EPSON

CMOS 32-BIT SINGLE CHIP MICROCOMPUTER

# S1C33 Family Application Note for Standard Core

(S5U1C33001C)

SEIKO EPSON CORPORATION

## Configuration of product number

### Devices

**S1 C 33209 F 00E1 00**

**Packing specifications**

|  |  |  |
|---|---|---|
| 00 | : Besides tape & reel |  |
| 0A | : TCP BL | 2 directions |
| 0B | : Tape & reel | BACK |
| 0C | : TCP BR | 2 directions |
| 0D | : TCP BT | 2 directions |
| 0E | : TCP BD | 2 directions |
| 0F | : Tape & reel | FRONT |
| 0G | : TCP BT | 4 directions |
| 0H | : TCP BD | 4 directions |
| 0J | : TCP SL | 2 directions |
| 0K | : TCP SR | 2 directions |
| 0L | : Tape & reel | LEFT |
| 0M | : TCP ST | 2 directions |
| 0N | : TCP SD | 2 directions |
| 0P | : TCP ST | 4 directions |
| 0Q | : TCP SD | 4 directions |
| 0R | : Tape & reel | RIGHT |
| 99 | : Specs not fixed |  |

**Specification**

**Package**
[D: die form; F: QFP]

**Model number**

**Model name**
[C: microcomputer, digital products]

**Product classification**
[S1: semiconductor]

### Development tools

**S5U1 C 33000 H2 1 00**

**Packing specifications**
[00: standard packing]

**Version**
[1: Version 1]

**Tool type**

|  |  |
|---|---|
| Hx | : ICE |
| Dx | : Evaluation board |
| Ex | : ROM emulation board |
| Mx | : Emulation memory for external ROM |
| Tx | : A socket for mounting |
| Cx | : Compiler package |
| Sx | : Middleware package |

**Corresponding model number**
[33L01: for S1C33L01]

**Tool classification**
[C: microcomputer use]

**Product classification**
[S5U1: development tool for semiconductor products]

## *PREFACE*

Written for developers of application systems incorporating the S1C33 Family of microcomputers, this manual explains how to write a program, design basic circuitry, and produce audio output using the S1C33 chips, particularly the S1C33209. The sample code provided in this manual is excerpted from S1C33 Family C Compiler Package (S5U1C33001C) Ver. 1 or later.

## *CONTENTS*

# 1 ABOUT THE S1C33000 CPU CORE

The S1C33000 is the CPU core shared by all chips in the S1C33 Family of 32-bit CMOS single-chip micro-computers. Arranged around this core are various peripheral components, such as ROM, RAM, DMA, A / D converters, and timers, which together make up the Seiko Epson line of S1C33 Family processors.

The main features of the S1C33000 are as follows.

- A highly code-efficient instruction set
- Fast operation and multiplier/accumulator function
- Small CPU core size
- Low current consumption

The S1C33000 supports a wide range of built-in applications, from portable to OA and FA equipment, and from digital signal processors to various types of controllers.

## 1.1 Outline

| | |
|---|---|
| • Type | Seiko Epson original 32-bit RISC core |
| • Operating frequency | DC to 60 MHz (varies with the type of S1C33xxx) |
| • Instruction set | 16-bit fixed length<br>105 discrete instructions<br>Main instructions can be executed in one cycle. |
| • Multiplier / accumulator function | MAC instruction (16 bits $\times$ 16 bits + 64 bits $\rightarrow$ 64 bits)<br>Executed in 2 cycles per MAC operation |
| • Register set | 32-bit general-purpose register $\times$ 16<br>32-bit special register $\times$ 5 |
| • Memory space | 28-bit (256 MB) space<br>Instruction, data, and I / O mixed type linear space<br>Divided into 19 areas, for which the select signal is output by the core |
| • Immediate extension | Immediate data of instructions are extended to 32 bits by EXT instruction. |
| • Interrupt | Reset, NMI, and external interrupt $\times$ 216 sources<br>Software exception $\times$ 4 sources, 2 types of instruction execution exception<br>Vectors are fetched from trap table when branching to the jump address. |
| • Reset | Cold reset (all internal circuits reset)<br>Hot reset (buses not reset)<br>Trap table can be selected between internal or external ROM at boot time and can then be relocated. |
| • Power-down mode | HALT instruction (only the core halted)<br>SLP instruction (all internal circuits halted) |
| • Other | Little endian (standard) / big endian<br>Harvard architecture |

