sign inversion	$x \leftarrow -a$
Single-precision floating-point calculation	<code>__addsf3</code>	Addition	$x \leftarrow a + b$
	<code>__subsf3</code>	Subtraction	$x \leftarrow a - b$
	<code>__mulsf3</code>	Multiplication	$x \leftarrow a * b$
	<code>__divsf3</code>	Division	$x \leftarrow a / b$
	<code>__negsf2</code>	Sign inversion	$x \leftarrow -a$
Type conversion	<code>__fixunsdfsi</code>	double \rightarrow unsigned int	$x \leftarrow a$
	<code>__fixdfsi</code>	double \rightarrow int	$x \leftarrow a$
	<code>__floatsidf</code>	int \rightarrow double	$x \leftarrow a$
	<code>__fixunssfsi</code>	float \rightarrow unsigned int	$x \leftarrow a$
	<code>__fixsfsi</code>	float \rightarrow int	$x \leftarrow a$
	<code>__floatsisf</code>	int \rightarrow float	$x \leftarrow a$
	<code>__truncdfsf2</code>	double \rightarrow float	$x \leftarrow a$
	<code>__extendsfdf2</code>	float \rightarrow double	$x \leftarrow a$
Double-precision floating-point comparison	<code>__fcmpd</code>	Comparison of double type	PSR change $\leftarrow a - b$
	<code>__eqdf2</code>	Comparison of double type ($a=b$)	PSR change $\leftarrow a - b, x \leftarrow 0 \mid 1^*$
	<code>__nedf2</code>	Comparison of double type ($a \neq b$)	PSR change $\leftarrow a - b, x \leftarrow 1 \mid 0^*$
	<code>__gtdf2</code>	Comparison of double type ($a > b$)	PSR change $\leftarrow a - b, x \leftarrow 1 \mid 0^*$
	<code>__gedf2</code>	Comparison of double type ($a \geq b$)	PSR change $\leftarrow a - b, x \leftarrow 0 \mid -1^*$
	<code>__ltdf2</code>	Comparison of double type ($a < b$)	PSR change $\leftarrow a - b, x \leftarrow -1 \mid 0^*$
	<code>__ledf2</code>	Comparison of double type ($a \leq b$)	PSR change $\leftarrow a - b, x \leftarrow 0 \mid 1^*$
Single-precision floating-point comparison	<code>__fcmps</code>	Comparison of float type	PSR change $\leftarrow a - b$
	<code>__eqsf2</code>	Comparison of float type ($a=b$)	PSR change $\leftarrow a - b, x \leftarrow 0 \mid 1^*$
	<code>__nesf2</code>	Comparison of float type ($a \neq b$)	PSR change $\leftarrow a - b, x \leftarrow 1 \mid 0^*$
	<code>__gtsf2</code>	Comparison of float type ($a > b$)	PSR change $\leftarrow a - b, x \leftarrow 1 \mid 0^*$
	<code>__gesf2</code>	Comparison of float type ($a \geq b$)	PSR change $\leftarrow a - b, x \leftarrow 0 \mid -1^*$
	<code>__ltsf2</code>	Comparison of float type ($a < b$)	PSR change $\leftarrow a - b, x \leftarrow -1 \mid 0^*$
	<code>__lesf2</code>	Comparison of float type ($a \leq b$)	PSR change $\leftarrow a - b, x \leftarrow 0 \mid 1^*$

* x = the value at left if true, x = the value at right if false

- 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* (a-b), 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

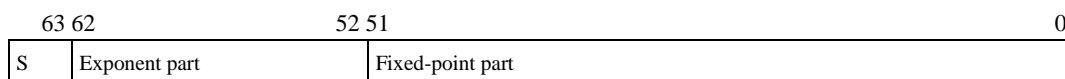
● Floating-point format

The **xgcc** C compiler supports the `float` type (32-bit single-precision) and the `double` type (64-bit double-precision) 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)

The result of a floating-point calculation is stored in the 64-bit area beginning with the address loaded in the `%r0` register.

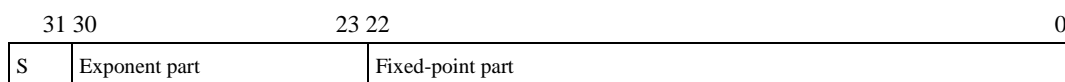
The following shows the relationship of the effective range, floating-point representation, and internal data of the `double` type.

+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)

This type of value occupies two registers. For example, the result of a floating-point calculation is stored in the `%r1` and `%r0` registers.

`%r1` register: Sign bit, exponent part, and 7 high-order bits of fixed-point part (22:16)

`%r0` register: 16 low-order bits of fixed-point part (15:0)

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 **xgcc** C compiler differ from the IEEE-based FPU in precision and functionality, including the manner in which infinity is handled.

5.3.3 Floating-point Number Processing Implementation Definition

The following processes are implementation-defined due to C language specifications. The package emulation library handles them as described below.

Floating-point value rounding method

In type conversion from integer type to floating-point type or from one floating-point type to another floatingpoint type or in floating-point calculations, if the target value is halfway between two adjacent values expressible by the intended format, whether the result is rounded to the larger value or smaller value is implementation-defined.

This package is designed to round values to yield even numbers.

In other words, if the LSB of the value before rounding is 0, no rounding is performed. If the LSB is 1, the value is rounded up by 1.

Conversion from floating-point type to integer type

In converting a floating-point number to an integral number, the fractional part will be truncated.

Following truncation, the action taken if the original value cannot be expressed in the intended format is implementation-defined.

- Conversion from single-/double-precision floating-point type to signed/unsigned long
 - If the original value is +NaN → 0x0 if the intended format is signed or 0x80000000 if unsigned
 - If the original value is -NaN → 0x0
 - If the original value is too large → Maximum value that can be expressed in the intended format
 - If the original value is too small → 0x80000000
- Conversion from single-/double-precision floating-point type to signed/unsigned long long
 - If the original value is +NaN → 0x80000000 80000000
 - If the original value is -NaN → 0x7fffffff 80000000 if the intended format is signed or 0x0 if unsigned
 - If the original value is too large → 0xffffffff ffffffff
 - If the original value is too small → 0x1 if the intended format is signed or 0x0 if unsigned

Conversion from one floating-point type to another floating-point type

If the original value cannot be expressed in the intended format in the conversion from one floating-point type to another floating-point type, the action taken is implementation-defined.

- Conversion from double-precision floating-point type to single-precision floating-point type
 - If the original value is +NaN → The significand of the double-precision floating-point number is shifted two bits to the left and the higher-order 32 bits obtained. If the lower-order 32 bits of the truncated significand are not 0x0, the LSB of the 32-bit significand is set to 1, and the logical sum of that value and 0x7f900000 is used (+NaN).
 - If the original value is -NaN → The significand of the double-precision floating-point number is shifted two bits to the left and the higher-order 32 bits obtained. If the lower-order 32 bits of the truncated significand are not 0x0, the LSB of the 32-bit significand is set to 1, and the logical sum of that value and 0xff900000 is used (-NaN).
 - If the original value is too large → 0x7f800000 (+∞)
 - If the original value is too small → 0xff800000 (-∞)
 - If the original value is too close to 0 (larger than 0) → 0x00000000 (+0)
 - If the original value is too close to 0 (less than 0) → 0x80000000 (-0)
- Conversion from single-precision floating-point type to double-precision floating-point type
 - If the original value is +NaN → The significand of the single-precision floating-point number is shifted two bits to the right, and the logical sum of that value and 0x7ff80000 00000000 is used.
 - If the original value is -NaN → The significand of the single-precision floating-point number is shifted two bits to the right, and the logical sum of that value and 0xfff80000 00000000 is used.

5.3.4 Integral Calculation Functions

Table 5.3.4.1 below lists the integral calculation functions.

Table 5.3.4.1 Integral calculation functions

Classification	Function name	Functionality	
Integral calculation	<code>__divsi3</code>	Signed 32-bit integral division	$x \leftarrow a / b$
	<code>__modsi3</code>	Signed 32-bit remainder calculation	$x \leftarrow a \% b$
	<code>__udivsi3</code>	Unsigned 32-bit integral division	$x \leftarrow a / b$
	<code>__umodsi3</code>	Unsigned 32-bit remainder calculation	$x \leftarrow a \% b$
	<code>__mulsi3</code>	32-bit multiplication	$x \leftarrow a * b$
	<code>__divhi3</code>	Signed 16-bit integral division	$x \leftarrow a / b$
	<code>__modhi3</code>	Signed 16-bit remainder calculation	$x \leftarrow a \% b$
	<code>__udivhi3</code>	Unsigned 16-bit integral division	$x \leftarrow a / b$
	<code>__umodhi3</code>	Unsigned 16-bit remainder calculation	$x \leftarrow a \% b$
	<code>__mulhi3</code>	16-bit multiplication	$x \leftarrow a / b$
Integral shift	<code>__ashlsi3</code>	32-bit arithmetical shift to left	$x \leftarrow a \gg b \text{ bits}$
	<code>__ashrsi3</code>	32-bit arithmetical shift to right	$x \leftarrow a \ll b \text{ bits}$
	<code>__lshrsi3</code>	32-bit logical shift to right	$x \leftarrow a \ll b \text{ bits}$
	<code>__ashlhi3</code>	16-bit arithmetical shift to left	$x \leftarrow a \gg b \text{ bits}$
	<code>__ashrhi3</code>	16-bit arithmetical shift to right	$x \leftarrow a \ll b \text{ bits}$
	<code>__lshrhi3</code>	16-bit logical shift to right	$x \leftarrow a \ll b \text{ bits}$
Integer comparison	<code>__cmpsi2</code>	Comparison (long)	$x \leftarrow 2 1 0 * 1$
	<code>__ucmpsi2</code>	Comparison (unsigned long)	$x \leftarrow 2 1 0 * 1$

5.3.5 long long Type Calculation Functions

Table 5.3.5.1 below lists the long long type calculation functions.

Table 5.3.5.1 long long type calculation functions

Classification	Function name	Functionality	
long long type calculation	<code>__mulldi3</code>	Signed 64-bit multiplication	$x \leftarrow a * b$
	<code>__divldi3</code>	Signed 64-bit division	$x \leftarrow a / b$
	<code>__udivldi3</code>	Unsigned 64-bit division	$x \leftarrow a / b$
	<code>__modldi3</code>	Signed 64-bit remainder calculation	$x \leftarrow a \% b$
	<code>__umodldi3</code>	Unsigned 64-bit remainder calculation	$x \leftarrow a \% b$
	<code>__negdi2</code>	Sign inversion	$x \leftarrow -a$
long long type shift	<code>__lshrdi3</code>	64-bit logical shift to right	$x \leftarrow a \gg b \text{ bits}$
	<code>__ashldi3</code>	64-bit arithmetical shift to left	$x \leftarrow a \ll b \text{ bits}$
	<code>__ashrdi3</code>	64-bit arithmetical shift to right	$x \leftarrow a \gg b \text{ bits}$
Type conversion	<code>__fixunsdfdi</code>	double → unsigned long long	$x \leftarrow a$
	<code>__fixdfdi</code>	double → long long	$x \leftarrow a$
	<code>__floatdidf</code>	long long → double	$x \leftarrow a$
	<code>__fixunssfdi</code>	float → unsigned long long	$x \leftarrow a$
	<code>__fixsfdi</code>	float → long long	$x \leftarrow a$
	<code>__floatdisf</code>	long long → float	$x \leftarrow a$
long long type comparison	<code>__cmpldi2</code>	Comparison (long long)	$x \leftarrow 2 1 0 * 1$
	<code>__ucmpldi2</code>	Comparison (unsigned long long)	$x \leftarrow 2 1 0 * 1$

*1 The integer comparison function and the long long comparison function return the following values based on the result of $op1 - op2$.

$op1 > op2 \rightarrow 2$
 $op1 = op2 \rightarrow 1$
 $op1 < op2 \rightarrow 0$

5.3.6 Compatibility with Coprocessor Instructions

The S1C17 core supports coprocessor instructions.

When using a library compatible with coprocessor instructions, add the "emu_copro_process" function in Vector Table No. 3 as shown below.

Example: Specifying vector tables (vector.c)

```
func *const vector[] = {
    VECTOR(boot),           // 0
    VECTOR(unalign),        // 1
    VECTOR(dummy),          // 2
    VECTOR(emu_copro_process) // 3
};
```

The startup processing library crt0.o already provides the "emu_copro_process" function for No. 3 in the vector table.

Note that some models do not support coprocessor instructions.

The libgcc.a library in the lib/M folder is compatible with coprocessor multiplication instructions. The libgcc.a library in the lib/MD folder is compatible with coprocessor multiplication, division, and remainder calculation instructions. The same applies to the libgcc.a library in the lib/MD2 folder. Note that model compatibility differs.

Table 5.3.6.1 lists functions that use coprocessor instructions in the emulation library. The functions call the "emu_copro_process" when coprocessor instructions are used. Calling the "emu_copro_process" function requires a disable interrupt interval of 15 to 40 cycles.

The specifics of the disable interrupt interval may vary from model to model. Use the "emu_copro_process" function on your model to confirm this.

Table 5.3.6.1 Functions using the coprocessor instructions in the emulation library

Function	Functionality	libgcc.a	M/libgcc.a	MD/libgcc.a	MD2/libgcc.a
__mulhi3	16-bit multiplication	-	✓	✓	✓
__mulsi3	32-bit multiplication	-	✓	✓	✓
__divhi3	Signed 16-bit division	-	-	✓	✓
__modhi3	Signed 16-bit remainder calculation	-	-	✓	✓
__udivhi3	Unsigned 16-bit division	-	-	✓	✓
__umodhi3	Unsigned 16-bit remainder calculation	-	-	✓	✓
__divsi3	Signed 32-bit division	-	-	-	✓
__modsi3	Signed 32-bit remainder calculation	-	-	-	✓
__udivsi3	Unsigned 32-bit division	-	-	-	✓
__umodsi3	Unsigned 32-bit remainder calculation	-	-	-	✓

5.4 ANSI Library

5.4.1 Overview

The S1C17 Family C Compiler Package contains an ANSI library.

Each function in this library has ANSI-standard functionality. Certain ANSI library functions not supported by this package are not included in the ANSI library. The client assumes responsibility for function implementation and prototype declarations when using ANSI library functions not listed in Section 5.4.2, "ANSI Library Function List."

For some ANSI library functions not supported by this package, the header files include only prototype declarations. In these cases, include the pertinent header file rather than declaring a prototype before implementing the function.

See the table in Section 2.2.2, "Library Functions and Header Files" for a discussion of ANSI library functions with prototype declarations only.

The `libc.a` ANSI library file is installed in separate directories (24bit and 16bit) for each memory model.

A long long-type ANSI library is included in `libgcc.a`.

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`
`setjmp.h` `smcvals.h` `stddef.h`

Registers used in the library

- The registers `%r0` to `%r7` are used.
- The registers `%r4` to `%r7` are protected by saving to the stack before execution of a function and by restoring from the stack after completion of the function.

5.4.2 ANSI Library Function List

The contents of the Reentrant column in the tables are as follows:

Reentrant: Reentrant function

Nonreentrant: Non-reentrant function

Conditional: Non-reentrant function (This function refers to a global variable. It can be used as a reentrant function if there is no change in the global variable, and your created `read()` and `write()` are reentrant functions.)

● Input/output functions

The table below lists the input/output functions included in `libc.a`.

Table 5.4.2.1 Input/output functions

Header file: `stdio.h`

Function	Functionality	Reentrant	Notes
<code>size_t fread(void *ptr, size_t size, size_t count, FILE *stream);</code>	Input array element from <code>stdin</code> .	Conditional	Refer to global variables <code>stdin</code> and <code>_iob</code> , and call <code>read</code> function.
<code>size_t fwrite(void *ptr, size_t size, size_t count, FILE *stream);</code>	Output array element to <code>stdout</code> .	Conditional	Refer to global variables <code>stdout</code> , <code>stderr</code> and <code>_iob</code> , and call <code>write</code> function.
<code>int fgetc(FILE *stream);</code>	Input one character from <code>stdin</code> .	Conditional	Refer to global variables <code>stdin</code> and <code>_iob</code> , and call <code>read</code> function.

Function	Functionality	Reentrant	Notes
<code>int getc(FILE *stream);</code>	Input one character from stdin.	Conditional	Refer to global variables <code>stdin</code> and <code>_iob</code> , and call <code>read</code> function.
<code>int getchar(void);</code>	Input one character from stdin.	Conditional	Refer to global variables <code>stdin</code> and <code>_iob</code> , and call <code>read</code> function.
<code>int ungetc(int c, FILE *stream);</code>	Push one character back to input buffer.	Nonreentrant	Refer to global variables <code>stdin</code> , <code>stdout</code> , <code>stderr</code> , and <code>_iob</code> , returned value overwrite.
<code>char *fgets(char *s, int n, FILE *stream);</code>	Input character string from stdin.	Conditional	Refer to global variables <code>stdin</code> and <code>_iob</code> , and call <code>read</code> function.
<code>char *gets(char *s);</code>	Input character string from stdin.	Conditional	Refer to global variables <code>stdin</code> and <code>_iob</code> , and call <code>read</code> function.
<code>int fputc(int c, FILE *stream);</code>	Output one character to stdout.	Conditional	Refer to global variables <code>stdout</code> , <code>stderr</code> and <code>_iob</code> , and call <code>write</code> function.
<code>int putc(int c, FILE *stream);</code>	Output one character to stdout.	Conditional	Refer to global variables <code>stdout</code> , <code>stderr</code> and <code>_iob</code> , and call <code>write</code> function.
<code>int putchar(int c);</code>	Output one character to stdout.	Conditional	Refer to global variables <code>stdout</code> , <code>stderr</code> and <code>_iob</code> , and call <code>write</code> function.
<code>int fputs(char *s, FILE *stream);</code>	Output character string to stdout.	Conditional	Refer to global variables <code>stdout</code> , <code>stderr</code> and <code>_iob</code> , and call <code>write</code> function.
<code>int puts(char *s);</code>	Output character string to stdout.	Conditional	Refer to global variables <code>stdout</code> , <code>stderr</code> and <code>_iob</code> , and call <code>write</code> function.
<code>void perror(const char *s);</code>	Output error information to stdout.	Nonreentrant	Refer to global variables <code>stdout</code> and <code>_iob</code> , change <code>errno</code> , and call <code>read</code> function.
<code>int fscanf(FILE *stream, const char *format, ...);</code>	Input from stdin with format specified.	Nonreentrant	Refer to global variables <code>stdout</code> and <code>_iob</code> , change <code>errno</code> , and call <code>read</code> function.
<code>int scanf(const char *format, ...);</code>	Input from stdin with format specified.	Nonreentrant	Refer to global variables <code>stdout</code> and <code>_iob</code> , change <code>errno</code> , and call <code>read</code> function.
<code>int sscanf(const char *s, const char *format, ...);</code>	Input from character string with format specified.	Nonreentrant	Change global variable <code>errno</code> .
<code>int fprintf(FILE *stream, const char *format, ...);</code>	Output to stdout with format specified.	Conditional	Refer to global variables <code>stdout</code> , <code>stderr</code> and <code>_iob</code> , and call <code>write</code> function.
<code>int printf(const char *format, ...);</code>	Output to stdout with format specified.	Conditional	Refer to global variables <code>stdout</code> , <code>stderr</code> and <code>_iob</code> , and call <code>write</code> function.
<code>int sprintf(char *s, const char *format, ...);</code>	Output to array with format specified.	Reentrant	Call <code>write</code> function.
<code>int vfprintf(FILE *stream, const char *format, va_list arg);</code>	Output conversion result to stdout.	Conditional	Refer to global variables <code>stdout</code> , <code>stderr</code> and <code>_iob</code> , and call <code>write</code> function.
<code>int vprintf(const char *format, va_list arg);</code>	Output conversion result to stdout.	Conditional	Refer to global variables <code>stdout</code> , <code>stderr</code> and <code>_iob</code> , and call <code>write</code> function.
<code>int vsprintf(char *s, const char *format, va_list arg);</code>	Output conversion result to array.	Reentrant	Call <code>write</code> function.

Note: The file system is disabled; `stdin` and `stdout` are enabled. When using `stdin` and `stdout`, the `read()` and `write()` functions are needed, respectively. Refer to Section 5.4.4 for more information.

● Utility functions

The table below lists the utility functions included in `libc.a`.

Table 5.4.2.2 Utility functions

Header file: `stdlib.h`

Function	Functionality	Reentrant	Notes
<code>void *malloc(size_t size);</code>	Allocate area.	Nonreentrant	Change global variables <code>errno</code> , <code>ansi_ucStartAlloc</code> , <code>ansi_ucEndAlloc</code> , <code>ansi_ucNxtAlcP</code> , <code>ansi_ucTblPtr</code> , and <code>ansi_ulRow</code> .
<code>void *calloc(size_t elt_count, size_t elt_size);</code>	Allocate array area.	Nonreentrant	Invalid for call from memory allocate.
<code>void free(void *ptr);</code>	Clear area.	Nonreentrant	Invalid for call from memory allocate.
<code>void *realloc(void *ptr, size_t size);</code>	Change area size.	Nonreentrant	Invalid for call from memory allocate.
<code>void exit(int status);</code>	Terminate program normally.	Reentrant	Refer to <code>exit</code> , terminates on the side of called later.
<code>void abort(void);</code>	Terminate program abnormally.	Reentrant	Refer to <code>exit</code> , terminates on the side of called later.
<code>void *bsearch(const void *key, const void *base, size_t count, size_t size, int (*compare)(const void *, const void *));</code>	Binary search.	Reentrant	
<code>void qsort(void *base, size_t count, size_t size, int (*compare)(const void *, const void *));</code>	Quick sort.	Reentrant	
<code>int abs(int x);</code>	Return absolute value (int type).	Reentrant	
<code>long labs(long x);</code>	Return absolute value (long type).	Reentrant	
<code>div_t div(int n, int d);</code>	Divide int type.	Nonreentrant	Change global variable <code>errno</code> .
<code>ldiv_t ldiv(int n, int d);</code>	Divide long type.	Nonreentrant	Change global variable <code>errno</code> .
<code>int rand(void);</code>	Return pseudo-random number.	Nonreentrant	Change global variable <code>errno</code> .
<code>void srand(unsigned int seed);</code>	Set seed of pseudo-random number.	Nonreentrant	Change global variable <code>errno</code> .
<code>long atol(const char *str);</code>	Convert character string into long type.	Nonreentrant	Change global variable <code>errno</code> .
<code>int atoi(const char *str);</code>	Convert character string into int type.	Nonreentrant	Change global variable <code>errno</code> .
<code>double atof(const char *str);</code>	Convert character string into double type.	Nonreentrant	Change global variable <code>errno</code> .
<code>double strtod(const char *str, char **ptr);</code>	Convert character string into double type.	Nonreentrant	Change global variable <code>errno</code> .
<code>long strtol(const char *str, char **ptr, int base);</code>	Convert character string into long type.	Nonreentrant	Change global variable <code>errno</code> .
<code>unsigned long strtoul(const char *str, char **ptr, int base);</code>	Convert character string into unsigned long type.	Nonreentrant	Change global variable <code>errno</code> .

● Non-local branch functions

The table below lists the non-local branch functions included in `libc.a`.

Table 5.4.2.3 Non-local branch functions

Header file: `setjmp.h`

Function	Functionality	Reentrant	Notes
<code>int setjmp(jmp_buf env);</code>	Non-local branch	Reentrant:	
<code>void longjmp(jmp_buf env, int status);</code>	Non-local branch	Reentrant:	

● Date and time functions

The table below lists the date and time functions included in `libc.a`.

Table 5.4.2.4 Date and time functions

Header file: `time.h`

Function	Functionality	Reentrant	Notes
<code>struct tm *gmtime(const time_t *t);</code>	Convert calendar time to standard time.	Nonreentrant	Change static variable.
<code>time_t mktime(struct tm *tmptr);</code>	Convert standard time to calendar time.	Nonreentrant	Locale information and daylight savings time settings are not applied.
<code>time_t time(time_t *tptr);</code>	Return current calendar time.	Conditional	Refer to global variable <code>gm_sec</code> .

● Mathematical functions

The table below lists the mathematical functions included in `libc.a`.

Table 5.4.2.5 Mathematical functions

Header file: `math.h`, `errno.h`, `float.h`, `limits.h`

Function	Functionality	Reentrant	Notes
<code>double fabs(double x);</code>	Return absolute value (double type).	Reentrant	
<code>double ceil(double x);</code>	Round up double-type decimal part.	Nonreentrant	Change global variable <code>errno</code> .
<code>double floor(double x);</code>	Round down double-type decimal part.	Nonreentrant	Change global variable <code>errno</code> .
<code>double fmod(double x, double y);</code>	Calculate double-type remainder.	Nonreentrant	Change global variable <code>errno</code> .
<code>double exp(double x);</code>	Exponentiate (e^x).	Nonreentrant	Change global variable <code>errno</code> .
<code>double log(double x);</code>	Calculate natural logarithm.	Nonreentrant	Change global variable <code>errno</code> .
<code>double log10(double x);</code>	Calculate common logarithm.	Nonreentrant	Change global variable <code>errno</code> .
<code>double frexp(double x, int *nptr);</code>	Return mantissa and exponent of floating-point number.	Nonreentrant	Change global variable <code>errno</code> .
<code>double ldexp(double x, int n);</code>	Return floating-point number from mantissa and exponent.	Nonreentrant	Change global variable <code>errno</code> .
<code>double modf(double x, double *nptr);</code>	Return integer and decimal parts of floating-point number.	Nonreentrant	Change global variable <code>errno</code> .
<code>double pow(double x, double y);</code>	Calculate x^y .	Nonreentrant	Change global variable <code>errno</code> .
<code>double sqrt(double x);</code>	Calculate square root.	Nonreentrant	Change global variable <code>errno</code> .
<code>double sin(double x);</code>	Calculate sine.	Nonreentrant	Change global variable <code>errno</code> .
<code>double cos(double x);</code>	Calculate cosine.	Nonreentrant	Change global variable <code>errno</code> .
<code>double tan(double x);</code>	Calculate tangent.	Nonreentrant	Change global variable <code>errno</code> .
<code>double asin(double x);</code>	Calculate arcsine.	Nonreentrant	Change global variable <code>errno</code> .
<code>double acos(double x);</code>	Calculate arccosine.	Nonreentrant	Change global variable <code>errno</code> .
<code>double atan(double x);</code>	Calculate arctangent.	Nonreentrant	
<code>double atan2(double y, double x);</code>	Calculate arctangent of y/x .	Nonreentrant	Change global variable <code>errno</code> .
<code>double sinh(double x);</code>	Calculate hyperbolic sine.	Nonreentrant	Change global variable <code>errno</code> .
<code>double cosh(double x);</code>	Calculate hyperbolic cosine.	Nonreentrant	Change global variable <code>errno</code> .
<code>double tanh(double x);</code>	Calculate hyperbolic tangent.	Nonreentrant	

● Character functions

The table below lists the character functions included in `libc.a`.

Table 5.4.2.6 Character functions

Header file: `string.h`

Function	Functionality	Reentrant	Notes
<code>void *memchr(const void *s, int c, size_t n);</code>	Return specified character position in the storage area.	Reentrant	
<code>int memcmp(const void *s1, const void *s2, size_t n);</code>	Compare storage areas.	Reentrant	
<code>void *memcpy(void *s1, const void *s2, size_t n);</code>	Copy storage area.	Reentrant	
<code>void *memmove(void *s1, const void *s2, size_t n);</code>	Copy the storage area (overlapping allowed).	Reentrant	
<code>void *memset(void *s, int c, size_t n);</code>	Set character in the storage area.	Reentrant	
<code>char *strcat(char *s1, const char *s2);</code>	Concatenate character strings.	Reentrant	
<code>char *strchr(const char *s, int c);</code>	Return specified character position found first in the character string.	Reentrant	
<code>int strcmp(const char *s1, const char *s2);</code>	Compare character strings.	Reentrant	
<code>char *strcpy(char *s1, const char *s2);</code>	Copy character string.	Reentrant	
<code>size_t strspn(const char *s1, const char *s2);</code>	Return number of characters from the beginning of the character string until the specified character appears (multiple choices).	Reentrant	
<code>char *strerror(int code);</code>	Return error message character string.	Reentrant	
<code>size_t strlen(const char *s);</code>	Return length of character string.	Reentrant	
<code>char *strncat(char *s1, const char *s2, size_t n);</code>	Concatenate character strings (number of characters specified).	Reentrant	
<code>int strncmp(const char *s1, const char *s2, size_t n);</code>	Compare character strings (number of characters specified).	Reentrant	
<code>char *strncpy(char *s1, const char *s2, size_t n);</code>	Copy character string (number of characters specified).	Reentrant	
<code>char *strpbrk(const char *s1, const char *s2);</code>	Return specified character position (multiple choices) found first in the character string.	Reentrant	
<code>char *strrchr(const char *str, int c);</code>	Return specified character position found last in the character string.	Reentrant	
<code>size_t strcspn(const char *s1, const char *s2);</code>	Return number of characters from the beginning of the character string until the non-specified character appears (multiple choices).	Reentrant	
<code>char *strstr(const char *s1, const char *s2);</code>	Return position where the specified character string appeared first.	Reentrant	
<code>char *strtok(char *s1, const char *s2);</code>	Divide the character string into tokens.	Nonreentrant	Change static variable.

● Character type determination/conversion functions

The table below lists the character type determination/conversion functions included in `libc.a`.

Table 5.4.2.7 Character type determination/conversion functions

Header file: `ctype.h`

Function	Functionality	Reentrant	Notes
<code>int isalnum(int c);</code>	Determine character type (decimal or alphabet).	Reentrant	
<code>int isalpha(int c);</code>	Determine character type (alphabet).	Reentrant	
<code>int iscntrl(int c);</code>	Determine character type (control character).	Reentrant	
<code>int isdigit(int c);</code>	Determine character type (decimal).	Reentrant	
<code>int isgraph(int c);</code>	Determine character type (graphic character).	Reentrant	
<code>int islower(int c);</code>	Determine character type (lowercase alphabet).	Reentrant	
<code>int isprint(int c);</code>	Determine character type (printable character).	Reentrant	
<code>int ispunct(int c);</code>	Determine character type (delimiter).	Reentrant	
<code>int isspace(int c);</code>	Determine character type (null character).	Reentrant	
<code>int isupper(int c);</code>	Determine character type (uppercase alphabet).	Reentrant	
<code>int isxdigit(int c);</code>	Determine character type (hexadecimal).	Reentrant	
<code>int tolower(int c);</code>	Convert character type (uppercase alphabet → lowercase).	Reentrant	
<code>int toupper(int c);</code>	Convert character type (lowercase alphabet → uppercase).	Reentrant	

● Variable argument macros

The table below lists the variable argument macros defined in `stdarg.h`.

Table 5.4.2.8 Variable argument macros

Header file: `stdarg.h`

Macro	Functionality
<code>void va_start(va_list ap, type lastarg);</code>	Initialize the variable argument group.
<code>type va_arg(va_list ap, type);</code>	Return the actual argument.
<code>void va_end(va_list ap);</code>	Return normally from the variable argument function.

5.4.3 Declaring and Initializing Global Variables

The ANSI library functions reference the global variables listed in Table 5.4.3.1. These variables are defined in the `crt0.o` library and are initialized on startup by the `_init_lib()` function also defined in `crt0.o`.

Table 5.4.3.1 Global variables required of declaration

Global variable	Initial setting	Related header file/function
FILE _iob[FOPEN_MAX +1]; FOPEN_MAX=3, defined in <code>stdio.h</code> File structure data for standard input/output streams	<code>_iob[N]._flg = _UGETN;</code> <code>_iob[N]._buf = 0;</code> <code>_iob[N]._fd = N;</code> (N=0-2) <code>_iob[0]: Input data for stdin</code> <code>_iob[1]: Output data for stdout</code> <code>_iob[2]: Output data for stderr</code>	stdio.h, smcvals.h <code>fgets, fread, fscanf,getc, getchar, gets, scanf, ungetc, perror, fprintf, fputs, fwrite, printf, putc, putchar, puts, fprintf, vprintf</code>
FILE *stdin; Pointer to standard input/output file structure data <code>_iob[0]</code>	<code>stdin = &_iob[0];</code>	stdio.h <code>fgets, fread, fscanf,getc, getchar, gets, scanf, ungetc</code>
FILE *stdout; Pointer to standard input/output file structure data <code>_iob[1]</code>	<code>stdout = &_iob[1];</code>	stdio.h <code>fprintf, fputs, fwrite, printf, putc, putchar, puts, fprintf, vprintf</code>
FILE *stderr; Pointer to standard input/output file structure data <code>_iob[2]</code>	<code>stderr = &_iob[2];</code>	stdio.h <code>fprintf, fputs, fwrite, printf, perror, putc, putchar, puts, fprintf, vprintf</code>
int errno; Variable to store error number	<code>errno = 0;</code>	errno.h <code>fopen, freopen, fseek, fsetpos, perror, remove, rename, tmpfile, tmpnam, fprintf, printf, sprintf, vprintf, fprintf, 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</code>
unsigned int seed; Variable to store seed of random number	<code>seed = 1;</code>	stdlib.h <code>rand, srand</code>
time_t gm_sec; Elapsed time of timer function in seconds from 0:00:00 on January 1, 1970	<code>gm_sec = -1;</code>	time.h <code>time</code>

Among the global variables referenced by the ANSI library functions, those that are used by each function (`malloc`, `calloc`, `realloc`, and `free`) are initialized using the initialization function shown below. This function is defined in `stdlib.h`.

```
int ansi_InitMalloc(unsigned long START_ADDRESS, unsigned long END_ADDRESS);
```

For the `START_ADDRESS` and `END_ADDRESS`, set the start and end addresses of the memory used, respectively. These addresses are adjusted to the 4-byte boundaries within the function.

The following global variables are initialized. These variables are defined in `stdlib.h`.

<code>unsigned char *ansi_ucStartAlloc;</code>	Pointer to indicate the start address of the heap
<code>unsigned char *ansi_ucEndAlloc;</code>	Pointer to indicate the end address of the heap area
<code>unsigned char *ansi_ucNxtAlcP;</code>	Address pointer to indicate the beginning of the next new area mapped
<code>unsigned char *ansi_ucTblPtr;</code>	Address pointer to indicate the beginning of the next management area mapped
<code>unsigned long ansi_ulRow;</code>	Line pointer to indicate the next management area mapped

Each time storage is reserved for a heap area, eight-byte heap area management data is written from the ending address (`ansi_ucEndAlloc`) toward the beginning address. Be careful to avoid rewriting areas specified as heap areas by the `ansi_InitMalloc()` function by a stack pointer, etc.

* The `ansi_InitMalloc()` function is not found in the `crt0.o` or `libstdio.a` libraries. Be aware that it must be called from the user routine before calling `malloc()` or a similar function. (A heap area cannot be allocated if the `ansi_InitMalloc()` function is not called.)

5.4.4 Lower-level Functions

The following three functions (`read`, `write`, and `_exit`) are the lower-level functions called by library functions. For these functions, the debugging implementation is defined within the `libg.a` library.

● 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`)

● 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 `putc`)
`printf`, `fprintf`, `sprintf`, `vprintf`, `vfprintf` (calls `_doprint`)
`perror` (calls `fprintf`)

- **_exit function**

Contents of _exit function

Format: **void _exit(void);**

Functionality: Performs program terminating processing.

Argument: None

Library function that calls the _exit function:

Direct call: `abort`, `exit`

6 Assembler

This chapter describes the functions of the **as** assembler. For the syntax of the assembly sources, refer to Section 2.3, "Grammar of Assembly Source".

6.1 Functions

The **as** assembler assembles (translates) assembly source files that are delivered by the C compiler and creates object files in the machine language. It can also deliver debugging information for purposes of symbolic debugging.

This assembler is based on the gnu assembler (as). For details about the **as** assembler, refer to the documents for the gnu assembler. The documents can be acquired from the GNU mirror sites located in various places around the world through Internet, etc.

6.2 Input/Output Files

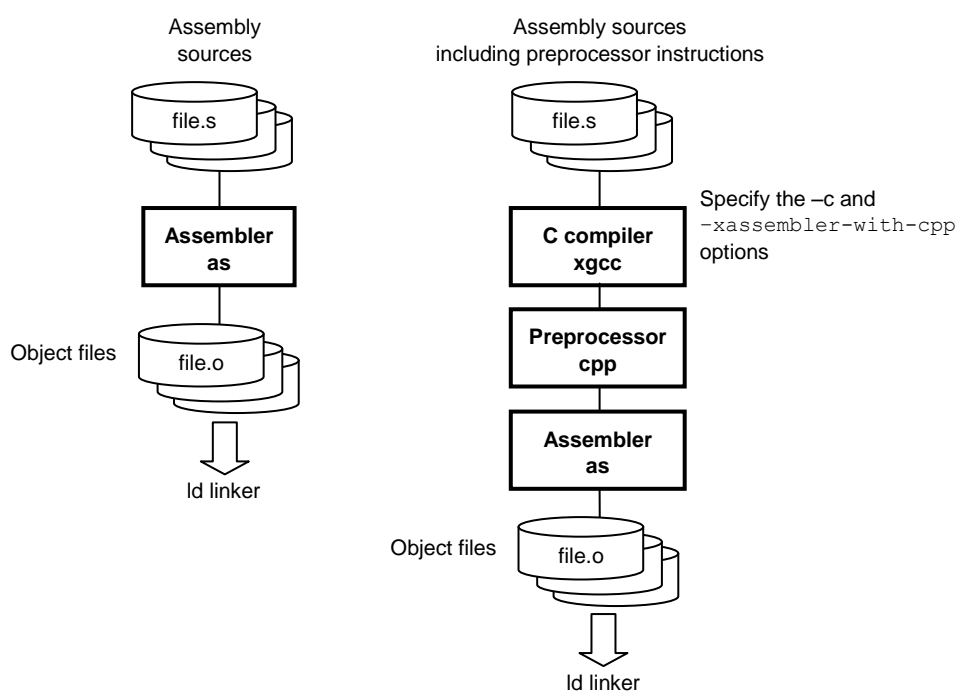


Figure 6.2.1 Flowchart

6.2.1 Input File

● Assembly source file

File format: Text file
File name: *<filename>.s* (Other extenders than ".s" can be used. A path can also be specified.)
Description: File in which basic instructions and assembler directives are described. Usually, a file delivered by the **xgcc** C compiler is input there.
If source files were created that only describe basic instructions and assembler directives, they can be input into the **as** assembler directly.

6.2.2 Output File

● Object file

File format: Binary file in elf format
File name: *<filename>.o* (The *<filename>* is the same as that of the input file.)
Description: File in which symbol information and debugging information are added to the program code (machine language).

6.3 Starting Method

6.3.1 Startup Format

To invoke the **as** assembler, use the command shown below.

```
as <options> <filename>
    <options>    See Section 6.3.2.
    <filename>   Specify assembly source file name(s) including the extension (.s).
```

6.3.2 Command-line Options

The **as** assembler accepts the **gnu** assembler standard options.

The following lists the principal options only. Refer to the **gnu** assembler manual for more information.

```
-o<filename>
    Function:    Specify output file name
    Description: This option is used to specify the name of the object file output by the as assembler.
                  The <filename> must be input immediately after -o.
    Default:     The default output file name is a.out.

-a [<sub-option>]
    Function:    Output assembly list file
    Description: Outputs an assembly list file. The <sub-option> controls the output contents.
                  Example: -adh1 Requests high-level assembly listing without debugging directives.
    Default:     No assembly list file is output.

--gstabs
    Function:    Add debugging information with relative path to source files
    Description: This option is used to creates an output file containing debugging information.
                  The source file location information is output as a relative path.
    Default:     No debugging information is output.
```

In addition to the standard options, the following **S1C17** option is available:

```
-mpointer16
    Function:    Specify 16-bit pointer mode
    Description: This option is used to generate object files for the 16-bit pointer mode (64KB memory model). This option just
                  sets a flag to indicate that the 16-bit pointer mode is specified and it does not affect the object code that will be
                  generated.
    Default:     The assembler generates object files for the 24-bit pointer mode (16MB memory model).
```

When entering options in the command line, you need to place one or more spaces before and after the option.

Example: **as -otest.o -adh1 test.s**

6.4 Scope

Symbols defined in each source file can freely be referred to within that file. Such reference range of symbols is termed scope.

Usually, reference can be made only within a defined file. If a symbol that does not exist in that file is referenced, the **as** assembler creates the object file assuming that the symbol is an undefined symbol, leaving the problem to be solved by the **ld** linker.

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 **as** assembler has the pseudo-instructions that can be used for this purpose.

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 **ld** linker.

Example:

file1: file in which global symbol is defined

```
.global SYMBOL      ...Global declaration of symbols that are to be defined in this file.
.global VAR1

SYMBOL:
:
:

LABEL:              ...Local symbol
                   (Can be referred to only in this file)

.section .bss
.align 2

VAR1:
.zero 4
```

file2: file in which a global symbol is referred

```
xcall SYMBOL      ...Symbol externally referred
:
xld.a %r1,VAR1     ...Symbol externally referred
LABEL:            ...Local symbol
                  (Treated as a different symbol from LABEL of file1)
```

The **as** assembler 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 **ld** linker.

6.5 Assembler Directives

The assembler directives 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 directives begin with a period (.).

Describe the directives in lowercase unless otherwise specified. Parameters are discriminated between uppercase and lowercase.

The **as** assembler supports all the gnu assembler directives. Refer to the gnu assembler manual for details of the assembler directives. The following explains the often-utilized directives.

6.5.1 Text Section Defining Directive (`.text`)

- **Instruction format**

`.text`

- **Description**

Declares the start of a `.text` section. Statements following this instruction are assembled as those to be mapped in the `.text` section, until another section is declared.

6.5.2 Data Section Defining Directives (.rodata, .data)

● List of data section defining directives

- .rodata** Declares a .rodata section in which constants are located.
- .data** Declares a .data section in which data with initial values are located.

● Instruction format

```
.section .rodata
.section .data
```

● Description

(1) .section .rodata

Declares the start of a constant data section. Statements following this instruction are assembled as those to be mapped in the .rodata section, until another section is declared. Usually, this section will be mapped into a read-only memory at the stage of linkage.

Example: .section .rodata Defines a .rodata section.

(2) .section .data

Declares the start of a data section with an initial value. Statements following this instruction are assembled as those to be mapped in the .data section, until another section is declared. Usually, this section will be mapped into a read-only memory at the stage of linkage and data in this section must be copied to a read/ write memory such as a RAM by the software before using.

Example: .section .data Defines a .data section.

● Note

The data space allocated by the data-define directive is as follows:

```
1 byte: .byte
2 bytes: .short, .hword, .word, .int
4 bytes: .long
```

6.5.3 Bss Section Defining Directive (`.bss`)

- **List of data section defining directives**

`.bss` Declares a `.bss` section for data without an initial value.

- **Instruction format**

`.section .bss`

- **Description**

Declares the start of a uninitialized data section. Statements following this instruction are assembled as those to be mapped in the `.bss` section, until another section is declared.

Example: `.section .bss` Defines a `.bss` section.

- **Note**

- The labels described in the `.bss` section will be defined as local symbols by default. To define a global symbol, use the `.global` directive.

Example: `.section .bss`

`.align 2`

`VAR1:`

`.skip 4` Defines the 4-byte local variable `VAR1`.

`.section .bss`

`.global VAR2`

`.align 2`

`VAR2:`

`.skip 4` Defines the 4-byte global variable `VAR2`.

- Areas in `.bss` sections can be secured using the `.skip` directive. The `.space` directive cannot be used because it has an initial data.

6.5.4 Data Defining Directives (`.long`, `.short`, `.byte`, `.ascii`, `.space`)

The following assembler directives are used to define data in `.data` or `.text` sections:

● List of data section defining directives

<code>.long</code>	Define 4-byte data.
<code>.short</code>	Define 2-byte data.
<code>.byte</code>	Define 1-byte data.
<code>.ascii</code>	Define ASCII character strings.
<code>.space</code>	Fills an area with a byte data.

● Instruction format

<code>.long</code>	<code><4-byte data>[, <4-byte data> ... , <4-byte data>]</code>
<code>.short</code>	<code><2-byte data>[, <2-byte data> ... , <2-byte data>]</code>
<code>.byte</code>	<code><1-byte data>[, <1-byte data> ... , <1-byte data>]</code>
<code>.ascii</code>	<code>"<character string>"[, "<character string>" ... , "<character string>"]</code>
<code>.space</code>	<code><length>[, <1-byte data>]</code>

<code><4-byte data></code>	<code>0x0—0xffffffff</code>
<code><2-byte data></code>	<code>0x0—0xffff</code>
<code><1-byte data></code>	<code>0x0—0xff</code>
<code><character string></code>	ASCII character string
<code>< length></code>	Area size to be filled

● Description

(1) `.long`, `.short`, `.byte`

Defines one or more 4-byte data, 2-byte data, or 1-byte data. When specifying two or more data, separate them with a comma. The defined data is located beginning with a boundary address matched to the data size by the data defining directive unless it is immediately preceded by the `.align` directive. If the current position is not a boundary address, 0x00 is set in the interval from that position to the nearest boundary address.

Example: `.long 0x0, 0x1, 0x2`
`.byte 0xff`

In addition to these directives, the directives listed below can also be used.

<code>.hword</code>	same as <code>.short</code>
<code>.word</code>	same as <code>.short</code>
<code>.int</code>	same as <code>.short</code>

(2) `.ascii`

Defines one or more string literals. Enclose a character string in double quotes. 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 quote in a character string, write `\"`; to set a `\`, write `\\`. When specifying two or more strings, separate them with a comma. The defined data is located beginning with the current address first, unless it is immediately preceded by the `.align` directive.

Example: `.ascii "abc", "xyz"`
`.ascii "abc\"D\"efg" (= abc"D"efg)`

(3) `.space`

An area of the specified `<length>` bytes long is set to `<1-byte data>`. The area begins from the current address unless it is immediately preceded by the `.align` directive.

If `<1-byte data>` is omitted, the area is filled with 0x0. To fill the area with 0x0, the `.zero` directive (see the next page) can also be used.

Example: `.space 4, 0xff` Sets 0xff to the 4-byte area beginning from the current address.
`.zero 4` (= `.space 4, 0x0`)

6.5.5 Area Securing Directive (`.zero`)

● Instruction format

`.zero <length>`

`<length>` Area size in bytes

● Description

This directive secures a `<length>` bytes of blank area in the current `.bss` section. The area begins from the current address unless it is immediately preceded by the `.align` directive.

Example:

```
.section .bss
.global VAR1
.align 2
VAR1:
.zero 4           Secures an space for the 4-byte global variable VAR1.
```

6.5.6 Alignment Directive (`.align`)

- **Instruction format**

.align *<alignment>*

<code><alignment></code>	Value to specify a boundary
--------------------------------	-----------------------------

● Description

The data that appears immediately after this directive is aligned to a 2^n byte boundary ($n = \langle alignment \rangle$).

Example: `.align 2` Aligns the following data to a 4-byte boundary.

- **Note**

The `.align` directive is valid for only the immediately following data definition or area securing directive. Therefore, when defining data that requires alignment, you need to use the `.align` directive for each data definition directive.

6.5.7 Global Declaring Directive (`.global`)

- **Instruction format**

`.global <symbol>`

`<symbol>` Symbol to be defined in the current file

- **Description**

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.

Example: `.global SUB1`

- **Note**

The symbols are always defined as a local symbol unless it is declared using this directive.

6.5.8 Symbol Defining Directive (`.set`)

- **Instruction format**

`.set <symbol>, <address>`

`<symbol>` Symbol for memory access (address reference)

`<address>` Absolute address

- **Description**

Defines a symbol with an absolute address (24-bit).

Example: `.set DATA1, 0x80000` Defines the symbol `DATA1` that represents absolute address `0x80000`.

- **Note**

The symbol is defined as a local symbol. To use it as a global symbol, global declaration using the `.global` directive is necessary.

6.6 Extended Instructions

The **as** assembler supports the extended instructions explained below. 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 before assembling.

Symbols used in explanation

<i>immX</i>	Unsigned X-bit immediate
<i>signX</i>	Signed X-bit immediate
<i>symbol</i>	Symbol to indicate memory address
<i>label</i>	Jump address label
<i>(X:Y)</i>	Bit field from bit X to bit Y

6.6.1 Arithmetic Operation Instructions

● Types and functions of extended instructions

Extended instruction	Function	Expansion
<code>sadd %rd, imm16</code>	$\%rd \leftarrow \%rd + imm16$	(1)
<code>sadc %rd, imm16</code>	$\%rd \leftarrow \%rd + imm16 + C$	(1)
<code>sadd.a %rd, imm20</code>	$\%rd \leftarrow \%rd + imm20$	(2)
<code>sadd.a %sp, imm20</code>	$\%sp \leftarrow \%sp + imm20$	(2)
<code>ssub %rd, imm16</code>	$\%rd \leftarrow \%rd - imm16$	(1)
<code>ssbc %rd, imm16</code>	$\%rd \leftarrow \%rd - imm16 - C$	(1)
<code>ssub.a %rd, imm20</code>	$\%rd \leftarrow \%rd - imm20$	(2)
<code>ssub.a %sp, imm20</code>	$\%sp \leftarrow \%sp - imm20$	(2)
<code>xadd %rd, imm16</code>	$\%rd \leftarrow \%rd + imm16$	(1)
<code>xadc %rd, imm16</code>	$\%rd \leftarrow \%rd + imm16 + C$	(1)
<code>xadd.a %rd, imm24</code>	$\%rd \leftarrow \%rd + imm24$	(3)
<code>xadd.a %sp, imm24</code>	$\%sp \leftarrow \%sp + imm24$	(3)
<code>xsub %rd, imm16</code>	$\%rd \leftarrow \%rd - imm16$	(1)
<code>xsbc %rd, imm16</code>	$\%rd \leftarrow \%rd - imm16 - C$	(1)
<code>xsub.a %rd, imm24</code>	$\%rd \leftarrow \%rd - imm24$	(3)
<code>xsub.a %sp, imm24</code>	$\%sp \leftarrow \%sp - imm24$	(3)

These extended instructions allow a 16-bit/20-bit/24-bit immediate to be specified directly in an add or subtract operation. A conditional operation option (`/c`, `/nc`) cannot be specified in the extended instructions.

● Basic instructions after expansion

sadd, xadd	Expanded into the <code>add</code> instruction
sadc, xadc	Expanded into the <code>adc</code> instruction
sadd.a, xadd.a	Expanded into the <code>add.a</code> instruction
ssub, xsub	Expanded into the <code>sub</code> instruction
ssbc, xsbc	Expanded into the <code>sbc</code> instruction
ssub.a, xsub.a	Expanded into the <code>sub.a</code> instruction

● Expansion formats

- (1) **sOP %rd,imm16 / xOP %rd,imm16** (OP = add, adc, sub, sbc)

Example: xadd %rd,imm16

$imm16 \leq 0x7f$	$0x7f < imm16$
add %rd,imm16(6:0)	ext imm16(15:7) add %rd,imm16(6:0)

- (2) **sOP.a %rd,imm20 / sOP.a %sp,imm20** (OP = add, sub)

Example: sadd.a %rd,imm20

$imm20 \leq 0x7f$	$0x7f < imm20$
add.a %rd,imm20(6:0)	ext imm20(19:7) add.a %rd,imm20(6:0)

- (3) **xOP.a %rd,imm24 / xOP.a %sp,imm24** (OP = add, sub)

Example: xadd.a %rd,imm24

$imm24 \leq 0x7f$	$0x7f < imm24 \leq 0xffff$	$0xffff < imm24$
add.a %rd,imm24(6:0)	ext imm24(19:7) add.a %rd,imm24(6:0)	ext imm24(23:20) ext imm24(19:7) add.a %rd,imm24(6:0)

6.6.2 Comparison Instructions

● Types and functions of extended instructions

Extended instruction	Function	Expansion
<code>scmp %rd, imm16</code>	<code>%rd-imm16</code> (Sets/resets C, V, Z and N flags in PSR)	(1)
<code>scmc %rd, imm16</code>	<code>%rd-imm16-C</code> (Sets/resets C, V, Z and N flags in PSR)	(1)
<code>scmp.a %rd, imm20</code>	<code>%rd-imm20</code> (Sets/resets C, V, Z and N flags in PSR)	(2)
<code>xcmp %rd, imm16</code>	<code>%rd-imm16</code> (Sets/resets C, V, Z and N flags in PSR)	(1)
<code>xcmc %rd, imm16</code>	<code>%rd-imm16-C</code> (Sets/resets C, V, Z and N flags in PSR)	(1)
<code>xcmp.a %rd, imm24</code>	<code>%rd-imm24</code> (Sets/resets C, V, Z and N flags in PSR)	(3)

These extended instructions let you compare a general-purpose register and a signed 16-bit/20-bit/24-bit immediate. A conditional operation option (`/c`, `/nc`) cannot be specified in the extended instructions.

● Basic instructions after expansion

`scmp, xcmp` Expanded into the `cmp` instruction
`scmc, xcmc` Expanded into the `cmc` instruction
`scmp.a, xcmp.a` Expanded into the `cmp.a` instruction

● Expansion formats

(1) `sOP %rd, imm16 / xOP %rd, imm16` ($OP = \text{cmp, cmc}$)

Example: `xcmp %rd, imm16`

$imm16 \leq 0x7f$	$0x7f < imm16$
<code>cmp %rd, imm16(6:0)</code>	<code>ext imm16(15:7)</code> <code>cmp %rd, imm16(6:0)</code>

(2) `scmp.a %rd, imm20`

$imm20 \leq 0x7f$	$0x7f < imm20$
<code>cmp.a %rd, imm20(6:0)</code>	<code>ext imm20(19:7)</code> <code>cmp.a %rd, imm20(6:0)</code>

(3) `xcmp.a %rd, imm24`

$imm24 \leq 0x7f$	$0x7f < imm24 \leq 0xffff$	$imm24 > 0xffff$
<code>cmp.a %rd, imm24(6:0)</code>	<code>ext imm24(19:7)</code> <code>cmp.a %rd, imm24(6:0)</code>	<code>ext imm24(23:20)</code> <code>ext imm24(19:7)</code> <code>cmp.a %rd, imm24(6:0)</code>

6.6.3 Logic Operation Instructions

● Types and functions of extended instructions

Extended instruction	Function	Expansion
<code>sand %rd, imm16</code>	$\%rd \leftarrow \%rd \& imm16$	(1)
<code>soor %rd, imm16</code>	$\%rd \leftarrow \%rd imm16$	(1)
<code>sxor %rd, imm16</code>	$\%rd \leftarrow \%rd \wedge imm16$	(1)
<code>snot %rd, imm16</code>	$\%rd \leftarrow !imm16$	(1)
<code>xand %rd, imm16</code>	$\%rd \leftarrow \%rd \& imm16$	(1)
<code>xoor %rd, imm16</code>	$\%rd \leftarrow \%rd imm16$	(1)
<code>xxor %rd, imm16</code>	$\%rd \leftarrow \%rd \wedge imm16$	(1)
<code>xnot %rd, imm16</code>	$\%rd \leftarrow !imm16$	(1)

These extended instructions allow a signed 16-bit immediate to be specified directly in a logical operation.
A conditional operation option (`/c`, `/nc`) cannot be specified in the extended instructions.

● Basic instructions after expansion

sand, xand Expanded into the `and` instruction
soor, xoor Expanded into the `or` instruction
sxor, xxor Expanded into the `xor` instruction
snot, xnot Expanded into the `not` instruction

● Expansion formats

(1) `sOP %rd, imm16 / xOP %rd, imm16` ($OP = \text{and, oor, xor, not}$)

Example: `xand %rd, imm16`

$imm16 \leq 0x7f$	$0x7f < imm16$
<code>and %rd, imm16(6:0)</code>	<code>ext imm16(15:7)</code> <code>and %rd, imm16(6:0)</code>

6.6.4 Data Transfer Instructions (between Stack and Register)

• Types and functions of extended instructions

Extended instruction	Function	Expansion
<code>sld.b %rd, [%sp+imm20]</code>	$\%rd \leftarrow B[\%sp+imm20]$ (with sign extension)	(1)
<code>sld.ub %rd, [%sp+imm20]</code>	$\%rd \leftarrow B[\%sp+imm20]$ (with zero extension)	(1)
<code>sld %rd, [%sp+imm20]</code>	$\%rd \leftarrow W[\%sp+imm20]$	(1)
<code>sld.a %rd, [%sp+imm20]</code>	$\%rd \leftarrow A[\%sp+imm20](23:0)$, ignored $\leftarrow A[\%sp+imm20](31:24)$	(1)
<code>sld.b [%sp+imm20], %rs</code>	$B[\%sp+imm20] \leftarrow \%rs(7:0)$	(1)
<code>sld [%sp+imm20], %rs</code>	$W[\%sp+imm20] \leftarrow \%rs(15:0)$	(1)
<code>sld.a [%sp+imm20], %rs</code>	$A[\%sp+imm20](23:0) \leftarrow \%rs(23:0)$, $A[\%sp+imm20](31:24) \leftarrow 0$	(1)
<code>xld.b %rd, [%sp+imm24]</code>	$\%rd \leftarrow B[\%sp+imm24]$ (with sign extension)	(2)
<code>xld.ub %rd, [%sp+imm24]</code>	$\%rd \leftarrow B[\%sp+imm24]$ (with zero extension)	(2)
<code>xld %rd, [%sp+imm24]</code>	$\%rd \leftarrow W[\%sp+imm24]$	(2)
<code>xld.a %rd, [%sp+imm24]</code>	$\%rd \leftarrow A[\%sp+imm24](23:0)$, ignored $\leftarrow A[\%sp+imm24](31:24)$	(2)
<code>xld.b [%sp+imm24], %rs</code>	$B[\%sp+imm24] \leftarrow \%rs(7:0)$	(2)
<code>xld [%sp+imm24], %rs</code>	$W[\%sp+imm24] \leftarrow \%rs(15:0)$	(2)
<code>xld.a [%sp+imm24], %rs</code>	$A[\%sp+imm24](23:0) \leftarrow \%rs(23:0)$, $A[\%sp+imm24](31:24) \leftarrow 0$	(2)

These extended instructions allow you to directly specify a displacement of up to 20 bits/24 bits. Specification of *imm20/imm24* can be omitted.

Basic instructions after expansion

sld.b, xld.b	Expanded into the <code>ld.b</code> instruction
sld.ub, xld.ub	Expanded into the <code>ld.ub</code> instruction
sld, xld	Expanded into the <code>ld</code> instruction
sld.a, xld.a	Expanded into the <code>ld.a</code> instruction

• Expansion formats

If *imm20/imm24* is omitted, the **as** assembler assumes that `[%sp+0x0]` is specified as it expands the instruction.

- (1) **sOP %rd, [%sp+imm20]** (*OP* = `ld.b`, `ld.ub`, `ld`, `ld.a`)
sOP [%sp+imm20], %rs (*OP* = `ld.b`, `ld`, `ld.a`)
 Example: `sld.a %rd, [%sp+imm20]`

$imm20 \leq 0x7f$	$0x7f < imm20$
<code>ld.a %rd, [%sp+imm20(6:0)]</code>	ext <code>imm20(19:7)</code> <code>ld.a %rd, [%sp+imm20(6:0)]</code>

- (2) **xOP %rd, [%sp+imm24]** (*OP* = `ld.b`, `ld.ub`, `ld`, `ld.a`)
xOP [%sp+imm24], %rs (*OP* = `ld.b`, `ld`, `ld.a`)
 Example: `xld.a %rd, [%sp+imm24]`

$imm24 \leq 0x7f$	$0x7f < imm24 \leq 0xffff$	$0xffff < imm24$
<code>ld.a %rd, [%sp+imm24(6:0)]</code>	ext <code>imm24(19:7)</code> <code>ld.a %rd, [%sp+imm24(6:0)]</code>	ext <code>imm24(23:20)</code> ext <code>imm24(19:7)</code> <code>ld.a %rd, [%sp+imm24(6:0)]</code>

6.6.5 Data Transfer Instructions (between Memory and Register)

● Types and functions of extended instructions

Extended instruction	Function	Expansion
<code>sld.b %rd, [imm20]</code>	$\%rd \leftarrow B[imm20]$ (with sign extension)	(1)
<code>sld.ub %rd, [imm20]</code>	$\%rd \leftarrow B[imm20]$ (with zero extension)	(1)
<code>sld %rd, [imm20]</code>	$\%rd \leftarrow W[imm20]$	(1)
<code>sld.a %rd, [imm20]</code>	$\%rd \leftarrow A[imm20](23:0)$, ignored $\leftarrow A[imm20](31:24)$	(1)
<code>sld.b [imm20], %rs</code>	$B[imm20] \leftarrow \%rs(7:0)$	(1)
<code>sld [imm20], %rs</code>	$W[imm20] \leftarrow \%rs(15:0)$	(1)
<code>sld.a [imm20], %rs</code>	$A[imm20](23:0) \leftarrow \%rs(23:0)$, $A[imm20](31:24) \leftarrow 0$	(1)
<code>xld.b %rd, [imm24]</code>	$\%rd \leftarrow B[imm24]$ (with sign extension)	(2)
<code>xld.ub %rd, [imm24]</code>	$\%rd \leftarrow B[imm24]$ (with zero extension)	(2)
<code>xld %rd, [imm24]</code>	$\%rd \leftarrow W[imm24]$	(2)
<code>xld.a %rd, [imm24]</code>	$\%rd \leftarrow A[imm24](23:0)$, ignored $\leftarrow A[imm24](31:24)$	(2)
<code>xld.b [imm24], %rs</code>	$B[imm24] \leftarrow \%rs(7:0)$	(2)
<code>xld [imm24], %rs</code>	$W[imm24] \leftarrow \%rs(15:0)$	(2)
<code>xld.a [imm24], %rs</code>	$A[imm24](23:0) \leftarrow \%rs(23:0)$, $A[imm24](31:24) \leftarrow 0$	(2)

These extended instructions allow memory locations to be accessed by specifying the address with a 20-bit/24-bit immediate. However, the postincrement function (`[]+`) cannot be used.

● Basic instructions after expansion

sld.b, xld.b Expanded into the `ld.b` instruction
sld.ub, xld.ub Expanded into the `ld.ub` instruction
sld, xld Expanded into the `ld` instruction
sld.a, xld.a Expanded into the `ld.a` instruction

● Expansion formats

- (1) **sOP %rd, [imm20]** ($OP = \text{ld.b, ld.ub, ld, ld.a}$)
sOP [imm20], %rs ($OP = \text{ld.b, ld, ld.a}$)

Example: `sld.a %rd, [imm20]`

$imm20 \leq 0x7f$	$0x7f < imm20$
<code>ld.a %rd, [imm20(6:0)]</code>	ext $imm20(19:7)$ <code>ld.a %rd, [imm20(6:0)]</code>

- (2) **xOP %rd, [imm24]** ($OP = \text{ld.b, ld.ub, ld, ld.a}$)
xOP [imm24], %rs ($OP = \text{ld.b, ld, ld.a}$)

Example: `xld.a %rd, [imm24]`

$imm24 \leq 0x7f$	$0x7f < imm24 \leq 0xffff$	$0xffff < imm24$
<code>ld.a %rd, [imm24(6:0)]</code>	ext $imm24(19:7)$ <code>ld.a %rd, [imm24(6:0)]</code>	ext $imm24(23:20)$ ext $imm24(19:7)$ <code>ld.a %rd, [imm24(6:0)]</code>

6.6.6 Immediate Data Load Instructions

● Types and functions of extended instructions

Extended instruction	Function	Expansion
<code>sld %rd, imm16</code>	$\%rd \leftarrow imm16$	(1)
<code>sld.a %rd, imm20</code>	$\%rd \leftarrow imm20$	(2)
<code>sld.a %sp, imm20</code>	$\%sp \leftarrow imm20$	(2)
<code>sld %rd, symbol+imm16</code>	$\%rd \leftarrow symbol+imm16(15:0)$	(4)
<code>sld.a %rd, symbol+imm20</code>	$\%rd \leftarrow symbol+imm20(19:0)$	(5)
<code>sld.a %sp, symbol+imm20</code>	$\%sp \leftarrow symbol+imm20(19:0)$	(5)
<code>xld %rd, imm16</code>	$\%rd \leftarrow imm16$	(1)
<code>xld.a %rd, imm24</code>	$\%rd \leftarrow imm24$	(3)
<code>xld.a %sp, imm24</code>	$\%sp \leftarrow imm24$	(3)
<code>xld %rd, symbol+imm16</code>	$\%rd \leftarrow symbol+imm16(15:0)$	(4)
<code>xld.a %rd, symbol+imm24</code>	$\%rd \leftarrow symbol+imm24(23:0)$	(6)
<code>xld.a %sp, symbol+imm24</code>	$\%sp \leftarrow symbol+imm24(23:0)$	(6)

These extended instructions allow a 16-bit/20-bit/24-bit immediate to be loaded directly into a general-purpose register. A symbol also can be used for immediate specification.

● Basic instructions after expansion

`sld, xld` Expanded into the `ld` instruction
`sld.a, xld.a` Expanded into the `ld.a` instruction

● Expansion formats

(1) `sld %rd, imm16 / xld %rd, imm16`

Example: `xld %rd, imm16`

$imm16 \leq 0x7f$	$0x7f < imm16$
<code>ld %rd, imm16(6:0)</code>	<code>ext imm16(15:7)</code> <code>ld %rd, imm16(6:0)</code>

(2) `sld.a %rd, imm20 / sld.a %sp, imm20`

Example: `sld.a %rd, imm20`

$imm20 \leq 0x7f$	$0x7f < imm20$
<code>ld.a %rd, imm20(6:0)</code>	<code>ext imm20(19:7)</code> <code>ld.a %rd, imm20(6:0)</code>

(3) `xld.a %rd, imm24 / xld.a %sp, imm24`

Example: `xld.a %rd, imm24`

$imm24 \leq 0x7f$	$0x7f < imm24 \leq 0xffff$	$0xffff < imm24$
<code>ld.a %rd, imm24(6:0)</code>	<code>ext imm24(19:7)</code> <code>ld.a %rd, imm24(6:0)</code>	<code>ext imm24(23:20)</code> <code>ext imm24(19:7)</code> <code>ld.a %rd, imm24(6:0)</code>

- (4) **sld %rd, symbol±imm16 / xld %rd, symbol±imm16**

Example: sld %rd, symbol±imm16

Unconditional
ext (symbol±imm16) (15:7)
ld %rd, (symbol±imm16) (6:0)

- (5) **sld.a %rd, symbol±imm20 / sld.a %sp, symbol±imm20**

Example: sld.a %rd, symbol±imm20

Unconditional
ext (symbol±imm20) (19:7)
ld.a %rd, (symbol±imm20) (6:0)

- (6) **xld.a %rd, symbol±imm24 / xld.a %sp, symbol±imm24**

Example: xld.a %rd, symbol±imm24

Unconditional
ext (symbol±imm24) (23:20)
ext (symbol±imm24) (19:7)
ld.a %rd, (symbol±imm24) (6:0)

6.6.7 Branch Instructions

● Types and functions of extended instructions

Extended instruction	Function	Expansion
<code>scall label+imm20</code>	PC relative subroutine call	(1)
<code>sjpr label+imm20</code>	PC relative unconditional jump	(1)
<code>sjreq label+imm20</code>	PC relative conditional jump	(2)
<code>sjrne label+imm20</code>	PC relative conditional jump	(2)
<code>sjrgt label+imm20</code>	PC relative conditional jump	(2)
<code>sjrge label+imm20</code>	PC relative conditional jump	(2)
<code>sjrlt label+imm20</code>	PC relative conditional jump	(2)
<code>sjrle label+imm20</code>	PC relative conditional jump	(2)
<code>sjrugt label+imm20</code>	PC relative conditional jump	(2)
<code>sjruge label+imm20</code>	PC relative conditional jump	(2)
<code>sjrult label+imm20</code>	PC relative conditional jump	(2)
<code>sjrule label+imm20</code>	PC relative conditional jump	(2)
<code>scalla label+imm20</code>	PC absolute subroutine call	(3)
<code>sjpa label+imm20</code>	PC absolute unconditional jump	(3)
<code>scall sign20</code>	PC relative subroutine call	(4)
<code>sjpr sign20</code>	PC relative unconditional jump	(4)
<code>sjreq sign20</code>	PC relative conditional jump	(5)
<code>sjrne sign20</code>	PC relative conditional jump	(5)
<code>sjrgt sign20</code>	PC relative conditional jump	(5)
<code>sjrge sign20</code>	PC relative conditional jump	(5)
<code>sjrlt sign20</code>	PC relative conditional jump	(5)
<code>sjrle sign20</code>	PC relative conditional jump	(5)
<code>sjrugt sign20</code>	PC relative conditional jump	(5)
<code>sjruge sign20</code>	PC relative conditional jump	(5)
<code>sjrult sign20</code>	PC relative conditional jump	(5)
<code>sjrule sign20</code>	PC relative conditional jump	(5)
<code>scalla imm20</code>	PC absolute subroutine call	(6)
<code>sjpa imm20</code>	PC absolute unconditional jump	(6)
<code>xcall label+imm24</code>	PC relative subroutine call	(7)
<code>xjpr label+imm24</code>	PC relative unconditional jump	(7)
<code>xjreq label+imm24</code>	PC relative conditional jump	(8)
<code>xjrne label+imm24</code>	PC relative conditional jump	(8)
<code>xjrgt label+imm24</code>	PC relative conditional jump	(8)
<code>xjrge label+imm24</code>	PC relative conditional jump	(8)
<code>xjrlt label+imm24</code>	PC relative conditional jump	(8)
<code>xjrle label+imm24</code>	PC relative conditional jump	(8)
<code>xjrugt label+imm24</code>	PC relative conditional jump	(8)
<code>xjruge label+imm24</code>	PC relative conditional jump	(8)
<code>xjrult label+imm24</code>	PC relative conditional jump	(8)
<code>xjrule label+imm24</code>	PC relative conditional jump	(8)
<code>xcalla label+imm24</code>	PC absolute subroutine call	(9)
<code>xjpa label+imm24</code>	PC absolute unconditional jump	(9)
<code>xcall sign24</code>	PC relative subroutine call	(10)
<code>xjpr sign24</code>	PC relative unconditional jump	(10)
<code>xjreq sign24</code>	PC relative conditional jump	(11)
<code>xjrne sign24</code>	PC relative conditional jump	(11)

xjrgt <i>sign24</i>	PC relative conditional jump	(11)
xjrge <i>sign24</i>	PC relative conditional jump	(11)
xjrlt <i>sign24</i>	PC relative conditional jump	(11)
xjrle <i>sign24</i>	PC relative conditional jump	(11)
xjrugt <i>sign24</i>	PC relative conditional jump	(11)
xjruge <i>sign24</i>	PC relative conditional jump	(11)
xjrult <i>sign24</i>	PC relative conditional jump	(11)
xjrle <i>sign24</i>	PC relative conditional jump	(11)
xcalla <i>imm24</i>	PC absolute subroutine call	(12)
xjpa <i>imm24</i>	PC absolute unconditional jump	(12)

These extended instructions allow a branch destination to be specified using a 20-bit/24-bit immediate or a label. The branch conditions of these conditional jump instructions are the same as those of the basic instructions.

The extended instructions can be used as delayed branch instructions by adding ".d".

Example: `xcall.d sign24`

● Basic instructions after expansion

scall, scall.d, xcall, xcall.d	Expanded into the <code>call/call.d</code> instruction
scalla, scalla.d, xcalla, xcalla.d	Expanded into the <code>calla/calla.d</code> instruction
sjpa, sjpa.d, xjpa, xjpa.d	Expanded into the <code>jpa/jpa.d</code> instruction
sjpr, sjpr.d, xjpr, xjpr.d	Expanded into the <code>jpr/jpr.d</code> instruction
sjreq, sjreq.d, xjreq, xjreq.d	Expanded into the <code>jreq/jreq.d</code> instruction
sjrne, sjrne.d, xjrne, xjrne.d	Expanded into the <code>jrne/jrne.d</code> instruction
sjrgt, sjrgt.d, xjrgt, xjrgt.d	Expanded into the <code>jrgt/jrgt.d</code> instruction
sjrge, sjrge.d, xjrge, xjrge.d	Expanded into the <code>jrge/jrge.d</code> instruction
sjrlt, sjrlt.d, xjrlt, xjrlt.d	Expanded into the <code>jrlt/jrlt.d</code> instruction
sjrle, sjrle.d, xjrle, xjrle.d	Expanded into the <code>jrle/jrle.d</code> instruction
sjrugt, sjrugt.d, xjrugt, xjrugt.d	Expanded into the <code>jrugt/jrugt.d</code> instruction
sjruge, sjruge.d, xjruge, xjruge.d	Expanded into the <code>jruge/jruge.d</code> instruction
sjrult, sjrult.d, xjrult, xjrult.d	Expanded into the <code>jruult/jruult.d</code> instruction
sjrule, sjrule.d, xjrle, xjrle.d	Expanded into the <code>jrle/jrule.d</code> instruction

● Expansion formats

- (1) **sOP *label+imm20*** (*OP* = `call, call.d, jpr, jpr.d`)

Example: `scall label+imm20`

Unconditional	
ext	(<i>label+imm20</i>) (19:12)
call	(<i>label+imm20</i>) (11:1)

- (2) **sOP *label+imm20*** (*OP* = `jr*, jr*.d`)

Example: `sjreq label+imm20`

Unconditional	
ext	(<i>label+imm20</i>) (19:8)
jreq	(<i>label+imm20</i>) (7:1)

- (3) **sOP *label+imm20*** (*OP* = `calla, calla.d, jpa, jpa.d`)

Example: `scalla label+imm20`

Unconditional	
ext	(<i>label+imm20</i>) (19:7)
calla	(<i>label+imm20</i>) (6:0)

- (4)
- sOP sign20**
- (OP= call, call.d, jpr, jpr.d)

Example: scall sign20

$-1024 \leq \text{sign20} \leq 1023$	$\text{sign20} < -1024$ or $1023 < \text{sign20}$
call sign20(11:1)	ext sign20(23:12) call sign20(11:1)

- (5)
- sOP sign20**
- (OP= jr*, jr*.d)

Example: sjreq sign20

$-128 \leq \text{sign20} \leq 127$	$\text{sign20} < -128$ or $127 < \text{sign20}$
jreq sign20(7:1)	ext sign20(19:8) jreq sign20(7:1)

- (6)
- sOP imm20**
- (OP= calla, calla.d, jpa, jpa.d)

Example: scalla imm20

$\text{imm20} \leq 0x7f$	$0x7f < \text{imm20}$
calla imm20(6:0)	ext imm20(19:7) calla imm20(6:0)

- (7)
- xOP label±imm24**
- (OP= call, call.d, jpr, jpr.d)

Example: xcall label±imm24

Unconditional
ext (label±imm24) (23:12) call (label±imm24) (11:1)

- (8)
- xOP label±imm24**
- (OP= jr*, jr*.d)

Example: xjreq label±imm24

Unconditional
ext (label±imm24) (23:21) ext (label±imm24) (20:8) jreq (label±imm24) (7:1)

- (9)
- xOP label±imm24**
- (OP= calla, calla.d, jpa, jpa.d)

Example: xcalla label±imm24

Unconditional
ext (label±imm24) (23:20) ext (label±imm24) (19:7) calla (label±imm24) (6:0)

- (10)
- xOP sign24**
- (OP= call, call.d, jpr, jpr.d)

Example: xcall sign24

$-1024 \leq \text{sign24} \leq 1023$	$\text{sign24} < -1024$ or $1023 < \text{sign24}$
call sign24(11:1)	ext sign24(23:12) call sign24(11:1)

(11) **xOP sign24** (OP = jr*, jr*.d)

Example: xjreq sign24

$-128 \leq \text{sign24} \leq 127$	$-1048576 \leq \text{sign24} < -128$ or $127 < \text{sign24} \leq 1048575$	$\text{sign24} < -1048576$ or $1048575 < \text{sign24}$
jreq sign24(7:1)	ext sign24(20:8) jreq sign24(7:1)	ext sign24(23:21) ext sign24(20:8) jreq sign24(7:1)

(12) **xOP imm24** (OP = calla, calla.d, jpa, jpa.d)

Example: xcalla imm24

$\text{imm24} \leq 0x7f$	$0x7f < \text{imm24} \leq 0xffff$	$0xffff < \text{imm24}$
calla imm24(6:0)	ext imm24(19:7) calla imm24(6:0)	ext imm24(23:20) ext imm24(19:7) calla imm24(6:0)

6.6.8 Coprocessor Instructions

● Types and functions of extended instructions

Extended instruction	Function	Expansion
<code>sld.cw %rd, imm20</code>	Coprocessor \leftarrow %rd & imm20	(1)
<code>sld.ca %rd, imm20</code>	Coprocessor \leftarrow %rd & imm20, get results and flag statuses	(1)
<code>sld.cf %rd, imm20</code>	Coprocessor \leftarrow %rd & imm20, get flag statuses	(1)
<code>sld.cw %rd, symbol+imm20</code>	Coprocessor \leftarrow %rd & symbol+imm20	(2)
<code>sld.ca %rd, symbol+imm20</code>	Coprocessor \leftarrow %rd & symbol+imm20, get results and flag statuses	(2)
<code>sld.cf %sp, symbol+imm20</code>	Coprocessor \leftarrow %rd & symbol+imm20, get flag statuses	(2)
<code>xld.cw %rd, imm24</code>	Coprocessor \leftarrow %rd & imm24	(3)
<code>xld.ca %rd, imm24</code>	Coprocessor \leftarrow %rd & imm24, get results and flag statuses	(3)
<code>xld.cf %rd, imm24</code>	Coprocessor \leftarrow %rd & imm24, get flag statuses	(3)
<code>xld.cw %rd, symbol+imm24</code>	Coprocessor \leftarrow %rd & symbol+imm24	(4)
<code>xld.ca %rd, symbol+imm24</code>	Coprocessor \leftarrow %rd & symbol+imm24, get results and flag statuses	(4)
<code>xld.cf %rd, symbol+imm24</code>	Coprocessor \leftarrow %rd & symbol+imm24, get flag statuses	(4)

These extended instructions allow a 20-bit/24-bit immediate to be transferred to the coprocessor. A symbol also can be used for immediate specification.

● Basic instructions after expansion

<code>sld.cw, xld.cw</code>	Expanded into the <code>ld.cw</code> instruction
<code>sld.ca, xld.ca</code>	Expanded into the <code>ld.ca</code> instruction
<code>sld.cf, xld.cf</code>	Expanded into the <code>ld.cf</code> instruction

● Expansion formats

- (1) **sOP %rd, imm20** ($OP = \text{ld.cw, ld.ca, ld.cf}$)

Example: `sld.ca %rd, imm20`

$imm20 \leq 0x7f$	$0x7f < imm20$
<code>ld.ca %rd, imm20(6:0)</code>	ext <code>imm20(19:7)</code> <code>ld.ca %rd, imm20(6:0)</code>

- (2) **sOP %rd, symbol+imm20** ($OP = \text{ld.cw, ld.ca, ld.cf}$)

Example: `sld.ca %rd, symbol+imm20`

Unconditional
ext <code>(symbol+imm20)(19:7)</code> <code>ld.ca %rd, (symbol+imm20)(6:0)</code>

- (3) **xOP %rd, imm24** ($OP = \text{ld.cw, ld.ca, ld.cf}$)

Example: `xld.ca %rd, imm24`

$imm24 \leq 0x7f$	$0x7f < imm24 \leq 0xffff$	$0xffff < imm24$
<code>ld.ca %rd, imm24(6:0)</code>	ext <code>imm24(19:7)</code> <code>ld.ca %rd, imm24(6:0)</code>	ext <code>imm24(23:20)</code> ext <code>imm24(19:7)</code> <code>ld.ca %rd, imm24(6:0)</code>

- (4) **xOP %rd, symbol+imm24** ($OP = \text{ld.cw, ld.ca, ld.cf}$)

Example: `xld.ca %rd, symbol+imm24`

Unconditional
ext <code>(symbol+imm24)(23:20)</code> ext <code>(symbol+imm24)(19:7)</code> <code>ld.ca %rd, (symbol+imm24)(6:0)</code>

6.6.9 Xext Instructions

● Types and functions of extended instruction

Extended instruction	Function	Expansion
Xext <i>imm24</i>	Expanded into the ext instruction	(1)

Combined with the instructions below, this instruction functions as the offset.

Xext *imm24*

OP [%rd], %rs ==> [%rd+imm24] ← Functions as %rs

Xext *imm24*

OP %rd, [%rs] ==> %rd ← Functions as [%rs+imm24]

(OP = ld.b, ld.ub, ld, ld.a)

● Basic instructions after expansion

Xext Expanded into the ext instruction

● Expansion formats

(1) **Xext** *imm24*

$imm24 \leq 0x1fff$	$0x1fff < imm24 \leq 0xffff$
ext <i>imm24</i> (12:0)	ext <i>imm24</i> (23:13) ext <i>imm24</i> (12:0)

6.7 Error/Warning Messages

The following shows the principal error and warning messages output by the assembler **as**:

Table 6.7.1 Error messages

Error message	Description
Error: Unrecognized opcode: 'XXXXX'	The operation code XXXXX is undefined.
Error: junk at end of line: 'XXXXX'	A format error of the operand.
Error: XXXXXX: invalid register name	The specified register cannot be used.

Table 6.7.2 Warning messages

Warning message	Description
Warning: Unrecognized .section attribute: want a, w, x	The section attribute is not a, w or x.
Warning: Bignum truncated to AAA bytes	The constant declared (e.g. .long, .int) exceeds the maximum size. It has been corrected to AAA-byte size. (e.g. 0x100000012 → 0x12)
Warning: Value XXXX truncated to AAA	The constant declared exceeds the maximum value AAA. It has been corrected to AAA. (e.g. .byte 0x100000012 → .byte 0xff)
Warning: operand out of range (XXXXXX: XXX not between AAA and BBB)	The value specified in the operand is out of the effective range.

6.8 Precautions

- To perform assembly source level debugging with the debugger **gdb**, specify the `--gstabs` assembler option to add the source information to the output object file when assembling the source file.
- Always be sure to use the **xgcc** compiler and/or **as** assembler to add debugging information (`.stab` directive) in the source file and do not use any other method. Also be sure not to correct the debugging information that is output. Corrections could cause the **as**, **ld** or **gdb** to malfunction.
- To prevent errors during linkage, be sure to write the `.section` directive with the `.align` directive to clearly define the section boundary.

Example:

```
        .section .rodata
        .align 2          ; ←Essential
.long   data1
.long   data2
```

7 Linker

This chapter describes the functions of the **ld** linker.

7.1 Functions

The **ld** linker 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.

This linker is based on the gnu linker (ld). For details about the **ld** linker or method for describing linker script, refer to the documents for the gnu linker. The documents can be acquired from the GNU mirror sites located in various places around the world through Internet, etc.

7.2 Input/Output Files

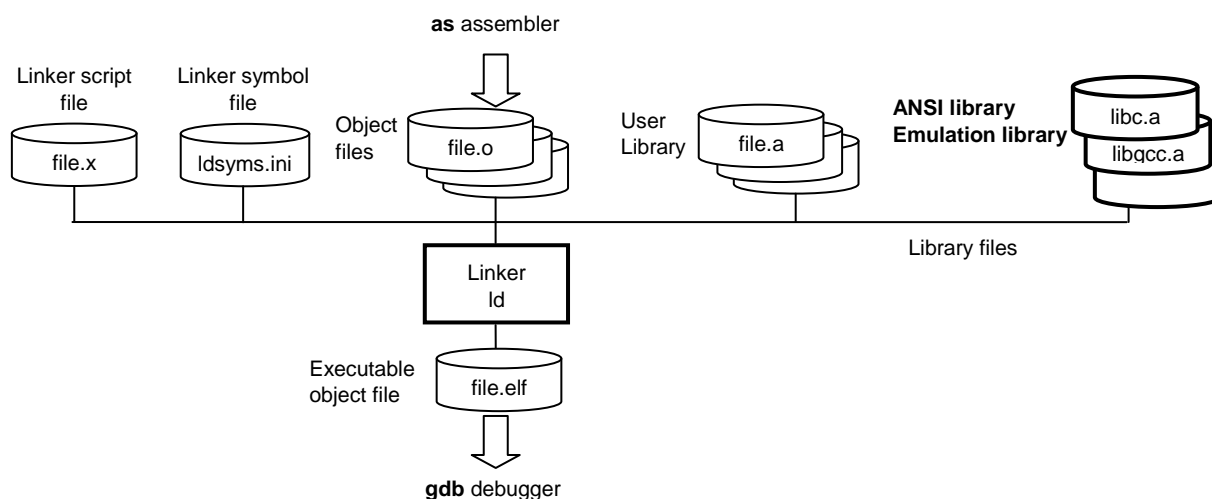


Figure 7.2.1 Flowchart

7.2.1 Input Files

● Object file

File format: Binary file in elf format
 File name: `<filename>.o`
 Description: Object file of individual modules created by the **as** assembler.

● Library file

File format: Binary file in library format
 File name: `<filename>.a`
 Description: ANSI library files, emulation library files and user library files.

● Linker script file

File format: Text file
 File name: `<filename>.x`
 Description: File to specify the start address of each section and other information for linkage.
 It is input to the **ld** linker when the `-T` option is specified.

● Linker symbol file

File format: Text file
 File name: `ldsyms.ini`
 Description: File to specify the address corresponding to the symbol name.
 This is input to the **ld** linker when the `-R` option is specified.

7.2.2 Output Files

● Executable object file

File format: Binary file in elf format
 File name: `<filename>.elf`
 Description: Object file in executable format that can be input in the **gdb** debugger. 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 elf format.
 The default file name is `a.out` when no output file name is specified using the `-o` option.

● Link map file

File format: Text file
 File name: `<filename>.map`
 Description: Mapping information file showing from which address of a section each input file was mapped.
 The file is delivered when the `-M` or `-Map` option is specified.

7.3 Starting Method

7.3.1 Startup Format

To invoke the **ld** linker, use the command shown below.

ld *<options>* *<file names>*

<options> See Section 7.3.2.

<file names> Specify one or more object file names and/or one or more library file names.

Example: `ld -o sample.elf sample.o ../lib/24bit/libc.a ../lib/24bit/libgcc.a`

7.3.2 Command-line Options

The **ld** linker accepts the gnu linker standard options.

The following lists the principal options only. Refer to the gnu linker manual for more information.

-o *<filename>*

Function: **Specify output file name**

Explanation: This option is used to specify the name of the object file output by the **ld** linker.

Default: The default output file name is `a.out`.

-T *<linker script file name>*

Function: **Read linker script file**

Explanation: Specify this option when loading relocate-information into the **ld** linker using a linker script file.

Default: The default linker script (see Section 7.4.1) is used.

-R *<linker symbol file name>*

Function: **Read linker symbol file**

Explanation: This option is used to specify the address corresponding to the symbol name using the linker symbol file.

Default: The linker symbol file is not read.

-M

-Map *<filename>*

Function: **Output link map file**

Explanation: The **-M** option outputs the link map information to `stdio`.

The **-Map** option outputs the link map information to a file.

Default: No link map information is output.

-N

Function: **Disable data segment alignment check**

Explanation: When the **-N** option is specified, the linker does not check the alignment of data segments.

Default: The linker checks the alignment of data segments.

--relax

Function: **Optimize output code size**

Explanation: Specifying the **--relax** option optimizes output code size by deleting the `ext 0` instruction.

Default: The linker does not delete the `ext 0` instruction.

When inputting options in the command line, one or more spaces are necessary before and after the option.

Example: `ld -o sample.elf -T sample.lds -N boot.o sample.o ../lib/24bit/libc.a`

7.4 Linkage

7.4.1 Default Linker Script

Default linker script when the `-T` option is not specified

When the `-T` option is not specified, the `ld` linker uses the default script shown below for linkage.

```

OUTPUT_FORMAT("elf32-c17")
OUTPUT_ARCH(c17)
ENTRY(_start)
SEARCH_DIR(.);
MEMORY
{
    iram      : ORIGIN = 0,          LENGTH = 32K
    irom      : ORIGIN = 0x8000,     LENGTH = 4064K
}
SECTIONS
{
    .bss (NOLOAD) :
    {
        PROVIDE (__START_bss = .) ;
        *(.bss)
        *(.bss.*)
        *(COMMON)
        PROVIDE (__END_bss = .) ;
    } > iram
    .vector :
    {
        PROVIDE (__START_vector = .) ;
        KEEP (*crt0.o(.rodata))
        PROVIDE (__END_vector = .) ;
    } > irom
    .text :
    {
        PROVIDE (__START_text = .) ;
        *(.text.*)
        *(.text)
        PROVIDE (__END_text = .) ;
    } > irom
    .data :
    {
        PROVIDE (__START_data = .) ;
        *(.data)
        *(.data.*)
        PROVIDE (__END_data = .) ;
    } > iram AT > irom
    .rodata :
    {
        PROVIDE (__START_rodata = .) ;
        *(EXCLUDE_FILE (*crt0.o) .rodata)
        *(.rodata.*)
        PROVIDE (__END_rodata = .) ;
    } > irom

```

```

PROVIDE (__START_data_lma = LOADADDR(.data));
PROVIDE (__END_data_lma = LOADADDR(.data) + SIZEOF (.data));
PROVIDE (__START_stack = 0x0007C0);
}

```

In this script, data will be located from address 0 in order of .bss and .data sections, the vector table, program codes and constant data will be located from address 0x8000.

Figure 7.4.1.1 shows the memory map after linkage.

	.data (initial values)	LMA
	.rodata	VMA=LMA
0x8080	.text	VMA=LMA
0x8000	.vector	VMA=LMA
	Unused	
	.data	VMA
0x0000	.bss	VMA=LMA

Figure 7.4.1.1 Memory map configured by default script

7.4.2 Examples of Linkage

When virtual and shared sections are used

The following is a sample linker script when virtual and shared sections are used:

```

OUTPUT_FORMAT("elf32-c17")
OUTPUT_ARCH(c17)
ENTRY(_start)
SEARCH_DIR(.);
MEMORY
{
    iram      : ORIGIN = 0,          LENGTH = 32K
    irom      : ORIGIN = 0x8000,     LENGTH = 4064K
}
SECTIONS
{
    .bss (NOLOAD) :
    {
        PROVIDE (__START_bss = .) ;
        *(.bss)
        *(.bss.*)
        *(COMMON)
        PROVIDE (__END_bss = .) ;
    } > iram
    .vector :
    {
        PROVIDE (__START_vector = .) ;
        KEEP (*crt0.o(.rodata))
        PROVIDE (__END_vector = .) ;
    } > irom
    .text :
    {
        PROVIDE (__START_text = .) ;
        *(EXCLUDE_FILE (*foo1.o *foo2.o *foo3.o) .text.*)
        *(EXCLUDE_FILE (*foo1.o *foo2.o *foo3.o) .text)
        PROVIDE (__END_text = .) ;
    } > irom
    .data :
    {
        PROVIDE (__START_data = .) ;
        *(.data)
        *(.data.*)
        PROVIDE (__END_data = .) ;
    } > iram AT > irom
    OVERLAY ".":
    {
        .text_foo1 { *foo1.o(.text.*) *foo1.o(.text) }
        .text_foo2 { *foo2.o(.text.*) *foo2.o(.text) }
        .text_foo3 { *foo3.o(.text.*) *foo3.o(.text) }
    } > iram AT > irom
    .rodata :
    {
        PROVIDE (__START_rodata = .) ;
        *(EXCLUDE_FILE (*crt0.o) .rodata)
        *(.rodata.*)

```

```

    PROVIDE ( __END_roddata = . ) ;
} > irom
PROVIDE ( __START_data_lma = LOADADDR(.data));
PROVIDE ( __END_data_lma = LOADADDR(.data) + SIZEOF (.data));
PROVIDE ( __START_stack = 0x0007C0);
}

```

The section map is shown in Figure 7.4.2.1.

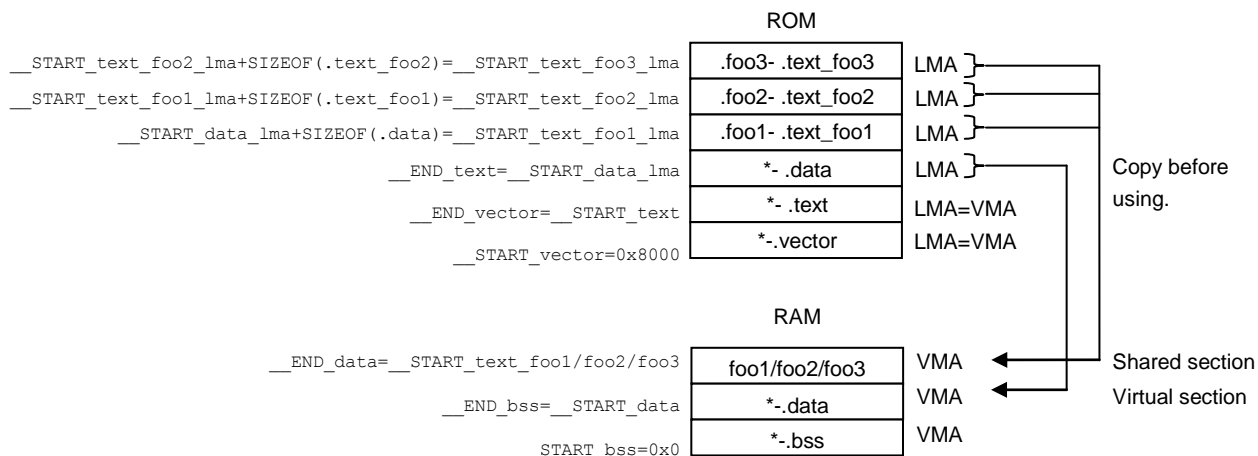


Figure 7.4.2.1 Memory map

The substance of the `.data` section is placed on the LMA in the ROM, and it must be copied to the VMA in the RAM (immediately following the `.bss` section) before it can be used. The `.data` section (VMA) in the RAM is a virtual section that does not exist when the program starts executing. This method should be used for handling variables that have an initial value. In this example, the `.data` sections in all the files are combined into one section.

`.text_foo1` is the `.text` section in the `foo1.o` file. Its actual code is located at the LMA in the ROM and is executed at the VMA in the RAM. Also the `.text_foo2` and `.text_foo3` sections are used similarly and the same VMA is set for these three sections. The RAM area for `.text_foo1/2/3` is a shared section used for executing multiple `.text` sections by replacing the codes. A program cache for high-speed program execution is realized in this method. The `.text` sections in other files than these three files are located in the `.text` section that follows the `.vector` section and are executed at the stored address in the ROM.

7.4.3 Link Maps

Linker **ld** will output link map information if **-M** (or **-Map**) is specified in the command line options.

The following shows an example link map output when the default linker script shown in Section 7.4.1 is used.

```
.bss      0x00000000      0xba
          0x00000000      PROVIDE ( __START_bss, .)
*(.bss)
.bss      0x00000000      0x30 crt0.o
          0x00000000      gm_sec
          0x00000004      seed
          0x00000008      stderr
          0x0000000c      stdout
          0x00000010      stdin
          0x00000014      _iob
          0x0000002c      errno
.bss      0x00000030      0x0 src\sample_gcc6.o
.bss      0x00000030      0x0 C:\EPSON\GNU17V3\gcc6\lib\24bit\libgcc.a(emu_copro_process.o)
.bss      0x00000030      0x0 C:\EPSON\GNU17V3\gcc6\lib\24bit\libc.a(puts.o)
.bss      0x00000030      0x0 C:\EPSON\GNU17V3\gcc6\lib\24bit\libc.a(memcpy.o)
.bss      0x00000030      0x0 C:\EPSON\GNU17V3\gcc6\lib\24bit\libc.a(memset.o)
.bss      0x00000030      0x0 C:\EPSON\GNU17V3\gcc6\lib\24bit\libc.a(putc.o)
.bss      0x00000030      0x8a C:\EPSON\GNU17V3\gcc6\lib\24bit\libg.a(debuglib.o)
          0x00000030      WRITE_SIZE
          0x00000032      WRITE_BUF
          0x00000074      READ_BUF_POS
          0x00000076      READ_EOF
          0x00000078      READ_BUF
*(.bss.*)
*(COMMON)
          0x000000ba      PROVIDE ( __END_bss, .)

.vector   0x00008000      0x80
          [!provide]      PROVIDE ( __START_vector, .)
*crt0.o(.rodata)
.rodata   0x00008000      0x80 crt0.o
          0x00008000      _vector
          [!provide]      PROVIDE ( __END_vector, .)

.text     0x00008080      0x3b6
          [!provide]      PROVIDE ( __START_text, .)
*(.text.*)
*(.text)
.text     0x00008080      0xac crt0.o
          0x00008080      _crt0_start0
          0x00008080      _start
          0x0000808a      _vector20_handler
          0x0000808a      _vector06_handler
          0x0000808a      _vector28_handler
          0x0000808a      _vector31_handler
          0x0000808a      _vector27_handler
          0x0000808a      _vector07_handler
          0x0000808a      _vector09_handler
          0x0000808a      _vector18_handler
          0x0000808a      _vector21_handler
          0x0000808a      _vector24_handler
          0x0000808a      _vector26_handler
          0x0000808a      _vector22_handler
          0x0000808a      _vector12_handler
          0x0000808a      _vector17_handler
          0x0000808a      _vector08_handler
          0x0000808a      _vector19_handler
          0x0000808a      _vector13_handler
          0x0000808a      _vector25_handler
          0x0000808a      _vector01_handler
          0x0000808a      _vector11_handler
```

```

0x0000808a      _vector14_handler
0x0000808a      _vector05_handler
0x0000808a      _vector29_handler
0x0000808a      _vector16_handler
0x0000808a      _vector04_handler
0x0000808a      _vector02_handler
0x0000808a      _vector23_handler
0x0000808a      _vector30_handler
0x0000808a      _vector15_handler
0x0000808a      _vector10_handler
0x0000808c      _crt0_init_dummy
0x0000808c      _init_device
0x0000808e      _start_device
0x0000808e      _crt0_start_device
0x00008092      _crt0_stop_device
0x00008092      _stop_device
0x00008096      _init_lib
0x00008096      _crt0_init_lib
0x000080de      _crt0_init_section
0x000080de      _init_section
0x00008114      _crt0_exit
0x00008116      _crt0_start1
0x00008116      _start1
.text          0x0000812c      0xa  src\sample_gcc6.o
              0x0000812c      main
.text          0x00008136      0x2  C:\EPSON\GNU17V3\gcc6\lib\24bit\libgcc.a(emu_copro_process.o)
              0x00008136      emu_copro_process
.text          0x00008138      0x76 C:\EPSON\GNU17V3\gcc6\lib\24bit\libc.a(puts.o)
              0x00008138      puts
.text          0x000081ae      0x10 C:\EPSON\GNU17V3\gcc6\lib\24bit\libc.a(memcpy.o)
              0x000081ae      memcpy
.text          0x000081be      0xe  C:\EPSON\GNU17V3\gcc6\lib\24bit\libc.a(memset.o)
              0x000081be      memset
.text          0x000081cc      0x30 C:\EPSON\GNU17V3\gcc6\lib\24bit\libc.a(putc.o)
              0x000081cc      putc
.text          0x000081fc      0x23a C:\EPSON\GNU17V3\gcc6\lib\24bit\libg.a(debuglib.o)
              0x000081fc      write
              0x0000823a      WRITE_FLASH
              0x00008330      _init_sys
              0x00008340      read
              0x000083d2      READ_FLASH
              0x00008432      _exit
              [!provide]      PROVIDE ( __END_text, .)

.data          0x000000ba      0x0  load address 0x00008436
              0x000000ba      PROVIDE ( __START_data, .)
*(.data)
.data          0x000000ba      0x0  crt0.o
.data          0x000000ba      0x0  src\sample_gcc6.o
.data          0x000000ba      0x0  C:\EPSON\GNU17V3\gcc6\lib\24bit\libgcc.a(emu_copro_process.o)
.data          0x000000ba      0x0  C:\EPSON\GNU17V3\gcc6\lib\24bit\libc.a(puts.o)
.data          0x000000ba      0x0  C:\EPSON\GNU17V3\gcc6\lib\24bit\libc.a(memcpy.o)
.data          0x000000ba      0x0  C:\EPSON\GNU17V3\gcc6\lib\24bit\libc.a(memset.o)
.data          0x000000ba      0x0  C:\EPSON\GNU17V3\gcc6\lib\24bit\libc.a(putc.o)
.data          0x000000ba      0x0  C:\EPSON\GNU17V3\gcc6\lib\24bit\libg.a(debuglib.o)
*(.data.*)
              0x000000ba      PROVIDE ( __END_data, .)

.rodata        0x00008436      0xc
              [!provide]      PROVIDE ( __START_rodata, .)
*(EXCLUDE_FILE(*crt0.o) .rodata)
.rodata        0x00008436      0xc  src\sample_gcc6.o
*(.rodata.*)
              [!provide]      PROVIDE ( __END_rodata, .)
              0x00008436      PROVIDE ( __START_data_lma, LOADADDR (.data))
              [!provide]      PROVIDE ( __END_data_lma, (LOADADDR (.data) + SIZEOF (.data)))
              [!provide]      PROVIDE ( __START_stack, 0x7c0)

```



```
OUTPUT(sample_gcc6.elf elf32-c17)
```

```
.stab      0x00000000  0xf78
.stab      0x00000000  0x570 crt0.o
.stab      0x00000570  0x21c src\sample_gcc6.o
               0x228 (size before relaxing)
.stab      0x0000078c 0x24 C:\EPSON\GNU17V3\gcc6\lib\24bit\libgcc.a(emu_copro_process.o)
               0x30 (size before relaxing)
.stab      0x000007b0 0x7c8 C:\EPSON\GNU17V3\gcc6\lib\24bit\libg.a(debuglib.o)
               0x7d4 (size before relaxing)

.comment    0x00000000  0x11
.comment    0x00000000  0x11 crt0.o
               0x12 (size before relaxing)
.comment    0x00000011  0x12 src\sample_gcc6.o
.comment    0x00000011  0x12 C:\EPSON\GNU17V3\gcc6\lib\24bit\libc.a(puts.o)
.comment    0x00000011  0x12 C:\EPSON\GNU17V3\gcc6\lib\24bit\libc.a(putc.o)
.comment    0x00000011  0x12 C:\EPSON\GNU17V3\gcc6\lib\24bit\libg.a(debuglib.o)

.stabstr    0x00000000  0xecb
.stabstr    0x00000000  0xecb crt0.o
```

As specified by the linker script, the `.bss` section starts at address `0x00000000`, followed by the `.data` section. The `.vector` section then starts at address `0x00008000`, and the `.text` section starts at address `0x00008080`, followed by the `.data` section initial value and `.rodata` section.

The symbol `__START_stack` is located at address `0x7c0`, as specified by the linker script.

Likewise, for the other symbols `__START_bss`, `__END_bss`, `__START_data`, `__END_data`, `__START_vector`, `__END_vector`, `__START_text`, `__END_text`, `__START_rodata`, `__END_rodata`, `__START_data_lma`, and `__END_data_lma`, the location address specified by linker is indicated. Referring to these symbols indicates the address and size of the individual sections within the program.

The `.stab` and `.stabstr` sections are used to embed debugging information used by the debugger into the program. The contents of these sections are not loaded to the target.

7.5 Error/Warning Messages

Error and warning messages are displayed/output through the Standard Output (stdout).

In the **ld** linker, the following error and warning messages are added to the standard error messages of the gnu linker:

Table 7.5.1 Error messages

Error message	Description
Warning: out of range error.	The address of the symbol exceeds the 16-bit address (when -mpointer16 is specified) or 24-bit address space.
Error: The offset value of a symbol is over 24bit.	The address of the symbol exceeds the 24-bit address space.
Error: section <i>XXX</i> is not within 16bit address.	The address of the <i>XXX</i> section exceeds the 16-bit address space.
Error: section <i>XXX</i> is not within 24bit address.	The address of the <i>XXX</i> section exceeds the 24-bit address space.
Error: Input object file <objectfile> [included from <archivefile>] is not for C17.	The object file is not compatible with the C17.
Error: Input object file <objectfile> is not 16bit nor 24bit address mode.	The object file is neither in 16-bit or 24-bit mode.
Error: Cannot link 16bit object <objectfile16> [included from <archivefile16>] with 24bit object <objectfile24> [included from <archivefile24>]	Object files created in 16-bit pointer mode and object files created in 24-bit pointer mode cannot be linked.

7.6 Linker Script Generation Wizard

This IDE features a linker script generation wizard that creates linker script files.

7.6.1 Output File

● Linker script file

File format: Text file

File name: < *filename* >.x

Description: This file specifies the start of each section and other linkage information.

The default file name is the name of the project. Assign a new name, if you wish.

The generated linker script file can be selected in the [GNU17 Setting] page in the [Properties] dialog box of the project.

7.6.2 Starting and Terminating the Linker Script Generation Wizard

● Starting up the linker script generation wizard

Launch the linker script generation wizard by one of the following methods:

- (1) Select [GNU17 Linker Script] from the [New] shortcut in the toolbar.
- (2) Select [New] > [GNU17 Linker Script] from the [File] menu.
- (3) Select [New] > [Project...] > [C/C++] > [GNU17 Linker Script] from the [File] menu.
- (4) Select and right-click on the project, then select [New] > [Project...] > [C/C++] > [GNU17 Linker Script].

● Terminating the linker script generation wizard

Close the linker script generation wizard by one of the following methods:

- (1) Click the [Finish] button. In this case, a linker script file will be created.
- (2) Click the [Cancel] button. In this case, no linker script file will be generated.

7.6.3 Menu

● Linker Script File:

Enter a linker script file name. The default linker script file name is the project name. A linker script file named "< specified filename >.x" is created in the currently selected project folder.

● Entry routine:

Enter the program start address to be set by the ENTRY command. If you use the "crt0.o" startup processing library, select the [use crt0.o] checkbox and set "_start".

● MCU memory regions

Enter the memory regions (region name, start address, size) to be described by the MEMORY command. By default, the RAM and ROM regions for the MCU selected as the target model in the project properties are defined as iram and irom, respectively.

To change the ORIGIN or LENGTH setting, click and select the numeric value in [Start Address] or [Length] and enter a new value.

To add a new memory region, click the [Add Region] button. Clicking this button adds a new region named "regionX" in the list of memory regions. Edit this description.

To delete a memory region, select the region to delete and click the [Del Region] button.