## 1.2 Memory Map

| Address | Area | | Area size |
|---|---|---|---|
| 0xFFFFFFF | Area 18 | External memory | 64MB |
| | Area 17 | External memory | 64MB |
| | Area 16 | External memory | 32MB |
| | Area 15 | External memory | 32MB |
| | Area 14 | External memory | 16MB |
| | Area 13 | External memory | 16MB |
| | Area 12 | External memory | 8MB |
| 0x1000000 | Area 11 | External memory | 8MB |
| 0x0C00000 | Area 10 | External memory | 4MB |
| | Area 9 | External memory | 4MB |
| | Area 8 | External memory | 2MB |
| | Area 7 | External memory | 2MB |
| | Area 6 | External I/O | 1MB |
| | Area 5 | External memory | 1MB |
| 0x0100000 | Area 4 | External memory | 1MB |
| 0x0080000 | Area 3 | On-chip ROM | 512KB |
| 0x0060000 | Area 2 | Reserved | 128KB |
| 0x0040000 | Area 1 | Internal I/O | 128KB |
| 0x0000000 | Area 0 | On-chip RAM | 256KB |

## 1.3 Trap Table

| | Address offset |
|---|---|
| Reserved | 1023 |
| External maskable interrupt 215<br>:<br>External maskable interrupt 0 | 929<br><br>64 |
| Software exception 3<br>:<br>Software exception 0 | 60<br><br>48 |
| Reserved | 32–44 |
| NMI | 28 |
| Address error | 24 |
| Reserved | 20 |
| Zero division | 16 |
| Reserved | 4–12 |
| Reset | 0 |

Trap table start address
At cold-reset, it is set to 0x0C00000.
The trap table can be relocated using the trap table base register TTBR (memory-mapped register) after resetting the CPU.
Vectors will be fetched from the trap table for booting and interrupts.

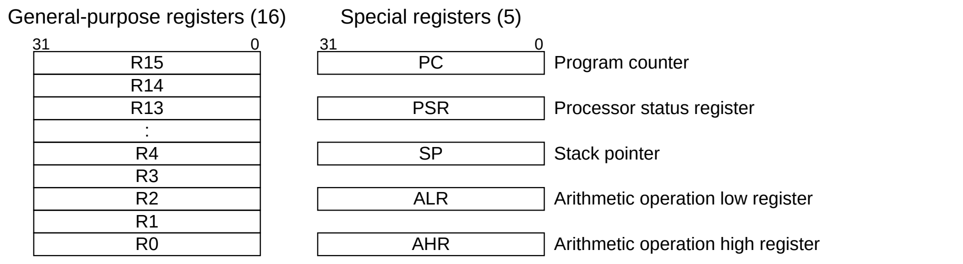
Interrupt sequence

1) The PC is saved to the stack.
2) The PSR is saved to the stack and IE is disabled.
3) The vector is fetched from the trap table.
4) Control jumps to the vector address.

Reset sequence

1) The reset vector is fetched.
2) Control jumps to the vector address.

## 1.4 CPU Registers


General-purpose registers (16)
31
0
R15
R14
R13
:
R4
R3
R2
R1
R0
Special registers (5)
31
0
PC
Program counter
PSR
Processor status register
SP
Stack pointer
ALR
Arithmetic operation low register
AHR
Arithmetic operation high register

(AHR, ALR: Option for Multiplication & Accumulation, Multiplication, and Division)

PSR

| 31–12 | 11–8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| Reserved | IL | MO | DS | – | IE | C | V | Z | N |

| | | |
|---|---|---|
| IL: | Interrupt level | (0–15: Enabled interrupt level) |
| MO: | MAC overflow flag | (1: MAC overflow, 0: Not overflown) |
| DS: | Dividend sign flag | (1: Negative, 0: Positive) |
| 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) |

## 1.5 Instruction Set Features

### ● Types of instructions

Instructions are functionally classified as one of the following eight types:

- **8, 16, or 32-bit data transfer instructions**
  **LD.B, LD.UB, LD.H, LD.UH, LD.W**
  Performs 8, 16, or 32-bit data transfers between the register and memory, or between two registers.
- **32-bit arithmetic/logic operation instructions**
  **AND, OR, XOR, NOT, ADD, ADC, SUB, SBC, CMP, MLT.H, MLTU.H (16-bit), MLT.W, MLTU.W, DIV0S, DIV1S, DIV2S, DIV3S**
  Performs 32-bit arithmetic/logic operation on two register values, or on register and immediate values.
- **32-bit shift and rotate instructions**
  **SRL, SLL, SRA, SLA, RR, RL**
  Shifts or rotates 32-bit register data by 0 to 8 bits.
- **Bit-manipulating instructions**
  **BTST, BSET, BCLR, BNOT**
  Operates on byte data in memory to set or reset bitwise.
- **Stack-manipulating instructions**
  **PUSHN, POPN**
  Saves or restores the contents of R0 to Rn successively to or from the stack.
- **Branch instructions**
  **JRGT, JRGE, JRLT, JRLE, JRUGT, JRUGE, JRULT, JRULE, JREQ, JRNE, CALL, JP, RET, RETI, RETD, INT, BRK**
  Performs various conditional jump, call, or return operations.
- **System control instructions**
  **HALT, SLP, NOP**
  Used to place the device in power-down mode or inserted to perform no operation.
- **Other instructions**
  **MAC, SCAN0, SCAN1, SWAP, MIRROR, EXT**
  Performs a MAC, data scan, or replacement operation.