● Output sections and their input patterns

Enter the output and input sections to be described by the SECTIONS command. By default, the default linker script input and output sections described in Section 7.4.1, "Default Linker Script" are defined.

To add a new output section, click the [Add Output Section] button. A new output section with an output section named ".sectionX" and a VMA region named "iram" will be defined and added.

To delete an output section, select the output section to delete and click the [Del Output Section] button. Note that deleting an output section will also delete the input section defined for that output section.

Double-clicking an output section opens the [Output Section Description] dialog box. Enter the following settings for the output section:

	Change location	Setting method
Output section name	Section name	Enter the name of the output section.
VMA region name	VMA region	Select a VMA region from the list. The list shows the regions defined in the "MCU memory regions".
LMA region name (Option)	LMA region	To use an LMA region, select the checkbox and choose an LMA region from the list. The list shows the regions defined in the "MCU memory regions".
Alignment value (Option)	Alignment	To set alignment, select the checkbox and choose a value from the list.
Fill value (Option)	Fill value	To set a value used for filling open areas, select the checkbox and enter a value.
NOLOAD setting (Option)	This section is not loaded at run time (NOLOAD).	To set the section as a NOLOAD section, select the checkbox.
OVERLAY setting (Option)	This section is overlay section (OVERLAY).	To set the section as an OVERLAY section, select the checkbox. The section will be combined as an OVERLAY section.

To add a new input section, select an output section and click the [Add Input Pattern] button. An input section will be added for the selected output section. To delete an input section, select the input section to delete and click the [Del Input Pattern] button.

Double-clicking an input section opens the [Input Section Description] dialog box. Enter the following settings for the input section:

	Change location	Setting method
Output section name	Output Section	Select an output section name from the list.
Input section name	Input Sections	Select an input section name from the list. To select multiple sections, enter the section names directly.
Input file name	Input Files	Enter the input file name.
Exclusion file name (Option)	Exclude Files	To set exclusion files, select the checkbox and enter the names of the files to exclude.
KEEP setting (Option)	This section should not be eliminated (KEEP).	To set the section as a KEEP section, select the checkbox.

For definitions that cannot be defined using the wizard, open and edit the generated linker script file in an editor.

7.7 Precautions

- When the linker is executed, an error message as shown below may appear.
ld: test.elf: Not enough room for program header, try linking with -N
This error occurs in the alignment check for the data segment. The linker's alignment check can be disabled with the -N option, so normally specify the -N option when invoking the linker.
- The object file names are case-sensitive. It is necessary to specify the exact same file name in the **ld** command line and the linker script file. If the upper/lower case is different, **ld** considers them as two different files.

Example:

Command line

```
ld -T sample.x -o sample.elf prg1.o prg2.o
```

Linker script file (sample.x)

```

:
.text 0xc00000:
{
PRG1.o (.text) ← PRG1.o must be changed to prg1.o
prg2.o (.text)
}
:

```

- Linking two or more library files (*.a) that contain the same function does not cause an error (no double linkage performed).
Note that an error occurs when two or more object files (*.o) that contain the same function are linked.
- If the located address, which is specified by a variable or the result of a calculation with a variable, is higher than the 24-bit limit (0xfffff) or lower than 0x0, the address bits that exceed 24 bits are masked with 0 and no error occurs.

Example: `xadd.a %r0,symbol-5`

If the `symbol` is located at address 0, the specified absolute address is $0 - 5 = 0xfffffb$ (-5). Therefore, this code will be assembled as `"xadd.a %r0,0xfffffb"`.

8 Debugger

This chapter describes how to use the debugger **gdb**.

8.1 Features

The debugger **gdb** is software used to debug a program after loading an elf-format object file created by the linker.

This debugger has the following features and functions:

- Debugs using the integrated development environment (IDE) debugging function.
- Can reference various types of data at one time, thanks to a multi-window facility.
- In addition to debugging programs using the ICDmini (S5U1C17001H), the debugger incorporates a software simulator function for debugging programs on a personal computer.
- Capable of C source and assembly source level debugging.
- Supports C source and assembler level single-stepping functions, in addition to continuous program execution.
- Supports hardware and software PC break functions.
- The EmbSysRegView function lets you refer to the peripheral circuit control register.
- The LCD panel simulator function simulates an LCD panel on a PC even if the actual device is not equipped with an LCD panel.

8.2 Input/Output Files

8.2.1 Input Files

● Object file

File format: elf format binary file

File name: `<filename>.elf`

Description: This is the elf format absolute object file created by the linker **ld**. This file is loaded in the debugger by using the `file` and `load` commands. Source display and symbolic debugging are made possible by loading an object file that contains debug information.

● Source files

File format: Text file

File name: `<filename>.c` (C source)

`<filename>.s` (assembly source)

Description: These are source files for the object file above, loaded in the debugger to generate source display.

● Startup command file

File format: Text file

File name: `<filename>.ini`

Description: This file contains a description of the debugging commands executed on startup. This file is loaded and executed by the startup option `-x`.

● ROM data

File format: Motorola (S1 to S3) format

File name: Desired filename (e.g., `.psa`, `.sa`, or `.saf`)

Description: This object file does not contain debug information created from an object file (`.elf`). Source level debugging is not performed for ROM data, since debugging information is not included. It is used with the `load` command to load programs and data to target MCU memory.

8.2.2 Output File

- Log file

File format: Text file

File name: gdb.txt

Description: The commands executed and execution results are output to this file. Output by executing the set logging on command within the startup command file.

8.3 Starting the Debugger

8.3.1 Startup Format

● General command line format

```
gdb [<startup option>]
```

The brackets [] denote that the specification can be omitted.

● Operation on IDE

- (1) Select [Debug Configurations...] on the [Run] menu to display the launch configuration dialog box. It can also be opened from the [Debug] button menu on the toolbar.
- (2) Select the C/C++ Application type and select the Debug configuration that corresponds to the project name.
- (3) Select the GDB command file within the project using GDB Command file on the Debugger tab.
 gdbsim.ini: When debugging in simulator mode
 gdbmini2.ini: When debugging using the ICDmini2 (S5U1C17001H2)
 gdbmini3.ini: When debugging using the ICDmini3 (S5U1C17001H3)
- (4) Click the [Debug] button to start debugging.

The debug start/launch configuration dialog box can be opened via the [Debug] button menu on the toolbar.

To use debugging on the **IDE** after launching the debugger, open the debug perspective.

● Selecting connect mode

gdb supports two connect modes, of which the mode used is set by the target command.

GNU17 projects created on the **IDE** include the startup command file and debugging configuration supporting the connect mode.

ICD Mini mode

In this mode, ICDmini (S5U1C17001H) is used to perform debugging. The program is executed on the target board.

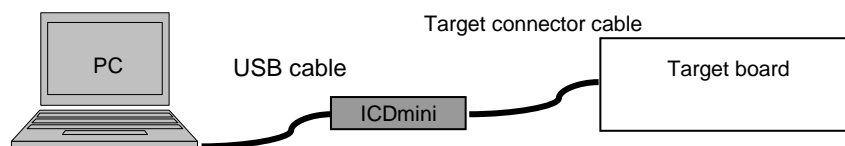


Figure 8.3.1. Example of debugging system using an ICD

Specification method

Command: (gdb)
 target icd icdminix

Specification in IDE:

Select the gdbminix.ini corresponding to the ICDmini used by the GDB command file on the Debugger tab.

To start in ICD Mini mode, connect the ICDmini to the host PC using a USB cable, connect the ICDmini and target board correctly, and turn on the power for these units.

[When using ICDmini3]

If the ICDmini3 is not connected to the RESET terminal on the target board and the EMU LED (red) on the ICDmini3 is turned off after starting the debugger, reset the target MCU while the LED is off. The EMU LED (red) is off for two seconds. The target board must be reset during this period.

Simulator (SIM) mode

In simulator mode, target program execution is simulated in the internal memory of a personal computer. No other tools are required. Note that ICDmini-dependent functions cannot be used in this mode.

Specification method

Command: (gdb) target sim

Specification in IDE:

Select [gdbsim.ini] for GDB command file on the Debugger tab.

8.3.2 Startup Options

The debugger includes startup options. These startup options need not be specified when executing on the **IDE**.

--command=<command filename>

-x <command filename>

Function: **Specifies a command file**

Explanation: When this option is specified, the debugger loads the specified command file at startup and executes the commands written in the file.

--cd=<directory path string>

Function: **Changes the current directory**

Explanation: When this option is specified, the debugger sets the specified path for the current directory at startup. If this option is omitted, the directory containing **gdb.exe** is assumed.

--directory=<directory path string>

Function: **Changes the source file directory**

Explanation: This option can be used to specify the directory containing the source files. Multiple instances of this option can be specified.

--model_path=<directory path string>

Function: **Sets the directory containing the model information file.**

Explanation: This option can be used to specify the directory containing the model-specific information file when the debugger performs operations corresponding to MCU models.
If this option is omitted, the sub-directory **mcu_model** in the directory containing **gdb.exe** is assumed.

--model=<MCU model name string>

Function: **Sets the MCU model name.**

Explanation: This option can be used to specify debugger operations corresponding to MCU models. To load a program to the Flash ROM inside the MCU, either specify this option or execute the **c17** model command. Loading to the Flash ROM is not possible unless you specify the MCU model.

8.3.3 Executing Command Files

You can use a command file to execute a series of debugging commands written in the file. To start debugging using IDE operations, use a command file at startup to execute the `c17model_path` command, `c17_model` command, `target` command, and `load` command.

● Creating a command file

Create a command file as a text file using a general-purpose editor, etc.

● Example of a command file

Only one command can be written per line.

Example:

<code>c17 model_path c:/EPSON/GNU17V3/mcu_model</code>	Specifies the model-specific information file directory.
<code>17 model 17W23@NOVCCIN</code>	Specifies the model name. (Voltage level 3.3 V)
<code>target icd icdmini3</code>	Connects the target MCU.
<code>load</code>	Loads a program.

● Loading/executing a command file

There are three ways to load and execute a command file:

1. Execution by a startup option

By specifying the `-x` option (or `—command` option) in the debugger startup command, you can execute one command file at debugger startup.

Example: `c:\EPSON\GNU17V3\gdb -x icdmini3.ini`

2. Setting using debugging configurations on IDE

Select [Run] > [Debug Configurations...] on the IDE menu, and specify the command file for GDB command file in the Debugger tab.

3. Execution by a command

A source command is available to execute a command file.

The source command loads a specified file and executes the commands in it in the order written.

Example: `(gdb) source gdbmini3.ini`

8.3.4 Quitting the Debugger

● Command operation

Debugging is terminated using the `quit` command input.

```
(gdb) quit
```

● IDE operation

Debugging can be terminated using any of the methods below.

The [Debug] view display will change to the terminated state after debugging has ended.

- Select [Terminate] on the [Run] menu
- Click the [Terminate] button in [Debug] view.
- Click the [Terminate] button in [Console] view.
- Select [Terminate] from the Context menu in [Debug] view.

Note: When using an ICDmini to debug a program, always be sure to close the debugger before turning off power to the ICDmini. Should you turn off power to the ICDmini while running the debugger, you will be unable to reconnect it.

8.4 Method of Executing Commands

The debug functions can be executed on the IDE debug perspective.

This section describes the method for executing commands without using the IDE. For command parameters and other details, see the explanation of each command described later in this manual.

8.4.1 Entering Commands From the Keyboard

Commands can be input if the debugger prompt "(gdb) " appears and the cursor after it is blinking.

Enter the debugging command here in lower case.

- **General command input format**

(gdb) command [parameter [parameter ... parameter]]

A space is required between the command and a parameter, and between parameters.

If you have entered an incorrect command by mistake, use the arrow (←, →), [Backspace], or [Delete] keys to correct it.

When you have finished entering a command, press the [Enter] key to execute the command.

Example: (gdb) continue (entry of command only)
(gdb) target icd icdmini3 (entry of command and parameters)

8.4.2 Parameter Input Format

● Numeric input

Parameters used to specify an address or data in a command must be entered in decimal (by default). To enter a parameter in hexadecimal, add 0x (or 0X) to the beginning of the value. Only characters 0 to 9, 'a' to 'f' and 'A' to 'F' are recognized as hexadecimal.

To specify an immediate address in a command that causes the program to break, add * to the beginning of the value, as shown below.

Example: (gdb) break *0xc00040

You need not add this asterisk for address parameters not preceded by * in the explanation of each command format.

● Specifying a source line number

For commands that cause the program to break, you can specify a breakpoint by source line number. However, this is limited to only when debugging an elf format object file that includes information on source line numbers.

To specify a line number, use the format shown below.

Filename:LineNo.

Filename Source file name

Filename: can be omitted when specifying a line number existing in the current file (one that includes code for the current PC).

LineNo. Line number

Line numbers can only be specified in decimal.

Example: main.c:100

● Address specification by a symbol

You can use a symbol to specify an address. However, this is limited to only when debugging an elf format object file that includes symbol information.

● Entering a file name

For file names in other than the current directory, always be sure to specify a path.

Only characters 'a' to 'z,' 'A' to 'Z,' 0 to 9, /, and _ can be used.

Drive names must be specified in /<drive name>/format, with / instead of \ used for delimiting the path.

Example: (gdb) file /c/EPSON/gnu17v3/sample/txt/sample.elf

8.5 Command Reference

8.5.1 List of Commands

Table 8.5.1.1 List of commands

Classification	Command	Operation	Supported modes	
			ICD Mini	SIM
Memory manipulation	c17 fb	Fill area (in bytes)	○	○
	c17 fh	Fill area (in 16 bits)	○	○
	c17 fw	Fill area (in 32 bits)	○	○
	x /b	Memory dump (in bytes)	○	○
	x /h	Memory dump (in 16 bits)	○	○
	x /w	Memory dump (in 32 bits)	○	○
	set {char}	Data input (in bytes)	○	○
	set {short}	Data input (in 16 bits)	○	○
	set {long}	Data input (in 32 bits)	○	○
	c17 mvb	Copy area (in bytes)	○	○
	c17 mvh	Copy area (in 16 bits)	○	○
	c17 mvw	Copy area (in 32 bits)	○	○
	c17 df	Save memory contents	○	○
Register manipulation	info reg	Display register	○	○
	set \$	Modify register	○	○
Program execution	continue	Execute continuously	○	○
	until	Execute continuously with temporary break	○	○
	step	Single-step (every line)	○	○
	stepi	Single-step (every mnemonic)	○	○
	next	Single-step with skip (every line)	○	○
	nexti	Single-step with skip (every mnemonic)	○	○
	finish	Quit function	○	○
CPU reset	c17 rst	Reset	○	○
	c17 rstt	Reset target	○	—
Interrupt	c17 int	Interrupt	—	○
	c17 intclear	Clear interrupt	—	○
Break	break	Set software PC break	○	○
	tbreak	Set temporary software PC break	○	○
	hbreak	Set hardware PC break	○	○
	thbreak	Set temporary hardware PC break	○	○
	delete	Clear break by break number	○	○
	clear	Clear break by break position	○	○
	enable	Enable breakpoint	○	○
	disable	Disable breakpoint	○	○
	ignore	Disable breakpoint with ignore counts	○	○
	info breakpoints	Display breakpoint list	○	○
	commands	Set command to execute at break	○	○
Symbol information	info locals	Display local symbol	○	○
	info var	Display global symbol	○	○
	print	Alter symbol value	○	○

Classification	Command	Operation	Supported modes	
			ICD Mini	SIM
File loading	file	Load debugging information	○	○
	load	Load program	○	○
Trace	c17 tm	Set trace mode	—	○
Other	set output-radix	Change variable display format	○	○
	set logging	Log output setting	○	○
	source	Execute command file	○	○
	target	Connect target	○	○
	detach	Disconnect target	○	○
	pwd	Display current directory	○	○
	cd	Change current directory	○	○
	c17 ttbr	Set TTBR	—	○
	c17 cpu	Set CPU type	—	○
	c17 chgclkmd	DCLK change mode	○	—
	c17 pwul	Unlock flash security password	○	—
	c17 help	Help	○	○
	c17 model_path	Model-specific information file directory setting	○	○
	c17 model	MCU model name setting	○	○
	c17 flv	Flash programming power setting	○	—
	c17 flvs	Flash programming power setting cancellation	○	—
	c17 stdin	Input of data using input/output functions	○	○
	c17 stdout	Output of data using input/output functions	○	○
	c17 lcdsim	LCD panel simulator setting	○	—
	quit	Quit debugger	○	○

Supported modes: ○= Can be used, — = Cannot be used.

* The GNU17V3 does not support commands other than those shown above.

8.5.2 Detailed Description of Commands

This chapter describes in detail each debugger command using the format shown below.

Command name (operation of command) [Supported modes]

A detailed description of each command begins with the command name in this format.

[Supported modes] shows such modes as [ICD Mini / SIM] in which the command can be used. You cannot use the command in modes other than those written here.

Basically, each command is described separately. However, two or more commands (belonging to the same operation group) that differ only slightly or which can be better understood when explained together are described collectively.

Operation

Explains the operation of the command.

Format

Shows the format in which the command is entered at the command prompt and the specifics of the parameters. Parameters enclosed in brackets [] can be omitted. Otherwise, no parameters can be omitted. The *italicized* characters denote parameters specified with numeric values or symbols.

Usage example

Shows an example of how to enter the command and the results of command execution, etc.

Notes

Describes limitations on use of the command or precautions to be taken when using the command.

Some commands have additional items other than those described above when needed for explanatory purposes.

8.5.3 Memory Manipulation Commands

c17 fb (fill area, in bytes)

c17 fh (fill area, in 16 bits)

c17 fw (fill area, in 32 bits)

[ICD Mini / SIM]

Operation

- c17 fb** Rewrites specified memory area with specified byte data.
- c17 fh** Rewrites specified memory area with specified 16-bit data.
- c17 fw** Rewrites specified memory area with specified 32-bit data.

Format

c17 fb *StartAddr EndAddr Data*
c17 fh *StartAddr EndAddr Data*
c17 fw *StartAddr EndAddr Data*

StartAddr: Start address of area to be filled (decimal, hexadecimal, or symbol)

EndAddr: End address of the area to be filled (decimal, hexadecimal, or symbol)

Data: The data to write (decimal or hexadecimal)

Conditions: $0 \leq \text{StartAddr} \leq \text{EndAddr} \leq 0\text{ffffff}$, $0 \leq \text{Data} \leq 0\text{xff}$ (**c17 fb**), $0 \leq \text{Data} \leq 0\text{xffff}$ (**c17 fh**),
 $0 \leq \text{Data} \leq 0\text{xffffffff}$ (**c17 fw**)

Usage example

Example 1

```
(gdb) c17 fb 0x0 0xf 0x1
Start address = 0x0, End address = 0xf, Fill data = 0x1 .....done
(gdb) x /16b 0x0          (memory dump command)
0x0: 0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01
0x8: 0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01
```

The entire memory area from address 0x0 to address 0xf is rewritten with byte data 0x01.

Example 2

```
(gdb) c17 fh 0x0 0xf 0x1
Start address = 0x0, End address = 0xe, Fill data = 0x1 .....done
(gdb) x /8h 0x0           (memory dump command)
0x0: 0x0001 0x0001 0x0001 0x0001 0x0001 0x0001 0x0001 0x0001
```

The entire memory area from address 0x0 to address 0xf is rewritten with 16-bit data 0x0001.
(This applies to when using little endian.)

Example 3

```
(gdb) c17 fw 0x0 0xf 0x1
Start address = 0x0, End address = 0xc, Fill data = 0x1 .....done
(gdb) x /4w 0x0           (memory dump command)
0x0: 0x00000001 0x00000001 0x00000001 0x00000001
```

The entire memory area from address 0x0 to address 0xf is rewritten with 32-bit data 0x00000001.
(This applies to when using little endian.)

Notes

- Writing in units of 16 bits or 32 bits is performed in little endian format.
- The data write memory section is aligned to boundary addresses according to the size of data.

```
(gdb) c17 fw 0x3 0x9 0x0
```

For example, when a write memory section is specified as shown above, and because start address 0x3 and end address 0x9 are not located on 32-bit data boundaries, both are aligned to boundary addresses by setting the 2 low-order bits to 00 (LSB = 0 for 16 bits). The following shows the actually executed command, where 32-bit data addresses 0x0 to 0x8 (byte data addresses 0x0 to 0xb) are rewritten with data 0x00000000.

```
(gdb) c17 fw 0x0 0x8 0x0
```

- If the specified address exceeds the 24-bit range, an error is assumed.
- Data parameters are only effective for the 8 low-order bits for `c17 fb`, 16 low-order bits for `c17 fh`, and 32 low-order bits for `c17 fw`, with excessive bits being ignored. For example, when data 0x100 is specified in `c17 fb`, it is processed as 0x00.
- If the end address is smaller than the start address, an error is assumed.

X (memory dump)

[ICD Mini / SIM]

Operation

Dumps memory contents (in hexadecimal) to a view. The data size, display start address, and display data counts can be specified.

Format

x [/ [*Length*] *Size*] [*Address*]

Length: Number of data items to display (in decimal)
1 when omitted.

Size: One of the following symbols that specify data size (in which units of data are displayed)

b In units of bytes

h In units of 16 bits

w In units of 32 bits (default)

i Disassemble

Address: Address from which to start displaying data (decimal, hexadecimal, symbol, or register name)

When omitted, the last address displayed when previously executing the **x** command is assumed.

The default address assumed at **gdb** startup is 0x0.

Conditions: $0 \leq \textit{Length} \leq 2147483647$, $0 \leq \textit{Address} \leq 0xffffffff$

Display

Memory contents are displayed as described below. (*Size* for b/h/w)

Address[<*Symbol*>]: *Data* [*Data* ...]

Address: The start address of each line of data is displayed in hexadecimal.

Symbol: When the address displayed at the beginning of a line has a symbol or label defined for it, the name of that symbol or label is displayed. When an intermediate address of a function or variable is specified, the specified symbol and a decimal offset (<*Symbol*+*n*>) are also displayed.

Data: Up to 16 bytes of data starting from *Address* are displayed on one line.

Disassemble is displayed as described below. (*Size* for i)

Address[<*Symbol*>]: *insn* [*ext insn*]

Address: The address of each line of code is displayed in hexadecimal notation.

Symbol: If the address displayed at the beginning of a line has a symbol or label defined for it, the name of that symbol is displayed. If an intermediate address of a function or variable is specified, the specified symbol and a decimal offset (<*Symbol*+*n*>) are also displayed.

insn: The assembler basic command starting from the *Address* is displayed.

ext insn: The *ext* extension command is displayed if present.

Usage example**■ Example 1**

```
(gdb) x
0x0: 0x00000000
```

When all parameters are omitted after startup, the command is executed as "**x** /1w 0x0".

■ Example 2

```
(gdb) x /b 0
0x0 <i>: 0xe3
(gdb) x /b 1
0x1 <i+1>: 0xa1
```

When *Size* is specified but *Length* omitted, one unit of data equal to the specified data size is displayed.

The letter **i** is a symbol defined at address 0x0. If any address other than the address at the beginning of a variable, etc. is specified, <*symbol*+*offset*> is displayed as the symbol.

■Example 3

```
(gdb) x /16h _START_text
0xc00000 <_START_text>: 0x0004 0x00c0 0xc020 0x6c0f 0xa0f1 0xc000 0xc000 0x6c0f
0xc00010 <boot+12>:      0xc000 0xc000 0x1c04 0xdff8 0xdfff 0x1ef5 0x0200 0x6c04
```

When *Length* is specified, the specified amount of data is displayed.

When a code area is displayed, *<label+offset>* is displayed as the symbol, even for addresses without defined symbols.

■Example 4

```
(gdb) x /4w 0
0x0 <i>: 0x00001ae3 0x00000000 0x00000000 0x00000000
(gdb) x
0x10:    0x00000000
(gdb) x
0x14:    0x00000000
```

When the *x* command is executed once, you can dump and display a single unit of data (having the same size as that of the previous address) from the address following the previous address by simply entering *x*.

■Example 5

```
(gdb) x /w &i
0x0 <i>: 0x00000010
(gdb) x /w i
0x10:    0x00000000
```

When specifying an address with a data symbol that references the assigned address, add *&* when you enter the command.

When only specifying a symbol, note that its data value is used as the address. In such case, *&* need not be added because labels in program code indicate assigned addresses.

■Example 6

```
(gdb) x /10i boot
0x8004 <boot>:      ext      0x20
0x8006 <boot+2>:    ld.a     %sp,0x0      sld.a   %sp,0x1000
0x8008 <boot+4>:    call    0x1        call   0x1    (0x00800C)    <main>
0x800a <boot+6>:    nop
0x800c <main>:      ld       %r0,[0x0]    ld       %r0,[0x0]    <p1>
0x800e <main+2>:    ld       %r1,[0x2]    ld       %r1,[0x2]    <p2>
0x8010 <main+4>:    ext      0x0
0x8012 <main+6>:    ld       %r2,0x64     sld      %r2,0x64
0x8014 <main+8>:    call    0x16     call   0x16    (0x008042)    <memcpy>
0x8016 <main+10>:   ld       %r0,[0x0]    ld       %r0,[0x0]    <p1>
```

The ten commands specified are displayed.

Notes

- Memory contents are displayed in little endian format.
- Even if the specified address is not a boundary address conforming to the data size, the *x* command starts displaying memory contents from that address.
- If the specified address exceeds the 24-bit range, an error is assumed.
- An error does not occur even if a value of 2147483648 or greater is specified for *Length*. The value for *Length* is set to 2147483647.

set { } (data input)

[ICD Mini / SIM]

Operation

Writes specified data to a specified address.

Format

set {Size}Address=Data

Size: One of the following symbols that specify data size

char In units of bytes
short In units of 16 bits (default)
int In units of 16 bits
long In units of 32 bits

Address: Address to which to write data (decimal, hexadecimal, or symbol)

Data: The data to write (decimal, hexadecimal, or symbol)

Conditions: $0 \leq \text{Address} \leq 0\text{ffffff}$, $0 \leq \text{Data} \leq 0\text{xff}$ (set {char}),
 $0 \leq \text{Data} \leq 0\text{xffff}$ (set {short/int}),
 $0 \leq \text{Data} \leq 0\text{xffffffff}$ (set {long})

Usage example**Example 1**

```
(gdb) set {char}0x1000=0x55
(gdb) x /b 0x1000
0x1000: 0x55
```

Byte data 0x55 is written to address 0x1000.

Example 2

```
(gdb) set {short}0x1000=0x5555
(gdb) x /h 0x1000
0x1000: 0x5555
```

16-bit data 0x5555 is written to address 0x1000.

Example 3

```
(gdb) set {long}&i=0x55555555
(gdb) x /w &i
0x0 <i>: 0x55555555
```

32-bit data 0x55555555 is written to long variable i.

Notes

- Writing in units of 16 bits or 32 bits is performed in little endian format.
- If the specified address exceeds the 24-bit range, an error is assumed.
- Data parameters are only effective for the 8 low-order bits for set {char}, 16 low-order bits for set {short} and set {int}, and 32 low-order bits for set {long}, with excessive bits being ignored.
 For example, when data 0x100 is specified in set {char}, it is processed as 0x00.

c17 mvb (copy area, in bytes)**c17 mvh** (copy area, in 16 bits)**c17 mvw** (copy area, in 32 bits)

[ICD Mini / SIM]

Operation

- c17 mvb** Copies the content of a specified memory area to another area in units of bytes.
c17 mvh Copies the content of a specified memory area to another area in units of 16 bits.
c17 mvw Copies the content of a specified memory area to another area in units of 32 bits.

Format

c17 mvb *SourceStart SourceEnd Destination*
c17 mvh *SourceStart SourceEnd Destination*
c17 mvw *SourceStart SourceEnd Destination*

SourceStart: Start address of area from which to copy (decimal, hexadecimal, or symbol)

SourceEnd: End address of area from which to copy (decimal, hexadecimal, or symbol)

Destination: Start address of area to which to copy (decimal, hexadecimal, or symbol)

Conditions: $0 \leq \text{SourceStart} \leq \text{SourceEnd} \leq 0\text{xfffff}$, $0 \leq \text{Destination} \leq 0\text{xfffff}$

Usage example**Example 1**

```
(gdb) x /16b 0
0x0: 0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07
0x8: 0x08 0x09 0x0a 0x0b 0x0c 0x0d 0x0e 0x0f
(gdb) c17 mvb 0x0 0x7 0x8
Start address = 0x0, End address = 0x7, Destination address = 0x8 .....done
(gdb) x /16b 0
0x0: 0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07
0x8: 0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07
```

The content of a memory area specified by addresses 0x0 to 0x7 is copied to an area beginning with address 0x8.

Example 2

```
(gdb) x /4w 0
0x0 <i>: 0x00000000 0x11111111 0x22222222 0x33333333
(gdb) c17 mvw i i i+4
Start address = 0x0, End address = 0x0, Destination address = 0x4 .....done
(gdb) x /4w 0
0x0 <i>: 0x00000000 0x00000000 0x22222222 0x33333333
```

The content of long variable *i* is copied to an area located four bytes after that int variable.

Notes

- When the source and destination have different endian formats, the data formats are converted when copied from the source to the destination.
- If the specified address exceeds the 24-bit range, an error is assumed.
- In **c17 mvh** and **c17 mvw**, addresses are adjusted to boundary addresses conforming to the data size. This is accomplished by processing the LSB address bit as 0 for **c17 mvh** and the 2 low-order address bits as 00 for **c17 mvw**.
- If the end address at the source is smaller than its start address, an error is assumed.
- When the start address at the destination is smaller than that of the source, data is copied sequentially beginning with the start address. Conversely, when the start address at the destination is larger than that of the source, data is copied sequentially beginning with the end address. Therefore, data is always copied even when the specified destination address exists within the source area.
- If the end address at the destination exceeds 0xfffff, data is only copied only up to 0xfffff.

c17 df (save memory contents)

[ICD Mini / SIM]

Operation

Outputs the specified range of memory contents to a file in binary, text, or Motorola S1/S2/S3 format.

Format

c17	df	<i>StartAddr</i>	<i>EndAddr</i>	<i>Type</i>	<i>Filename</i>	[Append]
------------	-----------	------------------	----------------	-------------	-----------------	----------

StartAddr: Start address (decimal, hexadecimal, or symbol)

EndAddr: End address (decimal, hexadecimal, or symbol)

Type: One of the following values that specify the type of file

- 1 Binary file
- 2 Text file
- 3 Motorola S1 file
- 4 Motorola S2 file
- 5 Motorola S3 file

Filename: File name

Append: **a** Append mode enabled

If *Type* = 1, dump data is appended to the end of a binary file when it is output.
 If *Type* = 2, dump data is appended to the end of a text file when it is output.
 If *Type* = 3, 4, or 5, no footer records are appended to the end of a Motorola file.

f Append mode enabled

If *Type* = 1, dump data is appended to the end of a binary file when it is output.

If *Type* = 2, dump data is appended to the end of a text file when it is output.

If *Type* = 3, 4, or 5, a footer record is appended to the end of a Motorola file.

If this specification is omitted, a new file is created.

Conditions: $0 \leq StartAddr \leq EndAddr \leq 0xffff$ (for Motorola S1 files)

$0 < StartAddr < EndAddr < 0xffffffff$ (for binary/text/Motorola S2/S3 files)

Usage example

■ Example 1

```
(gdb) c17 df 0x0 0xf 2 dump.txt
```

```
Start address = 0x0, End address = 0xf, File type = Text
Processing 00000000-0000000F address.
```

Contents at addresses 0x0–0xf are written to file "dump.txt" in text format.

(Contents of dump.txt)

addr	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
00000000	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F

■ Example 2

```
(gdb) c17 df 0x80000 0x80103 5 dump.mot
```

Start address = 0x80000, End address = 0x80103, File type = Motorola-S3

Processing 00080000-00080103 address.

Contents at addresses 0x80000–0x80103 are written to file "dump.mot" in Motorola S3 format.

In Motorola S3 format, data is output 32 bytes per line. If one line is less than 32 bytes, the number of bytes for the address range specified will be written.

(Contents of dump.mot)

S3250008000094D4BA020FCA086120800961881C6F0AA4D4BA020FCA086120800961881C6F0AA8
S3250008002008D3730A00000000000000000000000008D3730A09CA026139A505610CD309613F

S30800080100A4D5BABBB

S70500000000FA

■Example 3

```
(gdb) c17 df 0x10 0x1f 2 dump.txt a
```

The contents of addresses 0x10–0x1f are appended in text form to the end of the file "dump.txt" when it is output.

■Example 4

```
(gdb) c17 df 0x1000 0x1fff 5 dump.mot a ; Footer is not output. (First)
```

```
(gdb) c17 df 0x3000 0x3fff 5 dump.mot a ; Footer is not output.
```

```
(gdb) c17 df 0x5000 0x5fff 5 dump.mot a ; Footer is not output.
```

```
(gdb) c17 df 0x7000 0x7fff 5 dump.mot f ; Footer is not output. (Last)
```

The contents of addresses 0x1000–0x7fff (every 0x1000 addresses) are written out in Motorola S3 format to the file "dump.mot".

If no *Append* parameters exist or the parameter 'f' is specified, a footer record is output to a Motorola S3 format file.

Notes

- If the specified address exceeds the 24-bit range, an error is assumed.
- If the end address is smaller than the start address, an error is assumed.

8.5.4 Register Manipulation Commands

info reg (display register)

[ICD Mini / SIM]

Operation

Displays the contents of the CPU registers.

Format

info reg [*RegisterName*]

RegisterName: Name of register to display (specified in lowercase letters)

r0-r7, sp, pc, psr

If the above is omitted, the contents of all registers are displayed.

Display

Register contents are displayed as described below.

Register *Hexadecimal* *Decimal*

Register: This is a register name.

Hexadecimal: Shows the register value in hexadecimal.

Decimal: Shows the register value in decimal.

Usage example

■ Example 1

```
(gdb) info reg r1
r1                0xaaaaaa    1184810
(gdb) info reg pc
pc                0x4090      16528
```

When a register name is specified, only the content of that register is displayed.

■ Example 2

```
(gdb) info reg
r0                0xd20        3360
r1                0xaaaaaa    1184810
r2                0xaaaaaa    1184810
r3                0xaaaaaa    1184810
r4                0x690        1680
r5                0xaaaaaa    1184810
r6                0x0          0
r7                0xaaaaaa    1184810
sp                0x7f8        2040
pc                0xc00030    12582960
psr               0x2          2
```

Notes

Be sure to specify register names in lowercase letters. Using uppercase letters for register names or specifying nonexistent register names results in an error.

set \$ (modify register)

[ICD Mini / SIM]

Operation

Changes the values of the CPU registers.

Format

set \$RegisterName=Value

RegisterName: Name of register to change (specified in lowercase letters)

r0-r7, sp, pc, psr

Value: 24-bit data to set in the register (decimal, hexadecimal, or symbol)

Conditions: $0 \leq \text{Value} \leq 0\text{xfffff}$

Usage example

```
(gdb) set $r1=0x10000
(gdb) info reg r1
r1          0x10000  65536
(gdb) set $pc=main
```

In addition to numerals, symbols can also be used to set values.

Notes

- If the specified value exceeds the 24-bit range, the 24 low-order bits only will be effective.
- The contents of the set values are not checked internally. No errors are assumed even when values other than 16-bit or 32-bit boundary addresses are specified for PC or SP, respectively. However, when the registers are actually modified, values are forcibly adjusted to boundary addresses by truncating the lower bits.

8.5.5 Program Execution Commands

continue (execute continuously)

[ICD Mini / SIM]

Operation

Executes the target program from the current PC address.

The program is run continuously until it is made to break by one of the following causes:

- Already set break conditions are met.
- The [Suspend] button is clicked.

When reexecuting a target program halted because break conditions have been met, you can specify to disable the current breakpoint the specified number of times.

Format

```
continue [IgnoreCount]  
cont [IgnoreCount]      (abbreviated form)
```

IgnoreCount: Specifies the number of breaks (decimal or hexadecimal)

The program is run continuously until break conditions are met the specified number of times.

Usage example

Example 1

```
(gdb) continue  
Continuing.
```

```
Breakpoint 1, main () at main.c:13
```

When **continue** is executed with *IgnoreCount* omitted, the target program starts running from the current PC address and stops the first time break conditions are met.

Example 2

```
(gdb) cont 5  
Breakpoint 1, main () at main.c:13
```

Because value 5 is specified for *IgnoreCount*, break conditions that have been met four times (= 5 - 1) since the program started running are ignored, and the program breaks when break conditions are met the fifth time. In this example, the target program is restarted after being halted at the PC breakpoint (break 1) set at line 13 in `main.c`, and the program stops upon the fifth hit at that PC breakpoint.

The same effect is obtained by executing the following command:

```
(gdb) ignore 1 4  
(gdb) continue
```

Notes

- To run the program from the beginning, execute `c17 rst` (reset) before the **continue** command.
- The **continue** command with *IgnoreCount* specified can be executed on condition that the target program has been executed at least once and is currently halted because break conditions are met. In this case, a break caused by the [Suspend] button is not assumed since break conditions are met. If *IgnoreCount* is specified while the target program has never been made to break once, the specification is ignored.
- If the target program has been halted by one cause of a break, and the **continue** command is executed with *IgnoreCount* specified after clearing that break setting, an error is assumed. The same applies when other break conditions have been set.
- If break conditions other than the one that stopped the target program must be ignored a specified number of times, specify break conditions and the number of times that a break hit is to be ignored in the **ignore** command. Then execute the **continue** command without any parameters.

until (execute continuously with temporary break)

[ICD Mini / SIM]

Operation

Executes the target program from the current PC address.

A temporary break can be specified at one location, causing the program to stop before executing that breakpoint. A hardware PC break is used for this temporary break, which is cleared when the program breaks once. When a temporary break is specified, assembly sources other than the C source are executed continuously.

If the program does not pass the breakpoint set (a miss), the program runs continuously until made to break by one of the following causes:

- Other set break conditions are met.
- The [Suspend] button is clicked.
- Control is returned to a higher level from the current level (within the function).
- There is no assembly source or source information (in which case, only the current instruction is executed).

Format

until *Breakpoint*

Breakpoint: Temporary breakpoint

Can be specified by one of the following:

- Function name
- Source file name:line number, or line number only
- *Address (decimal, hexadecimal, or symbol)

Conditions: $0 \leq \text{address} \leq 0\text{xffffffff}$

Usage example**Example 1**

```
(gdb) until main
main () at main.c:10
```

The target program is run with a temporary break specified by a function name.

The program breaks before executing the first C instruction in `main()` (that is expanded to mnemonic). The PC on which the program has stopped displays the start address of that instruction (i.e., address of first mnemonic expanded).

Example 2

```
(gdb) until main.c:10
main () at main.c:10
```

The target program is run with a temporary break specified by line number. Although the breakpoint here is specified in "source file name:line number" format when the breakpoint is to be set in the C source containing the current PC address, it can be specified by simply using a line number like "until 10". For assembly sources, a source file name is always required. When this command is executed, the program breaks before executing the C instruction on line 10 in `main.c`. The PC on which the program has stopped displays the start address of that instruction (i.e., address of first mnemonic expanded). If no instructions exist on line 10 with actual code (i.e., not expanded to mnemonic), the program breaks at the beginning of the first instruction encountered with actual code thereafter.

Example 3

```
(gdb) until *0xc0001e
main () at main.c:10
```

The target program is run with a temporary break specified by address. The program breaks before executing the instruction stored at that address location. A symbol can also be used, as shown below.

```
(gdb) until *main
main () at main.c:7
```

Note that adding an asterisk (*) causes even the function name to be regarded as an address.

Notes

- To run the program from the beginning, execute `c17 rst` (reset) before the `until` command.
- If the location set as a temporary breakpoint is a C source line that does not expand to mnemonic, the program does not break at that line. The program breaks at the address of the first mnemonic executed thereafter.
- No temporary breakpoints can be set on the following lines, because an error is assumed.
 - Extended instruction lines (except for the `ext` instruction at the beginning)
 - Delayed instruction lines (next line after a delayed branch instruction)
- If temporary breakpoints are specified using a nonexistent function name or line number, an error is assumed.
- If temporary breakpoints are specified by an address value that exceeds the 24-bit range, an error is assumed.
- When specifying temporary breakpoints by address value and the address is specified with an odd value, the specified address is adjusted to the 16-bit boundary by assuming `LSB = 0`.

step (single-step, every line)

stepi (single-step, every mnemonic)

[ICD Mini / SIM]

Operation

Single-steps the target program from the current PC address. Lines and instructions in the called functions or subroutines also are single-stepped.

step: Single-steps the program by executing one source line at a time. In C sources, one line of C instruction (all multiple expanded mnemonics) are executed as one step. In assembly sources, instructions are executed the same way as for **stepi**.

stepi: Single-steps the program by executing one assembler instruction (in mnemonic units) at a time.

In addition to one line or instruction, a number of steps to execute can also be specified.

However, even before all specified steps are completed, the program may be halted by one of the following causes:

- Already set break conditions are met.
- The [Suspend] button is clicked.

Format

step [*Count*]

stepi [*Count*]

Count: Number of steps to execute (decimal or hexadecimal)

One step is assumed if omitted.

Conditions: $1 \leq \text{Count} \leq 0x7ffffff$

Usage example

■ Example 1

(gdb) **step**

The source line displayed on the current PC is executed.

■ Example 2

(gdb) **stepi**

The instruction (in mnemonic units) is executed at the address displayed on the current PC.

■ Example 3

(gdb) **step 10**

sub (k=5) at main.c:20

Ten lines are executed from the source line displayed on the current PC.

■ Example 4

(gdb) **stepi 10**

main () at main.c:13

Ten instructions (in mnemonic units) are executed from the address displayed on the current PC.

Notes

- The program cannot be single-stepped from an address that does not have source information (i.e., debugging information included in the object). The program can be run continuously, however, by using the `continue` command.
- To run the program from the beginning, execute `c17 rst` (reset) before **step** or **stepi**.
- Even with **stepi**, ext-based extended instructions are executed collectively (i.e., entire extended instruction set consisting of two or three instructions) as one step.
- Interrupts are accepted even while single-stepping the program.
Similarly, the `halt` and `slp` instructions are executed while single-stepping the program, causing the CPU to enter standby status. The CPU exits standby status when an external interrupt is generated. Clicking the [Suspend] button also releases the CPU from standby mode.

next (single-step with skip, every line)

nexti (single-step with skip, every mnemonic)

[ICD Mini / SIM]

Operation

Single-steps the target program from the current PC address. The basic operations here are the same as with `step` and `stepi`, except that when a function or subroutine call is encountered, all lines or instructions in the called function or subroutine are executed successively as one step until returning to a higher level.

next: Single-steps the program by executing one source line at a time. In C sources, one line of C instruction (all multiple expanded mnemonics) are executed as one step. In assembly sources, instructions are executed the same way as for `nexti`.

nexti: Single-steps the program by executing one assembler instruction (in mnemonic units) at a time.

In addition to one line or instruction, a number of steps to execute can also be specified.

However, even before all specified steps are completed, the program may be halted by one of the following causes:

- Already set break conditions are met.
- The [Suspend] button is clicked.

Format

next [*Count*]

nexti [*Count*]

Count: Number of steps to execute (decimal or hexadecimal)
One step is assumed if omitted.

Conditions: $1 \leq \textit{Count} \leq 0x7ffffff$

Usage example

■ Example 1

(gdb) **next**

The source line displayed on the current PC is executed. When the source is a function or subroutine call, the function or subroutine called is also executed until returning to a higher level.

■ Example 2

(gdb) **nexti**

The instruction (in mnemonic units) is executed at the address displayed on the current PC. When the instruction is a subroutine call, the subroutine called is also executed until returning to a higher level.

■ Example 3

(gdb) **next 10**

sub (k=5) at main.c:20

Ten lines are executed from the source line displayed on the current PC.

■ Example 4

(gdb) **nexti 10**

main () at main.c:13

Ten instructions (in mnemonic units) are executed from the address displayed on the current PC.

Notes

- The program cannot be single-stepped from an address that does not have source information (i.e., debugging information included in the object). The program can be run continuously, however, by using the `continue` command.
- To run the program from the beginning, execute `c17 rst` (reset) before `next` or `nexti`.
- Even with `nexti`, `ext`-based extended instructions are executed collectively (i.e., entire extended instruction set consisting of two or three instructions) as one step.
- Interrupts are accepted even while single-stepping the program.
Similarly, the `halt` and `slp` instructions are executed while single-stepping the program, causing the CPU to enter standby mode. The CPU exits standby mode when an external interrupt is generated. Clicking the [Suspend] button also releases the CPU from standby mode.

finish (finish function)

[ICD Mini / SIM]

Operation

Executes the target program from the current PC address and causes it stop upon returning from the current function to a higher level. The instruction at the return position is not executed.

Even before a return, however, the program may be halted by one of the following causes:

- Already set break conditions are met.
- The [Suspend] button is clicked.

Format

finish

Usage example

(gdb) **finish**

The target program is executed from the current PC address and halted after a return.

Notes

When the `finish` command is executed at the highest level (e.g., boot routine), the program does not stop. If no breaks are set, use the [Suspend] button to halt the program.

8.5.6 CPU Reset Commands

c17 rst (reset)

[ICD Mini / SIM]

Operation

Resets the CPU.
 Resets the target in ICD Mini mode.
 As a result, the CPU is reset to its initial state as shown below.

- (1) Internal registers of the CPU
 - r0–r7: 0x000000
 - pc: Boot address (reset vector in the trap table)
 - sp: 0xffffc
 - psr: 0x00 (IL = 000, IE = 0, CVZN = 0000)
- (2) The execution counter is cleared to 0.

Format

```
c17 rst
```

Usage example

```
(gdb) c17 rst
```

The CPU is reset.

Notes

- The contents of memory and debugging status of break and trace are not reset.
- When using **gdb** in ICD Mini mode, the bus status and I/O status are retained.

c17 rstt (reset target)

[ICD Mini]

Operation

Outputs the reset signal to the reset input pin on the target board.

Format

c17 rstt

Usage example

```
(gdb) c17 rstt  
TARGET resetting ..... done
```

The target is reset.

Message when target resetting fails:

```
TARGET resetting ..... failure
```

Notes

- The **c17 rstt** command can only be used in ICD Mini mode.
- To execute this command, a reset input pin is required on the target board.

8.5.7 Interrupt Commands

c17 int (interrupt)

[SIM]

Operation

Simulates the generation of an interrupt.

When an interrupt number is specified by this command, the specified interrupt is generated at next program startup.

Format

c17 int [*No Level*]

No: Interrupt number (decimal, hexadecimal, or symbol)

Level: Interrupt priority level (decimal, hexadecimal, or symbol)

Conditions: $0 \leq No \leq 0x1f$, $0 \leq Level \leq 7$

Usage example

■ Example 1

(gdb) **c17 int**

If no parameters are specified, an NMI is generated.

■ Example 2

(gdb) **c17 int 3 6**

Any maskable interrupt number and its priority level can be set.

Notes

- The **c17 int** command can only be used in simulator mode.
- Make sure the interrupt number is specified from 0 to 31. If this range is exceeded, an error is assumed.
- Make sure the interrupt priority level is specified from 0 to 7. If this range is exceeded, an error is assumed.
- TTBR is effective even in simulator mode.

c17 intclear (clear interrupt)

[SIM]

Operation

Simulates canceling interrupts.
The interrupt specified by the interrupt number is cleared.

Format

c17 intclear [*No*]

No: Interrupt number (decimal, hexadecimal, or symbol)

Conditions: $0 \leq No \leq 0xf$

Usage example

```
(gdb) c17 int 3 6
(gdb) continue
(gdb) c17 intclear 3
```

Cancels the interrupt of interrupt number 3.

Notes

- The `c17 intclear` command can only be used in simulator mode.
- Make sure the interrupt number is specified from 0 to 31. If this range is exceeded, an error is assumed.

8.5.8 Break Setup Commands

break (set software PC break)

tbreak (set temporary software PC break)

[ICD Mini / SIM]

Operation

Sets a software PC breakpoint. Up to 200 software PC breakpoints can be set. However, if breakpoints are set on the ROM on the target board in ICD Mini mode, this command functions in the same way as the `hbreak/tbreak` command and sets hardware PC breakpoints.

If the PC matches the address set during program execution, the program breaks before executing the instruction at that address. A breakpoint can be set using a function name, line number, or address.

The `break` and `tbreak` commands are functionally the same. The following describes the difference:

break: The breakpoints set by `break` are not cleared by a break that occurs when the set point is reached during program execution.

tbreak: The breakpoints set by `tbreak` are cleared by one occurrence of a break at the set point.

Format

break [*Breakpoint*]

tbreak [*Breakpoint*]

Breakpoint: Breakpoint

A breakpoint can be specified with one of the following:

- Function name
- Source file name:line number or line number only
- *Address (decimal, hexadecimal, or symbol)

When omitted, a breakpoint is set at the address displayed on the current PC.

Conditions: $0 \leq \text{address} \leq 0\text{xfffffe}$

Usage example

Example 1

```
(gdb) break main
Breakpoint 1 at 0xc0001e: file main.c, line 10.
(gdb) continue
Continuing.
Breakpoint 1, main () at main.c:10
```

A software PC breakpoint is set at the position specified using a function name.

When the target program is run, it breaks before executing the first C instruction (expanded to mnemonic) in `main()`. The PC on which the program has stopped displays the start address of that instruction (i.e., address of first mnemonic expanded).

Example 2

```
(gdb) tbreak main.c:10
Breakpoint 1 at 0xc0001e: file main.c, line 10.
```

A temporary software PC breakpoint is set at the position specified with a line number. Although the breakpoint here is specified in "source file name:line number" format when the breakpoint is to be set in the C source containing the current PC address, it can be specified by simply using a line number like "`tbreak 10`". For assembly sources, a source file name is always required.

If no instructions exist on the specified line with actual code (i.e., not expanded to mnemonic), a breakpoint is set at the beginning of the first instruction encountered with actual code thereafter.

When the target program is run, it breaks before executing the C instruction on line 10 in `main.c`. The PC on which the program has stopped displays the start address of that instruction (i.e., address of first mnemonic expanded). If no instructions exist on line 10 with actual code, the program breaks at the beginning of the first instruction encountered with actual code thereafter. Because the breakpoint is set by `tbreak`, it is cleared after a break.

Example 3

```
(gdb) break *0xc0001e
```

Note: breakpoint 1 also set at pc 0xc0001e.

Breakpoint 2 at 0xc0001e: file main.c, line 10.

A software PC breakpoint is set at the position specified using an address.

When the target program is run, it breaks before executing the instruction at that address. A symbol can also be used, as shown below.

```
(gdb) tbreak *main
```

Breakpoint 3 at 0xc0001c: file main.c, line 7.

Note that adding an asterisk (*) causes even a function name to be regarded as an address.

Breakpoint management

The breakpoints that you set are sequentially assigned break numbers beginning with 1, regardless of the types of breaks set (see the examples above). These numbers are required to disable/enable or delete breakpoints individually at a later time. Even when you delete breakpoints, the breakpoint numbers are not moved up (to reuse deleted numbers) until after you quit the debugger.

To manipulate the breakpoints you set, use the following commands:

<code>disable</code>	Disables a breakpoint.
<code>enable</code>	Enables a breakpoint.
<code>delete or clear</code>	Deletes a breakpoint.
<code>ignore</code>	Specifies the number of times a break is disabled.
<code>info breakpoints</code>	Displays a list of breakpoints.

For details, see the description of each command.

Notes

- Software PC breakpoints can be set at up to 200 locations.
If this limit is exceeded, an error is assumed. Note that this break count includes the software PC breakpoints used by the debugger in other functions.
- C source lines that are not expanded to mnemonic cannot be specified as a location at which to set a software PC breakpoint. Specifying such a C line sets a software PC breakpoint at the address of the first instruction to be executed next.
- When a function name or the beginning C source line in a function is specified as the position where to set a software PC breakpoint, the program execution will break at the start address of the first C source (i.e., instruction to be expanded to mnemonic) in the function. Although a `ld` instruction to save register contents is inserted at the beginning of the function during compilation, this instruction is executed before the program breaks. To make the program break before executing this instruction, specify a software PC breakpoint using the address value of that instruction.
- No software PC breakpoints can be set at the following lines.
 - Extended instruction lines (except for the `ext` instruction at the beginning)
 - Delayed instruction lines (next line after a delayed branch instruction)
- If software PC breakpoints are specified using a nonexistent function name or line number, an error is assumed.
- When specifying a software PC breakpoint by an address value, the specified address will be adjusted to the 16-bit boundary by assuming `LSB = 0` if it is an odd value. Furthermore, an error occurs if the specified address exceeds the 24-bit range.
- Software PC breaks are implemented by an embedded `brk` instruction and therefore cannot be used for target board ROM in which instructions cannot be embedded. If `brk` instructions cannot be embedded, hardware PC breaks are set in the same way as for the `hbreak/thbreak` command. Use the `hbreak/thbreak` command from the start if you know that `brk` instructions cannot be embedded.

- Processing resulting from the embedment of BRK command in source

When a BRK command embedded in a user application source, instead of a break command, PC+=2 is automatically performed following a software break (①+2 results in ② in the following example). By embedding a BRK command in a source, breaks can be implemented at a number of locations exceeding the number of hard breaks when ROM such as flash memory is used for execution.

Example:

```
sample.c
void main()
{
    •                ; < Continues here.
    •
    •
    a = b + 1        ;
    iRet = sub( a )  ; < ③
    asm( "brk" )     ; < ①Embedment of brk command
    if ( iRet - 1 ) { ; < ②Stops here. (BRK command address + 2)
        b -= 2      ;
    }
    •
    •
    •
```

In the above example, if a software break is set at ①, the process stops at ①. This is because the debugger cannot determine whether the BRK command embedded in memory is by a break command or one embedded in the source. Note that the breakpoint setting must be cleared before the next execution.

When a hardware break is set at ①, the process stops at ②. After ③ is processed by "next," the process stops at ②.

hbreak (set hardware PC break)

thbreak (set temporary hardware PC break)

[ICD Mini / SIM]

Operation

Sets a hardware PC breakpoint. The maximum number of hardware PC breakpoints that can be set is 1 to 4 for the ICD Mini mode, depending on the model, and 1 for the SIM mode.

When the PC matches the address set during program execution, the program breaks before executing the instruction at that address. A breakpoint can be set using a function name, line number, or address.

The **hbreak** and **thbreak** commands are functionally the same. The following describes the difference:

hbreak: The breakpoints set by **hbreak** are not cleared by a break that occurs when the set point is reached during program execution.

thbreak: The breakpoints set by **thbreak** are cleared by one occurrence of a break at the set point.

Format

hbreak [*Breakpoint*]

thbreak [*Breakpoint*]

Breakpoint: Breakpoint

A breakpoint can be specified with one of the following:

- Function name
- Source file name:line number or line number only
- *Address (decimal, hexadecimal, or symbol)

When omitted, a breakpoint is set at the address displayed on the current PC.

Conditions: $0 \leq \text{address} \leq 0\text{xfffffe}$

Usage example

Example 1

```
(gdb) hbreak main
```

Hardware assisted breakpoint 1 at 0xc0001e: file main.c, line 10.

```
(gdb) continue
```

Continuing.

Breakpoint 1, main () at main.c:10

A hardware PC breakpoint is set at the position specified using a function name.

When the target program is run, it breaks before executing the first C instruction (expanded to mnemonic) in `main()`. The PC on which the program has stopped displays the start address of that instruction (i.e., address of first mnemonic expanded).

Example 2

```
(gdb) thbreak main.c:10
```

Hardware assisted breakpoint 1 at 0xc0001e: file main.c, line 10.

A temporary hardware PC breakpoint is set at the position specified with a line number. Although the breakpoint here is specified in "source file name:line number" format when the breakpoint is to be set in the C source containing the current PC address, it can be specified by simply using a line number like "**thbreak 10**". For assembly sources, a source file name is always required.

If no instructions exist on the specified line with actual code (i.e., not expanded to mnemonic), a breakpoint is set at the beginning of the first instruction encountered with actual code thereafter.

When the target program is run, it breaks before executing the C instruction line 10 in `main.c`. The PC on which the program has stopped displays the start address of that instruction (i.e., address of first mnemonic expanded).

If no instructions exist on line 10 with actual code, the program breaks at the beginning of the first instruction encountered with actual code thereafter. Because the breakpoint is set by **thbreak**, it is cleared after a break.

■ Example 3

```
(gdb) hbreak *0xc0001e
```

Note: breakpoint 1 also set at pc 0xc0001e.

Hardware assisted breakpoint 2 at 0xc0001e: file main.c, line 10.

A hardware PC breakpoint is set at the position specified using an address.

When the target program is run, it breaks before executing the instruction at that address.

A symbol can also be used, as shown below.

```
(gdb) thbreak *main
```

Hardware assisted breakpoint 3 at 0xc0001c: file main.c, line 7.

Note that adding an asterisk (*) causes even a function name to be regarded as an address.

Breakpoint management

The breakpoints you set are sequentially assigned break numbers beginning with 1, regardless of which types of breaks you set (see the examples above).

These numbers are required when you disable/enable or delete breakpoints individually at a later time. Even when you delete breakpoints, the breakpoint numbers are not moved up (to reuse deleted numbers) until after you quit the debugger.

To manipulate the breakpoints you set, use the following commands:

disable	Disables a breakpoint.
enable	Enables a breakpoint.
delete or clear	Deletes a breakpoint.
ignore	Specifies the number of times a break is disabled.
info breakpoints	Displays a list of breakpoints.

For details, see the description of each command.

Notes

- The maximum number of enabled hardware PC breakpoints that can be set is 1 to 4 for ICD Mini mode, depending on the model, and 1 for the SIM mode. You can set more hardware PC breakpoints by setting the breakpoints in the disabled state. Keep in mind that this break count includes a temporary hardware PC breakpoint.
- C source lines that are not expanded to mnemonic cannot be specified as a location where to set a hardware PC breakpoint. Specifying such a C line sets a hardware PC breakpoint at the address of the first instruction to be executed next.
- If a function name or the beginning C source line in a function is specified as the position at which to set a hardware PC breakpoint, the program execution will break at the start address of the first C source (i.e., instruction to be expanded to mnemonic) in the function.

Although a `ld` instruction to save registers is inserted at the beginning of the function during compilation, this instruction is executed before the program breaks. To make the program break before executing this instruction, specify a breakpoint with the address value of that instruction.

- No hardware PC breakpoints can be set at the following lines, because an error is assumed and the target program can no longer be executed. (This problem may be resolved, however, by clearing the breakpoint.)
 - Extended instruction lines (except for the `ext` instruction at the beginning)
 - Delayed instruction lines (next line after a delayed branch instruction)
- If hardware PC breakpoints are specified using a nonexistent function name or line number, an error is assumed.
- When specifying hardware PC breakpoints by address value and the address is specified with an odd value, the specified address is adjusted to the 16-bit boundary by assuming `LSB = 0`. An error will occur if an address is specified exceeding 24 bits.

delete (clear break by break number)

[ICD Mini / SIM]

Operation

Deletes all breakpoints currently set or one or more breakpoints individually by specifying a break number.

Format

delete [*BreakNo*]

BreakNo: Break number (decimal)

When this entry is omitted, all breakpoints are deleted.

Usage example

```
(gdb) info breakpoints          (displays a breakpoint list.)
Num Type      Disp Enb Address What
1  breakpoint keep  y   0x00c00038 in sub at main.c:20
2  breakpoint keep  y   0x00c00030 in main at main.c:14
3  breakpoint keep  y   0x00c0003c in sub at main.c:22
```

Let's assume that breakpoints have been set as shown above.

■ Example 1

```
(gdb) delete 1 2
(gdb) info breakpoints
Num Type      Disp Enb Address What
3  breakpoint keep  y   0x00c0003c in sub at main.c:22
```

When you specify a break number, only that break can be cleared. You can specify multiple break numbers at a time.

■ Example 2

```
(gdb) delete
(gdb) info breakpoints
No breakpoints or watchpoints.
```

When a break number is omitted, all breakpoints are cleared.

Notes

- Break numbers are sequentially assigned to each breakpoint you set, beginning with 1. If you do not know the break number of a breakpoint you wish to delete, use the `info breakpoints` command to confirm as in the example above.
- The `delete` command clears all break settings. To disable a breakpoint temporarily, use the `disable` or `ignore` command.
- Note that specifying a break number not set displays the "No breakpoint number N." message, with no breakpoints being deleted.

clear (clear break by break position)

[ICD Mini / SIM]

Operation

Deletes PC breakpoints (including temporary breakpoints) currently set individually by specifying a set position (function name, line number, or address).

Format

clear *Breakpoint*

Breakpoint: Breakpoint

Can be specified by one of the following:

- Function name
- Source file name:line number or line number only
- *Address (decimal, hexadecimal, or symbol)

Conditions: $0 \leq \text{address} \leq 0\text{xfffffe}$

Usage example

```
(gdb) info breakpoints (displays a breakpoint list.)
Num Type      Disp Enb Address      What
1  breakpoint keep  y  0x00c0001e in main at main.c:10
2  breakpoint keep  y  0x00c00038 in sub at main.c:20
3  breakpoint keep  y  0x00c0003c in sub at main.c:22
4  breakpoint keep  y  0x00c00042 in sub at main.c:22
```

Let's assume that breakpoints have been set as shown above. Although break numbers 3 and 4 are at different addresses, the breakpoints are set on one line in terms of the C source. (This applies when breakpoints are set at addresses displayed in ASSEMBLY mode.)

Example 1

```
(gdb) clear main.c:22
Deleted breakpoints 4 3
(gdb) info breakpoints
Num Type      Disp Enb Address      What
1  breakpoint keep  y  0x00c0001e in main at main.c:10
2  breakpoint keep  y  0x00c00038 in sub at main.c:20
```

When you specify a line number, all breakpoints set on the source line are cleared.

Example 2

```
(gdb) clear main
Deleted breakpoint 1
(gdb) info breakpoints
Num Type      Disp Enb Address      What
2  breakpoint keep  y  0x00c00038 in sub at main.c:20
```

When you specify a function name, the breakpoint set in the first C instruction within the function (expanded to mnemonic) is cleared. Use this method to delete breakpoints that have been set by "break function name", etc.

Notes

- The **clear** command completely clears break settings. To disable a breakpoint temporarily, use the **disable** or **ignore** command.
- If you specify a function name, line number, or address for which no breakpoints are set, an error is assumed.

enable (enable breakpoint)**disable** (disable breakpoint)

[ICD Mini / SIM]

Operation**enable:** Enables a currently disabled breakpoint to make it effective again.**disable:** Disables a currently effective breakpoint to make it ineffective.

Breakpoints are effective when set by a break command and remain effective. The `disable` command disable these breakpoints without deleting them.

Once disabled, the breakpoints are ineffective and the program does not break until said breakpoints are reenabled by the `enable` command.

Format**enable** [*BreakNo*]**disable** [*BreakNo*]*BreakNo:* Break number (decimal)

When this entry is omitted, all breakpoints are disabled or enabled.

Usage example(gdb) `info breakpoints` (displays a breakpoint list.)

Num	Type	Disp	Enb	Address	What
1	breakpoint	keep	y	0x00c0001c	in main at main.c:7
2	breakpoint	keep	y	0x00c0001e	in main at main.c:10
3	breakpoint	keep	y	0x00c00028	in main at main.c:13
4	breakpoint	keep	y	0x00c00038	in sub at main.c:20

Let's assume that breakpoints have been set as shown above. The effective breakpoints are marked by 'y' in the `Enb` column.

Example 1(gdb) `disable 1 3`(gdb) `info breakpoints`

Num	Type	Disp	Enb	Address	What
1	breakpoint	keep	n	0x00c0001c	in main at main.c:7
2	breakpoint	keep	y	0x00c0001e	in main at main.c:10
3	breakpoint	keep	n	0x00c00028	in main at main.c:13
4	breakpoint	keep	y	0x00c00038	in sub at main.c:20

When executing the `disable` command with a break number attached, note that only the specified break is disabled.

You can specify multiple break numbers at a time. Ineffective breakpoints are marked by 'n' in the `Enb` column.

Example 2(gdb) `enable`(gdb) `info breakpoints`

Num	Type	Disp	Enb	Address	What
1	breakpoint	keep	y	0x00c0001c	in main at main.c:7
2	breakpoint	keep	y	0x00c0001e	in main at main.c:10
3	breakpoint	keep	y	0x00c00028	in main at main.c:13
4	breakpoint	keep	y	0x00c00038	in sub at main.c:20

When a break number is omitted, all breakpoints are enabled (or disabled) simultaneously.

Notes

- Break numbers are sequentially assigned to each breakpoint when set, beginning with 1. If you do not know the break number of a breakpoint you wish to disable or enable, use the `info breakpoints` command to confirm as in the example above.
- The number of breakpoints that can be set is limited. Use the `delete` command to delete unnecessary breakpoints.
- Note that specifying a break number not set displays the "No breakpoint number N." message, with no breakpoints being disabled or enabled.

ignore (disable breakpoint with ignore counts)

[ICD Mini / SIM]

Operation

Disables a specific break the number of times specified by a break hit count.

Format

ignore *BreakNo Count*

BreakNo: Break number (decimal)

Count: Number of break hits to be disabled (decimal or hexadecimal)

Usage example

```
(gdb) info breakpoints
Num Type      Disp Enb Address      What
1  breakpoint keep  y   0x00c0003c in sub at main.c:22
2  breakpoint keep  y   0x00c00030 in main at main.c:14
(gdb) ignore 2 2
```

Break number 2 is disabled twice.

```
(gdb) continue
Continuing.
Breakpoint 1, sub (k=1) at main.c:22
(gdb) continue
Continuing.
Breakpoint 1, sub (k=1) at main.c:22
(gdb) continue
Continuing.
Breakpoint 2, main () at main.c:14
```

Although the target program passes through the breakpoint twice as it is run twice (by `continue`) above, no break occurs. A break occurs when running the program a third time because the breakpoint is reenabled.

Notes

- Break numbers are sequentially assigned to each breakpoint when set, beginning with 1. If you do not know the break number of a breakpoint you wish to disable, use the `info breakpoints` command to confirm as in the example above.
- Count is used to count the number of times a specific break is hit, and not the number of times the target program is run. The count is not decremented unless the program passes through a specified breakpoint.
- The `ignore` command cannot be used to collectively disable multiple breakpoints.
- Note that specifying a break number not set displays the "No breakpoint number N." message, with program execution being aborted.

info breakpoints (display breakpoint list)

[ICD Mini / SIM]

Operation

Displays a list of breakpoints currently set.

Format

info breakpoints

Display

The breakpoint list is displayed as shown below.

```
(gdb) info breakpoints
Num Type      Disp Enb Address      What
1  breakpoint keep  y   0x00c00026 in main at main.c:11
   breakpoint already hit 1 time
2  hw breakpoint del   n   0x00c00038 in sub at main.c:20
```

Num: Indicates a break number.

Type: Indicates the type of breakpoint.

breakpoint Software PC breakpoint

hw breakpoint Hardware PC breakpoint

Disp: Indicates breakpoint status after a break hit.

keep The breakpoint will not be deleted.

del The breakpoint will be deleted. This means that the breakpoint is a temporary break.

Enb: Indicates whether the breakpoint is effective or ineffective.

y Effective

n Ineffective

Address: Indicates the address at which a breakpoint is set (in hexadecimal).

What: Indicates the location where a breakpoint is set.

This information is displayed in "in *function name* at *source file name:line number*" format.

Moreover, the number of times a breakpoint has thus been hit is displayed in "breakpoint already hit N times" format.

When breakpoints are not set at any location, the list is displayed as shown below.

```
(gdb) info breakpoints
No breakpoints or watchpoints.
```

commands (setting a command to execute after break)

[ICD Mini / SIM]

Operation

This command sets or cancels a command (multiple lines) to execute when execution halts at a specified breakpoint.

Format

commands [*BreakNo*]

After the command is entered, the prompt will change to a ">." Enter the command line to be set for a break.

The command line entered may consist of multiple lines. Enter "end" to end input.

Use the info breakpoint command to view the set command line.

Omitting the break number results in specification of the number of the most recently set breakpoint.

Command line cancellation:

When the prompt changes to ">," enter "end" on the first line.

Usage example

```
(gdb) break boot.s:16
```

```
(gdb) commands 1
```

```
>x /4b 0x100
```

```
>break main
```

```
>continue
```

```
>x /4b sub
```

```
>end
```

```
(gdb) continue
```

```
Continuing.
```

```
Breakpoint 1, boot () at boot.s:16
```

```
0x100: 0xaa 0xaa 0xaa 0xaa
```

```
Breakpoint 2 at 0x632: file main.c, line 18
```

```
Breakpoint 2, main () at main.c:18
```

```
Current language: auto; currenty c
```

```
0x658 <sub>: 0x00 0x40 0x25 0x18
```

```
(gdb)
```

Notes

- An error will be generated if a nonexistent numeric value is specified as a break number.
- If the break number is omitted, the number of the most recently set breakpoint will be specified.
- Proper operation is guaranteed for command lines of up to 50 lines. Even if a command line exceeds 50 lines, no error will be generated. Lines exceeding 50 lines are used as is.
- The program will halt immediately if an error occurs while a command is executing.
- The command line is not executed when a break occurs using a temporary breakpoint (tbreak, thbreak).
- The commands command cannot be nested. If a commands command occurs in the command line, the command line specified by the commands command cannot be entered during a break.

8.5.9 Symbol Information Display Commands

info locals (display local symbol)

info var (display global symbol)

[ICD Mini / SIM]

Operation

Displays a list of symbols.

info locals: Displays a list of local variables defined in the current function.

info var: Displays a list of global and static variables.

Format

```
info locals
info var
```

Usage example

■ Example 1

```
(gdb) info locals
i = 0
j = 2
```

All local symbols defined in the function that includes the current PC address are displayed along with symbol content.

■ Example 2

```
(gdb) info var
All defined variables:
```

```
File main.c:
int i;
```

```
Non-debugging symbols:
0x00000000 __START_bss
0x00000004 __END_bss
0x00000004 __END_data
0x00000004 __START_data
```

All defined global and static variables are displayed in list form separately for each source file. Displayed under the heading "Non-debugging symbols:" are such global symbols as section symbols defined in other than the source file.

Notes

If the current position indicated by the PC address is outside the function (stack frame) (e.g., in boot routine of an assembly source), local symbols are not displayed.

```
(gdb) info locals
No frame selected.
```


print (alter symbol value)

[ICD Mini / SIM]

Operation

Alters the value of a symbol.

Format

print *Symbol* [=*Value*]

Symbol: Variable name

Value: Value used to alter (decimal, hexadecimal, or symbol)

When this entry is omitted, the current symbol value is displayed.

Conditions: $0 \leq \textit{Value} \leq \text{valid range of type}$

Usage example**■ Example 1**

```
(gdb) info local
j = 0
(gdb) print j
$1 = 0
```

When you specify only a variable name, the value of that variable is displayed. The $\$N$ is a number used to reference this value at a later time. The contents displayed here can be referenced using `print print $1`.

■ Example 2

```
(gdb) print j=5
$2 = 5
(gdb) info local
j = 5
```

Note that specifying a value changes the variable value to that specified.

Notes

- If you specify an undefined symbol, an error is assumed.
- Even if the value you have specified exceeds the range of values for the type of variable you wish to alter, no errors are assumed. Only a finite number of low-order bits equivalent to the size of the variable are effective, with excessive bits being ignored. For example, specifying 0x10000 for variable `int` is processed as 0x0000.

8.5.10 File Loading Commands

file (load debugging information)

[ICD Mini / SIM]

Operation

Loads only debugging information from elf format object files.
Use the `load` command to load necessary object code.

Format

file *Filename*

Filename: Name of object file in elf format to be debugged (with path also specifiable)

Usage example

```
(gdb) file sample.elf
```

Debugging information is loaded from `sample.elf` in the current directory.

Notes

- The `file` command only loads debugging information; it does not load object code. Therefore, except when the program is written to target MCU ROM, you cannot start debugging by simply executing the `file` command.
- The `file` command must be executed before the `target` and `load` commands. The following shows the basic sequence of command execution:

(gdb) file sample.elf	(this command)
(gdb) target sim	(connects the target MCU.)
(gdb) load	(loads the program.)
(gdb) c17 rst	(resets the CPU.)
- Unless executed for elf object files in executable format (generated by the linker), the `file` command results in an error and no files can be loaded. If the loaded file contains no debugging information, an error also results.
- The elf format object files contain information on source files (including the directory structure). For this reason, unless the source files exist in a specified directory in the object file as viewed from the current directory, the source files cannot be loaded. Basically, the series of operations from compiling to debugging should be performed in the same directory.
- Once the `file` command is executed, operation cannot be aborted until the debugger finishes loading the file.
- An error will occur if an unsupported elf file (with no C17 flag) is specified.

load (load program)

[ICD Mini / SIM]

Operation

Loads the program and data from a file into target MCU memory.

Format

load [*Filename*]

Filename: Name of elf format or ROM data (Motorola format) file to be debugged (with path also specifiable)
When this entry is omitted, the file specified previously by the `file` command is loaded. This specification is usually omitted.

Usage example

```
(gdb) file sample.elf
(gdb) target sim
(gdb) load
```

The program and data are loaded from `sample.elf` in the current directory (specified by the `file` command) into target memory (computer memory in this example because the debugger operates in simulator mode).

Notes

- The `load` command must be executed after the `file` and `target` commands. The following shows the basic sequence of command execution:

(gdb) file sample.elf	(loads debugging information.)
(gdb) target sim	(connects the target MCU.)
(gdb) load	(this command)
(gdb) c17 rst	(resets the CPU.)
- The `load` command loads only several areas of an object file containing the code and data. All other areas are left intact in the previous state before `load` command execution.

8.5.11 Trace Command

c17 tm (set trace mode)

[SIM]

Operation

Sets the conditions below.

Turning trace on/off

When you turn trace on, trace information is sampled along with program execution.

Trace information items to be displayed

You can choose the items in the trace information to be displayed.

Output destination of trace information

You can choose a view or file as the destination at which sampled trace information is output. Choosing a file requires that you specify a file name.

Format

c17 tm on *Mode* [*Filename*] (sets trace mode.)

c17 tm off (clears trace mode.)

Mode: Trace mode (contents of trace information displayed)

Specify within the range from 0x00 to 0xff. Set the bit corresponding to the item to be displayed to 1.

Bit 0 Trace number

Bit 1 Clock number

Bit 2 PC value and instruction code

Bit 3 Bus information (address, R/W and access size, data)

Bit 4 Register values (R0–R7, SP)

Bit 5 PSR value (IE, IL, CVZN)

Bit 6 Disassembled contents and source code

Bit 7 Cumulative number of clocks (number of clocks spent by each instruction when set to 0)

Filename: Name of file to which trace information is output

When a file name is specified, sampled trace information is output to the specified file, and not displayed in the console.

When this entry is omitted, trace information is displayed in the console and not output to a file.

Usage example

■ Example 1

```
(gdb) c17 tm on 0xff trace.log
```

This example sets the trace mode for displaying all information and specifies the `trace.log` file in which to save the information. Running the program after setting these options outputs trace information to a file for each instruction executed. If a file name is omitted, the information is displayed in the console.

■ Example 2

```
(gdb) c17 tm off
```

Trace mode is turned off. From this time on, no trace information is sampled even when running the program.

Trace information

Running the target program after setting trace mode with this command displays trace information in the console for each instruction executed, or outputs it to a file.

The contents of trace information displayed or output are as follows:

Format of each trace information line

num clk pc code bus_addr/type/data r0 r1 r2 r3 r4 r5 r6 r7 sp ie/il/cvzn src_mix

num: Number of executed instructions (in decimal)
 Number of instructions executed since the CPU was reset

clk: Number of execution clocks (in decimal)
 Number of execution clocks since the CPU was reset

pc: Address of executed instructions

code: Instruction codes

bus_addr: Accessed memory addresses (in hexadecimal)

type: Type of bus operation
 r8: Byte data read
 r16: 16-bit data read
 r32: 32-bit data read
 w8: Byte data write
 w16: 16-bit data write
 w32: 32-bit data write

data: Read/written data

r0–r7: r0–r7 register values (in hexadecimal)

sp: sp register value

ie: IE bit value in psr

il: IL bit value in psr

cvzn: C, V, Z and N bit values in psr

src_mix: Disassembled contents and source codes of executed instructions

Display example

First half of information lines (trace number to register values)

num	clk	pc	code	bus_addr/type/data	r0	r1	r2	r3	r4	r5	r6	r7
652	1445	0040dc	9900	----- --	000094	000000	000000	00ffff	000000	000000	000000	000000
653	1446	0040de	4000	----- --	000094	000000	000000	00ffff	000000	000000	000000	000000
654	1447	0040e0	4000	----- --	000094	000000	000000	00ffff	000000	000000	000000	000000
655	1449	0040e2	d900	000000 w16	000094	000000	000000	00ffff	000000	000000	000000	000000
656	1450	0040e4	2a12	----- --	000094	000000	000000	00ffff	000000	000000	000000	000000
657	1451	0040e6	2814	----- --	000000	000000	000000	00ffff	000000	000000	000000	000000
658	1452	0040e8	4000	----- --	000000	000000	000000	00ffff	000000	000000	000000	000000
659	1457	0040ea	1805	003ef4 w32	000000	000000	000000	00ffff	000000	000000	000000	000000
660	1458	0040f6	a001	----- --	000000	000000	000000	00ffff	000000	000000	000000	000000
661	1459	0040f8	9000	----- --	000000	000000	000000	00ffff	000000	000000	000000	000000
662	1462	0040fa	0e0e	----- --	000000	000000	000000	00ffff	000000	000000	000000	000000
663	1466	004118	0120	003ef4 r32	000000	000000	000000	00ffff	000000	000000	000000	000000
664	1467	0040ec	8201	----- --	000000	000000	000000	00ffff	000001	000000	000000	000000
665	1468	0040ee	9205	----- --	000000	000000	000000	00ffff	000001	000000	000000	000000

Second half of information lines (SP value to source code)

sp	ie/il/cvzn	src	mix
003ef8	0 0 0010	ld	%r2,0x0 (main.c) 00012 i = 0;
003ef8	0 0 0010	ext	0x0
003ef8	0 0 0010	ext	0x0
003ef8	0 0 0010	ld	[0x0],%r2
003ef8	0 0 0010	ld	%r4,%r2 (main.c) 00014 for(j = 0; j < 6; ++j)
003ef8	0 0 0010	ld	%r0,%r4 (main.c) 00016 sub(j);
003ef8	0 0 0010	ext	0x0
003ef4	0 0 0010	call	0x5
003ef4	0 0 0010	and	%r0,0x1 (main.c) 00022 if(k & 0x1)
003ef4	0 0 0010	cmp	%r0,0x0
003ef4	0 0 0010	jreq	0xe
003ef8	0 0 0010	ret	(main.c) 00027 }
003ef8	0 0 0000	add	%r4,0x1 (main.c) 00014 for(j = 0; j < 6; ++j)
003ef8	0 0 1001	cmp	%r4,0x5

Notes

- This command cannot be used in ICD Mini mode.
- To change trace mode (with contents of trace information displayed), temporarily turn off trace mode (by executing `c17 tm off`), then set a new trace mode.

8.5.12 Other Commands

set output-radix (change of variable display format)

[ICD Mini / SIM]

Operation

Changes the format of variables displayed by the print command.

Selectable formats are hexadecimal, decimal, and octal.

Note that the display format will not change if the variable has a floating decimal point or pointer.

The format changed is not stored when the debugger ends, and variables are displayed in the default format (decimal) when the GDB is started next.

Format

set output-radix *Type*

Type: Display format

16 = Hexadecimal

10 = Decimal (default)

8 = Octal

Usage example

```
(gdb)print i
$1 = -21846
(gdb)set output-radix 16
(gdb)print i
$2 = 0xaaaa
(gdb)set output-radix 8
(gdb)print i
$3 = 0125252
```

Notes

- The debugger will not display correctly if binary is set (set output-radix 2).

set logging (log output setting)

[ICD Mini / SIM]

Operation

Saves the debugger command log as a file.

Format

set logging on (enables log output)

set logging off (disables log output)

Usage example

■ Example 1

(gdb) **set logging on**

Outputs a debugger command log. The log is saved as the file gdb.txt.

■ Example 2

(gdb) **set logging off**

Disables log output.

source (execute command file)

[ICD Mini / SIM]

Operation

Loads a command file and successively executes the debug commands written to the file.

Format

source *Filename*

Filename: Name of command file

Usage example

```
File name = src.cmd
# load symbol information
file c:/EPSON/gnu17v3/sample/tst/sample.elf
#decide debugger mode and its port
target sim
# load to memory
load c:/EPSON/gnu17v3/sample/tst/sample.elf
# reset
c17 rst
```

From # to the end of the line is interpreted as a comment.

```
(gdb) source src.cmd
(gdb)
(gdb) file c:/EPSON/gnu17v3/sample/tst/sample.elf
(gdb)
(gdb) target sim
boot () at boot.s:9
Connected to the simulator.
Current language: auto; currently asm
(gdb)
(gdb) load c:/EPSON/gnu17v3/sample/tst/sample.elf
Loading section .text, size 0xbc lma 0xc00000
Start address 0xc00000
Transfer rate: 1504 bits in <1 sec.
(gdb)
(gdb) c17 rst
CPU resetting ..... done
```

A specified command file is loaded and the commands contained in it are executed successively.

Notes

- If the command file contains a description error, the debugger stops executing the command file there. Because no error messages appear in this case, be very careful when creating a command file.
- The `source` commands can be nested so that `source` commands exist in the command file. There are no restrictions on the number of nests.
- Command files do not support control commands for if statements.

target (connect target MCU)

[ICD Mini / SIM]

Operation

Establishes connection to the target MCU and sets connect mode.

ICD Mini mode: Connected with the ICDmini via a USB interface.

Simulator mode: Debugger is set to simulator mode.

Format

target *Type* [*SubType*]

Type: One of the following symbols that specify the connect mode:

icd Connected with the ICDmini via a USB interface (in ICD Mini mode).

sim Simulator started (in SIM mode).

SubType: The following symbol specified when the connect mode is **icd**

icdmini2 Specify to use ICDmini2.

icdmini3 Specify to use ICDmini3.

Usage example**Example 1**

```
(gdb) target sim
Connected to the simulator.
```

The debugger is set to simulator mode.

Example 2

```
(gdb) target icd icdmini3
ICD hardware version ..... 1.0
ICD software version ..... 1.0
Hardware break MAX .... 1
```

The debugger is set to ICD Mini mode by ICDmini3.

Notes

Be sure to execute the **target** command before the **load** command, and the **file** command before the **target** command. The following shows the basic sequence of command execution:

```
(gdb) target sim           (this command)
(gdb) c17 ttbr 0x20000    (TTBR setting)
(gdb) load sample.elf     (loads the program.)
(gdb) c17 rst             (resets the CPU.)
```

detach (disconnect target MCU)

[ICD Mini / SIM]

Operation

Closes the port used to communicate with the target MCU and exits the current debug mode.

Format

detach

Usage example

```
(gdb) target icd icdmini3
      :
      Debug
      :
(gdb) detach
```

ICD Mini mode is exited.

Notes

This command can be used to turn the ICDmini off to switch between simulator mode and other modes, or perform operations on the target board. You need not execute this command to terminate debugging.

pwd (display current directory)

cd (change current directory)

[ICD Mini / SIM]

Operation

pwd: Displays the current directory.

cd: Changes the current directory.

Format

pwd (displays the current directory.)

cd *Directory* (changes the current directory.)

Directory: Character string used to specify a directory

Usage example

```
(gdb) pwd
```

```
Working directory c:/EPSON/gnu17/sample/tst.
```

```
(gdb) cd c:/EPSON/gnu17/sample/ansilib
```

```
Working directory c:/EPSON/gnu17/sample/ansilib.
```

After the current directory is confirmed, it is changed to "c:\EPSON\gnu17\sample\ansilib".

c17 ttbr (set TTBR)

[SIM]

Operation

Sets an address to TTBR.

When the reset command (`c17 rst`) is executed, the value (reset vector) that has been stored in the address represented by TTBR is set to the PC.

Format

c17 ttbr *Address*

Address: Address to be set to TTBR (decimal, hexadecimal, or symbol)

Conditions: $0 \leq \textit{Address} \leq 0\text{xffff}00$ (The eight low-order bits of *Address* must be 0x00.)

Usage example

(gdb) **c17 ttbr 0x8000**

Sets address 0x8000 to TTBR.

Notes

- This command can be used only in simulator mode.
- This command must be executed before the `target` command.

c17 cpu (set CPU type)

[SIM]

Operation

Sets the CPU type to the C17 core simulator.
 Changes coprocessor interface instruction operations for the C17 core simulator.

Format

c17 cpu [*CpuType*]

CpuType: The following symbols set the CPU type:

copro0	No coprocessor (default)
coprom	Model with multiplication coprocessor
copro	Model with COPRO
copro2	Model with COPRO2

The current CPU type is displayed if omitted.

Usage example

```
(gdb) c17 cpu copro2
```

Sets the CPU type to multiplication/division COPRO2 model.

Notes

- This command can be used only in simulator mode.
- This command must be executed before the `target` command.
- The debugger sets the CPU type using either this command or the model information acquired via the `c17 model` command. If both this command and the `c17 model` command are executed, the setting for the last command executed takes priority.

c17 chgclkmd (DCLK change mode)

[ICD Mini]

Operation

Sets DCLK (debug clock) change mode during break.

When change is enabled: DCLK automatically switches to high-speed clock if it is slow-speed clock during break.
The original DCLK is restored when the program is restarted.

When change is disabled: DCLK does not change during break.

Format

c17 chgclkmd [*Mode*]

Mode : 0 DCLK change mode enabled (default)

1 DCLK change mode disabled

Current mode is displayed if omitted.

When *Mode* = 0: "DCLK change mode ON"

When *Mode* = 1: "DCLK change mode OFF"

Usage example**<Example when target CPU is S1C17702>**

(gdb) continue	
Set to OSC1 in program	
Break occurs within program	Switch to HSCLK as Change mode is enabled.
(gdb) finish	Return to OSC1.
finish end	Switch to HSCLK.
(gdb) c17 chgclkmd 1	Disable Change mode and return to OSC1.
(gdb) continue	Continue with HSCLK as Change mode is disabled.
Break within program	Do not switch clock as Change mode is disabled.
(gdb)	

Notes

- The **c17 chgclkmd** command can be used only in ICD Mini mode.
- The **c17 chgclkmd** command can be used only with compatible target CPUs.
- When *Mode* = 0, the clock may not be switched correctly in the following cases:
 1. Overwriting clock control and clock source registers during break
 2. Breaking while switching the clock within the target program
 3. Step running the clock switching section within the target program

In the cases described above, set *Mode* = 1.

c17 pwul (unlock flash security password)

[ICD Mini]

Operation

Unlocks the password if a password has been set for a device that supports flash security.

Format

c17 pwul *Version Password*

Version: Flash security version (text)

Example:

M03 : Value indicating flash security version 3

Password: Password value

Specify alphanumeric characters (i.e., A–Z, a–z, 0–9).

The number of characters permitted will vary depending on the version.

Usage example

```
(gdb) c17 pwul M03 ABCD1234
```

Unlock flash security password was setted.

Unlocks the password with the preset password "ABCD1234".

Notes

- An error occurs for devices that do not support flash security.
- An error occurs if an undefined *version* is specified. The password will not be unlocked.
- An error occurs if invalid characters (non-alphanumeric characters) are specified for the *password*. The password is not unlocked.

c17 help (help)

[ICD Mini / SIM]

Operation

Displays a command description.

Format

c17 help [*Command*]

c17 help [*GroupNo.*]

Command: Name of command

GroupNo.: Command group number

Usage example**■ Example 1**

```
(gdb) c17 help
group 0: memory ..... c17 fb,c17 fh,c17 fw,x /b,x /h,x /w,set {char},
                        set {short},set {int},c17 mvb,c17 mvh, \n\
                        c17 mvw,c17 df
group 1: register ..... info reg,set $
group 2: execution ..... continue,until,step,stepi,next,nexti,finish
group 3: CPU reset ..... c17 rst, c17 rstt
group 4: interrupt ..... c17 int,c17 intclear
group 5: break ..... break,tbreak,hbreak,thbreak,delete,clear,enable,
                        disable,info breakpoints
group 6: symbol ..... info locals,info var,print
group 7: file ..... file,load
group 8: trace ..... c17 tm
group 9: others ..... source,target,detach,pwd,cd,set output-radix
                        c17 ttbr,c17 cpu,c17 chgclkmd,c17 help
                        c17 model_path,c17_model,quit

Please type \"c17 help 1\" to show group 1 or type \"c17 help c17 fb\" to get usage of command
\"c17 fb\"
```

When you omit parameters, a list of command groups is displayed.

■ Example 2

```
(gdb) c17 help 2
group 2: execution\n\
        continue          Execute continously
        until             Execute continously with temporary break
        step              Single-step every line
        stepi             Single-step every mnemonic
        next              Single-step with skip every line
        nexti             Single-step with skip every mnemonic
        finish            Quit function

Please type \"c17 help continue\" to get usage of command \"continu\"
```

When you specify a command group number, a list of commands belonging to that group is displayed.

■ Example 3

```
(gdb) c17 help step
step: Single-step,every line [ICD/SIM]

usage:      step [Count]
Count:      Number of steps to execute (decimal or hexadecimal)
             One step is assumed if ommitted.
Conditions: 1-0x7fffffff

example:
(gdb) step
(gdb) step 10
```

When you specify a command, a detailed description of that command is displayed.

Example 4

```
(gdb) c17 help c17 rst
c17 rst: Reset
```

[ICD/SIM]

```
usage: c17 rst
```

```
example:
```

```
(gdb)c17 rst
The CPU is reset.
```

To display a C17 command, specify the command name including "c17".

Notes

- Executing the `help` command (that comes standard with `gnu`) instead of the `c17 help` command displays help for the command classes and commands set in the `gnu` debuggers. This debugger does not support all of these command classes or commands. Note that device operation cannot be guaranteed for commands not described in this manual.
- A mode list (e.g. [ICD/SIM]) appears in the usage display (see Examples 3 and 4) indicating the modes in which the command is effective.
 - ICD: The command can be used in ICD Mini mode (when the ICDmini is used)
 - SIM: The command can be used in simulator mode (when debugging with the PC alone)
 If "[ICD]" is displayed, it indicates that the command cannot be executed in modes other than ICD Mini mode.

c17 model_path (model-specific information file directory setting)

[ICD Mini / SIM]

Operation

Sets the directory containing the model-specific information file.
This function is the same as the startup option `--model_path`.

Format

c17 model_path *ModelPath*

ModelPath: Character string used to specify the directory containing the model-specific information file

Usage example

```
(gdb) c17 model_path C:/EPSON/GNU17V3/mcu_model
(gdb) c17 model 17W23
(gdb) target icd icdmini3
```

Sets the directory containing the model-specific information file to `c:\EPSON\GNU17V3\mcu_model`.

Notes

- If this command is not executed, the directory containing the model-specific information file is set to the `mcu_model` sub-directory in the directory containing the debugger `gdb.exe`. Debugger `gdb.exe` is normally located in `c:\EPSON\GNU17V3`, so the setting in this usage example will be the same as when this command is not executed.
- Settings made by this command are referenced when the `c17 model` command is executed. If this command is to be executed, be sure to execute before executing the `c17 model` command.

c17 model (MCU model name setting)

[ICD Mini / SIM]

Operation

Sets the debugging target MCU model name.

This function is the same as the startup option `--model`.

Format

c17 model *ModelName* [*@Detail*]

ModelName: Character string used to specify the target MCU model

Example: 17W23

Detail : Character string specifying the following Detail options:

ModelName and *Detail* are separated by "@" instead of a space.

Detail option	Description
NOVCCIN	Specify this option if the TARGET VCC IN terminal of the ICDmini3 POWER I/F is not connected. ■ Operation when omitted Communicates with the target CPU at the voltage level of the TARGET VCC IN terminal. The target power must be connected to the TARGET VCC IN terminal. ■ Parameter None
FLS	Specify the FLS (flash deletion/rewrite) program you want to use. ■ Operation when omitted The standard FLS program (booster circuit not used) is used. ■ Parameter FLS program file name (*.saf) ■ Example FLS=fwr17W36_32bv11.saf; specifies the FLS program that uses the booster circuit.
NOREAD	Specify this option to prohibit the comparison of the write data with the data stored in flash memory. ■ Operation when omitted The data to be written is compared against data stored in flash memory. If they match, the write operation is skipped. ■ Parameter None
NOWRITE	Specify this option to prohibit writing of the data to flash memory. ■ Operation when omitted The data is written to flash memory when the debugger load command is executed. (Exception: when the data to be written matches the data stored in flash memory) ■ Parameter None
NOERASE	Specify this option to block deletion of data in flash memory. ■ Operation when omitted The data in flash memory is deleted before the data is written. ■ Parameter None
VPP	Specify this option to set flash memory delete/write voltage. This option can be used only if the target MCU is equipped with a flash programming power terminal. ■ Operation when omitted The voltage applied is determined by model specifications. ■ Parameter 7.5 [V], 7.0 [V] If an invalid value is specified, the operation will be the same as when this option is omitted. ■ Example VPP=7.5; sets the voltage to 7.5 V.

BREAKWAIT	<p>Specify this option to set the maximum wait time [msec] when forcibly made to break.</p> <ul style="list-style-type: none"> ■ Operation when omitted The default value (3000 msec) is set. ■ Parameter 5 [msec] to 300000 [msec] If a value outside the valid range is specified, the default value will be set. ■ Example BREAKWAIT=3000; sets the maximum wait time to 3000 msec.
TIMEOUT	<p>Specify this option to set the communication timeout [msec] for communications between ICDmini3 and the target CPU.</p> <ul style="list-style-type: none"> ■ Operation when omitted The default value (10 msec) is set. ■ Parameter 5 [msec] to 300000 [msec] If a value outside the valid range is specified, the default value will be set. ■ Example TIMEOUT=10; sets the communication timeout to 10 msec.

Usage example

■ Example 1: When the target MCU model is S1C17W23

```
(gdb) c17 model 17W23
(gdb) target icd icdmini3
```

The same specification applies when setting the debug I/F voltage to the target board voltage.

* The target power supply must be connected to the ICDmini POWER I/F (TARGET VCC IN pin).

■ Example 2: When FLS (Flash writing routine) is specified

```
(gdb) c17 model 17W23@FLS=FLS17W23.saf
(gdb) target icd icdmini3
```

■ Example 3: When the debug I/F voltage is 3.3 V

```
(gdb) c17 model 17W23@NOVCCIN
(gdb) target icd icdmini3
```

■ Example 4: When the debug I/F voltage is 3.3 V and FLS is also specified

```
(gdb) c17 model 17W23@FLS=FLS17W23.saf,NOVCCIN
(gdb) target icd icdmini3
```

If multiple Detail options are specified in *Detail*, they must be separated by commas.

Notes

- This command must be executed before the target command.

c17 flv (flash programming power setting)

[ICD Mini]

Operation

Supplies the voltage from the ICDmini required to delete data from and to write data to a microcomputer equipped with a flash programming power terminal. This command executes before the load command. This command generally does not need to be specified because voltage control is performed automatically within the load command.

Format

c17 flv 7.5 (setting)

Usage example

```
(gdb) target icd icdmini3
(gdb) c17 flv 7.5
(gdb) load
(gdb) c17 flvs
```

The supply of 7.5 V flash programming power starts before the load command. The supply of flash programming power stops after the load command is executed.

Notes

- The model-specific information file for microcomputers equipped with a flash programming power terminal is required to use this command.
- This command cannot be used in simulator mode.
- After data is written to or deleted from flash memory, use the **c17 flvs** command to cancel the write/delete voltage setting.
- If the **load** command fails to execute, the voltage supply will be cancelled automatically. (In this case, the processing will be the same as that for **c17 flvs**.)

c17 flvs (flash programming power setting cancellation)

[ICD Mini]

Operation

Stops the flash programming power supply set by the c17 flv command. This command executes after the load command. Execute this command after c17 flv executes.

Format

c17 flvs (cancellation)

Usage example

```
(gdb) target icd icdmini3
(gdb) c17 flv 7.5
(gdb) load
(gdb) c17 flvs
```

The supply of 7.5 V flash programming power starts before the load command. The supply of flash programming power stops after the load command is executed.

Notes

- The model-specific information file for microcomputers equipped with a flash programming power terminal is required to use this command.
- This command cannot be used in simulator mode.

c17 stdin (input of data using input/output functions)

[ICD Mini / SIM]

Operation

Enters settings for inputting data from a file or the standard input application (stdin.exe) and transferring it to the program. Use the input/output function to load data into the user program.

Format

```
c17 stdin 1 READ_FLASH READ_BUF [Filename] (setting)
c17 stdin 2 (cancellation)
```

Filename: Input file name. If no file name is provided, data will be input from [Console] view.

Usage example**■ Example 1**

```
(gdb) c17 stdin 1 READ_FLASH READ_BUF input.txt
```

This sets the data input function that inputs data from a file.

If the program continues to run after this setting is made, the debugger will abort execution at the location of the `READ_FLASH` label described in the `libg.a` library. At that location, the one-line data in the `input.txt` file is loaded into the input buffer (`READ_BUF`), and the program resumes operating.

■ Example 2

```
(gdb) c17 stdin 1 READ_FLASH READ_BUF
```

This sets a data input function that uses the standard input application.

If the program continues to run after this setting is made, the debugger will abort execution at the location of the `READ_FLASH` label described in the `libg.a` library and launch the standard input application. When data is input into the standard input application and the user clicks the [OK] button, the input data is loaded into the input buffer (`READ_BUF`), and the program resumes operating.

■ Example 3

```
(gdb) c17 stdin 2
```

This cancels the data input function. In the case of data input from a file, the specified file closes.

Notes

- To use the `c17 stdin` command, the user program must be linked to `libg.a`.
- The `c17 stdin` command uses one hardware breakpoint. If the maximum number of hardware breakpoints is exceeded, an error will be generated.
- The input buffer (`READ_BUF`) accepts up to 62 characters. Any characters beyond this limit are discarded.
- The standard input application permits the input of only alphanumeric characters and symbols.
- The `c17 stdin 1` command can not be executed continuously. Use the `c17 stdin 1` and the `c17 stdin 2` as a set.

c17 stdout (output of data using input/output functions)

[ICD Mini / SIM]

Operation

Enters the settings for outputting data from the user program to a file or [Console] view. Use the input/output function to output data from the user program.

Format

```
c17 stdout 1 WRITE_FLASH WRITE_BUF [Filename] (setting)
c17 stdout 2 (cancellation)
```

Filename: Input file name. If the file name is omitted, data will be output to [Console] view.

Usage example**■ Example 1**

```
(gdb) c17 stdout 1 WRITE_FLASH WRITE_BUF output.txt
```

This sets the data output function that outputs data to a file.

If the program continues to run after this setting is made, the debugger will abort execution at the location of the `WRITE_FLASH` label described in the `libg.a` library. At that location, the data in the specified buffer (`WRITE_BUF`) is output to the specified file, after which the program resumes operating.

■ Example 2

```
(gdb) c17 stdout 1 WRITE_FLASH WRITE_BUF
```

This sets the data output function that outputs data to [Console] view.

If the program continues to run after this setting is made, the debugger will abort execution at the location of the `WRITE_FLASH` label described in the `libg.a` library. At that location, [Console] view opens and displays the data in the specified buffer (`WRITE_BUF`), after which the program resumes operating.

■ Example 3

```
(gdb) c17 stdout 2
```

This cancels the data output function. For data output to a file, the specified file closes.

Notes

- To use the `c17 stdout` command, the user program must be linked to `libg.a`.
- The `c17 stdout` command uses one hardware breakpoint. If the maximum number of hardware breakpoints is exceeded, an error will be generated.
- The output buffer (`READ_BUF`) accepts up to 62 characters. Any characters beyond this limit are discarded.
- The `c17 stdout 1` command can not be executed continuously. Use the `c17 stdout 1` and the `c17 stdout 2` as a set.

c17 lcdsim (LCD panel simulator setting/cancellation)

[ICD Mini]

Operation

Sets or cancels the LCD panel simulator function.

Format

```
c17 lcdsim on (setting)
c17 lcdsim off (cancellation)
```

Usage example

■ Example 1

```
(gdb) c17 lcdsim on
```

This sets the LCD panel simulation function and opens the [ES-Sim] window.

If the program continues to run after this setting is made, the debugger will abort execution at the location of the `LCD_SIM` label described in the `liblcdsim.a` library.

■ Example 2

```
(gdb) c17 lcdsim off
```

This cancels the LCD panel simulator function and closes the [ES-Sim] window.

Notes

- To use the `c17 lcdsim` command, the user program must be linked to `liblcdsim.a`.
- The `c17 lcdsim` command uses one hardware breakpoint. If the maximum number of hardware breakpoints is exceeded, an error will be generated.
- If the model does not support the LCD panel simulator, the following error will be generated.

quit (quit debugger)

[ICD Mini / SIM]

Operation

Terminates the debugger.
Any ports or files used by the debugger that remain open are closed.

Format

quit
q (abbreviated form)

Usage example

(gdb) **q**

8.6 Status and Error Messages

8.6.1 Status Messages

When the target program breaks, one of the following messages is displayed, indicating the cause of the break immediately before entering the command input wait state.

Table 8.6.1.1 Status messages

Message	Description
Breakpoint #, <i>function at file:line</i>	Made to break at a set breakpoint
Illegal instruction.	Made to break by executing invalid instruction in simulator mode
Illegal delayed instruction.	Made to break by executing invalid delayed instruction in simulator mode
Break by key break.	Forcibly made to break by [Suspend] button (in simulator mode)
Break by key break. Program received signal SIGINT, Interrupt.	Forcibly made to break by [Suspend] button (in ICD Mini mode)

8.6.2 Error Messages

Table 8.6.2.1 Error messages (in alphabetical order)

Message	Description
Address is 24bit over.	The specified address is out of the 24-bit range. The maximum S1C17 address size is 24 bits (0xFFFFF).
Address(0x%lx) is ext or delayed instruction	The specified address cannot be set due to an <code>ext</code> or delayed instruction.
C17 command error, command is not supported in present mode.	The input command cannot be executed in the current mode (ICD MINI or SIM mode, or neither).
C17 command error, invalid command.	The command is erroneous.
C17 command error, invalid parameter.	The command is specified with an invalid argument.
C17 command error, number of parameter.	The command is specified with an invalid number of arguments.
C17 command error, start address > end address.	The specified start address is greater than the end address.
Cannot set hard pc break.	Cannot set a hard break at the address specified.
Cannot set hard pc break any more.	The number of hardware PC breakpoints set exceeds the limit.
Cannot set soft pc break.	Cannot set a soft break at the address specified.
Cannot set soft pc break any more.	The number of software PC breakpoints set exceeds the limit (up to 200).
Cannot write file	Cannot write to the file.
command result error!	An error occurred on executing an undefined command.
icdmini3 dll open failure.	Failed to connect to ICD mini Ver3.
C17 command error, command is not supported in present target CPU.	The selected model does not support the LCD panel simulator.

8.7 Run Time Measurement

This function measures the program run time from the execution of the user program to a break.

8.7.1 Display Method

Register the following four symbols in [Expressions] view.

- \$icdLastLapTime ... Lap time (hr, min, sec, us)
- \$icdLastLapUs ... Lap time (us)
- \$icdTotalLapTime ... Cumulative lap time (hr, min, sec, us)
- \$icdTotalLapUs ... Cumulative lap time (us)

8.7.2 Restrictions

- This function is unavailable in simulator mode.
- Since a measurement errors may occur during a switchover between break and continuous execution (resumption), use this function for measurements for extended periods, not for short-time execution of only several commands.
- This function does not support the LCD panel simulator function.

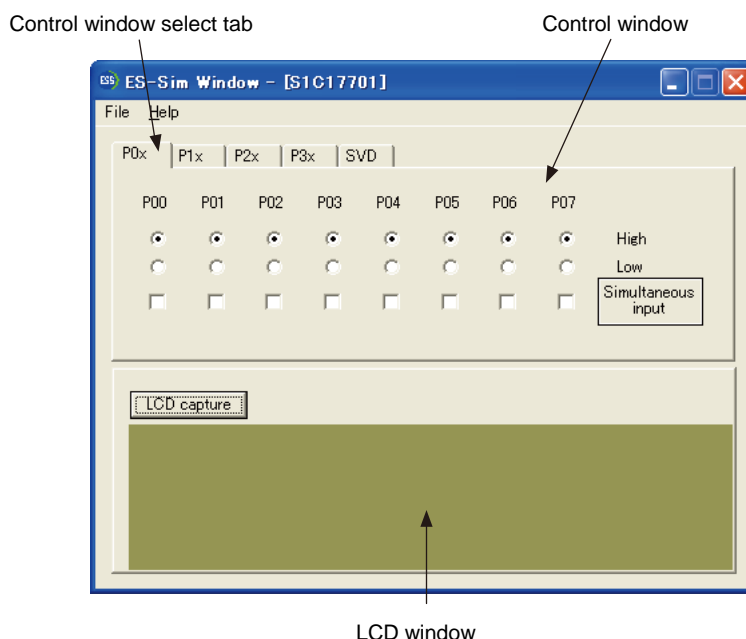
8.8 Peripheral Circuit Simulator (ES-Sim17)

The embedded system simulator (**ES-Sim17**) provides a feature to simulate the S1C17 hardware in a PC. It runs with simulator mode in the debugger **gdb** allowing practical debugging for application systems using a PC only.

The features of the **ES-Sim17** are as follows:

1. Indicates general-purpose port outputs status and simulates general-purpose port inputs.
2. Sets supply voltage level to evaluate the SVD operation.
3. Simulates display by the LCD driver built into a target model.
* The number of ports and whether with or without SVD will be determined by the model.

The [ES-Sim] window shown below is used for all operations and display.



[ES-Sim] window (sample for S1C17701)

The **ES-Sim17** can simulate operations with the OSC1 clock in real time. For operations with the OSC3 clock, refer to "simulator_readme.txt".

Note: The **ES-Sim17** is a simulator that runs on a PC, therefore, it has some restrictions. Refer to Section 8.7.8, "Restrictions", and "simulator_readme.txt".