## ● Addressing modes

### (1) Basic addressing modes

These addressing modes can be implemented in one instruction.

- **6-bit immediate data addressing**

| | |
|---|---|
| `LD.W  %R1,sign6` | Sign extends 6-bit data before loading it into the R1 register. |
| `ADD   %R2,imm6` | Adds 6-bit data to the R2 register. |

In this mode, the operations are performed upon 6-bit signed / unsigned immediate data and register.

- **Register direct addressing**

| | |
|---|---|
| `LD.W  %R1,%R2` | Transfers data from the R2 to the R1 register. |
| `JP    %R3` | Jumps to the address held by the R3 register. |

In this mode, operations are performed only on register values.

- **Register indirect addressing**

| | |
|---|---|
| `LD.B  %R2,[%R15]` | Loads signed 8-bit data from the address specified by R15. |
| `LD.W  %R2,[%R15]+` | Loads 32-bit data from the address specified by R15 and then increments the R15 register. |

In this mode, a memory address is set in a register and operations are performed on data at that address.

- **SP indirect addressing with displacement**

| | |
|---|---|
| `LD.UB %R15,[%SP+imm6]` | Loads unsigned 8-bit data from the address indicated by SP + imm6. |
| `LD.W  %R15,[%SP+imm6]` | Loads 32-bit data from the address indicated by SP + (imm6 × 4). |

In this mode, an offset address is specified from the stack pointer and operations performed on data within the stack.

- **Signed 8-bit PC relative addressing**

| | |
|---|---|
| `JP    sign8` | Jumps to a location up to 127 instructions ahead of or 128 instructions behind the current PC address. |
| `CALL  sign8` | Calls a subroutine located up to 127 instructions ahead of or 128 instructions behind the current PC address. |

In this mode, the jump address is specified by a relative address from the PC.

### (2) Extended addressing modes

The basic addressing modes can be extended with the EXT instruction.

- **Extended immediate data addressing**

```
EXT  imm13 + ADD  %R1,imm6                → ADD %R1,imm19
EXT  imm13 + EXT  imm13 + ADD  %R1,imm6   → ADD %R1,imm32
```

The immediate size can be extended to 19 or 32 bits with the EXT instruction.

- **Extended register indirect addressing**

```
EXT  imm13 + LD.W  %R2,[%R15]+                  → LD.W  %R2,[%R15+imm13]
EXT  imm13 + EXT  imm13 + LD.W  %R2,[%R15]+     → LD.W  %R2,[%R15+imm26]
```

A 13-bit or 26-bit offset address can be added using the EXT instruction.

- **SP indirect addressing with extended displacement**

```
EXT  imm13 + LD.B  %R15,[%SP+imm6]                  → LD.B  %R15,[%SP+imm19]
EXT  imm13 + EXT  imm13 + LD.B  %R15,[%SP+imm6]     → LD.B  %R15,[%SP+imm32]
```

The offset can be extended to 19 or 32 bits by the EXT instruction.

- **Extended PC relative addressing**

```
EXT  imm13 + CALL  sign8                  → CALL  sign22
EXT  imm13 + EXT  imm13 + CALL  sign8     → CALL  sign32
```

The address range to which to branch may be extended to 22 or 32 bits by the EXT instruction.

## ● High code density for C language

Based on the following two concepts, the S1C33 CPU core creates high code density for C language.

1. As often as possible, frequent operation patterns in C are processed by one instruction.
2. Other operation patterns are suppressed to as few instructions as possible using the EXT instruction, preventing worsening code density in less frequently used patterns.

### (1) Branch patterns

- **Conditional branch**

`JRNE sign8` (Jump area of +127 to -128 instructions)
Supports more than 90% of conditional branching cases with one instruction (2 bytes).

`EXT imm13 + JRNE sign8` → `JRNE sign22` (±2M jump area)
Supports other conditional branching with two instructions (4 bytes).

- **Subroutine call**

`EXT imm13 + CALL sign8` → `CALL sign22` (±2M jump area)
Supports almost all subroutine calls with two instructions (4 bytes).

`EXT imm13 + EXT imm13 + JRNE sign9` → `JRNE sign32` (Can jump to any area)
Supports other subroutine calls with three instructions (6 bytes).

### (2) Variable access patterns

- **Auto variable access**

`LD.W %R2,[%SP+imm6]` (Accesses SP + 0 to 255 area for int access)
Supports more than 80% of auto-variable access cases with one instruction (2 bytes).

`EXT imm13 + LD.W %R2,[%SP+imm6]` → `LD.W %R2,[%SP+imm19]` (Accesses 512K-byte area)
Supports other auto-variable access cases with two instructions (4 bytes).

- **Pointer variable access**

`LD.B %R2,[%R3]`
One instruction (2 bytes)

- **Static variable access (based on global pointer)**

`EXT imm13 + LD.H %R2,[%R15]` → `LD.H %R2,[%R15+imm13]` (Accesses 8K-byte area from R15)
Two instructions (4 bytes)

`EXT imm13 + EXT imm13 + LD.H %R2,[%R15]` → `LD.W %R2,[%R15+imm26]`
Three instructions (6 bytes)

### (3) Arithmetic patterns

- **2-operand, register to immediate**

`ADD %R2,imm6` (Adds 0–63 to R2)
One instruction (2 bytes)

`EXT imm13 + ADD %R2,imm6` → `ADD %R2,imm19` (Adds 0–512K to R2)
Two instructions (4 bytes)

`EXT imm13 + EXT imm13 + ADD %R2,imm6` → `ADD %R2,imm32`
Three instructions (6 bytes)

- **2-operand, register to register**

`ADD %R2,%R3` (Adds R3 to R2)
One instruction (2 bytes)

### (4) Other

- **Call, return**

| | |
|---|---|
| `CALL sign8` | Saves PC automatically |
| `RET` | Restores PC automatically |

  One instruction reduced for each

- **Push, pop**

| | |
|---|---|
| `PUSHN %Rn` | Saves R0–Rn to the stack |
| `POPN  %Rn` | Restores R0–Rn from the stack |

  Number of instructions reduced for each subroutine

- **Data conversion**

| | |
|---|---|
| `LD.B %R2,%R3` | Converts signed 8-bit data to 32-bit data |
| `LD.UB/LD.H/LD.UH` | Also supports signed / unsigned 8-bit and 16-bit data |

  Ideal for data cast in C

- **Bit manipulation**

| | |
|---|---|
| `BSET [%R5],2` | Sets bit 2 of [%R5] (memory data in bytes) to 1 |
| `BCLR/BTST/BNOT` | Clears, tests, or inverts a bit |

  Permits read-modify-write operation with one instruction.

## *1.6 Instruction Execution Speed*

The following shows the number of instruction cycles. Note that these apply when the program resides in internal ROM and data exists in RAM operating in the Harvard architecture. Wait cycles are added for access to external memory.

- **Register to register operation (arithmetic, logic, system, etc.)**

  AND, OR, XOR, NOT, ADD, ADC, SUB, SBC, CMP, MLT.H, MLTU.H, DIV0S, DIV1S, DIV2S, DIV3S, SRL, SLL, SRA, SLA, RR, RL, HALT, SLP, NOP, LD.B, LD.UB, LD.H, LD.UH, LD.W

  One cycle per instruction

  MLT.W, MLTU.W

  Five cycles per instruction

- **Memory to register operation (ld.w, ld.b, ld.ub, ld.h, ld.uh)**

  %RD, [%RB] (without interlock), [%RB], %RS, %RD, [%SP+imm6], [%SP+imm6], %RS, [%RB]+, %RS

  One cycle per instruction

  %RD, [%RB]+, %RD, [%RB] (with interlock)

  Two cycles per instruction

- **Memory to memory operation**

  BTST, BSET, BCLR, BNOT

  Three cycles per instruction

- **Branch operation**

  JRGT, JRGE, JRLT, JRLE, JRUGT, JRUGE, JRULT, JRULE, JREQ, JRNE, JP

  Ordinary branching: Two cycles per instruction; delayed jump (xxx.d): One cycle per instruction