8.8.1 Input/Output files

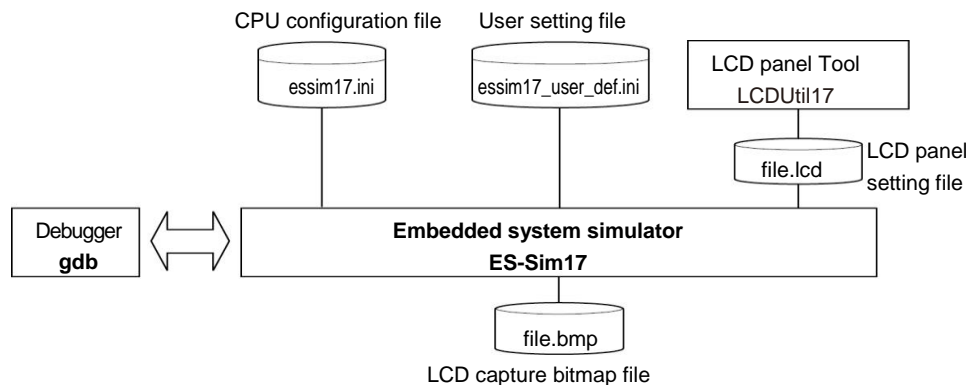


Figure 8.7.1.1 Input/output files

● Input files

CPU configuration file

File format: Text file
 File name: `essim17.ini` (fixed)
 Description: This file contains the hardware configuration for the target model to be simulated in the **ES-Sim17**. This file may be part of the model-specific information file for the target MCU. If not, obtain the latest model-specific information file (`gnu17_mcu_model_xxx.zip`) by visiting the Seiko Epson website or contacting the Seiko Epson sales operations.

Note: Do not modify this file, as the **ES-Sim17** may not run normally.

User setting file

File format: Text file
 File name: `essim17_user_def.ini` (fixed)
 Description: This file retains the settings for the target model used by the **ES-Sim17** as well as the user settings. This file must be part of the model-specific information file for the target MCU. If not, obtain the latest model-specific information file (`gnu17_mcu_model_xxx.zip`) by visiting the Seiko Epson website or contacting the Seiko Epson sales operations.

LCD panel setting file

File format: Binary file
 File name: `<filename>.lcd`
 Description: This is an LCD panel setting file for **ES-Sim17** created by `LcdUtil17`.
ES-Sim17 can simulate dot-matrix and segment LCDs.

● Output file

LCD screen-capture bitmap file

File format: Bitmap file
 File name: `<filename>.bmp`
 Description: This is a bitmap file that contains an LCD screen image simulated and can be generated by the **ES-Sim17**.

8.8.2 Starting and Terminating ES-Sim17

● Starting up ES-Sim17

The debugger launches the **ES-Sim17** when the following two conditions are met:

1. An `essim17.ini` file exists in the MCU model specified by the `c17` model command.
2. The `target sim` command (to set the debugger in simulator mode) is executed.

When the **ES-Sim17** starts up, the [ES-Sim] window appears.

● Terminating ES-Sim17

The **ES-Sim17** terminates in the following two cases:

1. When the `detach` command is executed in the debugger
2. When the debugger is terminated

● Opening/closing the [ES-Sim] window

The [ES-Sim] window can be closed by clicking the [Close] button. (This operation does not terminate the **ES-Sim17**.)

To reopen the window, execute `\essim17\EssWnd.exe`. Note, however, that the **ES-Sim17** must be running at that point. If the **ES-Sim17** has already terminated, execute the `target sim` command again.

The [ES-Sim] window cannot be opened twice. If you attempt to open the window when it is already opened, the [ES-Sim] window moves to the foreground but a new window does not appear.

8.8.3 Menus

[File] menu

File

Load lcd file

[Load lcd file]

Open an LCD file (.lcd) created in LCDUtil17.

For information on LCDUtil17 and LCD files, see Section 10.8, "LCDUtil17 (LCD Panel Customizing Tool)."

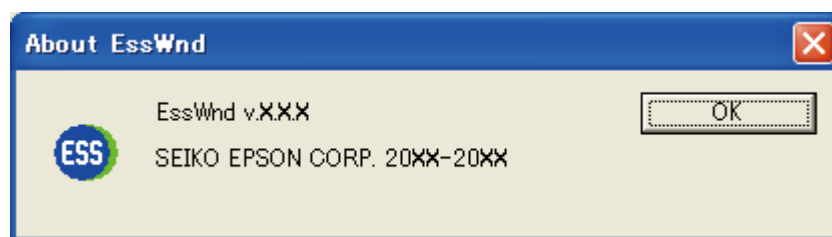
[Help] menu

Help

About EssWnd

[About EssWnd]

Shows ES-Sim Window version information.

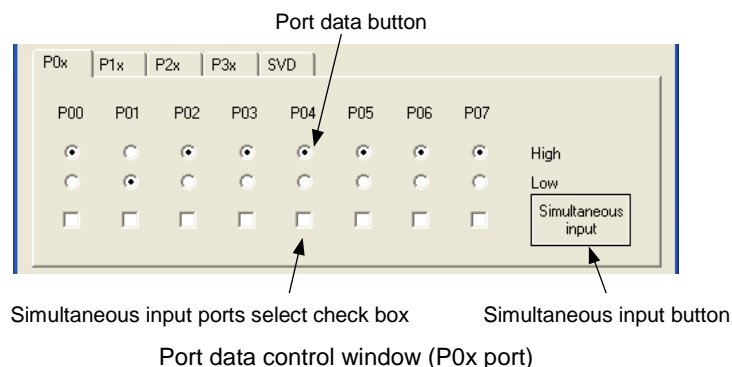


8.8.4 Simulating I/O Ports

The [ES-Sim] window allows control of the input status for the ports that have been set for general-purpose input. It also provides indicators to monitor the output status for the ports that have been set for general-purpose output.

● Port data control window

Click on a control window select tab to select the port group (P0x, P1x, P2x, P3x) you want to operate or display.



The **ES-Sim17** obtains the information, such as selected I/O port functions and I/O directions, from the emulation memory in the PC to determine the port configuration to be displayed in the port data control window.

The port data buttons and simultaneous input ports select check boxes for the ports configured as general-purpose input becomes effective and are used to set input levels. When you change the input level in the window, the **ES-Sim17** updates the input data register in the emulation memory through the debugger.

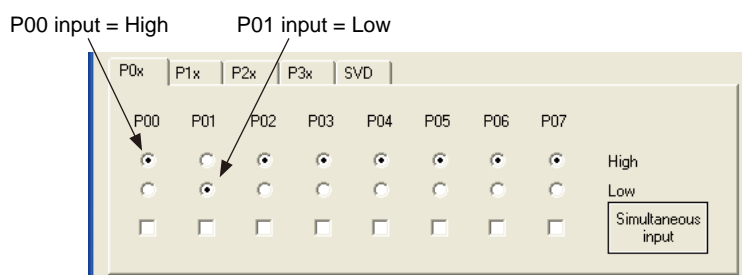
The port data buttons and simultaneous input ports select check boxes for the ports configured as a general-purpose output are grayed out to disable operations. However, the port data buttons indicate the current output status. When the output data register for the port configured as a general-purpose output is altered by the program, its status is reflected to the port data button.

The port data buttons and simultaneous input ports select check boxes for the ports configured to an internal peripheral input/output are not displayed.

The port data buttons and simultaneous input ports select check boxes for the ports that do not exist in the target model are not displayed.

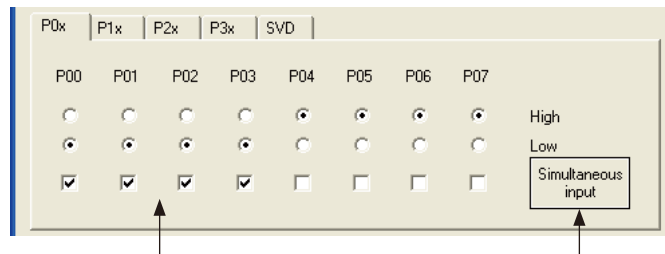
● Setting the port input status

Select either High or Low port data button. This determines the current port input level.



● Simultaneous multiple key inputs

To simulate an operation press two or more keys simultaneously, first select the simultaneous input ports select check boxes for those ports. Then click the simultaneous input button. The port input levels are reversed from the status set with the port data buttons.



(1) Select the ports used for simultaneous input.

(2) Click the button to reverse the input levels.

This operation affects ports not contained in the tab page being currently displayed. The simultaneous input button located in any page reverses all the ports that have been selected with the simultaneous input ports select check boxes regardless of whether its tab page is displayed or not.

Even if multiple ports are selected with the simultaneous input ports select check boxes, the port data button can be used to control each port individually.

● Port output status

When a port changes its output level by executing the program in the debugger, the output status is reflected to the display of the port data button immediately.

The port data button for the ports configured as a general-purpose output cannot be operated using the mouse.

● P0 port key entry reset

If the target model supports the P0 port key entry reset function, the CPU can be reset by entering the active level signals to the ports specified with software. To evaluate this function, use the same way as the simultaneous multiple key inputs described above.

● Port input interrupts

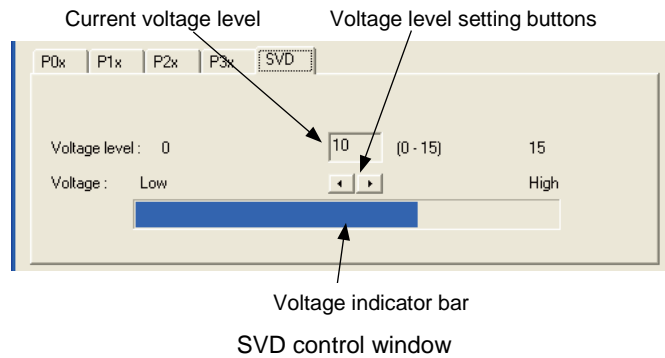
Changing the input status by an operation in the port data control window can generate a port input interrupt.

8.8.5 Simulating SVD

The [ES-Sim] window allows control of the supply voltage level for evaluating the SVD operation.

● SVD control window

Click on the SVD control window select tab to display the SVD control window.



The SVD control window is initialized with voltage level 15 (maximum level).

● Setting voltage level

The voltage level can be set within 16 steps* from 0 (low) to 15 (high) using the voltage level setting buttons.

* The number of voltage levels is equivalent to the number of valid SVD compare voltages supported in the target model. The number of available levels may be changed depending on the model.

Clicking the button changes the current voltage level and voltage indicator bar. At the same time, the compare voltage set in the SVD control register in the emulation memory and the voltage level set in this window are compared and the result is written to the SVD detection result register.

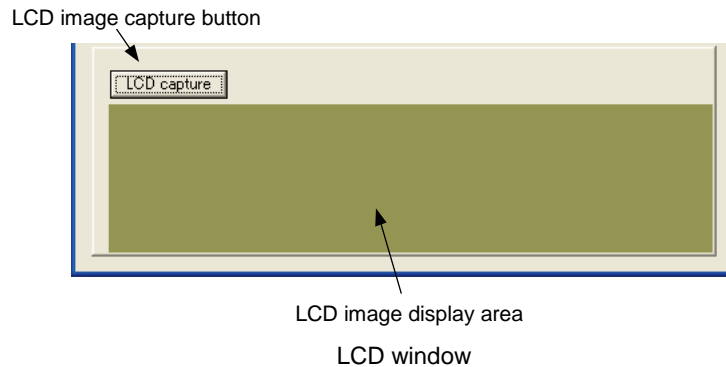
● SVD interrupt

If the target model supports the SVD interrupt, setting a voltage level lower than the SVD compare voltage in this window can generate an interrupt.

8.8.6 Simulating an LCD Driver

The **ES-Sim17** simulates display on an LCD panel according to control of the LCD driver and display memory.

● LCD window



This window simulates display on a dot-matrix type LCD driver.

The **ES-Sim17** reads the contents of the display memory in 32-Hz cycles to redraw this window.

The drive duty setting and display control (display on/off, contrast adjustment, display area selection, etc.) in the program are reflected to this window.

● Saving LCD screen

The screen image being currently displayed in the LCD window can be saved to a bitmap file (.bmp).

Click the [LCD capture] button when the screen you want to capture is displayed. When the file save dialog box appears, select the directory and enter the file name you want to save.

The screen data is captured at the point the [LCD capture] button is clicked and the LCD window stops refreshing the display until the file save has completed.

The whole panel image is saved even if the LCD window does not display a part of the screen.

The **ES-Sim17** generates a Windows standard bitmap file (.bmp).

● Restrictions

- The dot size, contrast, and background color are different from those of the actual LCD panels.
- The LCD display refresh times differ from actual LCD panels.

8.8.7 ES-Sim17 Error Messages

Table 8.7.7.1 Error messages (displayed in the **gdb** [Console] window)

Message	Description
ES-Sim error 01 : Loading the dll file was failed.	The dll file for ES-Sim17 does not exist in the default location or cannot be loaded normally.
ES-Sim error 02 : Opening the CPU construction file was failed.	The CPU configuration file does not exist in the specified location or cannot be loaded normally.
ES-Sim error 03 : Generating the CPU components was failed.	The ES-Sim17 has failed generation of the CPU module as the CPU module definition is incorrect or no required dll file exists.
ES-Sim error 04 : Connecting the CPU components was failed.	The ES-Sim17 has failed correction to the CPU module generation as the connect destination in the CPU module definition is incorrect.
ES-Sim error 05 : Opening the "user.ini" was failed.	The user setting file does not exist in the specified location or cannot be opened normally.
ES-Sim error 06 : Setting of the "user.ini" is invalid.	The setting value written in the user setting file is incorrect.

Table 8.7.7.2 Error messages (displayed in a dialog box)

Message	Description
Failed to save "path\file"	The ES-Sim17 has failed saving the captured image to the file.
Failed to open "path\file". Please confirm the file is fitting with the CPU type, or the file is existing.	The LCD file could not be opened.

8.8.8 Restrictions

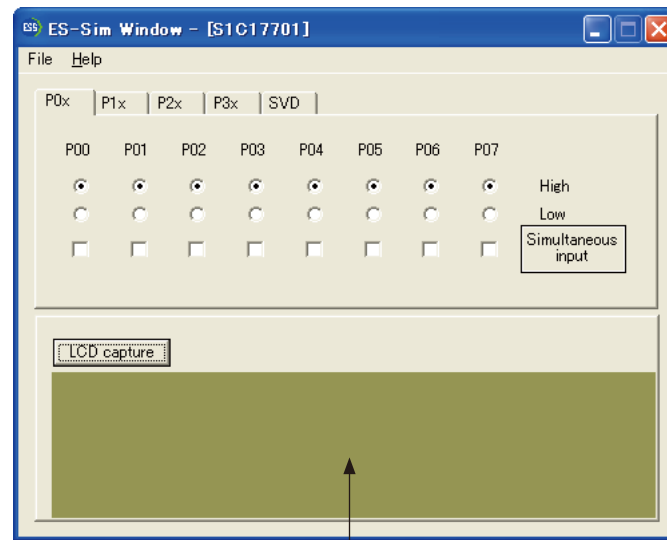
- The **ES-Slim17** supports monochrome dot-matrix LCD and monochrome segment LCD panels as external devices.
- The dot size, contrast, and panel color of the LCD window are different from those of the actual LCD panels.
- The **ES-Sim17** performs simulation on an instruction cycle basis. Therefore, operation cycles lower than the instruction cycle cannot be simulated.
- The **ES-Sim17** simulates the operation clock based on the instruction cycles. Therefore, the operation timings are not the same as those of the actual hardware.
- The functions listed below cannot be simulated.
 1. Timer clock and oscillation clock external outputs
 2. Data transfer using the UART, I²C and SPI
 3. Noise and chattering filters
- More than one **ES-Sim17** cannot be run on a PC for simulation.
- Setting higher oscillation clock frequency causes degradation of simulation performance.
- The I/O control registers that are not supported by the **ES-Sim17** function as general-purpose read/write registers. Also they are not initialized at a reset.
- Some peripheral circuits, such as the oscillator and SVD circuits, need time until their operations stabilize. In the simulation by the **ES-Sim17**, they can operate with stability immediately after they start.

* For the restrictions in the latest version of **ES-Sim17** and model dependent restrictions, refer to "simulator_readme.txt".

8.9 LCD Panel Simulator

The LCD panel simulator simulates an LCD panel display on a PC. This function runs the program on an actual device and simulates only LCD panel display operations from the PC. This allow confirmation of the LCD panel display even if the actual device is not equipped with an LCD panel.

The simulated LCD panel display appears in the LCD window in the [ES-Sim] window as shown below.



LCD window

[ES-Sim] window

For more information on using the LCD window, refer to Section 8.7.6, "Simulating an LCD Panel."

Note: Since the LCD panel display in the LCD window is simulated by the computer, certain restrictions apply. For more information, refer to Section 8.8.4, "Restrictions."

8.9.1 Input Files

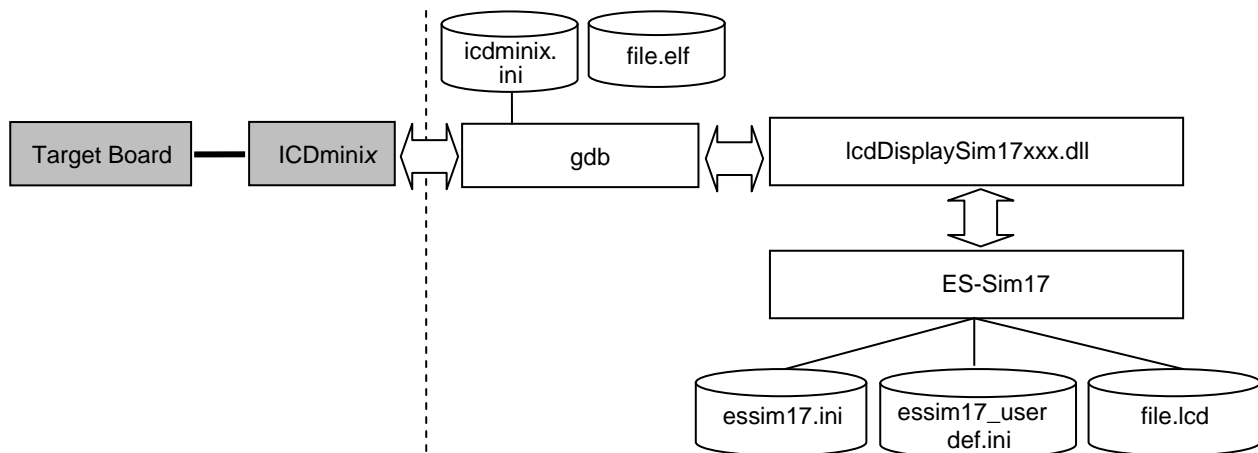


Figure 8.9.1.1 Input files

CPU configuration file

File format: Text file
 File name: `essim17.ini`
 Description: This file describes the hardware configuration for the target model to be simulated by the LCD panel simulator. It is part of the model-specific information file for the model.

Note: Do not modify this file. The LCD panel simulator may not run normally.

User setting file

File format: Text file
 File name: `essim17_user_def.ini`
 Description: This file describes user settable values for the target model to be simulated by the LCD panel simulator. It is part of the model-specific information file for the model.

GDB command file

File format: Text file
 File name: `gdbminix.ini`
 Description: This is a GDB command file for ICDminix. To launch the LCD panel simulator when the debugger starts up, add the following command:

`cl7 lcdsim on` Enables the LCD panel simulator.

LCD panel setting file

File format: Binary file
 File name: `<filename>.lcd`
 Description: This LCD panel setting file is created by LcdUtil17. It can simulate dot-matrix and segment LCDs.

8.9.2 Starting and Terminating the LCD Panel Simulator

● Starting up the LCD panel simulator

The LCD panel simulator starts when the following conditions are met:

1. An `lcdDisplaySim17xxx.DLL` file is found in the MCU model specified by the `c17` model command.
2. An LCD file exists and its file path is specified by `essim17_user_def.ini`.
3. The `c17 lcdsim on` command is executed.

When the LCD panel simulator starts up, the [ES-Sim] window appears.

● Terminating the LCD panel simulator

The LCD panel simulator terminates in the following two cases:

1. The `c17 lcdsim off` command is executed.
2. The debugger is terminated.

● Opening/closing the [ES-Sim] window

Click the [Close] button to close the [ES-Sim] window. (Closing the [ES-Sim] window does not terminate the LCD panel simulator.)

To reopen the [ES-Sim] window, execute `\essim17\EssWnd.exe`. Note, however, that the LCD panel simulator must be running at this point. If the LCD panel simulator has already terminated, execute the `c17 lcdsim on` command once again.

The [ES-Sim] window cannot be opened twice. If you attempt to open the window when it is already opened, the [ES-Sim] window will move to the foreground. No new window will appear.

8.9.3 Procedure for Modifying the Program

The program must be modified to use the LCD panel simulator. Include the LCD panel simulator library, and call the LCD panel simulator display update function at the point at which you want to refresh the LCD window display. The procedure for modifying the program is given below.

● Including the LCD panel simulator library

Using the `#include` command, include the header file (`lcdsim.h`) in the program file that calls the LCD panel simulator library function.

Example: `#include "lcdsim.h"`

● Inserting the LCD panel simulator display update function

The LCD window of the LCD panel simulator is refreshed when the LCD panel simulator display update function (`lcdsimUpdate`) is executed. Insert the LCD panel simulator display update function at the point in the program at which you want to refresh the LCD window display.

```
Example: LCD24DSP.DSPC = 0x2      // Turns on all dots of LCD
        lcdsimUpdate();          // Refreshes the LCD window.
        LCD24DSP.DSPC = 0x0      // Turns off all dots of LCD
        lcdsimUpdate();          // Refreshes the LCD window display.
        LCD24DSP.DSPC = 0x1      // Normal LCD display
        lcdsimUpdate();          // Refreshes the LCD window display.
```

● Setting the linker option

To build a program that links to the LCD panel simulator library, set the linker option in the project properties.

From the [Properties] dialog box, select C/C++ Build > Settings > [Tool Settings] > [Cross GCC Linker] > [Libraries] and specify the LCD panel simulator library. Add "lcdsim".

If the target model selected in creating a project supports the LCD panel simulator library, the linker option will be set automatically.

Note: To run the program on an actual device equipped with an LCD panel, delete the include command for the LCD panel simulator library and the LCD panel simulator display update function from the program. Leaving these in the program will result in needless processing.

8.9.4 Restrictions

- This function does not support the simulator mode that uses a target sim.
- This function supports only monochromatic dot matrix LCD and monochromatic segment LCD panels.
- The dot size, contrast, and panel color of the LCD window differ from those of actual LCD panels.
- The LCD display refresh timing differs from that of actual LCD panels.
- Enabling this function occupies one hardware breakpoint.
- This function does not refer to the peripheral device status (GPIO, CLG, etc.).

8.10 Profiler Coverage

The main purpose of the profiler coverage is to detect the functions in the program that cause bottlenecks and reduce program performance. Once the profiler coverage function is launched by `c17debug.exe` included in this package, it performs measurements using the actual device and simulator. Note that `c17debug.exe` is a debug function but an external tool not equivalent to the `gdb` debugger.

8.10.1 Input/Output Files

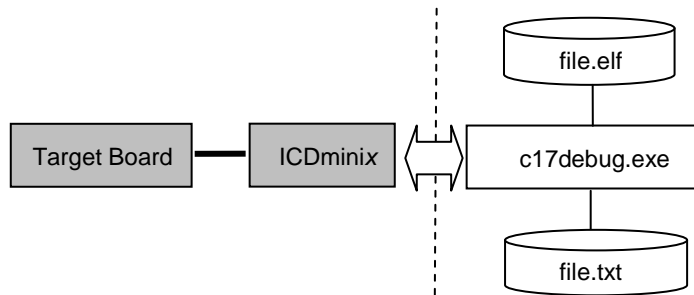


Figure 8.10.1.1 Input/output files

● Input file

Object file

File format: Binary file in elf format

File name: `<filename>.elf`

Description: This absolute object file in elf format is generated by the linker `ld`. It must contain debug information, such as symbols.

● Output file

Log file

File format: Text file

File name: `<filename>.txt`

Description: This file stores contents output to [Console] view as a log.

8.10.2 Starting and Terminating the Profiler Coverage

The profiler coverage function is provided by c17debug.exe, which can be launched as an external tool.

● Starting up c17debug.exe

Terminate the gdb debugger if it is running. Launch c17debug.exe as follows:

1. Select [External Tools] > [External Tools Configuration] from the [Run] menu.
2. Click [New launch configurations] to create “New_configuration”.
3. Make the following changes in “New_configuration”.

Tab	Setting item	Design description
-	Name:	Enter an arbitrary file name.
[Menu] tab	Location:	From [Browse File System], select c17debug.exe in the following folder: Folder name: C:\EPSON\GNU17V3
	Working Directory:	From [Browse File System], select the target project folder as the working folder.
	Arguments:	Assign a name to the folder to which the log is output.
[Common] tab	Save as:	Click the “Shared file” checkbox, then select the target project from [Browse].

4. Click [Apply] to apply the settings.
5. Click [Run] to launch c17debug.exe.

Note that c17debug.exe is controlled from [Console] view. When the c17debug.exe is executed, it outputs a prompt (“\$”) to [Console] view and awaits user input.

● Terminating c17debug.exe

Click the [Terminate] button in [Console] view.

8.10.3 Preparation

Before executing the profiler coverage function, load the program as follows:

● Procedure

1. Connect the ICDmini3 and target board to the PC.

*1 Note that c17debug.exe supports only ICDmini3.

2. Enter the following commands in the [Console] view opened by c17debug.exe.

```
$ model 17xxx           ... Specifies the target model.
$ icd                  ... Connects the ICDmini3 and target MCU.
$ load Debug/project_name.elf ... Loads the target object file (.elf) and the support function used for
                             detection into the target MCU.
```

*2 If there is no response after the icd command executes, the connection to the ICDmini3 and target MCU has failed.

*3 Using the load command, specify the path to the target object file.

*4 Include the symbol information in the target object file (.elf).

Preparations are successful when the following responses are returned:

● Response example

```
$ load Debug/sample_gcc6.elf
  VAddr    LAddr    FileSize MemSize
00: 0x000000 0x000000 0x000000 0x000050
01: 0x008000 0x008000 0x000280 0x000280
* Control flash:
  Successfully erased.
  Successfully wrote.
* Support routines: 0x000050 - 0x0000e9
  0x000090: resume, 0x00008c: step , 0x0000b8: go1 , 0x0000b4: go2
  0x0000bc: go0 , 0x000074: read , 0x00005e: write
```

Memory allocations

Results of loading the object file

Results of loading the support function used for detection

8.10.4 Coverage Function

The coverage (code coverage) refers to a code coverage rate. This function provides information on the program code executed and the number of times executed. The function executes the command from the input symbol name or address to the termination of the target function, and then shows the number of executions in each interval within the function.

In the [Console] view prepared in advance, enter and execute the following command:

```
$ coverage symbol ... Specifies the symbol name or address to be executed.
```

● Execution example

```
$ coverage _crt0_start0
target paused
register    PC: 0x008116 ... Information concerning the register after execution
           PSR: 0x00
           R0: 0x000000
           cycle : 2 ... Number of cycles from the start of execution to the
                           termination of the function
           status : 9 (9: break, 10: timeout, 11: uncertain PC, 12/13: wrong PC) ... Target MCU state
0x008080 - 0x008088: 1, _crt0_start0 (8/8) ... Measurement results
0x0000 - 0x0008: 1, _crt0_start0
```

● How to read measurement results

Function map address	Number of executions	Executed function	Number of executed blocks	
0x008080 - 0x008088:	1,	_crt0_start0	(8/8)	← Overall
0x0000 - 0x0008:	1,	_crt0_start0		← Breakdown

8.10.5 Profiler Function

The profiler function analyzes performance. This function employs a statistical profiler (sampling profiler) and determines the run time of each function by sampling. It executes the command from the input symbol name or address for 10 seconds and send and displays the sampling results from the PC (program counter) at 50 ms intervals.

In the [Console] view prepared in advance, enter and execute the following command:

```
$ clear ... Clears breakpoints to prevent breaks during measurement.
$ profile symbol ... Specifies the symbol name or address to be executed.
```

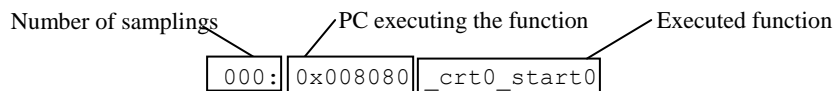
*1 Nothing is displayed during the 10 seconds of sampling operations.

*2 After the 10 seconds, the target MCU breaks, and the sampling results are displayed.

● Execution example

```
$ clear
$ profile _crt0_start0
000: 0x008080 _crt0_start0 ... Current sampling location
001: 0x008270 _exit
002: 0x008270 _exit
003: 0x008270 _exit
:
197: 0x008270 _exit
198: 0x008270 _exit
199: 0x008270 _exit
target paused
register PC: 0x008272 ... Information concerning the register after execution
      PSR: 0x0a
      R0: 0x000000
```

● How to read measurement results



The above execution example shows that samplings 001 to 199 occurred within the `_exit` function. This indicates most of the time was spent within the `_exit` function.

8.10.6 Restrictions

- Make sure there is at least 200 bytes of free RAM for the user program. This function requires a sufficient amount of free RAM to load subroutines for measurement.
- This function does not support measurement by functions that contain interrupt processing.
- The object file (`.elf`) must contain symbol information.
- ICDmini3 is the only ICDmini supported by this function.

9 Creating Data to Be Submitted

If using the service to load user programs to the internal ROM or Flash of the CPU at the Seiko Epson factory, a PA file (Data to be submitted) must be created and submitted to Seiko Epson.

Creating a PA file requires the following files.

- FDC file (Function option document)
(Only for CPUs requiring function option selection)
- PSA file (ROM data)

This section describes how to create PA files.

9.1 Outline of Tools for Creating Data to Be Submitted

The following two tools are used to create PA files (data to be submitted).

1. Function option generator (`winfog17.exe`)

This tool is used with CPUs requiring function option selection.

`Winfog17` is a tool for creating FDC files (Function option documents) for generating the IC mask pattern. Function options can be set simply by selecting the checkbox items displayed in the window.

FDC files are not required for CPUs that do not require function option setting.

2. Data checker (`winmdc17.exe`)

`Winmdc17` is a tool which checks the data of PSA and FDC files after development and creates PA files to be submitted to Seiko Epson.

9.2 Procedure for Creating Data to Be Submitted

Figure 9.2.1 shows the flowchart for creating a PA file (Data to be submitted).

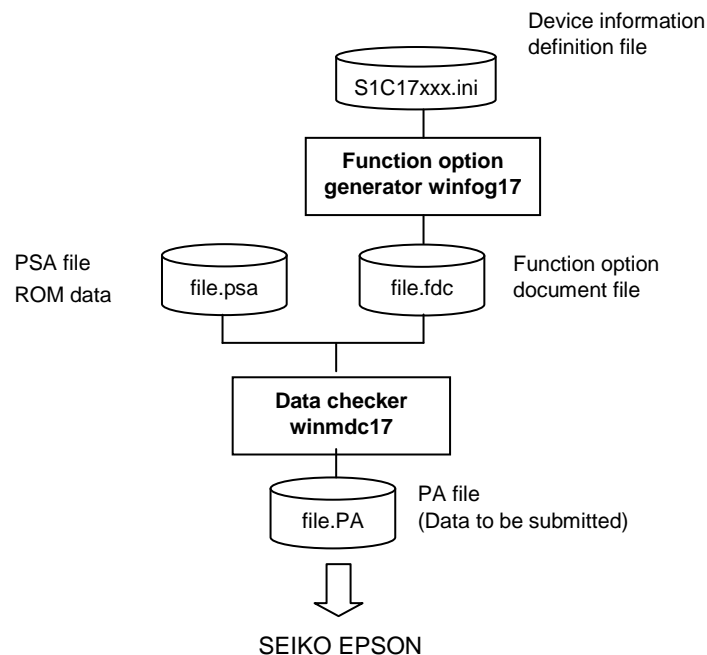


Figure 9.2.1 PA file creation flowchart

9.2.1 Creating FDC Files (Function Option Documents) Using winfog17

This procedure is necessary for CPUs that require function option selection.

FDC files (Function option documents) are files in which the specific function options are selected for the target CPU. For more information of the function options that can be selected, refer to the target CPU technical manual.

Do the following to select a function option and generate an FDC file using winfog17.exe.

- (1) Launch winfog17

Double-click winfog17.exe in the user folder \EPSON\GNU17V3\dev\Bin.

If the model-specific information file (S1C17xxx.ini) was read in for the previous execution, the same file will be loaded automatically when winfog17 is launched.

- (2) Load S1C17xxx.ini

Click the [Device INI Select] button in winfog17 or select [Device INI Select] from the [Tool] menu.

Enter the filename and path in the text box in the dialog box displayed, or click the [Ref] button to load S1C17xxx.ini for the target CPU. The S1C17xxx.ini corresponding to the specific model can be found in the folder below.

User_folder\EPSON\GNU17V3\mcu_model

If no corresponding S1C17xxx.ini exists for the target CPU, contact the Seiko Epson sales operations.

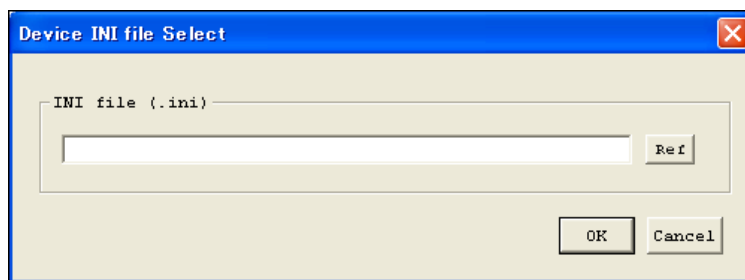


Figure 9.2.1.1 Load window for model-specific information definition file

If the CPU does not require function option selection, a dialog box is displayed indicating "INI file does not include FOG information".

(3) Select function options

Click the checkboxes in the option list area to select the required options. Changing the items selected in the option list area displays the selected function options in the function option document area.

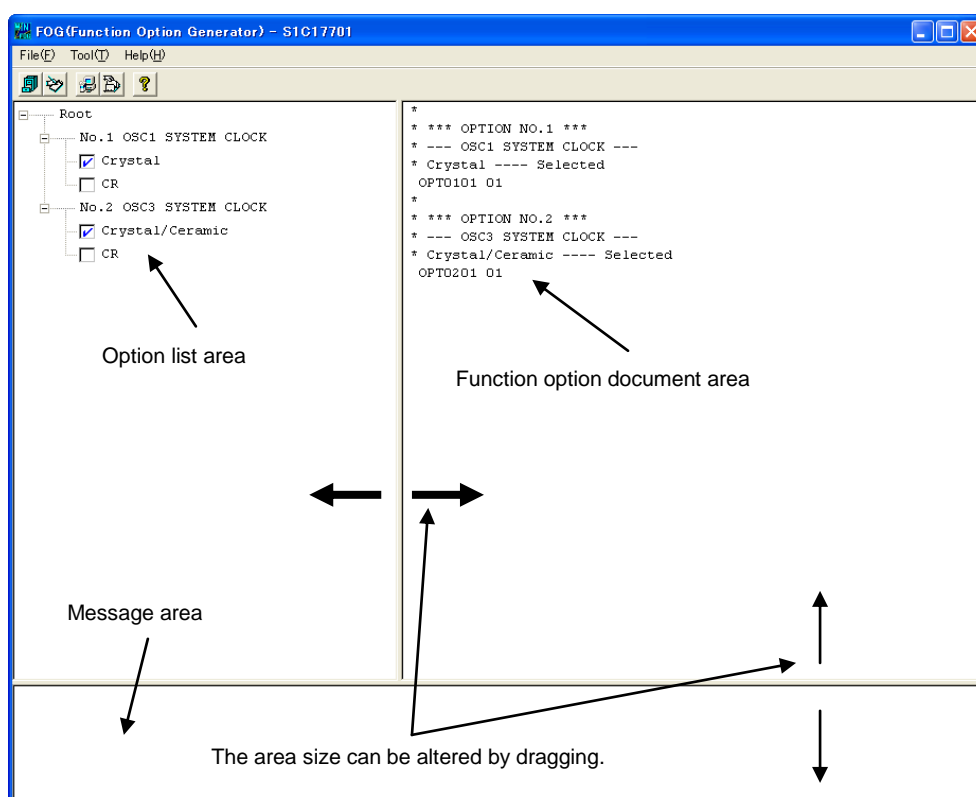


Figure 9.2.1.2 Function option display window

(4) Generate FDC file

Click the [Generate] button in winfog17 or select [Generate] from the [Tool] menu.

Once the file has been correctly generated, the message "Making file(s) is completed" is displayed in the message area. The FDC files are generated in the folder below.

User_folder\EPSON\GNU17V\dev

Other winfog17.exe functions

Functions of winfog17.exe other than those described above are explained here. The following functions can be used as required.

[File] menu

Open

Opens an FDC file. It is used to modify an existing file. The [Open] button has the same function.



[Open] button

End

Quits winfog17.

[Tool] menu

Setup

Sets details such as the creation data, output filename, and comments to be included in the FDC file. The [Setup] button has the same function.



[Setup] button

The following dialog box is displayed.

Date

Displays the current date. This can be changed as required.

Function Option Document file

Specifies the FDC file name to be created. The default name displayed should be edited for use. Other folders can be viewed using the [Ref] button.

Function Option HEX

This function is not used with the S1C17 Family. "No" should be selected.

User's Name

Enter the user's company name. The name can be up to 40 characters long, and any more than this will be ignored. The name may contain alphanumeric characters, symbols, and spaces. The details entered here are recorded in the "USER'S NAME" field of the FDC file.

Comment

Enter comments. Up to 10 lines can be entered with a maximum of 50 characters per line. The comment may contain alphanumeric characters, symbols, and spaces. Use the [Enter] key for line breaks. The details entered here are recorded in the "COMMENT" field of the FDC file.

Click [OK] after the necessary items described above have been entered to save these settings and close the dialog box. Clicking [Cancel] closes the dialog box without changing the current settings.

Note: • The following restrictions apply to file names.

1. File names cannot be more than 2,048 characters long, including the path.
 2. File names (excluding the file extension) cannot be more than 15 characters long. The extension cannot be more than 3 characters long.
 3. A hyphen (-) cannot be used at the beginning of a file name. The following symbols cannot be used in directory names (folder names), file names, or file extensions.
/ : , ; * ? " < > |
- The following symbols cannot be used in User's Name or Comment.
\$ \ | `

9.2.2 Creating PSA Files (ROM Data)

PSA files (ROM data) are Motorola S2 format files with the same file name as the elf format object files and with the extension ".psa". The procedure for creating PSA files is described below.

The created PSA file must be verified on the actual device.

- (1) Confirm target CPU model selection
Check on the IDE that the target CPU for the target project corresponds to the desired device. If the required model has not been selected, select the correct device. If the desired device is not listed, obtain the model-specific information file (gnu17_mcu_model_xxx.zip) by visiting the Seiko Epson website or contacting the Seiko Epson sales operations.
- (2) Generate PSA file
Build the target project. The PSA file will be created within the target project folder.

Creating an elf format object file in the project build on the IDE also creates PSA and PA files.

9.2.3 Creating PA Files (Data to Be Submitted) Using windmc17

PA files may be configured as follows.

- If no FDC file (Function option document) exists
PA file is generated from a PSA file (ROM data).
- If an FDC file exists
One PA file is generated from PSA and FDC files.

In either case, windmc17 creates a PA file in accordance with the target CPU settings when a project is built on the IDE.

If using the service to load user programs to the internal ROM or Flash of the CPU at the Seiko Epson factory, the PA files generated should be submitted to Seiko Epson.

9.2.4 PA File (Data to Be Submitted) Separation Procedure

PA files (Data to be submitted) generated (packed) using winmdc17 can be separated (unpacked) into function option documents and ROM data.

Use winmdc17.exe as shown below to separate a PA file into a UFD file (Function option document) and USA file (ROM data).

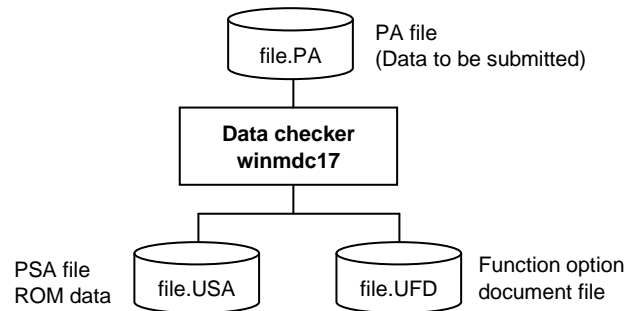


Figure 9.2.4.1 Flowchart for PA file unpacking

- (1) Launch winmdc17.exe

Double-click winmdc17.exe in *User_folder\EPSON\GNU17V3\dev\Bin*.

If the model-specific information file (S1C17xxx.ini) was read in for the previous execution, the same file will be loaded automatically when winmdc17 is launched.

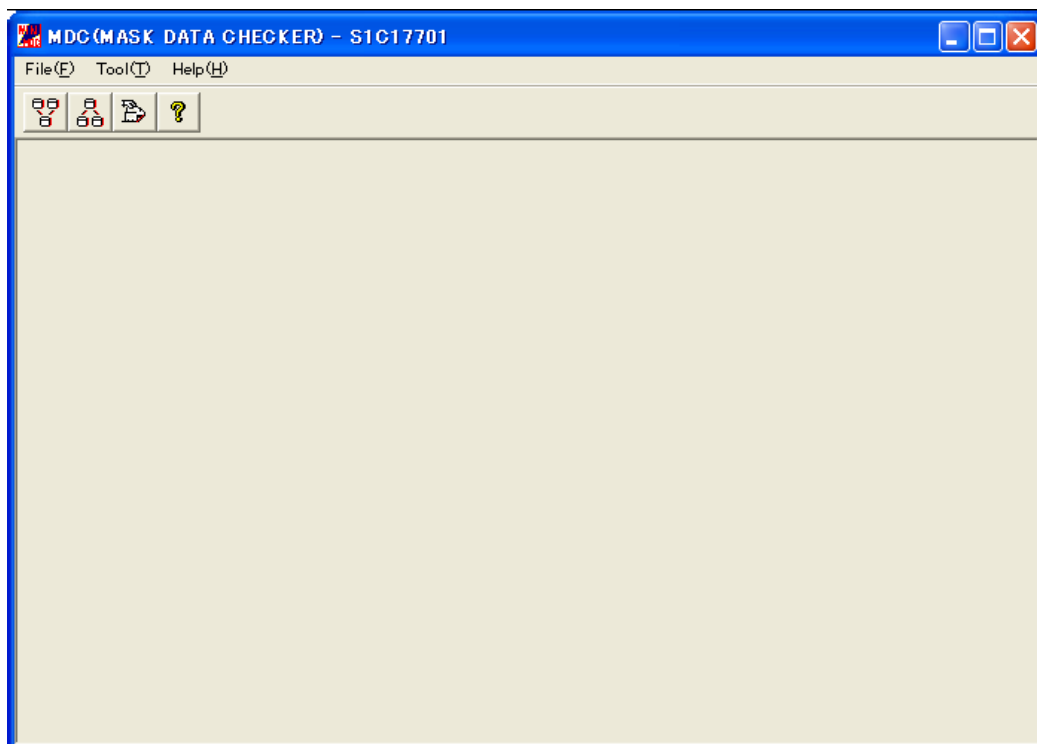


Figure 9.2.4.2 Initial window

(2) Load S1C17xxx.ini

Click the [Device INI Select] button in winmdc17.exe or select [Device INI Select] from the [Tool] menu.

Enter the filename and path in the text box in the dialog box displayed, or click the [Ref] button to load S1C17xxx.ini for the target CPU. The S1C17xxx.ini corresponding to the specific model can be found in the folder below.

User_folder\EPSON\GNU17V3\mcu_model

If no corresponding S1C17xxx.ini exists for the target CPU, contact the Seiko Epson sales operations.

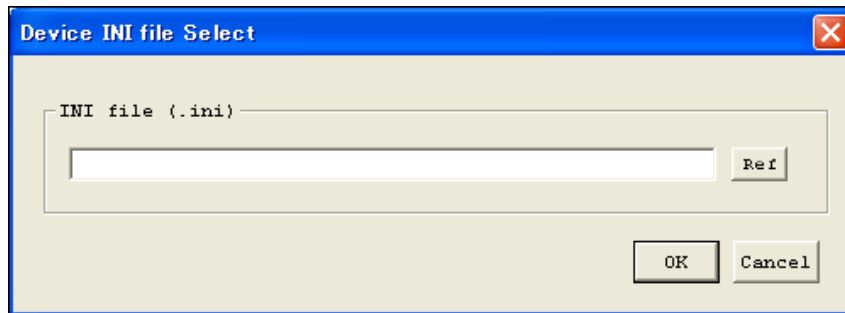


Figure 9.2.4.3 Load window for model-specific information definition file

(3) Select input file

Click the [Unpack] button in winmdc17.exe or select [Unpack] from the [Tool] menu.

Select the file to be unpacked. Click the [Ref] button for [Packed Input Files] to select the PA file to be unpacked.

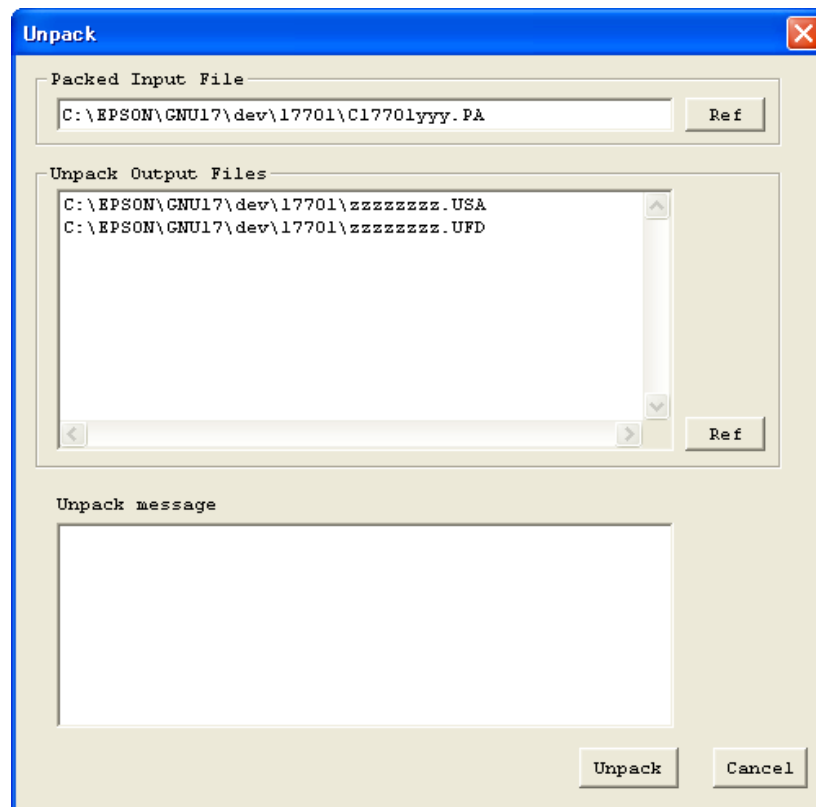


Figure 9.2.4.4 Input file selection window

[Unpack Output Files] can be used to specify the file name to be output. Enter the file name with extension by clicking the [Ref] button for the file listed and select the folder for output. The file extension cannot be changed.

(4) Separate PA file

Click the [Unpack] button on the input file selection window to execute unpacking. The message "Unpack completed!" is displayed in the [Unpack message] area once unpacking has successfully been completed.

With the default settings, the UFD and USA files will be generated in the following folder.

User_folder\EPSON\GNU17V3\dev

Click the [Cancel] button to close the dialog box.

9.3 Error Messages for Submitted Data Creation Tools

9.3.1 winfog17 Error Messages

The error messages output by winfog17 are listed below. "Dialog" in the Display column indicates that the message is displayed in a dialog box, and "Message" indicates that the message is displayed in the [FOG] window message area.

Table 9.3.1.1 List of error messages

Message	Description	Display
File name error	Number of characters in the file name or extension exceeds the limit.	Dialog
Illegal character	Prohibited characters have been entered.	Dialog
Please input file name	File name has not been entered.	Dialog
Can't open File : xxxx	File (xxxx) cannot be opened. Abnormality message (this output is produced, for example, when a file is deleted during debugging).	Dialog
INI file is not found	Specified model-specific information definition file (.ini) does not exist.	Dialog
INI file does not include FOG information	Specified model-specific information definition file (.ini) does not contain function option information.	Dialog
Function Option document file is not found	Specified function option document file does not exist.	Dialog
Function Option document file does not match INI file	Contents of the specified function option document file do not match device information definition file (.ini).	Dialog
A lot of parameter	Too many command line parameters are specified.	Dialog
Making file(s) is completed [xxxx is no data exist]	Finished creating the file, but the created file (xxxx) does not contain any data.	Message
Can't open File: xxxx Making file(s) is not completed	Cannot open file (xxxx) when executing Generate.	Message
Can't write File: xxxx Making file(s) is not completed	Cannot write in file (xxxx) when executing Generate.	Message

Table 9.3.1.2 Warning messages

Message	Description	Display
Are you file update? xxxx is already exist	Overwrite confirmation message (Specified file already exists.)	Dialog

9.3.2 winmdc17 Error Messages

The error messages output by winmdc17 are listed below. "Dialog" in the Display column indicates that the message is displayed in a dialog box, and "Message" indicates that the message is displayed in the [Pack] or [Unpack] dialog box message area.

Table 9.3.2.1 List of I/O error messages

Message	Description	Display
File name error	Number of characters in the file name or extension exceeds the limit.	Dialog
Illegal character	Prohibited characters have been entered.	Dialog
Please input file name	File name has not been entered.	Dialog
INI file is not found	Specified model-specific information definition file (.ini) does not exist.	Dialog
INI file does not include MDC information	Specified model-specific information definition file (.ini) does not contain MDC information.	Dialog
Can't open file : xxxx	Cannot open file (xxxx).	Dialog
Can't write file: xxxx	Cannot write in file (xxxx).	Dialog

Table 9.3.2.2 List of ROM data error messages

Message	Description	Display
Hex data error: Not S record.	Data does not begin with "S".	Message
Hex data error: Data is not sequential.	Data is not listed in ascending order.	Message
Hex data error: Illegal data.	Invalid character is included.	Message
Hex data error: Too many data in one line.	Too many data entries exist in one line.	Message
Hex data error: Check sum error.	Checksum does not match.	Message
Hex data error: ROM capacity over.	Data is large. (Greater than ROM size)	Message
Hex data error: Not enough the ROM data.	Data is small. (Smaller than ROM size)	Message
Hex data error: Illegal start mark.	Start mark is incorrect.	Message
Hex data error: Illegal end mark.	End mark is incorrect.	Message
Hex data error: Illegal comment.	Model name shown at the beginning of data is incorrect.	Message

Table 9.3.2.3 List of function option data error messages

Message	Description	Display
Option data error : Illegal model name.	Model name is incorrect.	Message
Option data error : Illegal version.	Version is incorrect.	Message
Option data error : Illegal option number.	Option No. is incorrect.	Message
Option data error : Illegal select number.	Selected number is incorrect.	Message
Option data error : Mask data is not enough.	ROM data is insufficient.	Message
Option data error : Illegal start mark.	Start mark is incorrect.	Message
Option data error : Illegal end mark.	End mark is incorrect.	Message

9.4 Sample Output for Submitted Data Creation Tools

The file formats are shown below for the FDC (Function option document) and PA (Data to be submitted) files generated using winfog17.exe and winmdc17.exe.

Note: The configuration and contents of data will vary depending on the device type.

● Example of an FDC file

* S1C17xxx_xxxKB FUNCTION OPTION DOCUMENT Vx.xx	←Version
*	
* FILE NAME zzzzzzzz.FDC	←File name (specified in [Setup])
* USER'S NAME SEIKO EPSON CORPORATION	←User name (specified in [Setup])
* INPUT DATE yyyy/mm/dd	←Creation date (specified in [Setup])
* COMMENT SAMPLE DATA	←Comment (specified in [Setup])
*	
* *** OPTION NO.1 ***	←Option No.
* --- OSC1 SYSTEM CLOCK ---	←Option name
* Crystal(32.768KHz) ---- Selected	←Selected specification
OPT0101 01	←Mask data
*	
* *** OPTION NO.2 ***	
* --- OSC3 SYSTEM CLOCK ---	
* CR 200KHz ---- Selected	
OPT0201 01	
*	
* *** OPTION NO.3 ***	
* --- INPUT PORT PULL UP RESISTOR ---	
* K00 With Resistor ---- Selected	
* K01 With Resistor ---- Selected	
* K02 With Resistor ---- Selected	
* K03 With Resistor ---- Selected	
* K04 With Resistor ---- Selected	
* K05 With Resistor ---- Selected	
* K06 With Resistor ---- Selected	
* K07 With Resistor ---- Selected	
OPT0301 01	
OPT0302 01	
OPT0303 01	
OPT0304 01	
OPT0305 01	
OPT0306 01	
OPT0307 01	
OPT0308 01	
*	
* *** OPTION NO.4 ***	
* --- OUTPUT PORT OUTPUT SPECIFICATION ---	
* R00 Complementary ---- Selected	
* R01 Complementary ---- Selected	
* R02 Complementary ---- Selected	
* R03 Complementary ---- Selected	
OPT0401 01	
OPT0402 01	
OPT0403 01	
OPT0404 01	
*	
:	
*	

```

* *** OPTION NO.8 ***
* --- SOUND GENERATOR POLARITY ---
* NEGATIVE ---- Selected
OPT0801 01
*EOF

```

←End mark

● Example of a PA file

```

*
* S1C17xxx_xxB xPCS xBITW xBITR MASK DATA VER x.xx
*
\ROM1
S1C17xxxxyy PROGRAM ROM
S224008000
: : : : : "xxxxxxxx.psa"
S804000000FB
\END
\OPTION1
* S1C17xxx FUNCTION OPTION DOCUMENT V x.xx
*
* FILE NAME zzzzzzzz.FDC
* USER'S NAME SEIKO EPSON CORPORATION
* INPUT DATE yyyy/mm/dd
* COMMENT SAMPLE DATA
* "xxxxxxxx.fdc"
* *** OPTION NO.1 ***
* --- OSC1 SYSTEM CLOCK ---
* Crystal(32.768KHz) ---- Selected
OPT0101 01
: : : : :
OPTnn01 01
*EOF
\END

```

←Version

←ROM data start mark

←Model name

←ROM data end mark

←Function option start mark

←Model name/version

←Function option end mark

10 Other Tools

This chapter explains the other tools that are included in the S1C17 Family C Compiler Package.