  CALL, JP, RET, RETI, RETD, INT, BRK

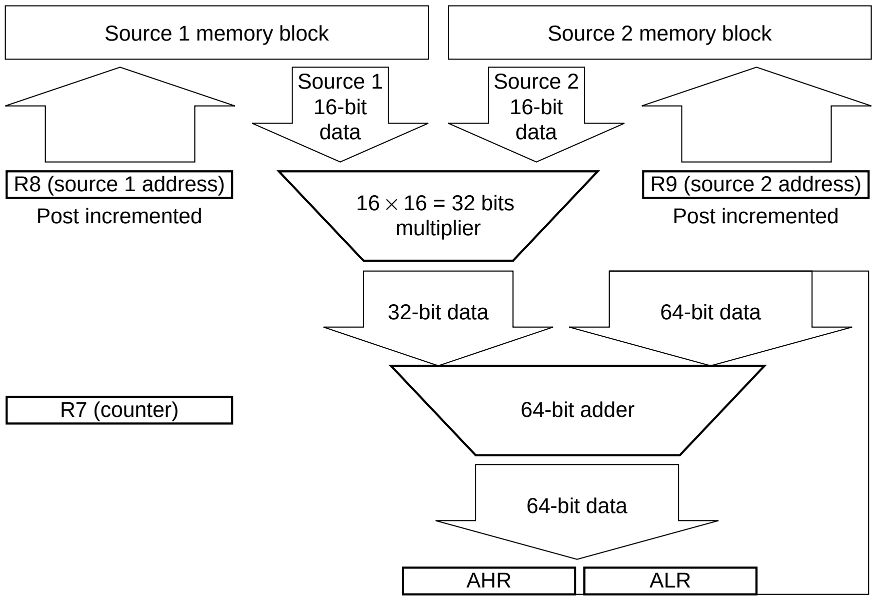
  Two to 10 cycles per instruction

- **Other operations**

| | |
|---|---|
| MAC | $2 \times N + 4$ cycles |
| PUSHN, POPN | $1 \times N$ cycles |
| SCAN0, SCAN1, SWAP, MIRROR | One cycle per instruction |

## 1.7 Multiplier/Accumulator Functions

The MAC instruction is capable of executing a 16 bits × 16 bits + 64 bits sum-of-products operation in one instruction every 2 clock periods, up to 2 G times.



Example: **MAC %R7**
R7: Repetition counter (maximum 4 G)
R8: Source 1 address (post incremented)
R9: Source 2 address (post incremented)

The source 1 and source 2 16-bit data are read from each memory location and multiplied. The 32-bit data resulting from the multiplication is added to a 64-bit register consisting of AHR:ALR. This is repeated once every 2 clock periods (given that source 1 and source 2 both exist in the internal RAM).

## 1.8 Instruction Set List

### ● Instruction format and operation

(The number of execution cycles applies here when the internal RAM is accessed for data with instructions residing in internal ROM.)

| Classification | Instruction | Typical instruction format | Operation | Number of cycles |
|---|---|---|---|---|
| Relative branch | jp, jrgt, jrge, jrlt, jrle, jrugt, jruge, jrult, jrule, jreq, jrne, call | jp sing8 | Branch to PC + (sign8 × 2) | 1,2(when branching)<br>3 for call |
| Relative delayed branch | jp.d, jrgt.d, jrge.d, jrlt.d, jrle.d, jrugt.d, jruge.d, jrult.d, jrule.d, jreq.d, jrne.d, call.d | jp.d sing8 | Branch to PC + (sign8 × 2)<br>Execute next instruction upon branching | 1<br>2 for call |
| Absolute branch | call, jp, call.d, jp.d | call %rb | Branch to address indicated by %rb | 1–3 |
| Special branch | ret, ret.d, int imm2, reti, brk, retd | | Return, interrupt, etc. | 3–10 |
| Logic operation | and, or, xor, not | and %rd, %rs<br>and %rd, sign6 | %rd = %rd & %rs<br>%rd = %rd & sign6 | 1 |
| Arithmetic operation | add, sub | add %rd, %rs<br>add %rd, imm6<br>add %sp, imm12 | %rd = %rd + %rs<br>%rd = %rd + imm6<br>%sp = %sp + imm12 | 1 |
| Compare operation | cmp | cmp %rd, %rs<br>cmp %rd, sign6 | %rd - %rs, flag only changes<br>%rd - sign6 | 1 |
| Carry operation | adc, sbc | adc %rd, %rs | %rd = %rd + %rs + carry flag | 1 |
| Multiplication | mlt.h, mlt.uh (16bit)<br>mlt.w, mlt.uw (32bit) | mlt.h %rd, %rs | %alr = %rd × %rs (32 = 16 × 16)<br>%ahr:%alr = %rd × %rs (64 = 32 × 32) | 1<br>5 |
| Division | div0s, div0u, div1, div2s, div3s | | Execute division using these in combination | 1 |
| Shift | srl, sll (logical shift)<br>sra, sla (arithmetical shift)<br>rr, rl (rotate) | srl %rd, imm4<br>srl %rd, %rs | %rd = %rd >> imm4<br>%rd = %rd >> %rs<br>Shift by 0 to 8 bits | 1 |
| Memory load | ld.b (signed 8bit load)<br>ld.ub (unsigned 8bit load)<br>ld.h (signed 16bit load)<br>ld.uh (unsigned 16bit load)<br>ld.w (32bit load) | ld.w %rd, [%sp+imm6]<br>ld.w [%sp+imm6], %rs<br>ld.w %rd, [%rb]<br>ld.w %rd, [%rb]+<br>ld.w [%rb], %rs<br>ld.w [%rb]+, %rs | %rd = [%sp+imm6], stack relative access<br>[%sp+imm6] = %rs<br>%rd = [%rb], register address access<br>%rd = [%rb], %rb = %rb + 4, post inc.<br>[%rb] = %rs<br>[%rb] = %rs, %rb = %rb + 4 | 1–2 |
| Register load | ld.w | ld.w %rd, %rs<br>ld.w %rd, sign6<br>ld.w %rd, %ss<br>ld.w %ss, %rs | Copy between registers<br>Store immediate value<br>Copy from special register<br>Copy to special register | 1 |
| Conversion | ld.b, ld.ub, ld.h, ld.uh | ld.b %rd, %rs | Convert types | 1 |
| Bit operation | btst, bset, bclr, bnot | btst [%rb], imm3 | Test, set, clear, or invert a bit | 3 |
| System | nop, slp, hlt | | No operation, stock clock | 1 |
| Mac operation | mac | | Repeat %ahr:%alr= [%r14] × [%r15] + %ahr:%alr %r13 times | 2 × N + 4 |
| Stack operation | pushn, popn | pushn %rs | Successively push/pop from %r0 to %rs | 1 × N |
| Scan | scan0, scan1 | scan0 %rd, %rs | Scan 1 or 0 from MSB, up to 8 bits | 1 |
| Swap | swap, miror | swap %rd, %rs | Swap or mirror bits bytewise | 1 |
| Extention | ext | ext imm13 | Extend immediate data of instruction | 1 |

signX, immX: immediate value, %XX: register

### ● Immediate extension by EXT instruction

Example:

| Instruction only | One EXT instruction is added | Two EXT instructions are added |
|---|---|---|
| call sign8 | ext imm13<br>call sign8 (= call sign22) | ext imm13<br>ext imm13<br>call sign8 (= call sign32) |

| Classification | Instruction | Typical format for 1 instruction | Typical operation when 1 EXT instruction is added | Typical operation when 2 EXT instructions are added |
|---|---|---|---|---|
| Relative branch | jp, jrgt, jrge, jrlt, jrle, jrugt, jruge, jrult, jrule, jreq, jrne, call, and delayed branch inst. | jp sing8 | jp sign22 | jp sign32 |
| Operation | add, sub, and, or, xor, not, cmp, ld.w | add %rd, imm6 /sign6 | add %rd, imm19/sign19 | add %rd, imm32 |
| Stack load | ld.b, ld.ub, ld.h, ld.uh, ld.w | ld.w %rd, [%sp+imm6]<br>ld.w [%sp+imm6], %rs | [%sp+imm19]<br>Extend offset value | [%sp+imm32]<br>Extend offset value |
| Absolute load | ld.b, ld.ub, ld.h, ld.uh, ld.w | ld.w %rd, [%rb]<br>ld.w %rd, [%rb]+<br>ld.w [%rb], %rs<br>ld.w [%rb]+, %rs | [%rb+imm13]<br>Add offset value | [%rb+imm26]<br>Add offset value |
| Bit operation | btst, bset, bclr, bnot | btst [%rb], imm3 | [%rb+imm13]<br>Add offset value | [%rb+imm26]<br>Add offset value |

signX, immX: immediate value, %XX: register

# 2 WRITING PROGRAMS FOR THE S1C33

This chapter explains how to write programs for the S1C33. The method described here applies to all microcomputers in the S1C33 Family.

## 2.1 Vector Table and Boot Routine

The S1C33 program must have at least a vector table and a boot routine. When cold reset at power-on, the S1C33 chip normally fetches the reset vector from address 0xC00000 and begins executing a program from that address. The simplest assembler resembles the one show below.

```
      .set SP_INI,0x0800         ; sp is in end of 2KB internal RAM
      .set DP_INI,0x0000         ; default data area pointer %r15 is 0x0

      .text
      .long BOOT                 ; BOOT VECTOR
BOOT:
      xld.w    %r15,SP_INI
      ld.w     %sp,%r15          ; set SP
      ld.w     %r15,DP_INI       ; set default data area pointer
      xcall    main              ; goto main
      xjp      BOOT              ; infinity loop
```

In addition, the actual application may require a vector table for exceptions and interrupts, and a boot routine that includes processing required to set up the BCU and initialize peripheral functions. Code examples, one in assembler and one in C, are provided below.