10.1 objdump.exe

10.1.1 Function

The **objdump** displays the internal data of binary files in elf format. Disassembled code, raw data, section configuration, section map addresses, data size and relocatable information symbol tables can be displayed.

Refer to the documents for the gnu utilities for details of **objdump**.

10.1.2 Input Files

● Executable object file

File format: Binary file in elf format
 File name: `<filename> .elf`
 Description: An executable object file after the linkage process by the linker has been completed.
 The contents will be displayed using the absolute addresses.

● Object file

File format: Binary file in elf format
 File name: `<filename> .o`
 Description: An object file after assembled.
 The contents will be displayed using the relative addresses from the beginning of the file or section.

10.1.3 Method for Using objdump

● Startup format

objdump *<options>* *<input file name>*
<input file name>: Specify a object file name to be dumped.

● Options

The following startup options can be specified:

-d

Function: **Display disassembled contents**

Explanation: Displays all the executable sections after disassembling the object code. No source is displayed together.

-h

Function: **Display section information**

Explanation: Displays the section configuration, section size and address.

-g (gcc4 only)

Function: **Display information converted from debugging information**

Explanation: Displays the relations between sources and addresses based on the debugging information. The data types of the global symbols are also displayed.

-t

Function: **Display global symbol information**

Explanation: Displays a list of the global symbols including the local labels.

-s

Function: **Display in hexadecimal dump format**

Explanation: Displays all the section information in hexadecimal dump format. Data corresponding to unresolved symbols cannot be displayed correctly.

-D

Function: **Display disassembled contents for all sections**

Explanation: Displays all the sections after disassembling the object code.

-G

Function: **Display raw data of debugging information**

Explanation: Displays the raw data of the debugging information in stab format.

-S

Function: **Mixed display**

Explanation: Displays all the executable sections after disassembling the object code. The source code is also displayed with the corresponding disassembled code if possible.

When entering an option, you need to place one or more spaces before and after the option.

Example: `c:\EPSON\GNU17V3\GCC6\objdump -S test.elf`

10.1.4 Error Message

The following shows the error message generated by **objdump**:

Table 10.1.4.1 Error message

Error message	Description
<code>/cygdrive/X/path to objdump/objdump: filename:</code> File format not recognized	An unrecognized file (<i>filename</i>) is specified. Specify an elf format file.

10.1.5 Precautions

- The disassembled display may be aborted halfway if the amount of information is too large.
- When a `.o` file before linking is dumped, the relative addresses from the beginning of each section are displayed, not the absolute addresses. In this case the beginning of each section is address `0x0`.

10.2 objcopy.exe

10.2.1 Function

The **objcopy** is the gnu standard object file format conversion utility, and it copies and converts data format of object files.

In application development for the S1C17 Family, this tool is used to convert an elf format object file into Motorola S3 format files so that data can be written to the ROMs.

Although **objcopy** supports many functions (options) and file formats, this section treats only the elf to Motorola S3 format file conversion function. Refer to the documents for the gnu utilities for details of **objcopy**.

10.2.2 Input/Output Files

Input file

- Object file

File format: Binary file in elf format

File name: *<filename>* .elf

Description: An executable object file after the linkage process by the linker has been completed.

Output file

- SA file (ROM data)

File format: Motorola S3 format file

File name: *<filename>* .sa

Description: A file for writing to the ROM. When the system uses two or more ROMs, create a data file for each ROM by extracting the section data to write to the ROM from the elf object file.

10.2.3 Method for Using objcopy

● Startup format

`objcopy <option> <input file name> [<output file name>]`

[] indicates the possibility to omit.

<input file name>: Specify an elf format object file name to be converted.

<output file name>: Specify the Motorola S3 format file name after conversion.

Note: When <output file name> is omitted, `objcopy` creates a temporary file used to output the converted data, and renames it with the input file name after the process has been completed. Therefore, the input file is destroyed.

● Options

The following options are mainly used in application development for the S1C17 Family:

-I elf32-little

Function: **Specifies the input file format**

Explanation: Specifies elf as the input file format.

-O srec

Function: **Output in Motorola format**

Explanation: Specifies the Motorola format as the output file format. This option must be specified together with '-I elf32-little'.

-O binary

Function: **Output in binary format**

Explanation: Specifies binary format as the output file format. This option must be specified together with '-I elf32-little'.

--srec-forceS3

Function: **Specify Motorola S3 format**

Explanation: Specifies the Motorola S3 format as the output file format. This option must be specified with the `-O srec` option.

Example: `-O srec --srec-forceS3 ...`

-R SectionName

Function: **Remove section**

Explanation: Specifies that the section named *SectionName* should not be included in the output file. This option can be specified multiple times in a command line. This option must be specified together with '-I elf32-little'.

-v (or --verbose)

Function: **Verbose output mode**

Explanation: Displays the converted object file names.

-V (or --version)

Function: **Display version number**

Explanation: Displays the version number of `objcopy`, and then terminates the process.

--help

Function: **Usage display**

Explanation: Displays the usage of `objcopy`, and then terminates the process

10.2.4 Creating SA Files (ROM Data)

Open the command prompt window and execute **objcopy** at the command line as shown below.

```
C:\EPSON\GNU17V3\gcc6\objcopy -I elf32-little -O srec -R SectionName --srec-forceS3 InputFile  
OutputFile
```

Running the above command converts sections other than those specified with the **-R** option into S3 records and generates an output file.

Example: Extract all section data from `input.elf` and write the data to `output.sa`.

```
C:\EPSON\GNU17V3\GCC6\objcopy -I elf32-little -O srec --srec-forceS3 input.elf  
output.sa
```

10.3 ar.exe

10.3.1 Function

The **ar** is the gnu standard utility for maintenance of archived files. This utility is used to create and update library files that can be used with the linker **ld**. Refer to the documents for the gnu utilities for details of **ar**.

10.3.2 Input/Output Files

● Object file

File format: Binary file in elf

File name: *<filename>.o*

Description: A relocatable object file.

The **ar** can add files in this format into an archive or extract an object from an archive to generate a file in this format.

● Archive file (library file)

File format: Archive file in binary format

File name: *<filename>.a*

Description: A library file that can be input to the linker **ld**.

10.3.3 Method for Using ar

● Startup format

ar *<key>* [*<modifier>*] [*<add position>*] *<archive>* [*<objects>*]

[] indicates the possibility to omit.

<key>, *<modifier>*: Specify a process.

<add position>: Specify the location in the archive for inserting *<objects>* using the object name in the archive.

<archive>: Specify an archive file to be edited.

<objects>: Specify object file names to be added, extracted, moved or removed. Multiple file names can be specified by separating between the file names with a space.

● Keys

- d** Removes *<objects>* from the archive.
- m** Moves *<objects>* to the end of the archive. By specifying with modifier 'a' or 'b', the location in the archive where *<objects>* are moved can be specified.
- q** Adds *<objects>* at the end of the archive. This function does not update the symbol table in the archive.
- r** Replaces *<objects>* in the archive with the object files with the same name. If the archive does not contain *<objects>*, the *<objects>* files are added at the end of the archive. (By specifying with modifier 'a' or 'b', the location in the archive where *<objects>* are added can be specified.)
- t** Displays the list of objects in the archive or the list of the specified *<objects>*.
- x** Extracts *<objects>* from the archive and creates the object files. When *<objects>* are omitted, all the objects in the archive are extracted to create the files.

● Modifiers

- a** Use this modifier with key 'r' or 'm' to place *<objects>* behind the *<add position>*. Specify an object name located at the *<add position>* in the archive file.
- b** This modifier has the same function as 'a' but *<objects>* are placed in front of the *<add position>*.
- s** Forcibly updates the symbol table in the archive.
- u** Use this modifier with key 'r' to replace only the updated objects in the *<objects>* that are newer than those included in the archive.
- v** Specifies verbose mode to display the executed processes.

Do not enter a space between the keys and modifiers.

● Usage examples

(1) Creating a new archive

```
ar rs mylib.a func1.o func3.o
(mylib.a: func1.o + func3.o)
```

When the specified archive (`mylib.a`) does not exist, a new archive is created and the specified object files (`func1.o` and `func3.o`) are added into it in the specified order.

(2) Adding objects

```
ar rs mylib.a func4.o func5.o
(mylib.a: func1.o + func3.o + func4.o + func5.o)
func4.o and func5.o are added at the end of mylib.a.
```

(3) Adding an object to the specified location

```
ar ras func1.o mylib.a func2.o
(mylib.a: func1.o + func2.o + func3.o + func4.o + func5.o)
func2.o is added behind the func1.o in mylib.a.
```

(4) Replacing objects

```
ar rus mylib.a func1.o func2.o func3.o func4.o func5.o
(mylib.a: func1.o + func2.o + func3.o + func4.o + func5.o)
```

If there are files from among `func1.o`, `func2.o`, `func3.o`, `func4.o` and `func5.o` that have been updated after they have been added into `mylib.a`, the objects in `mylib.a` are replaced with the newer files. The objects that have not been updated are not replaced.

(5) Extracting an object

```
ar x mylib.a func5.o
(mylib.a: func1.o + func2.o + func3.o + func4.o + func5.o)
func5.o is extracted from mylib.a and an object file is created. The archive is not modified.
```

(6) Removing an object

```
ar ds mylib.a func5.o
(mylib.a: func1.o + func2.o + func3.o + func4.o)
func5.o is removed from mylib.a.
```

10.4 moto2ff.exe

10.4.1 Function

The moto2ff loads a Motorola S3 format file with a given start address and block size and fills the unused area of the file with 0xff to generate an output file.

In applications development for the S1C17 Family, the moto2ff is used to retrieve ROM area data from the Motorola S3 format file generated by objcopy.

The ROM area data generated by moto2ff should be processed with sconv32 and winmdc17 to generate the PA file (Data to be submitted) to be ultimately submitted to Seiko Epson. For more information on the PA file generation procedure, refer to Section 9.2, "Procedure for Creating Data to Be Submitted".

10.4.2 Input/Output Files

Input file

● SA file (ROM data)

File format: Motorola S3 format file
 File name: *<filename> .sa*
 Description: A Motorola S3 format file converted from an elf format executable file by the **objcopy**.

Output file

● SAF file (ROM data)

File format: Motorola S3 format file
 File name: *<filename> .saf*
 Description: A data file of the specified address in which the unused area is filled with 0xff.

10.4.3 Startup Format

moto2ff *<data start address><data block size> <input file name>*

<data start address>: Specify the data output start address in the input file using a hexadecimal number.
<data block size> : Specify the output data block size in bytes using a hexadecimal number.
<input file name> : Specify the file name of the Motorola S3 format file to be filled with 0xff.
 The file name must be within 128 characters including a path and an extension. Path can be specified for the input file, note, however, that the output file will be located in the current directory.

- Usage will be displayed when no parameters are specified.
- If the output file already exists, it will be overwritten.
- When an error occurs, an error message is displayed and the output file is not generated.
- If the input Motorola S3 file contains data that exceeds the range specified by a start address and a block size, the following message appears and the output file is not generated.
 Error: FILENAME contains data outside of specified range (STARTADDR:SIZE)
- If Motorola S3 data records are in the same address, the first data is overwritten by the last.

- Make sure that the data start address and data block size are correct values for the model by referring its technical manual. If an incorrect value is input, an error will occur in the **winmdc17** process to generate final PA file.
- When **moto2ff** has completed successfully, the following message is shown in the standard output.
moto2ff : Convert Completed
- [-f]: Forced output option. Even when the input Motorola S3 format file has data in the range beyond that specified by the start address and the block size, the specified range is cut out, and the output file is generated with the unused area filled by 0xff. If out-of-range data is present, a warning is output.

10.4.4 Error/Warning Messages

The following shows the error and warning messages generated by the **moto2ff**:

Table 10.4.4.1 Error messages

Error message	Description
Input filename is over 128 letters.	The input file name has exceeded 128 characters.
Cannot open input file " <i>FILENAME</i> ".	The input file <i>FILENAME</i> cannot be opened.
Cannot open output file " <i>FILENAME</i> ".	The output file <i>FILENAME</i> cannot be opened.
Motorola S3 checksum error.	A checksum error occurred while reading Motorola S3 format file.
Cannot allocate memory.	Cannot allocate memory.
<i>FILENAME</i> contains data outside of specified range (" <i>STARTADDR</i> ";" <i>SIZE</i> ")	The input file <i>FILENAME</i> contains data that exceeds the specified range (<i>SIZE</i> bytes from <i>STARTADDR</i>). The output file is not generated.

Table 10.4.4.2 Warning messages

Warning message	Description
Invalid file format in " <i>FILENAME</i> " line " <i>NUMBER</i> ".	The input file <i>FILENAME</i> contains an invalid format data at line <i>NUMBER</i> .
<i>FILENAME</i> contains data outside of specified range (" <i>STARTADDR</i> ";" <i>SIZE</i> ")	Although <i>FILENAME</i> contains data outside the specified range (from " <i>STARTADDR</i> " to " <i>SIZE</i> "), an output file is generated due to the forced output option, -f.

10.4.5 Creating SAF File (ROM Data)

After a Motorola S3 format SA file (ROM data) has been generated by **objcopy**, create an SAF file using **moto2ff**.

Open the command prompt window and execute **moto2ff** as shown below.

Example: C:\EPSON\GNU17V3\>moto2ff 8000 10000 input.sa

The command above outputs the data of 0x10000 bytes starting from address 0x8000 contained in *input.sa* to *input.saf*.

The unused addresses within the range from addresses 0x8000 to 0x17fff are filled with 0xff.

The SAF file for the internal ROM is generated by the above procedure.

Next convert the SAF file generated here into the Motorola S2 format PSA file (ROM data) using **sconv32**. Then perform the final verification of program operation on the actual target board using that file.

Finally, pack the verified PSA file and the FDC file (Function option document) generated by **winfog17** into a single PA file using **winmdc17** and submit this to Seiko Epson.

10.5 sconv32.exe

10.5.1 Function

The **sconv32** is a tool to convert a Motorola S format into another S format. In an application development for the S1C17 Family, **sconv32** is used to convert the Motorola S3 format SAF file (ROM data) generated by **moto2ff** into the Motorola S2 format.

The file should be processed with **winmdc17** to generate the PA file (Data to be submitted) to be ultimately submitted to Seiko Epson after verifying program operation on the actual target board using the PSA file (ROM data). For more information on the PA file generation procedure, refer to Section 9.2, "Procedure for Creating Data to Be Submitted".

10.5.2 Input/Output Files

Input file

● SAF file

File format: Motorola S3 format file
 File name: *<filename>*.saf
 Description: A Motorola S3 format file generated by **moto2ff**.

Output file

● PSA file

File format: Motorola S2 format file
 File name: *<filename>*.psa
 Description: The Motorola S2 format file converted from the input file.

10.5.3 Startup Format

sconv32 S2 <input file name> <output file name>

S2: This is a switch to convert the input file into the Motorola S2 format.
<input file name> : Specify a SAF file generated by **moto2ff**.
<output file name> : Specify the output file name that will be converted into the Motorola S2 format.
 The file extension must be ".psa".

- Usage will be displayed when no parameters are specified.
- If the output file already exists, it will be overwritten.
- When an error occurs, an error message is output to the standard error device and the output file is not generated.
- The [Esc] key can be used to forcibly terminate the process while converting.
- When **sconv32** has completed successfully, the following message is displayed.

Sconv32 : Convert Completed. End message *1

*1: Output to the standard output

10.5.4 Error Messages

The following shows the error messages generated by the **sconv32**:

Table 10.5.4.1 Error messages

Error message	Description
INVALID SWITCH.	An invalid switch is specified.
COMPLEMENT SWITCH ERROR.	The specified complement of the checksum for the output file is incorrect.
S FORMAT TYPE ERROR.	The specified S format for the output file is incorrect.
NO INPUT FILE NAME.	An input file is not specified.
NO OUTPUT FILE NAME.	An output file is not specified.
INPUT SAME FILE.	The same file name is specified for input and output.
CANNOT OPEN SOURCE FILE (<i>filename</i>).	The specified input file cannot be found or cannot be opened.
CANNOT OPEN DESTINATION FILE (<i>filename</i>).	The output file cannot be opened.
SOURCE RECORD TYPE NOT SUPPORT.	The input file has an unsupported record type.
ADDRESS LENGTH RANGE OVER.	The address range of the input file exceeds the address range for the S format to be converted.
OTHER ERROR.	Another error has occurred.

10.6 gpdata.exe

10.6.1 Function

gpdata reads binary data (in binary format) to generate the "gpdata.bin" file including user setting information for use by Gang Programmer (S5U1C17001W2).

For details of **gpdata**, refer to the Gang Programmer user manual or the documentation provided with **gpdata**.

10.6.2 Input/Output Files

Input file

● BIN file

File format: Binary file
 File name: <filename>.bin
 Description: A binary file generated by **objcopy** etc.

Output file

● gpdata.bin file

File format: Gang Programmer user setting/program data file
 File name: gpdata.bin
 Description: A file containing user setting information added to binary data

10.6.3 Method for Using gpdata

● Startup format

gpdata <input file name> <options>

<input file name>: Specify binary data generated by **objcopy** etc.

● Options

The following startup options can be specified. For details, refer to the Gang Programmer user manual or the documentation provided with **gpdata**.

- v **Select verify method**
- d **Select interface voltage level (required)**
- b **Select whether buzzer sounds when program ends**
- t **Set model name (required)**
- a **User program location address (required)**
- i **Set serial number initial value**
- s **Set serial number write start address**
- p **Flash memory security password**
- f **Specify parameter input file**

10.7 ptd.exe

10.7.1 Function

ptd reads Motorola S format files, changes the data for the address specified, and re-outputs it in Motorola S format.

This is used to generate PSA files with flash protection set by embedding flash protection bits in Motorola S2 format PSA files (ROM data) generated with **sconv32** for S1C17 Family application development.

10.7.2 Input/Output Files

Input file

● PSA file

File format: Motorola S2 format file
 File name: *<filename>.psa*
 Description: A Motorola S2 format file generated by **sconv32**

Output file

● PSA file

File format: Motorola S2 format file
 File name: *<filename>_ptd.psa*
 Description: A file with flash protection set for the input data file

10.7.3 Method for Using ptd.exe

● Startup format

ptd *<options>* *<input file name>*

<input file name>: Specify PSA file name generated by **sconv32**.

- If no argument is specified, a help message is displayed.
- If there are no changes to the input file, processing will end normally, but no output file will be generated.
- If an error occurs, an error message is output to the standard error. No output file is generated.
- If processing ends successfully, the following message is output to standard output:

`ptd:Convert Completed: Completion message`

● Options

The following startup options can be specified. If an unknown option is specified, a help message is displayed.

-o *<output file name>*

Function: **Specify output file name**

Explanation: Specifies the name of the file to which data change results are output. If not specified, the output file "*<input>_ptd.psa*" is generated for the input file "*<input>.psa*." If an output file with the same name already exists, it will be overwritten.

-w *<address>=<data>* [*<data>*]

Function: **Specify data to be written to address**

Explanation: Specifies the data to be written to the address in 16-bit units. If multiple data is specified, data is written to subsequent addresses. An error will occur if the address specified does not exist in the input file.

10.7.4 Error Messages

The error messages output by **ptd** are as follows:

Table 10.7.4.1 Error messages

Error message	Details
Cannot open input file "FILENAME".	The input file FILENAME cannot be opened.
Cannot open output file "FILENAME".	The output file FILENAME cannot be opened.
S-format checksum error.	A checksum error occurred while reading an S-format file.
Unknown address format "ADDRESS"	The format of the specified address ADDRESS is incorrect.
Unknown data format "DATA"	The format of the specified data DATA is incorrect.
Address "ADDRESS" is out of range	The input data does not contain the specified address ADDRESS.

10.7.5 Method for Setting Flash Protection

The S1C17 Family includes models (e.g., S1C17554) that allow flash protection setting to protect the contents of the internal flash memory. To set flash protection, create a Motorola S2 format PSA file (ROM data) using **sconv32**, then create a PSA file with flash protection set using **ptd**.

Run **ptd** as follows at the command prompt.

Example: C:\EPSON\GNU17V3\>ptd s1c17554.psa -e 0x27ffc=0xff80,0xffff

This command rewrites the value at address 0x27ffc to 0xff80 and the value at address 0x27ffe to 0xffff inside s1c17554.psa and outputs to s1c17554_ptd.psa. The addresses for Flash Protection Bits in S1C17754 are 0x27ffc (write-protect) and 0x27ffe (data-read-protect). In this example, writing s1c17554_ptd.psa with the Flash Protection Bits modified to the internal flash memory in S1C17754 prevents data writing or sector deletion in the range 0x8000-0x23fff.

Note: The ability to set flash protection and the addresses of the Flash Protection Bits differ from model to model. For details, refer to the technical manual for the specific model.

10.8 LcdUtil17 (LCD Panel Customizing Tool)

10.8.1 Overview

The LCD panel customizing tool (LcdUtil17) produces an LCD file that describes the LCD panel layout and COM/SEG port allocation. This file is used by the ES-Sim17 (v.1.2 or newer) built-in simulator to simulate a monochrome LCD panel screen. LcdUtil17 produces a layout of the segment LCD from a bitmap file (.bmp), allowing the ES-Sim17 to simulate the screen that would appear on an actual product. This tool also lets users produce dot-matrix LCD layouts.

10.8.2 Input/Output files

Figure 10.8.2.1 shows the LcdUtil17 input and output files.

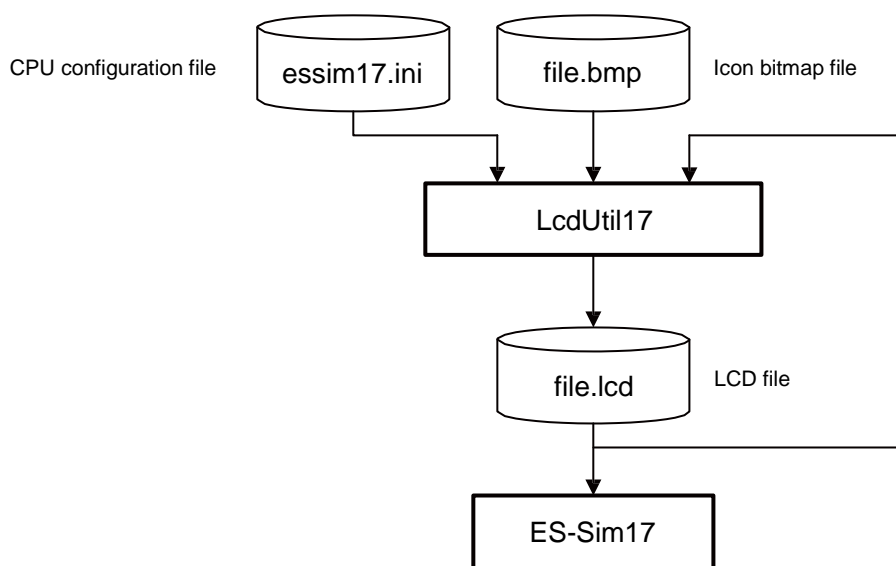


Figure 10.8.2.1 LcdUtil17 input/output files

● CPU configuration file (essim17.ini)

This file contains recorded information on the simulator model. Be sure to use the setting file provided by Seiko Epson. Modifying the contents of this file may prevent the LcdUtil17 and ES-Sim17 operating properly.

● Bitmap file (file_name.bmp)

This bitmap file contains an LCD panel image (in monochrome). LcdUtil17 reads this bitmap file, then loads each part as an icon, allowing editing of the layout in the LcdUtil17 window.

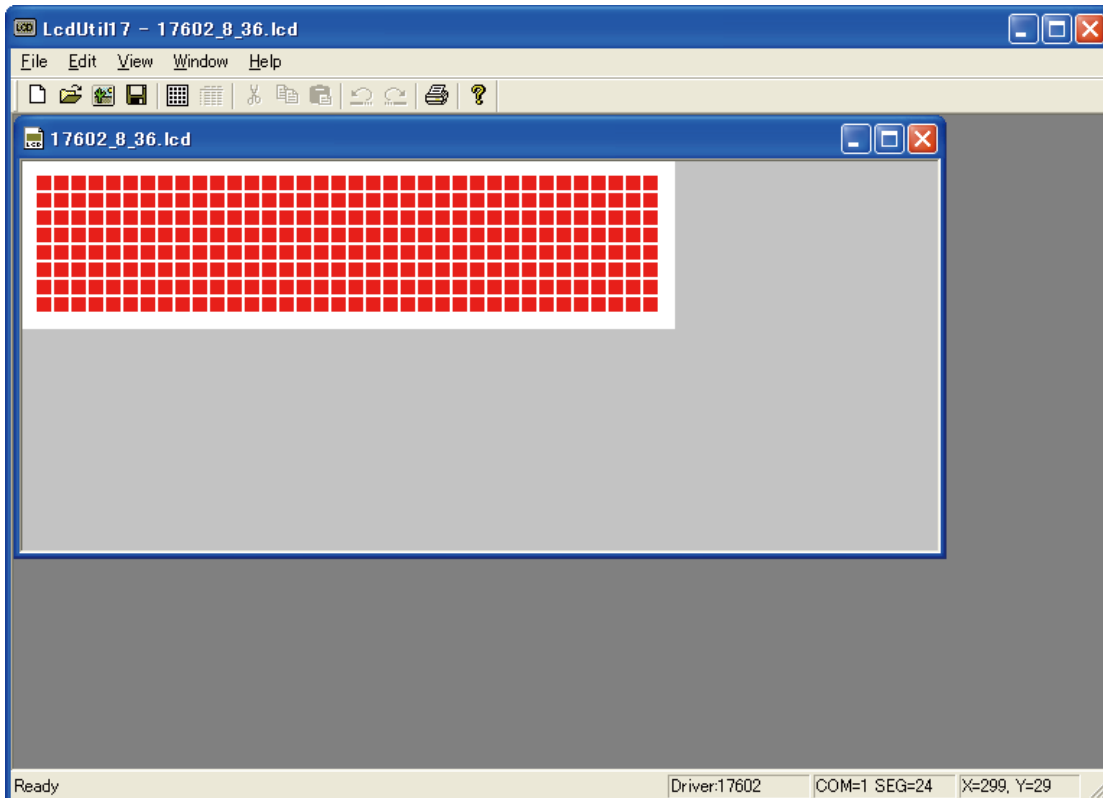
● LCD file

This file contains an LCD panel layout and COM/SEG allocation information and is loaded into the simulator for LCD screen simulation.

10.8.3 Starting and Closing LCDUtil17

To start LcdUtil17, select [Launch LcdUtility] from the [C17] menu of the IDE.
To end the LcdUtil17 program, select [Exit] from the [File] menu.

10.8.4 Window



● Panel editing window

Opening a bitmap file (.bmp) or LCD file (.lcd) will display the file in this window. This window is used to design an LCD panel layout and assign COM/SEG.

Two or more windows can be opened at the same time, and icons and dot matrices can be dragged and dropped between two windows.

10.8.5 Menus and Toolbar

10.8.5.1 Menus

[File] menu

File	
New	Ctrl+N
Open...	Ctrl+O
Open Bitmap File...	
Close	
Save	Ctrl+S
Save As...	
Print...	Ctrl+P
Print Preview	
Print Setup...	
1 17701_16_72.lcd	
2 17701_16_72_02.lcd	
3 seg_sample02.bmp	
4 17701_32_56.lcd	
5 17602_4_40.lcd	
6 SVT17701.lcd	
7 17602_8_36.lcd	
8 seg_sample.bmp	
Exit	

[New] ([Ctrl]+[N])

Opens a new panel editing window.

[Open...] ([Ctrl]+[O])

Opens an LCD file (.lcd).

[Open Bitmap File...]

Opens a bitmap file (.bmp).

[Close]

Closes the active panel editing window.

[Save] ([Ctrl]+[S])

Saves the contents of the active panel editing window to the LCD file (.lcd) (by overwriting).

[Save As...]

Saves the contents of the active panel editing window to an LCD file (.lcd) under a new name.

[Print...] ([Ctrl]+[P])

Prints the bitmap data in the active panel editing window.

[Print Preview]

Displays the print image of the active panel editing window. Displays the print image of the active panel editing window.

[Print Setup...]

Opens the dialog box used for selecting the paper size or printer to use.

File list

Displays up to eight previously opened files and enables access to those files.

[Exit]

Ends LcdUtil17.

[Edit] menu

Edit	
Cut	Ctrl+X
Copy	Ctrl+C
Paste	Ctrl+V
Insert dot matrix	Ctrl+M
Icon List	Ctrl+I
Resize LCD	
Group Icon	
Release Group	

[Cut] ([Ctrl]+[X])

Cuts the part selected in the panel editing window and copies it to the clipboard.

[Copy] ([Ctrl]+[C])

Copies the part selected in the panel editing window to the clipboard.

[Paste] ([Ctrl]+[V])

Pastes the part copied in the clipboard to the upper left corner of the panel editing window.

[Insert dot matrix] ([Ctrl]+[M])

Inserts a dot matrix in the panel editing window. Use the dialog box to change dimensional settings, if necessary.

[Icon List] ([Ctrl]+[I])

Displays a list of icons appearing in the active panel editing window. This list can also be used to allocate COM/SEG.

[Resize LCD]

Sets the size of the LCD panel. The default size of the new panel editing window is 640 x 480.

[Group Icon]

Sets multiple icons as a group.

[Release Group]

Cancels the grouping and returns the grouped icons to separate icons.

[View] menu

View	
✓ Toolbar	
✓ Status Bar	

[Toolbar]

Displays or hides the toolbar.

[Status Bar]

Displays or hides the status bar.

[Window] menu

Window	
Cascade	
Tile	
Arrange Icons	
✓ 1 17701_16_72.lcd	

[Cascade]

Rearranges all open panel editing windows in cascading format.

[Tile]

Rearranges all open panel editing windows in tiled format.

[Arrange Icons]

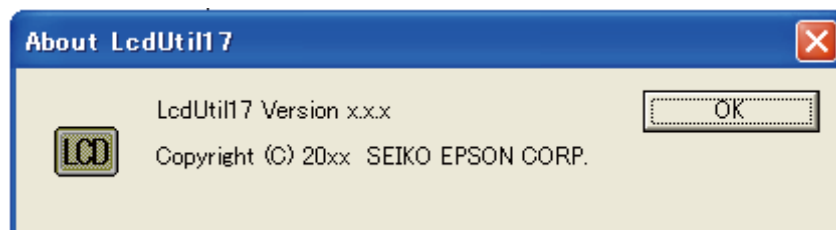
Minimizes all open panel editing windows to icons at the bottom of the window.

Window list

Displays a list of names of all currently open panel editing windows. Select a panel editing window in the list to activate the selected panel editing window.

[Help] menu**[About LcdUtil...]**

Displays LcdUtil17 version information.

**10.8.5.2 Toolbar Buttons****[New] button**

Opens a new panel editing window.

**[Open] button**

Opens an LCD file (.lcd).

**[Bitmap] button**

Opens a bitmap file (.bmp).

**[Save] button**

Saves the contents of the active panel editing window to the LCD file (.lcd) (by overwriting).

**[Dot Matrix] button**

Inserts a dot matrix in the panel editing window.

**[Icon List] button : [Edit]-[Icon List]**

Displays a list of icons appearing in the active panel editing window.

**[Cut] button**

Cuts the part selected in the panel editing window and copies it to the clipboard.

**[Copy] button**

Copies the part selected in the panel editing window to the clipboard.

**[Paste] button**

Pastes the part copied in the clipboard to the upper left corner of the panel editing window.

**[Undo] button**

Cancels up to three of the most recent operations.

Commands that can be undone include Move, Cut, and Paste for icon/dot matrix, Change of SEG/COM, Group Icon, Release Group

**[Redo] button**

Reperforms the command most recently undone with Undo.

**[Print] button**

Prints the bitmap image in the active panel editing window.

**[About] button**

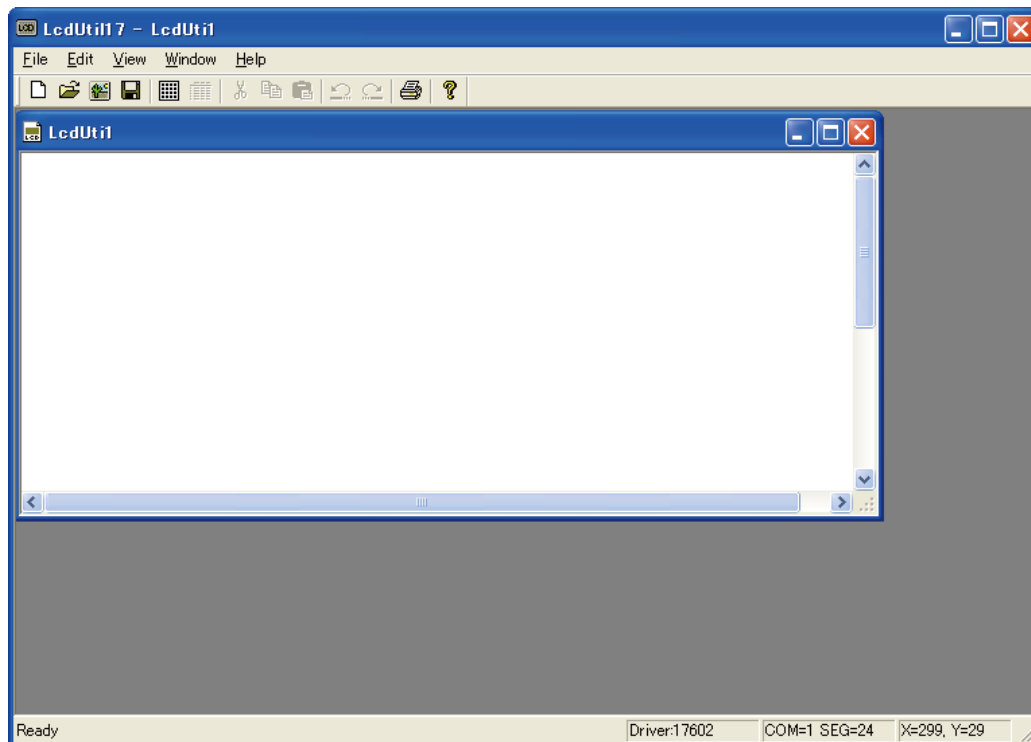
Displays LcdUtil17 version information.

10.8.6 Producing an LCD file

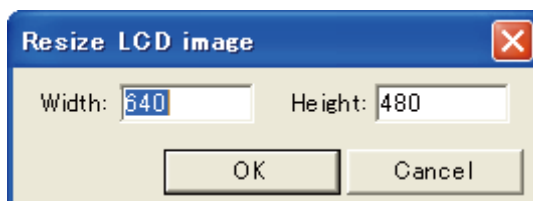
The panel editing window allows icons and dot matrices to be laid out in the same way as they would appear on an actual LCD panel. It also permits COM/SEG allocation. Described below are the procedures for producing an LCD file.

10.8.6.1 Producing a Dot Matrix LCD Panel

- 1) Select [New] from the [File] menu.
A blank panel editing window will open.

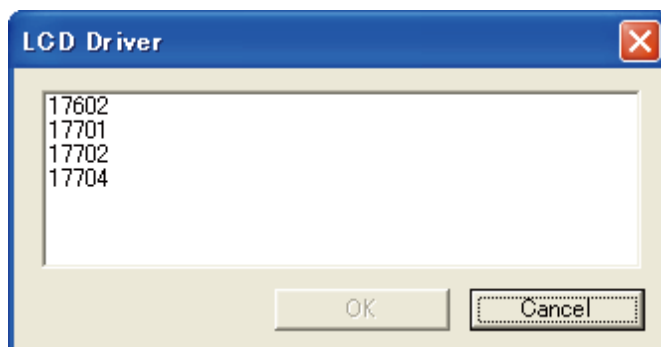


- 2) Select [Resize LCD] from the [Edit] menu.
The [Resize LCD image] dialog box will open.



Enter the LCD panel size, then click the [OK] button.
The default LCD size of a new panel editing window is 640 x 480 dots.
Calculate the dot matrix size and set the LCD panel size.

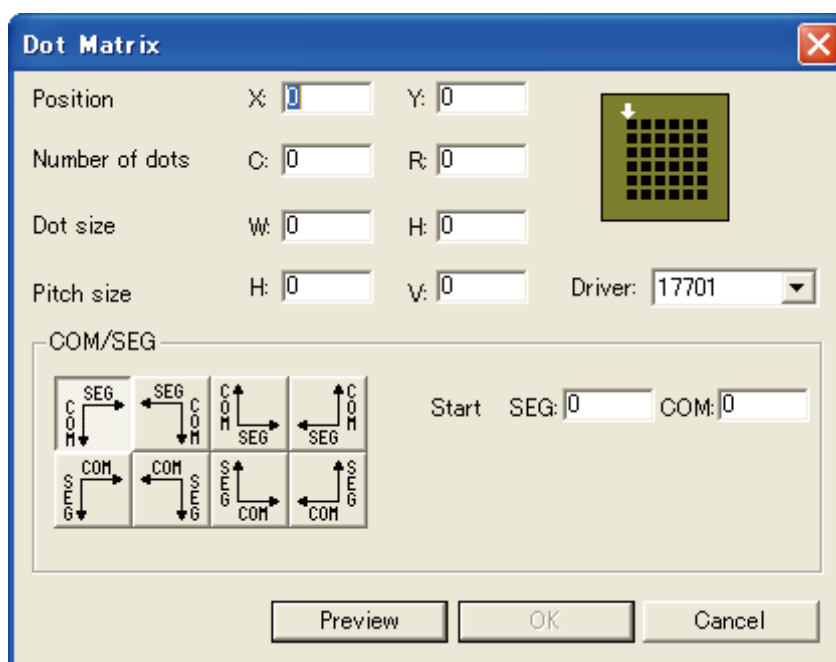
- 3) Select [Insert dot matrix] from the [Edit] menu.
 If no LCD driver is set, the [LCD Driver] dialog box appears.
 From the list, select the model for development, then click the [OK] button.



- * The [LCD Driver] dialog box appears only when no LCD driver has been set.
 When an already produced LCD file is read or when the [Start LcdUtility] button of the IDE is clicked while a project is selected by the IDE, an LCD driver is already set and this dialog box does not appear.

Note: Once set, an LCD driver cannot be changed. Be extremely careful when setting the LCD driver.

- 4) The [Dot matrix] dialog box appears.



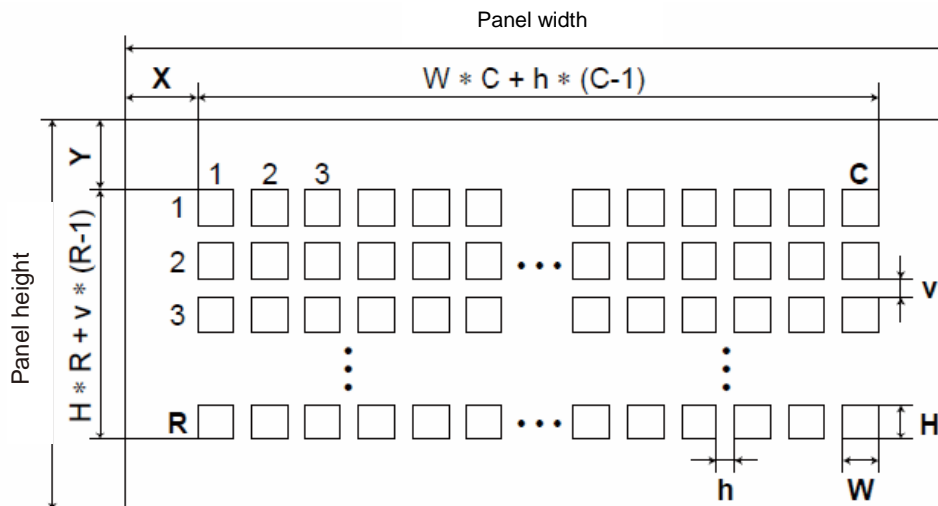


Figure 10.8.6.1.1 Dot matrix setting

Make the following settings in the [Dot matrix] dialog box.

Position

Specify the coordinates of the upper left corner of the dot matrix. (X and Y in the diagram)

Number of dots

Specify the number of dots along the horizontal and vertical axes of the dot matrix. (C and R in the diagram)

Dot size

Specify the size of each dot in the dot matrix by entering the number of dots. (W and H in the diagram)

Pitch size

Specify the interval between dots in the dot matrix by entering the number of dots. (h and v in the diagram)

Driver

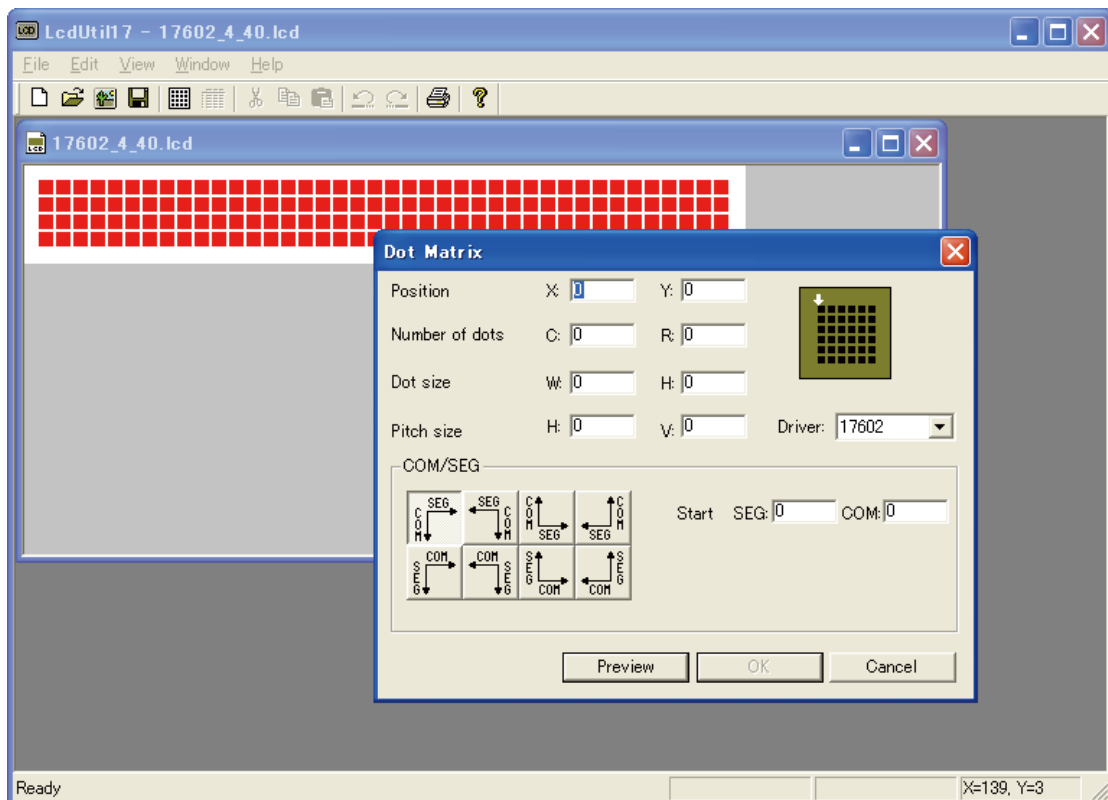
Indicates the name of the currently set LCD driver. This cannot be changed.

The maximum COM/SEG value is set by the LCD driver.

COM/SEG

Select the COM/SEG port allocating direction. Port numbers will be allocated in sequence based on the COM/SEG values specified in the [Start] text box.

Click the [Preview] button after making the settings above. A dot matrix will be displayed in the panel editing window for confirmation. Click the [OK] button to produce a dot matrix based on the settings entered.



Dot matrices allocated with a COM/SEG port are indicated in red.

Double-click a dot matrix to display the [Dot matrix] dialog to change settings.

Note: A dot matrix that is copied and pasted retains position and size information but discards port allocation information. Dot matrices without allocation information are indicated in black.

10.8.6.2 Producing a Segment LCD Panel

- 1) Prepare a bitmap file of an icon for segment LCD.

Although bitmap files can be created with an ordinary paint application, note the following when creating files:

Number of colors and file format

Set the background to white and create an icon in black. Save data in monochrome bitmap format (.bmp).

* You can read a color bitmap file, but binarization may not complete successfully, in certain cases.

Size

Keep the size of the bitmap file less than 1280 x 1024.

Number of files

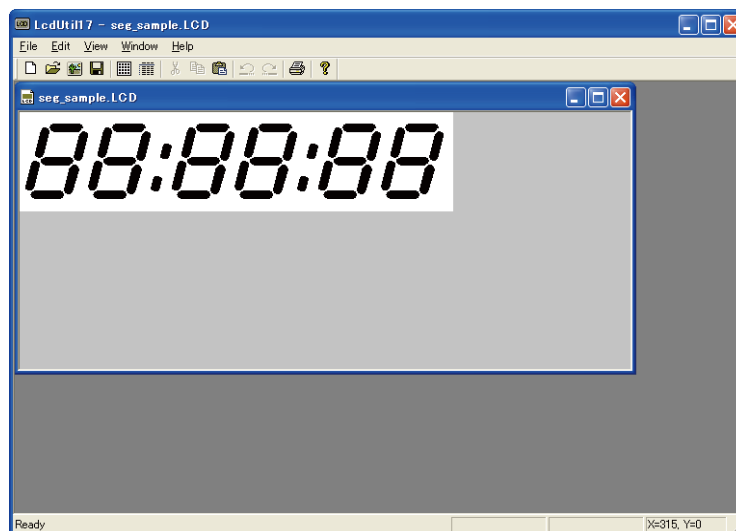
You can use several icons produced in several separate bitmap files to create a segment LCD panel in the LcdUtil17 window. However, since LcdUtil17 offers only simplified editing functions, we recommend producing one bitmap file.

- 2) Read a bitmap file.

Start LcdUtil17, and select [Open Bitmap File] from the [File] menu.

Select the bitmap file you created.

A panel editing window will open and display the read bitmap.



- 3) Edit the icon layout.

Icon condition

Icons can be in one of the following three states.

- Black: COM/SEG information not set
- Red: COM/SEG information set
- Blue: In selected state

Editing icons

You can perform the following operations on a selected icon.

Change of position

Drag with the mouse to move an icon. You can also move an icon from one panel editing window to another by dragging and dropping.

Cut, Copy

Select the command from the [Edit] menu or click the toolbar button.

Paste

Select the command from the [Edit] menu or click the toolbar button. The icon will be pasted at the upper left corner of the LCD panel.

Delete

Press the [Delete] key to delete the icon.

Group

[You can select multiple icons by holding down the [Ctrl] key while clicking. Set the multiple selected icons as a group by selecting [Group Icon] from the [Edit] menu. Once grouped, the icons can be treated as a single icon.

To cancel the grouping, select [Release Group] from the [Edit] menu.

Note: Grouped icons reflect the COM/SEG information corresponding to the icon selected last.

If the icons are then ungrouped, the individual icons will have the COM/SEG information corresponding to the grouped icons.

Note: LcdUtil17 offers only simplified functions. We recommend completing layout during bitmap production.

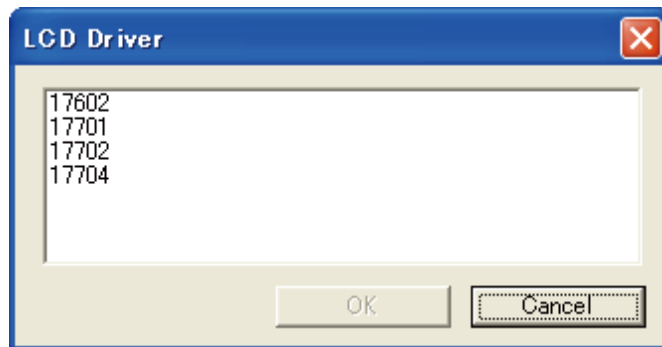
- 4) Allocate a port to the icon.

Double-click the icon.

The [LCD Driver] dialog box opens.

- 4-1) If no LCD driver has been set, the [LCD Driver] dialog box will appear.

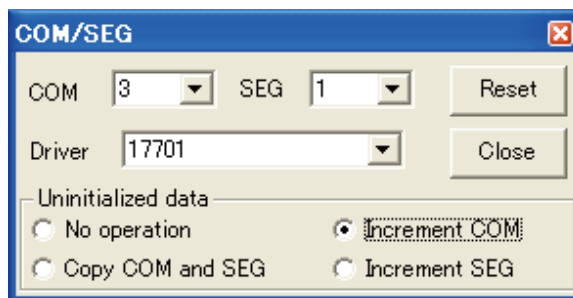
From the list, select the model for development, then click the [OK] button.



* The [LCD Driver] dialog box appears only when no LCD driver has been set. When an already produced LCD file is read or when the [Start LcdUtility] button of the IDE is clicked while a project is selected by the IDE, an LCD driver is already set and this dialog box does not appear.

Note: Once set, an LCD driver cannot be changed. Be extremely careful when setting the LCD driver.

4-2) The [COM/SEG] dialog box appears.



COM, SEG

Select from the pull-down list. COM/SEG allocation is applied immediately after the change is made.

[Reset] button

Click this button to clear COM/SEG settings and set COM/SEG in the "not set" state.

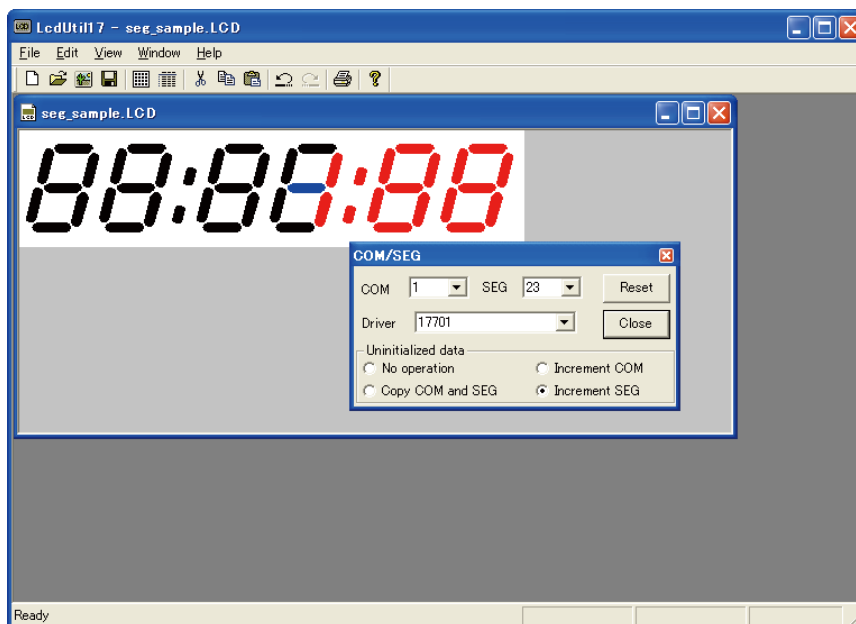
Driver

Indicates the name of the currently set LCD driver. This cannot be changed. The maximum COM/SEG value is set by the LCD driver.

Uninitialized data

You can initiate settings if you click an icon for which no COM/SEG has been set.

No operation	The COM/SEG information will not be changed.
Copy COM and SEG	The displayed COM/SEG information will be set to the icon.
Increment COM	The COM will be incremented (+1) and set to the displayed COM/SEG information.
Increment SEG	The SEG will be incremented (+1) and set to the displayed COM/SEG information.



Note: The port allocation information is discarded if you copy and paste an icon or move an icon from another panel editing window. Dot matrices without allocation information appear in black.

5) Display the icon list.

Select [Icon List] from the [Edit] menu.

A list appears and shows the icons in the currently active panel editing window.

Click an icon in the list to display the corresponding icon in the panel editing window in blue.

In this window, you can also change COM/SEG information for icons.

10.8.7 Shortcut Key list

The following shows a list of shortcut keys available with LcdUtil17.

Table 10.8.7.1 List of shortcut keys

Function	Shortcut key
Copy	Ctrl + C Ctrl + Insert
Cut	Ctrl + X Delete
Paste	Ctrl + V Shift + Insert
Icon List	Ctrl + I
Dot Matrix	Ctrl + M
New	Ctrl + N
Open	Ctrl + O
Save	Ctrl + S
Print	Ctrl + P
Undo	Ctrl + Z Alt + BS
Redo	Ctrl + Y

10.8.8 Warning Messages and Error Messages

The following is a list of warning messages displayed by LcdUtil17.

Table 10.8.8.1 List of warning messages

No.	When creating or opening a file
1	<ul style="list-style-type: none"> •A blank document could not be created. •Unable to create new document
	Cause: A new window could not be opened because more than 100 windows are currently open. A new window could not be opened due to insufficient memory.
	Corrective action: Close windows to reduce the number of open windows to 99 or less. Close other applications to increase available memory.
2	Unexpected file format
	Cause: The file was not in the correct lcd or bmp format. The file was an lcd file with an unsupported driver.
	Corrective action: Select a file supported by LcdUtil17.
No.	When entering an input
3	<ul style="list-style-type: none"> •Enter an integer value between ??? and ???. •Please enter a valid number between ??? and ???.
	Cause: A value outside the allowed range was entered.
	Corrective action: Enter a correct value.
4	<ul style="list-style-type: none"> •Enter an integer. •Please enter a valid number. Invalid numbers include: space, decimals, 0, +, -.
	Cause: A nonnumeric character was entered in the edit box for integer input.
	Corrective action: Enter a correct value.
No.	Dot matrix setting window (when [Preview] button is clicked)
5	Invalid ??? (??? = position, number of dots, dot size, pitch size, start COM/SEG)
	Cause: An invalid value is entered in ???.
	Corrective action: Enter a correct value.
6	Invalid ??? start number or Invalid number of dots (??? = COM, SEG)
	Cause: The number of COM/SEG allowed by the driver has been exceeded, based on settings for Start ??, dot count, and COM/SEG allocation direction.
	Corrective action: Set the Start ??, dot count, and COM/SEG allocation direction in accordance with the number of COM/SEG allowed by the driver.
No.	When using [Save as]
7	Some icons/matrixes are out of LCD panel. Remove them? [OK][Cancel]
	Cause: An attempt was made to save data while an icon or dot matrix extended from the LCD panel.
	Corrective action: Move the icon or dot matrix so that it does not extend from the LCD panel or click the [OK] button to remove the icon or dot matrix extending from the LCD panel before saving.
No.	When creating an icon or reading a file
8	The number of the icon is exceeding 4096 pieces.
	Cause: The total number of icons exceeded the maximum value of 4096.
	Corrective action: Reduce the total number of icons to less than 4096.
9	The number of the icon is exceeding 4096 pieces. Loaded the icons to 4096 pieces.
	Cause: The total number of icons exceeded the maximum value of 4096.
	Corrective action: Reduce the total number of icons to less than 4096.
No.	When creating a matrix or reading a file
10	The number of the matrix is exceeding 10 pieces.
	Cause: The total number of dot matrices exceeded the maximum value of 10.
	Corrective action: Reduce the total number of dot matrices to less than 10.
No.	When saving data or closing a window without saving changes
11	This file was not saved.
	Cause: The icon or dot matrix extending from the LCD panel was not removed.
	Corrective action: Data cannot be saved if an icon or dot matrix extends from the LCD panel.

The following shows a list of error messages displayed by LcdUtil17.

Table 10.8.8.2 List of error messages

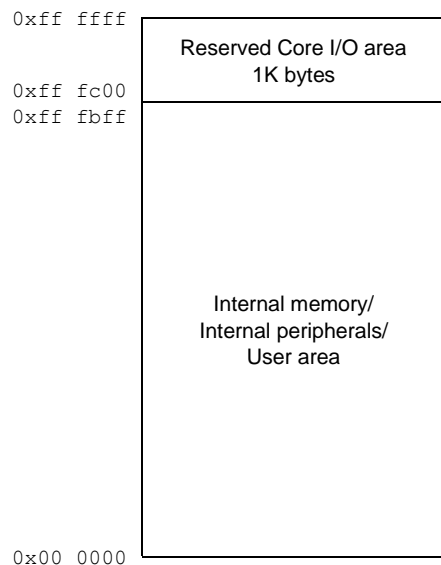
No.	During startup
1	Cannot open bitmap file. The size of the panel is too large.
	Cause: The vertical or horizontal size of the bitmap file is too large.
	Corrective action: Use a bitmap file with a horizontal size of no more than 1280 and a vertical size of no more than 1024.

11 Quick Reference

Memory Map and Trap Table (S1C17 Core)

S1C17 Core

Memory Map



Trap Table

No.		Vector address
0 (0x00)	Reset	TTBR + 0x00
1 (0x01)	Address misaligned interrupt	TTBR + 0x04
2 (0x02)	NMI	TTBR + 0x08
3 (0x03)	Maskable external interrupt 3	TTBR + 0x0c
:	:	:
31 (0x1f)	Maskable external interrupt 31	TTBR + 0x7c

TTBR: Trap table start address
(Can be read from address 0xffff80.)

Registers (S1C17 Core)

S1C17 Core

General-purpose Registers (8)

23	0
R7	
R6	
R5	
R4	
R3	
R2	
R1	
R0	

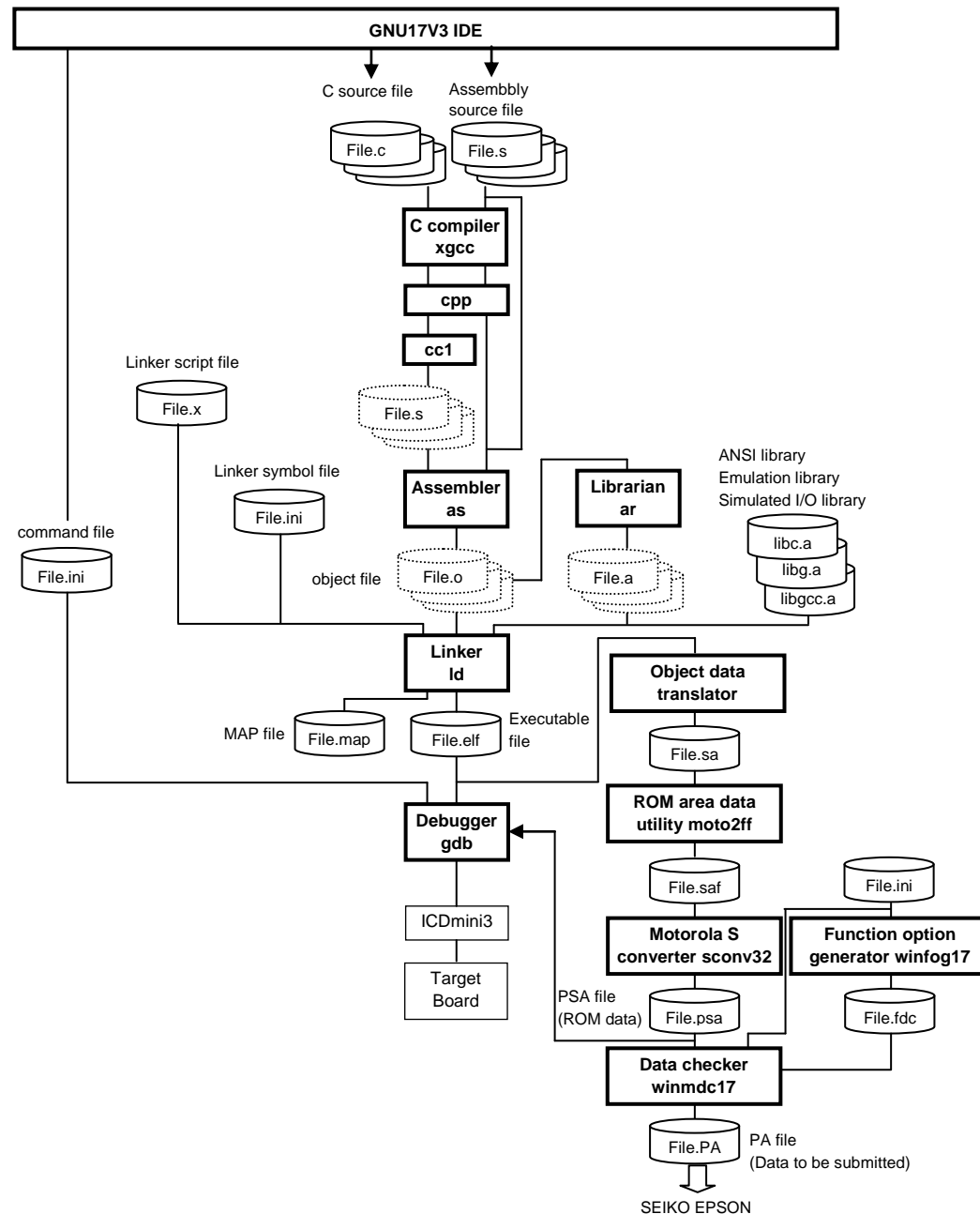
Special Registers (3)

23	0
PC	Program counter
23	0
SP	Stack pointer
7	0
PSR	Processor status register

PSR

7	6	5	4	3	2	1	0
IL[2:0]	IE	C	V	Z	N		
Initial value	0	0	0	0	0	0	0

IL[2:0]: Interrupt level (0–7: Enabled interrupt level)
 IE: Interrupt enable (1: Enabled, 0: Disabled)
 Z: Zero flag (1: Zero, 0: Non zero)
 N: Negative flag (1: Negative, 0: Positive)
 C: Carry flag (1: Carry/borrow, 0: No carry)
 V: Overflow flag (1: Overflow, 0: Not overflown)

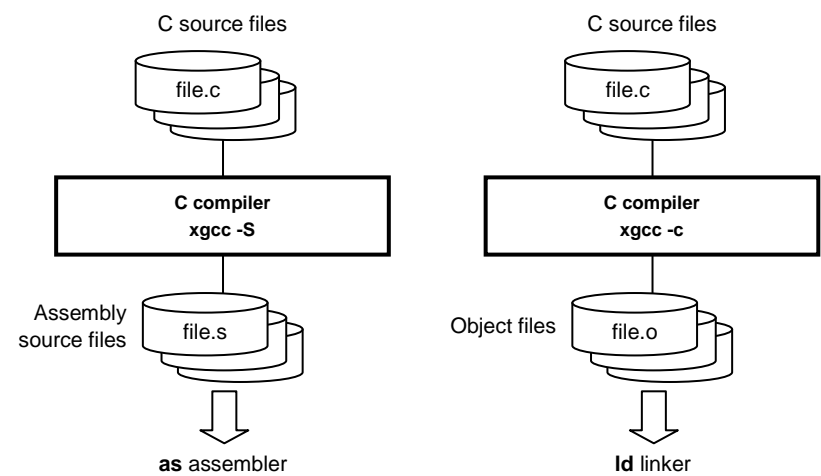


1. Creating a project
Use the **IDE** to create a new project or importing an existing project.
2. Editing source files
Edit resources such as the sources using the **IDE** editor or a general-purpose editor.
3. Building (Compile, Assemble, Link)
3-1) Edit the build options and linker script in the **IDE** project.
3-2) Execute the build using the **IDE**. The C Compiler (xgcc), assembler (as), and linker ld are executed in sequence by the builder to generate an executable object file (.elf).
4. Debugging
4-1) Edit the command file to be executed at gdb startup in the **IDE**.
4-2) Start up the debugger gdb from the **IDE**.
4-3) Debugging the program using the **IDE**.
5. PA file (Data to be submitted) creation
When program development is completed, create the data to be submitted.
5-1) Create the PSA file (ROM data) using **objcopy**, **moto2ff**, and **sconv32**.
5-2) Create the FDC file (Function option document) using **winfog17**
5-3) Pack the PSA and FDC files into a single PA file using **winmdc17**.
5-4) Submit the created PA file to Seiko Epson.

Outline

This tool is made based on GNU C Compiler and is compatible with ANSI C. This tool invokes **cpp.exe** and **cc1.exe** sequentially to compile C source files to the assembly source files for the S1C17 Family. It has a powerful optimizing ability that can generate minimized assembly codes. The **xgcc.exe** can also invoke the **as.exe** assembler to generate object files.

Flowchart



Start-up Command

```
xgcc <options> <filename>
```

<filename> C source file name

Example: xgcc -c -g test.c

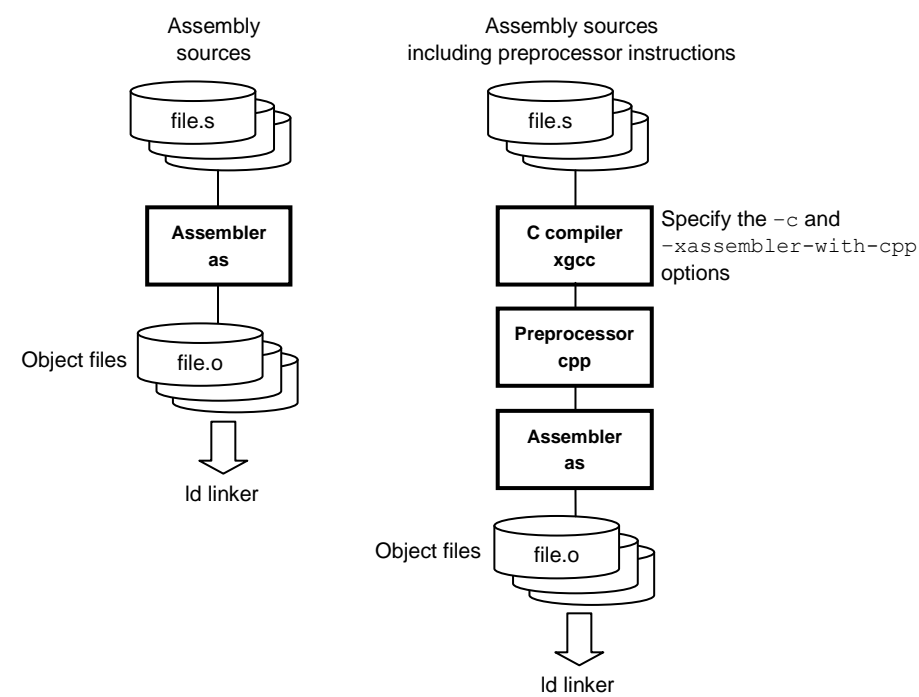
Major Command-line Options

-S	Output assembly code (.s)
-c	Output relocatable object file (.o)
-E	Execute C preprocessor only
-B<path>	Specify compiler search path
-I<path>	Specify include file directory
-fno-builtin	Disable built-in functions
-D<macro> [=<string>]	Define macro name
-O0, -O, -O1, -O2, -O3, -Os	Optimization
-g	Add debugging information with relative path to source files (for gcc6)
-gstabs	Add debugging information with relative path to source files (for gcc4)
-mpointer16	Generate code for 16-bit (64KB) data space
-mrelax	Output code size optimization
-Wall	Enables warning options
-Werror-implicit-function-declaration	Undeclared function error output
-xassembler-with-cpp	Invoke C preprocessor
-Wa, <option>	Specify assembler option

Outline

This tool assembles assembly source files output by the C compiler and converts the mnemonics of the source files into object codes (machine language) of the S1C17. The **as.exe** allows the user to invoke the assembler through **xgcc.exe**, this makes it possible to include preprocessor directives into assembly source files. The results are output in an object file that can be linked or added to a library.

Flowchart



Start-up Command

as <options> <filename>

<filename> Assembly source file name

Example: `as -o test.o -adh1 test.s`Assemblerasfile.

Major Command-line Options

-o<filename>	Specify output file name
-a[<suboption>]	Output assembly list file Example:-adh1 (high-level assembly listing without debugging directives)
--gstabs	Add debugging information with relative path to source files
-mpointer16	Specify 16-bit pointer mode

Major Preprocessor Pseudo-instructions

#include	Insertion of file
#define	Definition/macro definition of character string and numeric value
#if - #else - #endif	Conditional assembly (Can be used when the -c -xassembler-with-cpp option of xgcc is specified.)

Major Assembler Pseudo-instructions

.text	Declare .text section
.section .data	Declare .data section
.section .rodata	Declare .rodata section
.section .bss	Declare .bss section
.long <data>	Define 4-byte data
.short <data>	Define 2-byte data
.byte <data>	Define 1-byte data
.ascii <string>	Define ASCII character strings
.space <length>	Define blank area (0x0)
.zero <length>	Define blank area (0x0)
.align <value>	Alignment to specify boundary address
.global <symbol>	Global declaration of symbol
.set <symbol>, <address>	Define symbol with absolute address

Error/Warning messages

Error messages

Error: Unrecognized opcode: 'XXXXX'	The operation code XXXXX is undefined.
Error: junk at end of line: 'XXXXX'	A format error of the operand.
Error: XXXXXX: invalid register name	The specified register cannot be used.

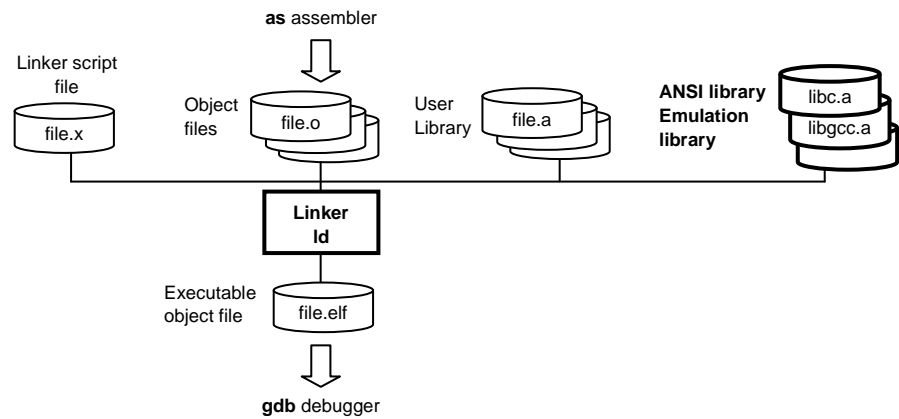
Warning messages

Warning: Unrecognized .section attribute: want a, w, x	The section attribute is not a, w or x.
Warning: Bignum truncated to AAA bytes	The constant declared (e.g. .long, .int) exceeds the maximum size. It has been corrected to AAA-byte size. (e.g. 0x100000012 → 0x12)
Warning: Value XXXX truncated to AAA	The constant declared exceeds the maximum value AAA. It has been corrected to AAA. (e.g. .byte 0x100000012 → .byte 0xff)
Warning: operand out of range (XXXXXX: XXX not between AAA and BBB)	The value specified in the operand is out of the effective range.

Outline

Defines the memory locations of object codes created by the C compiler and assembler, and creates executable object codes. This tool puts together multiple objects and library files into one file.

Flowchart



Start-up Command

```
ld <options> <filename>
```

<filename> Object and library files to be linked

```
Example: ld -o sample.elf boot.o sample.o ..\lib\24bit\libc.a
        ..\lib\24bit\libgcc.a ..\lib\24bit\libc.a
```

Major Command-line Options

-o <filename>	Specify output file name
-T <filename>	Read linker script file
-M	Link map stdout output
-Map <filename>	Link map file output
-N	Disable data segment alignment check
-R	Read linker symbol file
--relax	Optimize output code size

Error Messages

warning: out of range error.	The address of the symbol exceeds the 16-bit address (when -mpointer16 is specified) or 24-bit address space.
Error: The offset value of a symbol is over 24bit.	The address of the symbol exceeds the 24-bit address space.
Error: section XXX is not within 16bit address.	The address of the XXX section exceeds the 16-bit address space.
Error: section XXX is not within 24bit address.	The address of the XXX section exceeds the 24-bit address space.
Error: Input object file <objectfile> [included from <archivefile>] is not for C17.	The object file is not compatible with the C17.
Error: Input object file <objectfile> is not 16bit nor 24bit address mode.	The object file is neither in 16-bit or 24-bit mode.
Error: Cannot link 16bit object <objectfile16> [included from <archivefile16>] with 24bit object <objectfile24> [included from <archivefile24>]	Object files created in 16-bit pointer mode and object files created in 24-bit pointer mode cannot be linked.

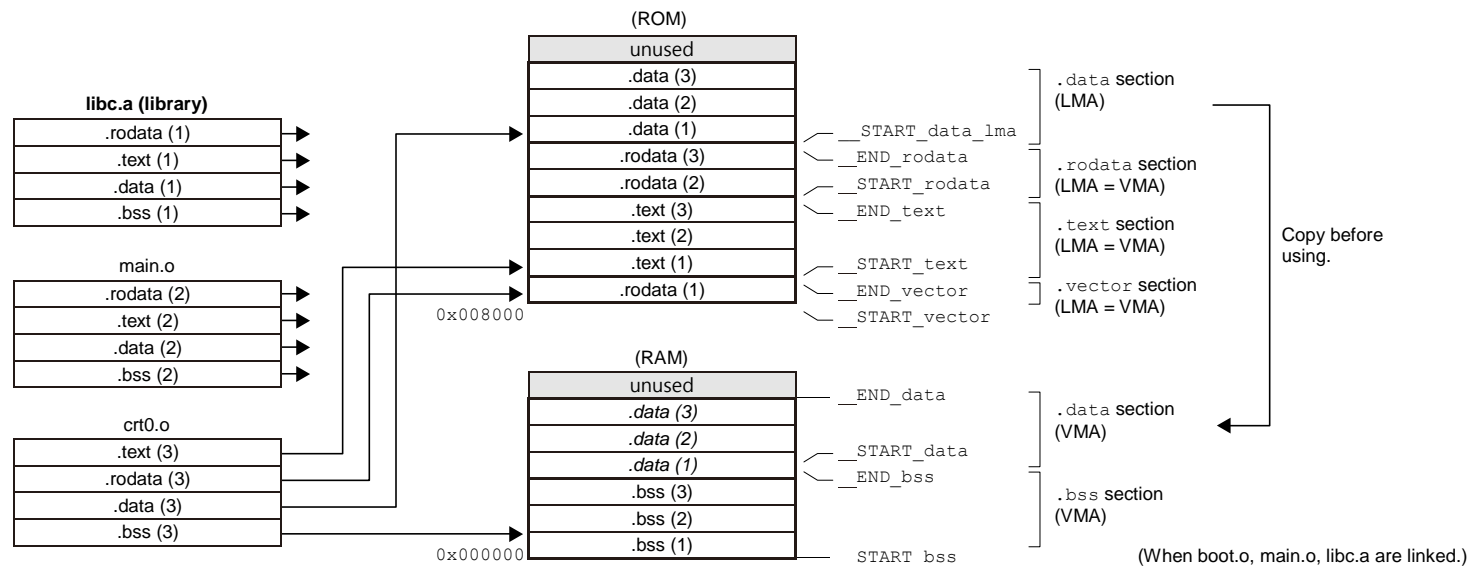
Default linker script file generated by the IDE

```

OUTPUT_FORMAT("elf32-c17")
OUTPUT_ARCH(c17)
ENTRY(_start)
SEARCH_DIR(.);
MEMORY
{
    iram          : ORIGIN = 0,          LENGTH = 32K
    irom          : ORIGIN = 0x8000,     LENGTH = 4064K
}
SECTIONS
{
    .bss (NOLOAD) :
    {
        PROVIDE (__START_bss = .) ;
        *(.bss)
        *(.bss.*)
        *(COMMON)
        PROVIDE (__END_bss = .) ;
    } > iram
    .vector :
    {
        PROVIDE (__START_vector = .) ;
        KEEP (*crt0.o(.rodata))
        PROVIDE (__END_vector = .) ;
    } > irom

    .text :
    {
        PROVIDE (__START_text = .) ;
        *(.text.*)
        *(.text)
        PROVIDE (__END_text = .) ;
    } > irom
    .data :
    {
        PROVIDE (__START_data = .) ;
        *(.data)
        *(.data.*)
        PROVIDE (__END_data = .) ;
    } > iram AT > irom
    .rodata :
    {
        PROVIDE (__START_rodata = .) ;
        *(EXCLUDE_FILE (*crt0.o) .rodata)
        *(.rodata.*)
        PROVIDE (__END_rodata = .) ;
    } > irom
    PROVIDE (__START_data_lma = LOADADDR(.data));
    PROVIDE (__END_data_lma = LOADADDR(.data) + SIZEOF (.data));
    PROVIDE (__START_stack = 0x0007C0);
}

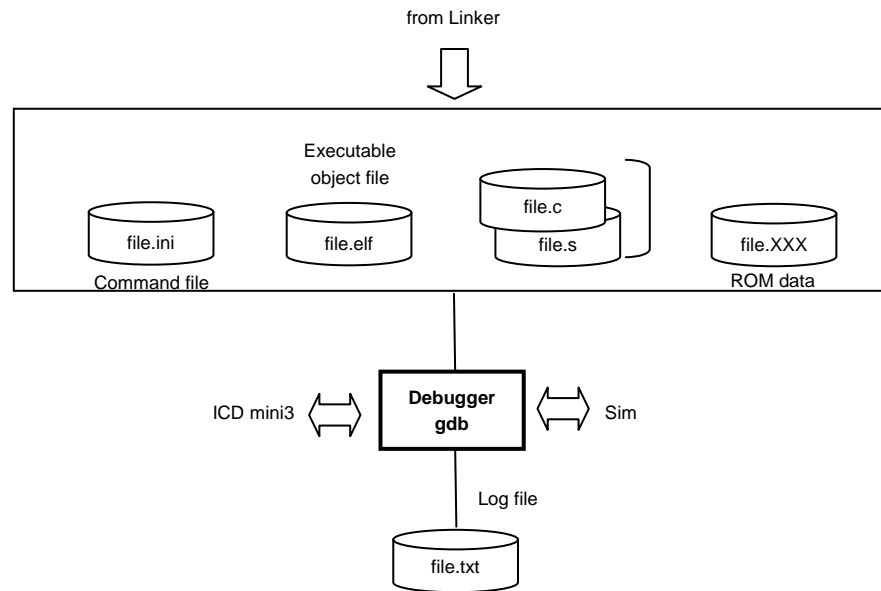
```



Outline

The gdb serves to perform source-level debugging by controlling an ICD. It also comes with a simulating function that allows you to perform debugging on a personal computer.

Flowchart



Start-up Options

`gdb[<Start-up options>]`

Specify the command file using start-up options when executing from within the IDE.

Example: `gdb -x gdbmini3.ini`

Command file

Example:

```
c17 model_path c:/EPSON/GNU17V3/mcu_model
```

Specifies the model information file directory.

```
c17 model 17W23@NOVCCIN
```

Specifies the model name.

(Voltage level 3.3 V)

```
target icd icdmini3
```

Connects the target.

```
load
```

Loads a program.

```
thbreak main
```

Sets the hardware PC breakpoint.

Debug Commands

Memory operation

c17 fb <i>addr1 addr2 data</i>	Fill memory area (8 bits)	ICD Mini/SIM
c17 fh <i>addr1 addr2 data</i>	Fill memory area (16 bits)	ICD Mini/SIM
c17 fw <i>addr1 addr2 data</i>	Fill memory area (32 bits)	ICD Mini/SIM
x <i>/[length]b [addr]</i>	Dump memory data (8 bits)	ICD Mini/SIM
x <i>/[length]h [addr]</i>	Dump memory data (16 bits)	ICD Mini/SIM
x <i>/[length]w [addr]</i>	Dump memory data (32 bits)	ICD Mini/SIM
set {char} <i>addr=data</i>	Set memory data (8 bits)	ICD Mini/SIM
set {short} <i>addr=data</i>	Set memory data (16 bits)	ICD Mini/SIM
set {long} <i>addr=data</i>	Set memory data (32 bits)	ICD Mini/SIM
c17 mvb <i>addr1 addr2 addr3</i>	Copy memory area (8 bits)	ICD Mini/SIM
c17 mvh <i>addr1 addr2 addr3</i>	Copy memory area (16 bits)	ICD Mini/SIM
c17 mvw <i>addr1 addr2 addr3</i>	Copy memory area (32 bits)	ICD Mini/SIM
c17 df <i>addr1 addr2 type file [append]</i>	Save memory data to file	ICD Mini/SIM

Register operation

info reg <i>[register]</i>	Display register data	ICD Mini/SIM
set \$ <i>register=data</i>	Set register data	ICD Mini/SIM

Program execution

continue <i>[Ignore]</i>	Execute program successively	ICD Mini/SIM
until <i>addr</i>	Execute program successively with temporary break	ICD Mini/SIM
step <i>[count]</i>	Execute source lines	ICD Mini/SIM
stepi <i>[count]</i>	Execute instruction steps	ICD Mini/SIM
next <i>[count]</i>	Execute source lines with function skip	ICD Mini/SIM
nexti <i>[count]</i>	Execute instruction steps with subroutine skip	ICD Mini/SIM
finish	Exit from function/subroutine	ICD Mini/SIM

CPU reset

c17 rst	Reset CPU (execute reset.gdb)	ICD Mini/SIM
c17 rstt	Reset target	ICD Mini

Interrupt

c17 int <i>[intNo. level]</i>	Generate interrupt	SIM
c17 intclear <i>[intNo.]</i>	Clear interrupt	SIM

Break

break <i>[addr]</i>	Set software PC breakpoint	ICD Mini/SIM
tbreak <i>[addr]</i>	Set temporary software PC breakpoint	ICD Mini/SIM
hbreak <i>[addr]</i>	Set hardware PC breakpoint	ICD Mini/SIM
thbreak <i>[addr]</i>	Set temporary hardware PC breakpoint	ICD Mini/SIM
delete <i>[breakNo.]</i>	Clear breakpoint by break number	ICD Mini/SIM
clear <i>addr</i>	Clear breakpoint by location	ICD Mini/SIM
enable <i>[breakNo.]</i>	Enable breakpoint	ICD Mini/SIM
disable <i>[breakNo.]</i>	Disable breakpoint	ICD Mini/SIM
ignore <i>breakNo. count</i>	Disable breakpoint with ignore count	ICD Mini/SIM
info breakpoints	Display breakpoint list	ICD Mini/SIM
commands	Set a command to execute after break	ICD Mini/Sim

S

info locals	Display local symbol information	ICD Mini/SIM
info var	Display global symbol information	ICD Mini/SIM
print <i>symbol[=value]</i>	Change symbol values	ICD Mini/SIM

File

file <i>file</i>	Load debug information	ICD Mini/SIM
load <i>[file]</i>	Load program	ICD Mini/SIM

Trace

c17 tm on/off mode <i>[file]</i>	Set trace mode	SIM
---	----------------	-----

Debug Commands

Others

set output-rad x	Change variable display format	ICD Mini/SIM
set logging on/off	Log output setting	ICD Mini/SIM
source file	Execute command file	ICD Mini/SIM
target type	Connect target	ICD Mini/SIM
detach	Disconnect target	ICD Mini/SIM
pwd	Display current directory	ICD Mini/SIM
cd directory	Change current directory	ICD Mini/SIM
c17 ttbr addr	Set TTBR	SIM
c17 cpu type	Set CPU type	SIM
c17 chgclkmd 0/1	DCLK change mode	ICD Mini
c17 pwul	Unlock flash security password	ICD Mini
c17 help [command/groupNo.]	Help	ICD Mini/SIM
c17 model_path addr	Model-specific information file directory setting	ICD Mini/SIM
c17 model name	MCU model name setting	ICD Mini/SIM
quit	Terminate debugger	ICD Mini/SIM

Status and Error Messages

Status messages

Breakpoint #, function at file:line	Made to break at a set breakpoint
Illegal instruction.	Made to break by executing invalid instruction in simulator mode
Illegal delayed instruction.	Made to break by executing invalid delayed instruction in simulator mode
Break by key break.	Forcibly made to break by [Suspend] button (in simulator mode)
Break by key break. Program received signal SIGINT, Interrupt.	Forcibly made to break by [Suspend] button (in ICD Mini mode)

Error Messages

Address is 24bit over.	The specified address is out of the 24-bit range. The maximum S1C17 address size is 24 bits (0xFFFFF).
Address(0x%x) is ext or delayed instruction	The specified address cannot be set due to an ext or delayed instruction.
C17 command error, command is not supported in present mode.	The input command cannot be executed in the current mode (ICD or SIM mode, or neither).
C17 command error, invalid command.	The command is erroneous.
C17 command error, invalid parameter.	The command is specified with an invalid argument.
C17 command error, number of parameter.	The command is specified with an invalid number of arguments.
C17 command error, start address > end address.	The specified start address is greater than the end address.
Cannot set hard pc break.	Cannot set a hard break at the address specified.
Cannot set hard pc break any more.	The number of hardware PC breakpoints set exceeds the limit (one location only).
Cannot set soft pc break.	Cannot set a soft break at the address specified.
Cannot set soft pc break any more.	The number of software PC breakpoints set exceeds the limit.
Cannot write file	Cannot write to the file.
command result error!	An error occurred on executing an undefined command.
icdmini3 dll open failure.	Failed to connect to ICD mini Ver3.

Floating-point Calculation Functions

Double-type operation

__adddf3	Addition	$x \leftarrow a + b$
__subdf3	Subtraction	$x \leftarrow a - b$
__muldf3	Multiplication	$x \leftarrow a * b$
__divdf3	Division	$x \leftarrow a / b$
__negdf2	Sign change	$x \leftarrow -a$

Float-type operation

__addsf3	Addition	$x \leftarrow a + b$
__subsf3	Subtraction	$x \leftarrow a - b$
__mulsf3	Multiplication	$x \leftarrow a * b$
__divsf3	Division	$x \leftarrow a / b$
__negsf2	Sign change	$x \leftarrow -a$

Type conversion

__fixunsdfsi	double \rightarrow unsigned long	$x \leftarrow a$
__fixdfsi	double \rightarrow long	$x \leftarrow a$
__floatsidf	long \rightarrow double	$x \leftarrow a$
__fixunssfsi	float \rightarrow unsigned long	$x \leftarrow a$
__fixsfsi	float \rightarrow long	$x \leftarrow a$
__floatsisf	long \rightarrow float	$x \leftarrow a$
__truncdfsf2	double \rightarrow float	$x \leftarrow a$
__extendsfdf2	float \rightarrow double	$x \leftarrow a$

Comparison

__**df2	double type	Changes %psr and x by a - b **=eq, ne, gt, ge, lt, le
__**sf2	float type	Changes %psr and x by a - %13 **=eq, ne, gt, ge, lt, le

Floating-point Data Format

Double-type data format

63	62	52	51	0
S	Exponent part			Fixed-point part

Double-type effective range

+0:	0.0e+0 0x00000000 00000000
-0:	-0.0e+0 0x80000000 00000000
Maximum normalized number:	1.79769e+308 0x7fefffff fffffff
Minimum normalized number:	2.22507e-308 0x00100000 00000000
Maximum unnormalized number:	2.22507e-308 0x000fffff fffffff
Minimum unnormalized number:	4.94065e-324 0x00000000 00000001
Infinity:	0x7ff00000 00000000
-Infinity:	0xfff00000 00000000

Float-type data format

31	30	23	22	0
S	Exponent part			Fixed-point part

Float-type effective range

+0:	0.0e+0f 0x00000000
-0:	-0.0e+0f 0x80000000
Maximum normalized number:	3.40282e+38f 0x7f7ffff
Minimum normalized number:	1.17549e-38f 0x00800000
Maximum unnormalized number:	1.17549e-38f 0x007ffff
Minimum unnormalized number:	1.40129e-45f 0x00000001
Infinity:	0x7f800000
-Infinity:	0xff800000

Integral Calculation Functions

Integral calculation

<code>__divsi3</code>	Signed 32-bit integral division	$x \leftarrow a / b$
<code>__modsi3</code>	Signed 32-bit remainder calculation	$x \leftarrow a \% b$
<code>__udivsi3</code>	Unsigned 32-bit integral division	$x \leftarrow a / b$
<code>__umodsi3</code>	Unsigned 32-bit remainder calculation	$x \leftarrow a \% b$
<code>__mulsi3</code>	32-bit multiplication	$x \leftarrow a * b$
<code>__divhi3</code>	Signed 16-bit integral division	$x \leftarrow b$
<code>__modhi3</code>	Signed 16-bit remainder calculation	$x \leftarrow a \% b$
<code>__udivhi3</code>	Unsigned 16-bit integral division	$x \leftarrow a / b$
<code>__umodhi3</code>	Unsigned 16-bit remainder calculation	$x \leftarrow a \% b$
<code>__mulhi3</code>	16-bit multiplication	$x \leftarrow a * b$

Integral shift

<code>__ashlsi3</code>	32-bit arithmetical shift to left	$x \leftarrow a \ll b \text{ bits}$
<code>__ashrsi3</code>	32-bit arithmetical shift to right	$x \leftarrow a \gg b \text{ bits}$
<code>__lshrsi3</code>	32-bit logical shift to right	$x \leftarrow a \gg b \text{ bits}$
<code>__ashlhi3</code>	16-bit arithmetical shift to left	$x \leftarrow a \ll b \text{ bits}$
<code>__ashrhi3</code>	16-bit arithmetical shift to right	$x \leftarrow a \gg b \text{ bits}$
<code>__lshrhi3</code>	16-bit logical shift to right	$x \leftarrow a \gg b \text{ bits}$

Integral comparison

<code>__cmpsi2</code>	Comparison (long)	$x \leftarrow 2 1 0$
<code>__ucmpsi2</code>	Comparison (unsigned long)	$x \leftarrow 2 1 0$

long long Type Calculation Functions

long long type calculation

<code>__mulldi3</code>	Signed 64-bit multiplication	$x \leftarrow a * b$
<code>__divldi3</code>	Signed 64-bit division	$x \leftarrow a / b$
<code>__udivldi3</code>	Unsigned 64-bit division	$x \leftarrow a / b$
<code>__modldi3</code>	Signed 64-bit remainder calculation	$x \leftarrow a \% b$
<code>__umodldi3</code>	Unsigned 64-bit remainder calculation	$x \leftarrow a \% b$
<code>__negdi2</code>	Sign inversion	$x \leftarrow -a$

long long type shift

<code>__lshrdi3</code>	64-bit logical shift to right	$x \leftarrow a \gg b \text{ bits}$
<code>__ashldi3</code>	64-bit arithmetical shift to left	$x \leftarrow a \ll b \text{ bits}$
<code>__ashrdi3</code>	64-bit arithmetical shift to right	$x \leftarrow a \gg b \text{ bits}$

Type conversion

<code>__fixunsdfdi</code>	double \rightarrow unsigned long long	$x \leftarrow a$
<code>__fixdfdi</code>	double \rightarrow long long	$x \leftarrow a$
<code>__floatdidf</code>	long long \rightarrow double	$x \leftarrow a$
<code>__fixunssfdi</code>	float \rightarrow unsigned long long	$x \leftarrow a$
<code>__fixsfdi</code>	float \rightarrow long long	$x \leftarrow a$
<code>__floatdisf</code>	long long \rightarrow float	$x \leftarrow a$

long long type comparison

<code>__cmpdi2</code>	Comparison (long long)	$x \leftarrow 2 1 0$
<code>__ucmpdi2</code>	Comparison (unsigned long long)	$x \leftarrow 2 1 0$

Input/Output Functions (header file: stdio.h)

fread()	size_t fread(void *ptr, size_t size, size_t count, FILE *stream);	*1, *2
fwrite()	size_t fwrite(const void *ptr, size_t size, size_t count, FILE *stream);	*1, *2
fgetc()	int fgetc(FILE *stream);	*2
getc()	int getc(FILE *stream);	*1, *2
getchar()	int getchar(void);	*1, *2
ungetc()	int ungetc(int c, FILE *stream);	*1
fgets()	char *fgets(char *s, int n, FILE *stream);	*1, *2
gets()	char *gets(char *s);	*1, *2
fputc()	int fputc(int c, FILE *stream);	*2
putc()	int putc(int c, FILE *stream);	*1, *2
putchar()	int putchar(int c);	*1, *2
fputs()	int fputs(char *s, FILE *stream);	*1, *2
puts()	int puts(char *s);	*1, *2
perror()	void perror(const char *s);	*1, *2
fscanf()	int fscanf(FILE *stream, const char *format, ...);	*1, *2
scanf()	int scanf(const char *format, ...);	*1, *2
sscanf()	int sscanf(const char *s, const char *format, ...);	*1, *2
fprintf()	int fprintf(FILE *stream, const char *format, ...);	*1, *2
printf()	int printf(const char *format, ...);	*1, *2
sprintf()	int sprintf(char *s, const char *format, ...);	*1, *2
vfprintf()	int vfprintf(FILE *stream, const char *format, va_list arg);	*1, *2
vprintf()	int vprintf(const char *format, va_list arg);	*1, *2
vsprintf()	int vsprintf(char *s, const char *format, va_list arg);	

Utility Functions (header file: stdlib.h)

malloc()	void *malloc(size_t size);	*1
calloc()	void *calloc(size_t elt_count, size_t elt_size);	*1
free()	void free(void *ptr);	*1
realloc()	void *realloc(void *ptr, size_t size);	*1
exit()	void exit(int status);	
abort()	void abort(void);	
bsearch()	void *bsearch(const void *key, const void *base, size_t count, size_t size, int (*compare)(const void *, const void *));	
qsort()	void qsort(void *base, size_t count, size_t size, int (*compare)(const void *, const void *));	
abs()	int abs(int x);	
labs()	long labs(long x);	
div()	div_t div(int n, int d);	*1
ldiv()	ldiv_t ldiv(long n, long d);	*1
rand()	int rand(void);	
srand()	void srand(unsigned int seed);	
atol()	long atol(const char *str);	
atoi()	int atoi(const char *str);	*1
atof()	double atof(const char *str);	*1
strtod()	double strtod(const char *str, char **ptr);	*1
strtol()	long strtol(const char *str, char **ptr, int base);	*1
strtoul()	unsigned long strtoul(const char *str, char **ptr, int base);	*1

Date and Time Functions (header file: time.h)

gmtime()	struct tm *gmtime(const time_t *t);	
mktime()	time_t mktime(struct tm *tmptr);	
time()	time_t time(time_t *t_ptr);	*1

Non-local Branch Functions (header file: setjmp.h)

setjmp()	int setjmp(jmp_buf env);	
longjmp()	void longjmp(jmp_buf env, int status);	

*1 These functions need to declare and initialize the global variables.

*2 These functions need to define the low-level functions and I/O buffers.

Mathematical Functions (header file: math.h, errno.h, float.h, limits.h)

fabs()	double fabs(double x);	*1
ceil()	double ceil(double x);	*1
floor()	double floor(double x);	*1
fmod()	double fmod(double x, double y);	*1
exp()	double exp(double x);	*1
log()	double log(double x);	*1
log10()	double log10(double x);	*1
frexp()	double frexp(double x, int *nptr);	*1
ldexp()	double ldexp(double x, int n);	*1
modf()	double modf(double x, double *nptr);	*1
pow()	double pow(double x, double y);	*1
sqrt()	double sqrt(double x);	*1
sin()	double sin(double x);	*1
cos()	double cos(double x);	*1
tan()	double tan(double x);	*1
asin()	double asin(double x);	*1
acos()	double acos(double x);	*1
atan()	double atan(double x);	
atan2()	double atan2(double y, double x);	*1
sinh()	double sinh(double x);	*1
cosh()	double cosh(double x);	*1
tanh()	double tanh(double x);	

Character Type Determination/Conversion Functions (header file: ctype.h)

isalnum()	int isalnum(int c);
isalpha()	int isalpha(int c);
iscntrl()	int iscntrl(int c);
isdigit()	int isdigit(int c);
isgraph()	int isgraph(int c);
islower()	int islower(int c);
isprint()	int isprint(int c);
ispunct()	int ispunct(int c);
isspace()	int isspace(int c);
isupper()	int isupper(int c);
isxdigit()	int isxdigit(int c);
tolower()	int tolower(int c);
toupper()	int toupper(int c);

Variable Argument Macros (header file: stdarg.h)

va_start()	void va_start(va_list ap, type lastarg);
va_arg()	type va_arg(va_list ap, type);
va_end()	void va_end(va_list ap);

*1 These functions need to declare and initialize the global variables.

Character Functions (header file: string.h)

memchr()	void *memchr(const void *s, int c, size_t n);
memcmp()	int memcmp(const void *s1, const void *s2, size_t n);
memcpy()	void *memcpy(void *s1, const void *s2, size_t n);
memmove()	void *memmove(void *s1, const void *s2, size_t n);
memset()	void *memset(void *s, int c, size_t n);
strcat()	char *strcat(char *s1, const char *s2);
strchr()	char *strchr(const char *s, int c);
strcmp()	int strcmp(const char *s1, const char *s2);
strcpy()	char *strcpy(char *s1, const char *s2);
strcspn()	size_t strcspn(const char *s1, const char *s2);
strerror()	char *strerror(int code);
strlen()	size_t strlen(const char *s);
strncat()	char *strncat(char *s1, const char *s2, size_t n);
strncmp()	int strncmp(const char *s1, const char *s2, size_t n);
strncpy()	char *strncpy(char *s1, const char *s2, size_t n);
strpbrk()	char *strpbrk(const char *s1, const char *s2);
strrchr()	char *strrchr(const char *str, int c);
strspn()	size_t strspn(const char *s1, const char *s2);
strstr()	char *strstr(const char *s1, const char *s2);
strtok()	char *strtok(char *s1, const char *s2);

***1 Declaring and Initializing Global Variables**

FILE _iob[FOPEN_MAX+1];	_iob[N]._flag= _UGETN; _iob[N]._buf=0; _iob[N]._fd=N; (N=0: stdin, N=1: stdout, N=2: stderr)
FILE *stdin;	stdin=&_iob[0];
FILE *stdout;	stdout=&_iob[1];
FILE *stderr;	stderr=&_iob[2];
int errno;	errno=0;
unsigned int seed;	seed=1;
time_t gm_sec;	gm_sec=-1;

***2 Definition of Lower-level Functions**

read()	int read(int fd, char *buf, int nbytes); unsigned char READ_BUF[65]; (Variable name is arbitrary) unsigned char READ_EOF;
write()	int write(int fd, char *buf, int nbytes); unsigned char WRITE_BUF[65]; (Variable name is arbitrary)

Symbols in the Instruction List

Registers/Register Data

%rd, rd:	A general-purpose register (R0–R7) used as the destination register or its contents
%rs, rs:	A general-purpose register (R0–R7) used as the source register or its contents
%rb, rb:	A general-purpose register (R0–R7) that has stored a base address to be accessed in the register indirect addressing mode or its contents
%sp, sp:	Stack pointer (SP) or its contents
%pc, pc:	Program counter (PC) or its contents

Memory/Addresses/Memory Data

[%rb], [%sp]:	Specification for register indirect addressing
[%rb]+, [%sp]+:	Specification for register indirect addressing with post-increment
[%rb]-, [%sp]-:	Specification for register indirect addressing with post-decrement
-%rb], -[%sp]:	Specification for register indirect addressing with pre-decrement
[%sp+immX]:	Specification for register indirect addressing with a displacement
[imm7]:	Specification for a memory address with an immediate data
B[XXX]:	An address specified with XXX, or the byte data stored in the address
W[XXX]:	A 16-bit address specified with XXX, or the word data stored in the address
A[XXX]:	A 32-bit address specified with XXX, or the 24-bit or 32-bit data stored in the address

Immediate

immX:	A X-bit unsigned immediate data
signX:	A X-bit signed immediate data

Symbol/Label

Symbol:	A symbol that points an address.
Label:	A branch destination label.

Bit Field

(X):	Bit X of data.
(X:Y):	A bit field from bit X to bit Y.
{X, Y...}:	Indicates a bit (data) configuration.

Functions

←:	Indicates that the right item is loaded or set to the left item.
+:	Addition
-:	Subtraction
&:	AND
:	OR
^:	XOR
!:	NOT

Flags

IL:	Interrupt level
IE:	Interrupt enable flag
C:	Carry flag
V:	Overflow flag
Z:	Zero flag
N:	Negative flag
-:	Not changed
↔:	Set (1), reset (0) or not changed
1:	Set (1)
0:	Reset (0)

D

○:	Indicates that the instruction can be used as a delayed instruction.
-:	Indicates that the instruction cannot be used as a delayed instruction.

Notes

- The instruction list contains the basic instructions in the S1C17 instruction set and the extended instructions (s... and x..., except for xor)
- "*Italic basic instructions*" indicate that the upper compatible extended instructions are provided.