### ● Code example in assembler

The following shows an example of vector table:

Vector table [vector.s]

```
      .text

      .long    RESET             ; Vector table
      .long    RESERVED
      .long    RESERVED
      .long    RESERVED
      .long    ZERODIV
      .long    RESERVED
      .long    ADDRERR
      .long    NMI
      .long    RESERVED
      .long    RESERVED
      .long    RESERVED
      .long    RESERVED
      .long    SOFTINT0
      .long    SOFTINT1
      .long    SOFTINT2
      .long    SOFTINT3
      .long    INT0
      .long    INT1
      .long    INT2
      .long    INT3
      .long    INT4
      .long    INT5
                 |
            (INT6–INT49)
                 |
      .long    INT50
      .long    INT51
      .long    INT52
      .long    INT53
      .long    INT54
      .long    INT55

RESET:                           ; Dummy label for undefined vector
ZERODIV:
```

```
ADDRERR:
NMI:
RESERVED:
SOFTINT0:
SOFTINT1:
SOFTINT2:
SOFTINT3:
INT0:
INT1:
INT2:
INT3:
INT4:
INT5:
  |
(INT6–INT49)
  |
INT50:
INT51:
INT52:
INT53:
INT54:
INT55:
      .global  INT_LOOP
INT_LOOP:                        ; Trap routine for undefined vector
      nop
      jp       INT_LOOP
      reti
```

In this file, the vector table for boot to hardware interrupts is defined in the format

**.long label**

This allows storage of 32-bit jump addresses in the vector table. For safety, addresses that are not specifically defined are vectored to INT_LOOP at the bottom of the file. Note that the program assumes the vectors actually used will be redefined by another name. (The processing routine may also be written by moving the jump address below to another location.)

When an invalid interrupt is generated, the CPU jumps to INT_LOOP. It may be convenient to have a breakpoint set here when debugging the program. The address error exception (ADDRERR), 7th from the top in the vector table, occurs especially frequently in undebugged code. Although the address error exception in the preceding sample code is not separated from other exceptions or interrupts, we recommend that address invalid exceptions be vectored to another routine. Note that an address error exception occurs when an attempt is made to access an odd address during 16-bit memory read / writes, or when accessing a nonword-aligned address (not a multiple of 4) during 32-bit memory read / writes. In the S1C33, these memory accesses are prohibited.

Redefinition of interrupt vectors [vector.h]

```
;; Vector define
#define     RESET    BOOT
#define     INT12    int_16timer_u00
#define     INT15    int_16timer_c01
#define     INT18    int_16timer_u11
#define     INT23    int_16timer_c21
#define     INT27    int_16timer_c31
```

Redefine the exception / interrupt vector labels actually used in vector.s by another name, letting the CPU jump to the appropriate routine. In the preceding example, the reset vector and 16-bit timer interrupt vectors are redefined using the label names of the actual processing routines.

## Boot routine [boot.s]

```
;     file name : boot.s
;
;     Coptright (C) SEIKO EPSON CORP. 2002
;
;   BOOT:
;     boot program  set, SP, default data area pointer(%r15)
;     call _init_sys() and _init_lib().
;     And call main.

.set SP_INI,0x0800               ; sp is in end of 2KB internal RAM
.set DP_INI,0x0000               ; global pointer is 0x0

      .text
      .long BOOT                 ; BOOT VECTOR
BOOT:
      xld.w    %r15,SP_INI
      ld.w     %sp,%r15          ; set SP
      ld.w     %r15,DP_INI       ; set default data area pointer
      xcall    _init_bcu         ; Initialize BCU on boot time
      xcall    _init_sys         ; call _init_sys() in sys.c
      xcall    main              ; goto main
      xcall    _exit             ; in last, goto _exit

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; _init_bcu
;;   Type :     void
;;   Ret val :  none
;;   Argument : void
;;   Function : Initialize BCU on boot time.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      .global  _init_bcu
_init_bcu:
;; Set area 9-10 setting
;;    Area 9-10 setting ... Device size 16 bits, output disable delay 1.5, wait control
2, burst ROM is not used in area 9-10, burst ROM burst read cycle wait control 0
      xld.w    %r11,BCU_A10_ADDR
      xld.w    %r10,BCU_BW_0|BCU_DRAH_NOT|BCU_DRAL_NOT|BCU_SZL_16|BCU_DFL_15|BCU_WTL_2
      ld.h     [%r11],%r10
```

This boot routine (BOOT) initializes the stack pointer, the default data-area pointer and the BCU before calling the main routine.

Since the CPU uses the stack if any exception or interrupt occurs, make sure the stack pointer is set before other processing. Always confirm that the BCU is set before accessing memory or device.

## ● Code example written in C

The following illustrative code is found in gnu33\sample\drv33209\.