Instruction List (2)

Assembly Programming

[illegible]

Instruction List (3)

Assembly Programming

Classification	Mnemonic		Function	Flags						D
	Opcode	Operand		IL	IE	C	V	Z	N	
16-bit data transfer	ld	%rd, %rs	rd(15:0)←rs(15:0), rd(23:16)←0	–	–	–	–	–	–	○
		%rd, sign7	rd(6:0)←sign7(6:0), rd(15:7)←sign7(6), rd(23:16)←0	–	–	–	–	–	–	○
		%rd, [%rb]	rd(15:0)←W[rb], rd(23:16)←0	–	–	–	–	–	–	○
		%rd, [%rb]+	rd(15:0)←W[rb], rd(23:16)←0, rb(23:0)←rb(23:0)+2	–	–	–	–	–	–	○
		%rd, [%rb]-	rd(15:0)←W[rb], rd(23:16)←0, rb(23:0)←rb(23:0)-2	–	–	–	–	–	–	○
		%rd, -[%rb]	rb(23:0)←rb(23:0)-2, rd(15:0)←W[rb], rd(23:16)←0	–	–	–	–	–	–	○
		%rd, [%sp+imm7]	rd(15:0)←W[sp+imm7], rd(23:16)←0	–	–	–	–	–	–	○
		%rd, [imm7]	rd(15:0)←W[imm7], rd(23:16)←0	–	–	–	–	–	–	○
		[%rb], %rs	W[rb]←rs(15:0)	–	–	–	–	–	–	○
		[%rb]+, %rs	W[rb]←rs(15:0), rb(23:0)←rb(23:0)+2	–	–	–	–	–	–	○
		[%rb]-, %rs	W[rb]←rs(15:0), rb(23:0)←rb(23:0)-2	–	–	–	–	–	–	○
		-%[rb], %rs	rb(23:0)←rb(23:0)-2, W[rb]←rs(15:0)	–	–	–	–	–	–	○
		[%sp+imm7], %rs	W[sp+imm7]←rs(15:0)	–	–	–	–	–	–	○
		[imm7], %rs	W[imm7]←rs(15:0)	–	–	–	–	–	–	○
	sld	%rd, imm16	%rd←imm16	–	–	–	–	–	–	–
		%rd, symbol±imm16	%rd←symbol±imm16(15:0)	–	–	–	–	–	–	–
		%rd, [%sp+imm20]	%rd←W[%sp+imm20]	–	–	–	–	–	–	–
		%rd, [imm20]	%rd←W[imm20]	–	–	–	–	–	–	–
		[%sp+imm20], %rs	W[%sp+imm20]←%rs(15:0)	–	–	–	–	–	–	–
		[imm20], %rs	W[imm20]←%rs(15:0)	–	–	–	–	–	–	–
	xld	%rd, imm16	%rd←imm16	–	–	–	–	–	–	–
		%rd, symbol±imm16	%rd←symbol±imm16(15:0)	–	–	–	–	–	–	–
		%rd, [%sp+imm24]	%rd←W[%sp+imm24]	–	–	–	–	–	–	–
		%rd, [imm24]	%rd←W[imm24]	–	–	–	–	–	–	–
		[%sp+imm24], %rs	W[%sp+imm24]←%rs(15:0)	–	–	–	–	–	–	–
		[imm24], %rs	W[imm24]←%rs(15:0)	–	–	–	–	–	–	–
32-bit data transfer	ld.a	%rd, %rs	rd(23:0)←rs(23:0)	–	–	–	–	–	–	○
		%rd, imm7	rd(6:0)←imm7(6:0), rd(23:7)←0	–	–	–	–	–	–	○
		%rd, [%rb]	rd(23:0)←A[rb](23:0), ignored←A[rb](31:24)	–	–	–	–	–	–	○
		%rd, [%rb]+	rd(23:0)←A[rb](23:0), ignored←A[rb](31:24), rb(23:0)←rb(23:0)+4	–	–	–	–	–	–	○
		%rd, [%rb]-	rd(23:0)←A[rb](23:0), ignored←A[rb](31:24), rb(23:0)←rb(23:0)-4	–	–	–	–	–	–	○
		%rd, -[%rb]	rb(23:0)←rb(23:0)-4, rd(23:0)←A[rb](23:0), ignored←A[rb](31:24)	–	–	–	–	–	–	○
		%rd, [%sp+imm7]	rd(23:0)←A[sp+imm7](23:0), ignored←A[sp+imm7](31:24)	–	–	–	–	–	–	○
		%rd, [imm7]	rd(23:0)←A[imm7](23:0), ignored←A[imm7](31:24)	–	–	–	–	–	–	○

Remarks

Instruction List (4)

Assembly Programming

Classification	Mnemonic		Function	Flags						D
	Opcode	Operand		IL	IE	C	V	Z	N	
32-bit data transfer	ld.a	[%rb], %rs	A[rb](23:0)←rs(23:0), A[rb](31:24)←0	–	–	–	–	–	–	○
		[%rb]+, %rs	A[rb](23:0)←rs(23:0), A[rb](31:24)←0, rb(23:0)←rb(23:0)+4	–	–	–	–	–	–	○
		[%rb]-, %rs	A[rb](23:0)←rs(23:0), A[rb](31:24)←0, rb(23:0)←rb(23:0)-4	–	–	–	–	–	–	○
		[-%rb], %rs	rb(23:0)←rb(23:0)-4, A[rb](23:0)←rs(23:0), A[rb](31:24)←0	–	–	–	–	–	–	○
		[%sp+imm7], %rs	A[sp+imm7](23:0)←rs(23:0), A[sp+imm7](31:24)←0	–	–	–	–	–	–	○
		[imm7], %rs	A[imm7](23:0)←rs(23:0), A[imm7](31:24)←0	–	–	–	–	–	–	○
		%rd, %sp	rd(23:2)←sp(23:2), rd(1:0)←0	–	–	–	–	–	–	○
		%rd, %pc	rd(23:0)←pc(23:0)+2	–	–	–	–	–	–	○
		%rd, [%sp]	rd(23:0)←A[sp](23:0), ignored←A[sp](31:24)	–	–	–	–	–	–	○
		%rd, [%sp]+	rd(23:0)←A[sp](23:0), ignored←A[sp](31:24), sp(23:0)←sp(23:0)+4	–	–	–	–	–	–	○
		%rd, [%sp]-	rd(23:0)←A[sp](23:0), ignored←A[sp](31:24), sp(23:0)←sp(23:0)-4	–	–	–	–	–	–	○
		%rd, [-%sp]	sp(23:0)←sp(23:0)-4, rd(23:0)←A[sp](23:0), ignored←A[sp](31:24)	–	–	–	–	–	–	○
		[%sp], %rs	A[sp](23:0)←rs(23:0), A[sp](31:24)←0	–	–	–	–	–	–	○
		[%sp]+, %rs	A[sp](23:0)←rs(23:0), A[sp](31:24)←0, sp(23:0)←sp(23:0)+4	–	–	–	–	–	–	○
		[%sp]-, %rs	A[sp](23:0)←rs(23:0), A[sp](31:24)←0, sp(23:0)←sp(23:0)-4	–	–	–	–	–	–	○
		[-%sp], %rs	sp(23:0)←sp(23:0)-4, A[sp](23:0)←rs(23:0), A[sp](31:24)←0	–	–	–	–	–	–	○
		%sp, %rs	sp(23:2)←rs(23:2)	–	–	–	–	–	–	○
		%sp, imm7	sp(6:2)←imm7(6:2), sp(23:7)←0	–	–	–	–	–	–	○
	sld.a	%rd, imm20	%rd←imm20	–	–	–	–	–	–	–
		%sp, imm20	%sp←imm20	–	–	–	–	–	–	–
		%rd, symbol±imm20	%rd←symbol±imm20(19:0)	–	–	–	–	–	–	–
		%sp, symbol±imm20	%sp←symbol±imm20(19:0)	–	–	–	–	–	–	–
		%rd, [%sp+imm20]	%rd←A[%sp+imm20](23:0), ignored←A[%sp+imm20](31:24)	–	–	–	–	–	–	–
		%rd, [imm20]	%rd←A[imm20](23:0), ignored←A[imm20](31:24)	–	–	–	–	–	–	–
		[%sp+imm20], %rs	A[%sp+imm20](23:0)←%rs(23:0), A[%sp+imm20](31:24)←0	–	–	–	–	–	–	–
		[imm20], %rs	A[imm20](23:0)←%rs(23:0), A[imm20](31:24)←0	–	–	–	–	–	–	–
	xld.a	%rd, imm24	%rd←imm24	–	–	–	–	–	–	–
		%sp, imm24	%sp←imm24	–	–	–	–	–	–	–
		%rd, symbol±imm24	%rd←symbol±imm24(23:0)	–	–	–	–	–	–	–
		%sp, symbol±imm24	%sp←symbol±imm24(23:0)	–	–	–	–	–	–	–
		%rd, [%sp+imm24]	%rd←A[%sp+imm24](23:0), ignored←A[%sp+imm24](31:24)	–	–	–	–	–	–	–
		%rd, [imm24]	%rd←A[imm24](23:0), ignored←A[imm24](31:24)	–	–	–	–	–	–	–
		[%sp+imm24], %rs	A[%sp+imm24](23:0)←%rs(23:0), A[imm24](31:24)←0	–	–	–	–	–	–	–
		[imm24], %rs	A[imm24](23:0)←%rs(23:0), A[%sp+imm24](31:24)←0	–	–	–	–	–	–	–

Remarks

Classification	Mnemonic		Function	Flags						D
	Opcode	Operand		IL	IE	C	V	Z	N	
Arithmetic operation	add	%rd, %rs	rd(15:0)←rd(15:0)+rs(15:0), rd(23:16)←0	–	–	↔	↔	↔	↔	○
	add/c	%rd, %rs	rd(15:0)←rd(15:0)+rs(15:0), rd(23:16)←0 if C = 1 (nop if C = 0)	–	–	–	↔	↔	↔	○
	add/nc	%rd, %rs	rd(15:0)←rd(15:0)+rs(15:0), rd(23:16)←0 if C = 0 (nop if C = 1)	–	–	–	↔	↔	↔	○
	<i>add</i>	<i>%rd, imm7</i>	rd(15:0)←rd(15:0)+imm7 (with zero extension), rd(23:16)←0	–	–	↔	↔	↔	↔	○
	sadd	%rd, imm16	rd(15:0)←rd(15:0)+imm16, rd(23:16)←0	–	–	↔	↔	↔	↔	–
	xadd	%rd, imm16	rd(15:0)←rd(15:0)+imm16, rd(23:16)←0	–	–	↔	↔	↔	↔	–
	add.a	%rd, %rs	rd(23:0)←rd(23:0)+rs(23:0)	–	–	–	–	–	–	○
	add.a/c	%rd, %rs	rd(23:0)←rd(23:0)+rs(23:0) if C = 1 (nop if C = 0)	–	–	–	–	–	–	○
	add.a/nc	%rd, %rs	rd(23:0)←rd(23:0)+rs(23:0) if C = 0 (nop if C = 1)	–	–	–	–	–	–	○
	add.a	%sp, %rs	sp(23:0)←sp(23:0)+rs(23:0)	–	–	–	–	–	–	○
		<i>%rd, imm7</i>	rd(23:0)←rd(23:0)+imm7 (with zero extension)	–	–	–	–	–	–	○
		<i>%sp, imm7</i>	sp(23:0)←sp(23:0)+imm7 (with zero extension)	–	–	–	–	–	–	○
	sadd.a	%rd, imm20	rd(23:0)←rd(23:0)+imm20 (with zero extension)	–	–	–	–	–	–	–
		%sp, imm20	sp(23:0)←sp(23:0)+imm20 (with zero extension)	–	–	–	–	–	–	–
	xadd.a	%rd, imm24	rd(23:0)←rd(23:0)+imm24	–	–	–	–	–	–	–
		%sp, imm24	sp(23:0)←sp(23:0)+imm24	–	–	–	–	–	–	–
	adc	%rd, %rs	rd(15:0)←rd(15:0)+rs(15:0)+C, rd(23:16)←0	–	–	↔	↔	↔	↔	○
	adc/c	%rd, %rs	rd(15:0)←rd(15:0)+rs(15:0)+C, rd(23:16)←0 if C = 1 (nop if C = 0)	–	–	–	↔	↔	↔	○
	adc/nc	%rd, %rs	rd(15:0)←rd(15:0)+rs(15:0)+C, rd(23:16)←0 if C = 0 (nop if C = 1)	–	–	–	↔	↔	↔	○
	<i>adc</i>	<i>%rd, imm7</i>	rd(15:0)←rd(15:0)+imm7 (with zero extension)+C, rd(23:16)←0	–	–	↔	↔	↔	↔	○
	sadc	%rd, imm16	rd(15:0)←rd(15:0)+imm16+C, rd(23:16)←0	–	–	↔	↔	↔	↔	–
	xadc	%rd, imm16	rd(15:0)←rd(15:0)+imm16+C, rd(23:16)←0	–	–	↔	↔	↔	↔	–
	sub	%rd, %rs	rd(15:0)←rd(15:0)-rs(15:0), rd(23:16)←0	–	–	↔	↔	↔	↔	○
	sub/c	%rd, %rs	rd(15:0)←rd(15:0)-rs(15:0), rd(23:16)←0 if C = 1 (nop if C = 0)	–	–	–	↔	↔	↔	○
	sub/nc	%rd, %rs	rd(15:0)←rd(15:0)-rs(15:0), rd(23:16)←0 if C = 0 (nop if C = 1)	–	–	–	↔	↔	↔	○
	<i>sub</i>	<i>%rd, imm7</i>	rd(15:0)←rd(15:0)-imm7 (with zero extension), rd(23:16)←0	–	–	↔	↔	↔	↔	○
	ssub	%rd, imm16	rd(15:0)←rd(15:0)-imm16, rd(23:16)←0	–	–	↔	↔	↔	↔	–
	xsub	%rd, imm16	rd(15:0)←rd(15:0)-imm16, rd(23:16)←0	–	–	↔	↔	↔	↔	–
	sub.a	%rd, %rs	rd(23:0)←rd(23:0)-rs(23:0)	–	–	–	–	–	–	○
	sub.a/c	%rd, %rs	rd(23:0)←rd(23:0)-rs(23:0) if C = 1 (nop if C = 0)	–	–	–	–	–	–	○
	sub.a/nc	%rd, %rs	rd(23:0)←rd(23:0)-rs(23:0) if C = 0 (nop if C = 1)	–	–	–	–	–	–	○
	sub.a	%sp, %rs	sp(23:0)←sp(23:0)-rs(23:0)	–	–	–	–	–	–	○
<i>%rd, imm7</i>		rd(23:0)←rd(23:0)-imm7 (with zero extension)	–	–	–	–	–	–	○	
<i>%sp, imm7</i>		sp(23:0)←sp(23:0)-imm7 (with zero extension)	–	–	–	–	–	–	○	
Remarks										

Instruction List (6)

Assembly Programming

Classification	Mnemonic		Function	Flags						D
	Opcode	Operand		IL	IE	C	V	Z	N	
Arithmetic operation	ssub.a	%rd, imm20	rd(23:0)←rd(23:0)-imm20 (with zero extension)	-	-	-	-	-	-	-
		%sp, imm20	sp(23:0)←sp(23:0)-imm20 (with zero extension)	-	-	-	-	-	-	-
	xsub.a	%rd, imm24	rd(23:0)←rd(23:0)-imm24	-	-	-	-	-	-	-
		%sp, imm24	sp(23:0)←sp(23:0)-imm24	-	-	-	-	-	-	-
	sbc	%rd, %rs	rd(15:0)←rd(15:0)-rs(15:0)-C, rd(23:16)←0	-	-	↔	↔	↔	↔	○
	sbc/c	%rd, %rs	rd(15:0)←rd(15:0)-rs(15:0)-C, rd(23:16)←0 if C = 1 (nop if C = 0)	-	-	-	↔	↔	↔	○
	sbc/nc	%rd, %rs	rd(15:0)←rd(15:0)-rs(15:0)-C, rd(23:16)←0 if C = 0 (nop if C = 1)	-	-	-	↔	↔	↔	○
	<i>sbc</i>	<i>%rd, imm7</i>	rd(15:0)←rd(15:0)-imm7 (with zero extension)-C, rd(23:16)←0	-	-	↔	↔	↔	↔	○
	ssbc	%rd, imm16	rd(15:0)←rd(15:0)-imm16-C, rd(23:16)←0	-	-	↔	↔	↔	↔	-
	xsbc	%rd, imm16	rd(15:0)←rd(15:0)-imm16-C, rd(23:16)←0	-	-	↔	↔	↔	↔	-
	cmp	%rd, %rs	rd(15:0)-rs(15:0)	-	-	↔	↔	↔	↔	○
	cmp/c	%rd, %rs	rd(15:0)-rs(15:0) if C = 1 (nop if C = 0)	-	-	-	↔	↔	↔	○
	cmp/nc	%rd, %rs	rd(15:0)-rs(15:0) if C = 0 (nop if C = 1)	-	-	-	↔	↔	↔	○
	<i>cmp</i>	<i>%rd, sign7</i>	rd(15:0)-sign7 (with sign extension)	-	-	↔	↔	↔	↔	○
	scmp	%rd, imm16	rd(15:0)-imm16	-	-	↔	↔	↔	↔	-
	xcmp	%rd, imm16	rd(15:0)-imm16	-	-	↔	↔	↔	↔	-
	cmp.a	%rd, %rs	d(23:0)-rs(23:0)	-	-	↔	-	↔	-	○
	cmp.a/c	%rd, %rs	rd(23:0)-rs(23:0) if C = 1 (nop if C = 0)	-	-	-	-	↔	-	○
	cmp.a/nc	%rd, %rs	rd(23:0)-rs(23:0) if C = 0 (nop if C = 1)	-	-	-	-	↔	-	○
	<i>cmp.a</i>	<i>%rd, imm7</i>	rd(23:0)-imm7 (with zero extension)	-	-	↔	-	↔	-	○
	scmp.a	%rd, imm20	rd(23:0)-imm20 (with zero extension)	-	-	↔	-	↔	-	-
	xcmp.a	%rd, imm24	rd(23:0)-imm24	-	-	↔	-	↔	-	-
	cmc	%rd, %rs	rd(15:0)-rs(15:0)-C	-	-	↔	↔	↔	↔	○
	cmc/c	%rd, %rs	rd(15:0)-rs(15:0)-C if C = 1 (nop if C = 0)	-	-	-	↔	↔	↔	○
	cmc/nc	%rd, %rs	rd(15:0)-rs(15:0)-C if C = 0 (nop if C = 1)	-	-	-	↔	↔	↔	○
	<i>cmc</i>	<i>%rd, sign7</i>	rd(15:0)-sign7 (with sign extension)-C	-	-	↔	↔	↔	↔	○
	scmc	%rd, imm16	rd(15:0)-imm16-C	-	-	↔	↔	↔	↔	-
	xcmc	%rd, imm16	rd(15:0)-imm16-C	-	-	↔	↔	↔	↔	-
Logic operation	and	%rd, %rs	rd(15:0)←rd(15:0)&rs(15:0), rd(23:16)←0	-	-	-	0	↔	↔	○
	and/c	%rd, %rs	rd(15:0)←rd(15:0)&rs(15:0), rd(23:16)←0 if C = 1 (nop if C = 0)	-	-	-	0	↔	↔	○
	and/nc	%rd, %rs	rd(15:0)←rd(15:0)&rs(15:0), rd(23:16)←0 if C = 0 (nop if C = 1)	-	-	-	0	↔	↔	○
	<i>and</i>	<i>%rd, sign7</i>	rd(15:0)←rd(15:0)&sign7 (with sign extension), rd(23:16)←0	-	-	-	0	↔	↔	○
	sand	%rd, imm16	rd(15:0)←rd(15:0)&imm16, rd(23:16)←0	-	-	-	0	↔	↔	-
	xand	%rd, imm16	rd(15:0)←rd(15:0)&imm16, rd(23:16)←0	-	-	-	0	↔	↔	-

Remarks

Classification	Mnemonic		Function	Flags						D
	Opcode	Operand		IL	IE	C	V	Z	N	
Logic operation	or	%rd, %rs	d(15:0)←rd(15:0) rs(15:0), rd(23:16)←0	–	–	–	0	↔	↔	○
	or/c	%rd, %rs	rd(15:0)←rd(15:0) rs(15:0), rd(23:16)←0 if C = 1 (nop if C = 0)	–	–	–	0	↔	↔	○
	or/nc	%rd, %rs	rd(15:0)←rd(15:0) rs(15:0), rd(23:16)←0 if C = 0 (nop if C = 1)	–	–	–	0	↔	↔	○
	or	%rd, sign7	rd(15:0)←rd(15:0) sign7 (with sign extension), rd(23:16)←0	–	–	–	0	↔	↔	○
	soor	%rd, imm16	rd(15:0)←rd(15:0) imm16, rd(23:16)←0	–	–	–	0	↔	↔	–
	xoor	%rd, imm16	rd(15:0)←rd(15:0) imm16, rd(23:16)←0	–	–	–	0	↔	↔	–
	xor	%rd, %rs	rd(15:0)←rd(15:0)^rs(15:0), rd(23:16)←0	–	–	–	0	↔	↔	○
	xor/c	%rd, %rs	rd(15:0)←rd(15:0)^rs(15:0), rd(23:16)←0 if C = 1 (nop if C = 0)	–	–	–	0	↔	↔	○
	xor/nc	%rd, %rs	rd(15:0)←rd(15:0)^rs(15:0), rd(23:16)←0 if C = 0 (nop if C = 1)	–	–	–	0	↔	↔	○
	xor	%rd, sign7	rd(15:0)←rd(15:0)^sign7 (with sign extension), rd(23:16)←0	–	–	–	0	↔	↔	○
	sxor	%rd, imm16	rd(15:0)←rd(15:0)^imm16, rd(23:16)←0	–	–	–	0	↔	↔	–
	xxor	%rd, imm16	rd(15:0)←rd(15:0)^imm16, rd(23:16)←0	–	–	–	0	↔	↔	–
	not	%rd, %rs	rd(15:0)←!rs(15:0), rd(23:16)←0	–	–	–	0	↔	↔	○
	not/c	%rd, %rs	rd(15:0)←!rs(15:0), rd(23:16)←0 if C = 1 (nop if C = 0)	–	–	–	0	↔	↔	○
	not/nc	%rd, %rs	rd(15:0)←!rs(15:0), rd(23:16)←0 if C = 0 (nop if C = 1)	–	–	–	0	↔	↔	○
	not	%rd, sign7	rd(15:0)←!sign7 (with sign extension), rd(23:16)←0	–	–	–	0	↔	↔	○
	snot	%rd, imm16	rd(15:0)←!imm16, rd(23:16)←0	–	–	–	0	↔	↔	–
	xnot	%rd, imm16	rd(15:0)←!imm16, rd(23:16)←0	–	–	–	0	↔	↔	–
Branch	jpr / jpr.d	%rb sign10	pc←pc+2+rb pc←pc+2+sign11; sign11={sign10,0}	–	–	–	–	–	–	–
	sjpr / sjpr.d	label±imm20	pc←label±imm20	–	–	–	–	–	–	–
		sign20	pc←pc+2+sign20	–	–	–	–	–	–	–
	xjpr / xjpr.d	label±imm24	pc←label±imm24	–	–	–	–	–	–	–
		sign24	pc←pc+2+sign24	–	–	–	–	–	–	–
	jpa / jpa.d	%rb	pc←rb	–	–	–	–	–	–	–
		imm7	pc←imm7	–	–	–	–	–	–	–
	sjpa / sjpa.d	label±imm20	pc←label±imm20	–	–	–	–	–	–	–
		imm20	pc←imm20	–	–	–	–	–	–	–
	xjpa / xjpa.d	label±imm24	pc←label±imm24	–	–	–	–	–	–	–
		imm24	pc←imm24	–	–	–	–	–	–	–
	jrgt / jrgt.d	sign7	pc←pc+2+sign8 if !Z&!(N^V) is true; sign8={sign7,0}	–	–	–	–	–	–	–
	sjrgt / sjrgt.d	label±imm20	pc←label±imm20 if !Z&!(N^V) is true	–	–	–	–	–	–	–
		sign20	pc←pc+2+sign20 if !Z&!(N^V) is true	–	–	–	–	–	–	–
xjrgt / xjrgt.d	label±imm24	pc←label±imm24 if !Z&!(N^V) is true	–	–	–	–	–	–	–	
	sign24	pc←pc+2+sign24 if !Z&!(N^V) is true	–	–	–	–	–	–	–	

Remarks

Instruction List (8)

Assembly Programming

Classification	Mnemonic		Function	Flags						D
	Opcode	Operand		IL	IE	C	V	Z	N	
Branch	<i>jrge / jrge.d</i>	<i>sign7</i>	$pc \leftarrow pc + 2 + sign8$ if $!(N^{\wedge}V)$ is true; $sign8 = \{sign7, 0\}$	-	-	-	-	-	-	-
	<i>sjrge / sjrge.d</i>	$label \pm imm20$	$pc \leftarrow label \pm imm20$ if $!(N^{\wedge}V)$ is true	-	-	-	-	-	-	-
		<i>sign20</i>	$pc \leftarrow pc + 2 + sign20$ if $!(N^{\wedge}V)$ is true	-	-	-	-	-	-	-
	<i>xjrge / xjrge.d</i>	$label \pm imm24$	$pc \leftarrow label \pm imm24$ if $!(N^{\wedge}V)$ is true	-	-	-	-	-	-	-
		<i>sign24</i>	$pc \leftarrow pc + 2 + sign24$ if $!(N^{\wedge}V)$ is true	-	-	-	-	-	-	-
	<i>jrlt / jrlt.d</i>	<i>sign7</i>	$pc \leftarrow pc + 2 + sign8$ if $N^{\wedge}V$ is true; $sign8 = \{sign7, 0\}$	-	-	-	-	-	-	-
	<i>sjrlt / sjrlt.d</i>	$label \pm imm20$	$pc \leftarrow label \pm imm20$ if $N^{\wedge}V$ is true	-	-	-	-	-	-	-
		<i>sign20</i>	$pc \leftarrow pc + 2 + sign20$ if $N^{\wedge}V$ is true	-	-	-	-	-	-	-
	<i>xjrlt / xjrlt.d</i>	$label \pm imm24$	$pc \leftarrow label \pm imm24$ if $N^{\wedge}V$ is true	-	-	-	-	-	-	-
		<i>sign24</i>	$pc \leftarrow pc + 2 + sign24$ if $N^{\wedge}V$ is true	-	-	-	-	-	-	-
	<i>jrle / jrle.d</i>	<i>sign7</i>	$pc \leftarrow pc + 2 + sign8$ if $Z \mid (N^{\wedge}V)$ is true; $sign8 = \{sign7, 0\}$	-	-	-	-	-	-	-
	<i>sjrle / sjrle.d</i>	$label \pm imm20$	$pc \leftarrow label \pm imm20$ if $Z \mid (N^{\wedge}V)$ is true	-	-	-	-	-	-	-
		<i>sign20</i>	$pc \leftarrow pc + 2 + sign20$ if $Z \mid (N^{\wedge}V)$ is true	-	-	-	-	-	-	-
	<i>xjrle / xjrle.d</i>	$label \pm imm24$	$pc \leftarrow label \pm imm24$ if $Z \mid (N^{\wedge}V)$ is true	-	-	-	-	-	-	-
		<i>sign24</i>	$pc \leftarrow pc + 2 + sign24$ if $Z \mid (N^{\wedge}V)$ is true	-	-	-	-	-	-	-
	<i>jrugt / jrugt.d</i>	<i>sign7</i>	$pc \leftarrow pc + 2 + sign8$ if $!Z \& !C$ is true; $sign8 = \{sign7, 0\}$	-	-	-	-	-	-	-
	<i>sjrugt / sjrugt.d</i>	$label \pm imm20$	$pc \leftarrow label \pm imm20$ if $!Z \& !C$ is true	-	-	-	-	-	-	-
		<i>sign20</i>	$pc \leftarrow pc + 2 + sign20$ if $!Z \& !C$ is true	-	-	-	-	-	-	-
	<i>xjrugt / xjrugt.d</i>	$label \pm imm24$	$pc \leftarrow label \pm imm24$ if $!Z \& !C$ is true	-	-	-	-	-	-	-
		<i>sign24</i>	$pc \leftarrow pc + 2 + sign24$ if $!Z \& !C$ is true	-	-	-	-	-	-	-
	<i>jruge / jruge.d</i>	<i>sign7</i>	$pc \leftarrow pc + 2 + sign8$ if $!C$ is true; $sign8 = \{sign7, 0\}$	-	-	-	-	-	-	-
	<i>sjruge / sjruge.d</i>	$label \pm imm20$	$pc \leftarrow label \pm imm20$ if $!C$ is true	-	-	-	-	-	-	-
		<i>sign20</i>	$pc \leftarrow pc + 2 + sign20$ if $!C$ is true	-	-	-	-	-	-	-
	<i>xjruge / xjruge.d</i>	$label \pm imm24$	$pc \leftarrow label \pm imm24$ if $!C$ is true	-	-	-	-	-	-	-
		<i>sign24</i>	$pc \leftarrow pc + 2 + sign24$ if $!C$ is true	-	-	-	-	-	-	-
	<i>jrult / jrult.d</i>	<i>sign7</i>	$pc \leftarrow pc + 2 + sign8$ if C is true; $sign8 = \{sign7, 0\}$	-	-	-	-	-	-	-
	<i>sjrult / sjrult.d</i>	$label \pm imm20$	$pc \leftarrow label \pm imm20$ if C is true	-	-	-	-	-	-	-
		<i>sign20</i>	$pc \leftarrow pc + 2 + sign20$ if C is true	-	-	-	-	-	-	-
	<i>xjrult / xjrult.d</i>	$label \pm imm24$	$pc \leftarrow label \pm imm24$ if C is true	-	-	-	-	-	-	-
		<i>sign24</i>	$pc \leftarrow pc + 2 + sign24$ if C is true	-	-	-	-	-	-	-
	<i>jrule / jrule.d</i>	<i>sign7</i>	$pc \leftarrow pc + 2 + sign8$ if $Z \mid C$ is true; $sign8 = \{sign7, 0\}$	-	-	-	-	-	-	-
	<i>sjrule / sjrule.d</i>	$label \pm imm20$	$pc \leftarrow label \pm imm20$ if $Z \mid C$ is true	-	-	-	-	-	-	-
		<i>sign20</i>	$pc \leftarrow pc + 2 + sign20$ if $Z \mid C$ is true	-	-	-	-	-	-	-
	<i>xjrule / xjrule.d</i>	$label \pm imm24$	$pc \leftarrow label \pm imm24$ if $Z \mid C$ is true	-	-	-	-	-	-	-
		<i>sign24</i>	$pc \leftarrow pc + 2 + sign24$ if $Z \mid C$ is true	-	-	-	-	-	-	-

Remarks

Instruction List (9)

Assembly Programming

Classification	Mnemonic		Function	Flags						D
	Opcode	Operand		IL	IE	C	V	Z	N	
Branch	<i>jreq / jreq.d</i>	<i>sign7</i>	$pc \leftarrow pc+2+sign8$ if Z is true; $sign8=\{sign7,0\}$	-	-	-	-	-	-	-
	<i>sjreq / sjreq.d</i>	$label \pm imm20$	$pc \leftarrow label \pm imm20$ if Z is true	-	-	-	-	-	-	-
		<i>sign20</i>	$pc \leftarrow pc+2+sign20$ if Z is true	-	-	-	-	-	-	-
	<i>xjreq / xjreq.d</i>	$label \pm imm24$	$pc \leftarrow label \pm imm24$ if Z is true	-	-	-	-	-	-	-
		<i>sign24</i>	$pc \leftarrow pc+2+sign24$ if Z is true	-	-	-	-	-	-	-
	<i>jrne / jrne.d</i>	<i>sign7</i>	$pc \leftarrow pc+2+sign8$ if !Z is true; $sign8=\{sign7,0\}$	-	-	-	-	-	-	-
	<i>sjrne / sjrne.d</i>	$label \pm imm20$	$pc \leftarrow label \pm imm20$ if !Z is true	-	-	-	-	-	-	-
		<i>sign20</i>	$pc \leftarrow pc+2+sign20$ if !Z is true	-	-	-	-	-	-	-
	<i>xjrne / xjrne.d</i>	$label \pm imm24$	$pc \leftarrow label \pm imm24$ if !Z is true	-	-	-	-	-	-	-
		<i>sign24</i>	$pc \leftarrow pc+2+sign24$ if !Z is true	-	-	-	-	-	-	-
	<i>call / call.d</i>	<i>%rb</i>	$sp \leftarrow sp-4$, $A[sp] \leftarrow pc+2(d=0)/4(d=1)$, $pc \leftarrow pc+2+rb$	-	-	-	-	-	-	-
		<i>sign10</i>	$sp \leftarrow sp-4$, $A[sp] \leftarrow pc+2(d=0)/4(d=1)$, $pc \leftarrow pc+2+sign11$; $sign11=\{sign10,0\}$	-	-	-	-	-	-	-
	<i>scall / scall.d</i>	$label \pm imm20$	$sp \leftarrow sp-4$, $A[sp] \leftarrow pc+2(d=0)/4(d=1)$, $pc \leftarrow label \pm imm20$	-	-	-	-	-	-	-
		<i>sign20</i>	$sp \leftarrow sp-4$, $A[sp] \leftarrow pc+2(d=0)/4(d=1)$, $pc \leftarrow pc+2+sign20$	-	-	-	-	-	-	-
	<i>xcall / xcall.d</i>	$label \pm imm24$	$sp \leftarrow sp-4$, $A[sp] \leftarrow pc+2(d=0)/4(d=1)$, $pc \leftarrow label \pm imm24$	-	-	-	-	-	-	-
		<i>sign24</i>	$sp \leftarrow sp-4$, $A[sp] \leftarrow pc+2(d=0)/4(d=1)$, $pc \leftarrow pc+2+sign24$	-	-	-	-	-	-	-
	<i>calla / calla.d</i>	<i>%rb</i>	$sp \leftarrow sp-4$, $A[sp] \leftarrow pc+2(d=0)/4(d=1)$, $pc \leftarrow rb$	-	-	-	-	-	-	-
		<i>imm7</i>	$sp \leftarrow sp-4$, $A[sp] \leftarrow pc+2(d=0)/4(d=1)$, $pc \leftarrow imm7$	-	-	-	-	-	-	-
	<i>scalla / scalla.d</i>	$label \pm imm20$	$sp \leftarrow sp-4$, $A[sp] \leftarrow pc+2(d=0)/4(d=1)$, $pc \leftarrow label \pm imm20$	-	-	-	-	-	-	-
		<i>imm20</i>	$sp \leftarrow sp-4$, $A[sp] \leftarrow pc+2(d=0)/4(d=1)$, $pc \leftarrow imm20$	-	-	-	-	-	-	-
	<i>xcalla / xcalla.d</i>	$label \pm imm24$	$sp \leftarrow sp-4$, $A[sp] \leftarrow pc+2(d=0)/4(d=1)$, $pc \leftarrow label \pm imm24$	-	-	-	-	-	-	-
		<i>imm24</i>	$sp \leftarrow sp-4$, $A[sp] \leftarrow pc+2(d=0)/4(d=1)$, $pc \leftarrow imm24$	-	-	-	-	-	-	-
	<i>ret / ret.d</i>		$pc \leftarrow A[sp](23:0)$, $sp \leftarrow sp+4$	-	-	-	-	-	-	-
	<i>int</i>	<i>imm5</i>	$sp \leftarrow sp-4$, $A[sp] \leftarrow \{psr, pc+2\}$, $pc \leftarrow vector$ (TTBR+imm5×4)	-	0	-	-	-	-	-
	<i>intl</i>	<i>imm5, imm3</i>	$sp \leftarrow sp-4$, $A[sp] \leftarrow \{psr, pc+2\}$, $pc \leftarrow vector$ (TTBR+imm5×4), $psr(IL) \leftarrow imm3$	↔	0	-	-	-	-	-
	<i>reti / reti.d</i>		$\{psr, pc\} \leftarrow A[sp]$, $sp \leftarrow sp+4$	↔	↔	↔	↔	↔	↔	-
	<i>brk</i>		$A[DBRAM] \leftarrow \{psr, pc+2\}$, $A[DBRAM+4] \leftarrow r0$, $pc \leftarrow 0xffffc00$	-	0	-	-	-	-	-
	<i>retd</i>		$r0 \leftarrow A[DBRAM+4](23:0)$, $\{psr, pc\} \leftarrow A[DBRAM]$	↔	↔	↔	↔	↔	↔	-
Shift and swap	<i>sr</i>	<i>%rd, %rs</i>	Logical shift to right: $rd(15:0) \leftarrow rd(15:0) \gg rs(15:0)$, $rd(23:16) \leftarrow 0$, zero enters to MSB (*1)	-	-	↔	-	↔	↔	○
		<i>%rd, imm7</i>	Logical shift to right: $rd(15:0) \leftarrow rd(15:0) \gg imm7$, $rd(23:16) \leftarrow 0$, zero enters to MSB (*1)	-	-	↔	-	↔	↔	○
	<i>sa</i>	<i>%rd, %rs</i>	Arithmetical shift to right: $rd(15:0) \leftarrow rd(15:0) \gg rs(15:0)$, $rd(23:16) \leftarrow 0$, sign copied to MSB (*1)	-	-	↔	-	↔	↔	○
		<i>%rd, imm7</i>	Arithmetical shift to right: $rd(15:0) \leftarrow rd(15:0) \gg imm7$, $rd(23:16) \leftarrow 0$, sign copied to MSB (*1)	-	-	↔	-	↔	↔	○
	<i>sl</i>	<i>%rd, %rs</i>	Logical shift to left: $rd(15:0) \leftarrow rd(15:0) \ll rs(15:0)$, $rd(23:16) \leftarrow 0$, zero enters to LSB (*1)	-	-	↔	-	↔	↔	○
		<i>%rd, imm7</i>	Logical shift to left: $rd(15:0) \leftarrow rd(15:0) \ll imm7$, $rd(23:16) \leftarrow 0$, zero enters to LSB (*1)	-	-	↔	-	↔	↔	○
	<i>swap</i>	<i>%rd, %rs</i>	$rd(15:8) \leftarrow rs(7:0)$, $rd(7:0) \leftarrow rs(15:8)$, $rd(23:16) \leftarrow 0$	-	-	-	-	-	-	○

Remarks

*1) Number of bits to be shifted: Zero to three bits when $rs/imm7 = 0-3$, four bits when $rs/imm7 = 4-7$, eight bits when $rs/imm7 \geq 8$

Instruction List (10)

Assembly Programming

Classification	Mnemonic		Function	Flags						D
	Opcode	Operand		IL	IE	C	V	Z	N	
Conversion	cv.ab	%rd, %rs	rd(23:8)←rs(7), rd(7:0)←rs(7:0)	–	–	–	–	–	–	○
	cv.as	%rd, %rs	rd(23:16)←rs(15), rd(15:0)←rs(15:0)	–	–	–	–	–	–	○
	cv.al	%rd, %rs	rd(23:16)←rs(7:0), rd(15:0)←rd(15:0)	–	–	–	–	–	–	○
	cv.la	%rd, %rs	rd(23:8)←0, rd(7:0)←rs(23:16)	–	–	–	–	–	–	○
	cv.ls	%rd, %rs	rd(23:16)←0, rd(15:0)←rs(15)	–	–	–	–	–	–	○
Imm extension	ext	imm13	Extends the immediate or operand of the following instruction.	–	–	–	–	–	–	–
System control	nop		No operation	–	–	–	–	–	–	○
	halt		HALT mode	–	–	–	–	–	–	–
	slp		SLEEP mode	–	–	–	–	–	–	–
	ei		psr(IE)←1	–	1	–	–	–	–	○
	di		psr(IE)←0	–	0	–	–	–	–	○
Coprorocessor	ld.cw	%rd, %rs	co_dout0←rd, co_dout1←rs	–	–	–	–	–	–	○
		%rd, imm7	co_dout0←rd, co_dout1←imm7	–	–	–	–	–	–	○
	sld.cw	%rd, imm20	co_dout0←rd, co_dout1←imm20	–	–	–	–	–	–	–
		%rd, symbol±imm20	co_dout0←rd, co_dout1←symbol±imm20	–	–	–	–	–	–	–
	xld.cw	%rd, imm24	co_dout0←rd, co_dout1←imm24	–	–	–	–	–	–	–
		%rd, symbol±imm24	co_dout0←rd, co_dout1←symbol±imm24	–	–	–	–	–	–	–
	ld.ca	%rd, %rs	co_dout0←rd, co_dout1←rs, rd←co_din, psr(C, V, Z, N)←co_cvzn	–	–	↔	↔	↔	↔	○
		%rd, imm7	co_dout0←rd, co_dout1←imm7, rd←co_din, psr(C, V, Z, N)←co_cvzn	–	–	↔	↔	↔	↔	○
	sld.ca	%rd, imm20	co_dout0←rd, co_dout1←imm20, rd←co_din, psr(C, V, Z, N)←co_cvzn	–	–	↔	↔	↔	↔	–
		%rd, symbol±imm20	co_dout0←rd, co_dout1←symbol±imm20, rd←co_din, psr(C, V, Z, N)←co_cvzn	–	–	↔	↔	↔	↔	–
	xld.ca	%rd, imm24	co_dout0←rd, co_dout1←imm24, rd←co_din, psr(C, V, Z, N)←co_cvzn	–	–	↔	↔	↔	↔	–
		%rd, symbol±imm24	co_dout0←rd, co_dout1←symbol±imm24, rd←co_din, psr(C, V, Z, N)←co_cvzn	–	–	↔	↔	↔	↔	–
	ld.cf	%rd, %rs	co_dout0←rd, co_dout1←rs, psr(C, V, Z, N)←co_cvzn	–	–	↔	↔	↔	↔	○
		%rd, imm7	co_dout0←rd, co_dout1←imm7, psr(C, V, Z, N)←co_cvzn	–	–	↔	↔	↔	↔	○
	sld.cf	%rd, imm20	co_dout0←rd, co_dout1←imm20, psr(C, V, Z, N)←co_cvzn	–	–	↔	↔	↔	↔	–
		%rd, symbol±imm20	co_dout0←rd, co_dout1←symbol±imm20, psr(C, V, Z, N)←co_cvzn	–	–	↔	↔	↔	↔	–
	xld.cf	%rd, imm24	co_dout0←rd, co_dout1←imm24, psr(C, V, Z, N)←co_cvzn	–	–	↔	↔	↔	↔	–
		%rd, symbol±imm24	co_dout0←rd, co_dout1←symbol±imm24, psr(C, V, Z, N)←co_cvzn	–	–	↔	↔	↔	↔	–

Remarks

Extended instruction		Expansion format		
Opcode	Operand	Condition 1	Condition 2	Condition 3
sld.b sld.ub sld sld.a	%rd, [%sp+imm20] Example) sld.b %rd, [%sp+imm20]	imm20≤0x7f	0x7f<imm20	—
		ld.b %rd, [%sp+imm20(6:0)] Example) sld.b %rd, [%sp+imm20(6:0)]	ext imm20(19:7) ld.b %rd, [%sp+imm20(6:0)]	
	%rd, [imm20] Example) sld %rd, [imm20]	imm20≤0x7f	0x7f<imm20	—
		ld %rd, [imm20(6:0)] Example) sld %rd, [imm20(6:0)]	ext imm20(19:7) ld %rd, [imm20(6:0)]	
sld.b sld sld.a	[%sp+imm20], %rs Example) sld.b [%sp+imm20], %rs	imm20≤0x7f	0x7f<imm20	—
		ld.b [%sp+imm20(6:0)], %rs Example) sld.b [%sp+imm20(6:0)], %rs	ext imm20(19:7) ld.b [%sp+imm20(6:0)], %rs	
	[imm20], %rs Example) sld [imm20], %rs	imm20≤0x7f	0x7f<imm20	—
		ld [imm20(6:0)], %rs Example) sld [imm20(6:0)], %rs	ext imm20(19:7) ld [imm20(6:0)], %rs	
sld	%rd, imm16 Example) sld %rd, imm16	imm16≤0x7f	0x7f<imm16	—
		ld %rd, imm16(6:0) Example) sld %rd, imm16(6:0)	ext imm16(15:7) ld %rd, imm16(6:0)	
	%rd, symbol±imm16 Example) sld %rd, symbol+imm16	Unconditional	—	—
		ext (symbol+imm16)(15:7) ld %rd, (symbol+imm16)(6:0)		
sld.a	%rd, imm20 Example) sld.a %rd, imm20	imm20≤0x7f	0x7f<imm20	—
		ld.a %rd, imm20(6:0) Example) sld.a %rd, imm20(6:0)	ext imm20(19:7) ld.a %rd, imm20(6:0)	
	%sp, imm20 Example) sld.a %sp, imm20	imm20≤0x7f	0x7f<imm20	—
		ld.a %sp, imm20(6:0) Example) sld.a %sp, imm20(6:0)	ext imm20(19:7) ld.a %sp, imm20(6:0)	
Remarks				

Extended instruction		Expansion format		
Opcode	Operand	Condition 1	Condition 2	Condition 3
sld.a	%rd, symbol±imm20 Example) sld.a %rd, symbol+imm20	Unconditional	—	—
		ext (symbol+imm20)(19:7) ld.a %rd, (symbol+imm20)(6:0)		
	%sp, symbol±imm20 Example) sld.a %sp, symbol-imm20	Unconditional	—	—
		ext (symbol-imm20)(19:7) ld.a %sp, (symbol-imm20)(6:0)		
xld.b xld.ub xld xld.a	%rd, [%sp+imm24] Example) xld.b %rd, [%sp+imm24]	imm24≤0x7f	0x7f<imm24≤0xffff	0xffff<imm24
		ld.b %rd, [%sp+imm24(6:0)]	ext imm24(19:7) ld.b %rd, [%sp+imm24(6:0)]	ext imm24(23:20) ext imm24(19:7) ld.b %rd, [%sp+imm24(6:0)]
	%rd, [imm24] Example) xld %rd, [imm24]	imm24≤0x7f	0x7f<imm24≤0xffff	0xffff<imm24
		ld %rd, [imm24(6:0)]	ext imm24(19:7) ld %rd, [imm24(6:0)]	ext imm24(23:20) ext imm24(19:7) ld %rd, [imm24(6:0)]
xld.b xld xld.a	[%sp+imm24], %rs Example) xld.b [%sp+imm24], %rs	imm24≤0x7f	0x7f<imm24≤0xffff	0xffff<imm24
		ld.b [%sp+imm24(6:0)], %rs	ext imm24(19:7) ld.b [%sp+imm24(6:0)], %rs	ext imm24(23:20) ext imm24(19:7) ld.b [%sp+imm24(6:0)], %rs
	[imm24], %rs Example) xld [imm24], %rs	imm24≤0x7f	0x7f<imm24≤0xffff	0xffff<imm24
		ld [imm24(6:0)], %rs	ext imm24(19:7) ld [imm24(6:0)], %rs	ext imm24(23:20) ext imm24(19:7) ld [imm24(6:0)], %rs
xld	%rd, imm 16 Example) xld %rd, imm16	imm16≤0x7f	0x7f<imm16	—
		ld %rd, imm16(6:0)	ext imm16(15:7) ld %rd, imm16(6:0)	
	%rd, symbol±imm16 Example) xld %rd, symbol+imm16	Unconditional	—	—
		ext (symbol+imm16)(15:7) ld %rd, (symbol+imm16)(6:0)		

Remarks

Extended instruction		Expansion format		
Opcode	Operand	Condition 1	Condition 2	Condition 3
xld.a	%rd, imm24 Example) xld.a %rd, imm24	imm24≤0x7f	0x7f<imm24≤0xffff	0xffff<imm24
		ld.a %rd, imm24(6:0)	ext imm24(19:7) ld.a %rd, imm24(6:0)	ext imm24(23:20) ext imm24(19:7) ld.a %rd, imm24(6:0)
	%sp, imm24 Example) xld.a %sp, imm24	imm24≤0x7f	0x7f<imm24≤0xffff	0xffff<imm24
		ld.a %sp, imm24(6:0)	ext imm24(19:7) ld.a %sp, imm24(6:0)	ext imm24(23:20) ext imm24(19:7) ld.a %sp, imm24(6:0)
	%rd, symbol±imm24 Example) xld.a %rd, symbol+imm24	Unconditional	—	—
		ext (symbol+imm24)(23:20) ext (symbol+imm24)(19:7) ld.a %rd, (symbol+imm24)(6:0)		
	%sp, symbol±imm24 Example) xld.a %sp, symbol-imm24	Unconditional	—	—
		ext (symbol-imm24)(23:20) ext (symbol-imm24)(19:7) ld.a %sp, (symbol-imm24)(6:0)		
sadd sadc ssub ssbc	%rd, imm16 Example) sadd %rd, imm16	imm16≤0x7f	0x7f<imm16	—
		add %rd, imm16(6:0)	ext imm16(15:7) add %rd, imm16(6:0)	
sadd.a ssub.a	%rd, imm20 Example) ssub.a %rd, imm20	imm20≤0x7f	0x7f<imm20	—
		sub.a %rd, imm20(6:0)	ext imm20(19:7) sub.a %rd, imm20(6:0)	
	%sp, imm20 Example) sadd.a %sp, imm20	imm20≤0x7f	0x7f<imm20	—
		add.a %sp, imm20(6:0)	ext imm20(19:7) add.a %sp, imm20(6:0)	
xadd xadc xsub xsbc	%rd, imm16 Example) xadc %rd, imm16	imm16≤0x7f	0x7f<imm16	—
		adc %rd, imm16(6:0)	ext imm16(15:7) adc %rd, imm16(6:0)	

Remarks

Extended instruction		Expansion format		
Opcode	Operand	Condition 1	Condition 2	Condition 3
xadd.a xsub.a	%rd, imm24 Example) xsub.a %rd, imm24	imm24≤0x7f	0x7f<imm24≤0xffff	0xffff<imm24
		sub.a %rd, imm24(6:0)	ext imm24(19:7) sub.a %rd, imm24(6:0)	ext imm24(23:20) ext imm24(19:7) sub.a %rd, imm24(6:0)
	%sp, imm24 Example) xadd.a %sp, imm24	imm24≤0x7f	0x7f<imm24≤0xffff	0xffff<imm24
		add.a %sp, imm24(6:0)	ext imm24(19:7) add.a %sp, imm24(6:0)	ext imm24(23:20) ext imm24(19:7) add.a %sp, imm24(6:0)
scmp scmc	%rd, imm16 Example) scmp %rd, imm16	imm16≤0x7f	0x7f<imm16	—
		cmp %rd, imm16(6:0)	ext imm16(15:7) cmp %rd, imm16(6:0)	
scmp.a	%rd, imm20 Example) scmp.a %rd, imm20	imm20≤0x7f	0x7f<imm20	—
		cmp.a %rd, imm20(6:0)	ext imm20(19:7) cmp.a %rd, imm20(6:0)	
xcmp xcmc	%rd, imm16 Example) xcmc %rd, imm16	imm16≤0x7f	0x7f<imm16	—
		cmc %rd, imm16(6:0)	ext imm16(15:7) cmc %rd, imm16(6:0)	
xcmp.a	%rd, imm24 Example) xcmp.a %rd, imm24	imm24≤0x7f	0x7f<imm24≤0xffff	0xffff<imm24
		cmp.a %rd, imm24(6:0)	ext imm24(19:7) cmp.a %rd, imm24(6:0)	ext imm24(23:20) ext imm24(19:7) cmp.a %rd, imm24(6:0)
sand soor sxor snot	%rd, imm16 Example) sand %rd, imm16	imm16≤0x7f	0x7f<imm16	—
		and %rd, imm16(6:0)	ext imm16(15:7) and %rd, imm16(6:0)	
xand xoor xxor xnot	%rd, imm16 Example) xoor %rd, imm16	imm16≤0x7f	0x7f<imm16	—
		or %rd, imm16(6:0)	ext imm16(15:7) or %rd, imm16(6:0)	

Remarks

Extended instruction		Expansion format		
Opcode	Operand	Condition 1	Condition 2	Condition 3
scall scall.d sjpr sjpr.d	label±imm20 Example) scall label+imm20	Unconditional	—	—
		ext (label+imm20)(19:12) call (label+imm20)(11:1)		
	sign20 Example) sjpr sign20	-1024≤sign20≤1023	sign20<-1024 or 1023<sign20	—
		jpr sign20(11:1)	ext sign20(19:12) jpr sign20(11:1)	
sjr*1 sjr*1.d	label±imm20 Example) sjreq label+imm20	Unconditional	—	—
		ext (labe+imm20)(19:8) jreq (label+imm20)(7:1)		
	sign20 Example) sjrne sign20	-128≤sign20≤127	sign20<-128 or 127<sign20	—
		jrne sign20(7:1)	ext sign20(19:8) jrne sign20(7:1)	
scalla scalla.d sjpa sjpa.d	label±imm20 Example) scalla label+imm20	Unconditional	—	—
		ext (label+imm20)(19:7) calla (label+imm20)(6:0)		
	imm20 Example) sjpa imm20	imm20≤0x7f	0x7f<imm20	—
		jpa imm20(6:0)	ext imm20(19:7) jpa imm20(6:0)	
xcall xcall.d xjpr xjpr.d	label±imm24 Example) xcall label+imm24	Unconditional	—	—
		ext (label+imm24)(23:12) call (label+imm24)(11:1)		
	sign24 Example) xjpr sign24	-1024≤sign24≤1023	sign24<-1024 or 1023<sign24	—
		jpr sign24(11:1)	ext sign24(23:12) jpr sign24(11:1)	

Remarks

*1) sjreq, sjreq.d, sjrne, sjrne.d, sjrgt, sjrgt.d, sjrge, sjrge.d, sjrlt, sjrlt.d, sjrle, sjrle.d, sjrugt, sjrugt.d, sjruge, sjruge.d, sjrult, sjrult.d, sjrule, sjrule.d

Extended instruction		Expansion format		
Opcode	Operand	Condition 1	Condition 2	Condition 3
xjr*1 xjr*1.d	label±imm24 Example) xjreq label+imm24	Unconditional	—	—
		ext (label+imm24)(23:21) ext (label+imm24)(20:8) jreq (label+imm24)(7:1)		
	sign24 Example) xjrne sign24	-128≤sign24≤127	-1048576≤sign24<-128 or 127<sign24≤1048575	sign24<-1048576 or 1048575<sign24
		jrne sign24(7:1)	ext sign24(20:8) jrne sign24(7:1)	ext sign24(23:21) ext sign24(20:8) jrne sign24(7:1)
xcalla xcalla.d xjpa xjpa.d	label±imm24 Example) xcalla label+imm24	Unconditional	—	—
		ext (label+imm24)(23:20) ext (label+imm24)(19:7) calla (label+imm24)(6:0)		
	imm24 Example) xjpa imm24	imm24≤0x7f	0x7f<imm24≤0xffff	0xffff<imm24
		jpa imm24(6:0)	ext imm24(19:7) jpa imm24(6:0)	ext imm24(23:20) ext imm24(19:7) jpa imm24(6:0)
sld.cw sld.ca sld.cf	%rd, imm20 Example) sld.cw %rd, imm20	imm20≤0x7f	0x7f<imm20	—
		ld.cw %rd, imm20(6:0)	ext imm20(19:7) ld.cw %rd, imm20(6:0)	
	%rd, symbol±imm20 Example) sld.ca %rd, symbol+imm20	Unconditional	—	—
		ext (symbol+imm20)(19:7) ld.ca %rd, (symbol+imm20)(6:0)		
xld.cw xld.ca xld.cf	%rd, imm24 Example) xld.cw %rd, imm24	imm24≤0x7f	0x7f<imm24≤0xffff	0xffff<imm24
		ld.cw %rd, imm24(6:0)	ext imm24(19:7) ld.cw %rd, imm24(6:0)	ext imm24(23:20) ext imm24(19:7) ld.cw %rd, imm24(6:0)
	%rd, symbol±imm24 Example) xld.ca %rd, symbol+imm24	Unconditional	—	—
		ext (symbol+imm24)(23:20) ext (symbol+imm24)(19:7) ld.ca %rd, (symbol+imm24)(6:0)		

Remarks

*1) xjreq, xjreq.d, xjrne, xjrne.d, xjrgt, xjrgt.d, xjrge, xjrge.d, xjrlt, xjrld.d, xjrle, xjrle.d, xjrugt, xjrugt.d, xjruge, xjruge.d, xjrult, xjrult.d, xjrle, xjrle.d

America

Epson America, Inc.

Headquarter:
3840 Kilroy Airport Way
Long Beach, California 90806-2452 USA
Phone: +1-562-290-4677

San Jose Office:
214 Devcon Drive
San Jose, CA 95112 USA
Phone: +1-800-228-3964 or +1-408-922-0200

Europe

Epson Europe Electronics GmbH

Riesstrasse 15, 80992 Munich,
Germany
Phone: +49-89-14005-0 FAX: +49-89-14005-110

Asia

Epson (China) Co., Ltd.

4th Floor, Tower 1 of China Central Place, 81 Jianguo Road, Chaoyang District, Beijing 100025 China
Phone: +86-10-8522-1199 FAX: +86-10-8522-1120

Shanghai Branch

Room 1701 & 1704, 17 Floor, Greenland Center II,
562 Dong An Road, Xu Hui District, Shanghai, China
Phone: +86-21-5330-4888 FAX: +86-21-5423-4677

Shenzhen Branch

Room 804-805, 8 Floor, Tower 2, Ali Center, No.3331
Keyuan South RD (Shenzhen bay), Nanshan District, Shenzhen 518054, China
Phone: +86-10-3299-0588 FAX: +86-10-3299-0560

Epson Taiwan Technology & Trading Ltd.

15F, No.100, Songren Rd, Sinyi Dist, Taipei City 110, Taiwan
Phone: +886-2-8786-6688

Epson Singapore Pte., Ltd.

1 HarbourFront Place,
#03-02 HarbourFront Tower One, Singapore 098633
Phone: +65-6586-5500 FAX: +65-6271-3182

Seiko Epson Corp.

Korea Office

19F, KLI 63 Bldg, 60 Yoido-dong,
Youngdeungpo-Ku, Seoul 150-763, Korea
Phone: +82-2-784-6027 FAX: +82-2-767-3677

Seiko Epson Corp.

Sales & Marketing Division

Device Sales & Marketing Department

421-8, Hino, Hino-shi, Tokyo 191-8501, Japan
Phone: +81-42-587-5816 FAX: +81-42-587-5116