Vector table, boot routine [drv33209\16timer\vector.c]

```
/************************************************************************
 *                                                                      *
 *      Copyright (C) SEIKO EPSON CORP. 2002                            *
 *                                                                      *
 *      File name: vector.c                                             *
 *        This is vector and interrupt program with C.                  *
 *                                                                      *
 ************************************************************************/

/* Prototype */
void boot(void);
void dummy(void);
extern void _init_bcu(void);
extern void _init_int(void);
extern void _init_sys(void);
extern void _exit(void);
extern void int_16timer_c0(void);
extern void int_16timer_u1(void);
extern void int_16timer_c2(void);
extern void int_16timer_u3(void);

/* vector table */
const unsigned long vector[] = {
      (unsigned long)boot,             // 0     0
      0,                               // 4     1
      0,                               // 8     2
      0,                               // 12    3
      (unsigned long)dummy,            // 16    4
      0,                               // 20    5
      (unsigned long)dummy,            // 24    6
      (unsigned long)dummy,            // 28    7
      0,                               // 32    8
      0,                               // 36    9
      0,                               // 40    10
      0,                               // 44    11
      (unsigned long)dummy,            // 48    12
      (unsigned long)dummy,            // 52    13
                |
          (56  14 – 120  30)
                |
      (unsigned long)int_16timer_c0,   // 124   31
      (unsigned long)dummy,            // 128   32
      (unsigned long)dummy,            // 132   33
      (unsigned long)int_16timer_u1,   // 136   34
      (unsigned long)dummy,            // 140   35
      (unsigned long)dummy,            // 144   36
      (unsigned long)dummy,            // 148   37
      (unsigned long)dummy,            // 152   38
      (unsigned long)int_16timer_c2,   // 156   39
      (unsigned long)dummy,            // 160   40
      (unsigned long)dummy,            // 164   41
      (unsigned long)int_16timer_u3,   // 168   42
      (unsigned long)dummy,            // 172   43
                |
          (176  44 – 268  67)
                |
      (unsigned long)dummy,            // 272   68
      (unsigned long)dummy,            // 276   69
      (unsigned long)dummy,            // 280   70
      (unsigned long)dummy             // 284   71
};
```

```
/*******************************************************************************
 * boot
 *   Type :     void
 *   Ret val :  none
 *   Argument : void
 *   Function : Boot program.
 *******************************************************************************/
void boot(void)
{
      asm("xld.w %r15,0x2000");  // Set SP in end of 8KB internal RAM
      asm("ld.w  %sp,%r15");
      asm("ld.w  %r15,0b10000"); // Set PSR to interrupt enable
      asm("ld.w  %psr,%r15");
      asm("xld.w %r15,0x600000");// Set DPR is 0x0
      _init_bcu();               // Initialize BCU on boot time
      _init_int();               // Initialize interrupt controller
      _init_sys();               // Initialize for sys.c
      main();                    // Call main
      _exit();                   // In last, go to exit in sys.c to use simulated I/O
}

/*******************************************************************************
 * dummy
 *   Type :     void
 *   Ret val :  none
 *   Argument : void
 *   Function : Dummy interrupt program.
 *******************************************************************************/
void dummy(void)
{
INT_LOOP:
      goto     INT_LOOP;
      asm("reti");
}
```

This file contains a vector table and a boot routine.

The vector table is defined as a const-type 32-bit array to allow storage of 32-bit jump addresses in ROM. The comment for each vector (//x y) is a decimal value indicating the offset address (x) from the top of the table and the vector number (y). In this example, the start addresses of externally-referenced interrupt processing functions are written directly. Unused interrupts are vectored to dummy routines.

The boot routine is functionally equivalent to the preceding example written in assembler. The SP and PSR are initialized using the asm() instruction.

The reti instruction for the dummy exception/interrupt handler routine is written using the asm() instruction.

## *2.2 Interrupt Handling Routines*

### ● Interrupt handling functions

In S5U1C33001C, interrupt handler functions can be implemented by declaring a function prototype with __attribute__ ((interrupt_handler)).
In addition, the "asm("reti");" line within the interrupt function can be deleted by declaring the function with __attribute__ ((interrupt_handler)).

### ● Declaring interrupt handler functions

Interrupt handler functions should be declared in the following format:

<Type><Function name>__attribute__ ((interrupt_handler));

Example for interrupt handling [gnu33\sample\int_c\int.c]

```
// int.c  1998.1.7
// vector and interrupt program with C

extern volatile int int_num;

// functions that jump from vector table

extern void boot() __attribute__ ((interrupt_handler));
void div0() __attribute__ ((interrupt_handler));
void unalign() __attribute__ ((interrupt_handler));
void nmi() __attribute__ ((interrupt_handler));
void softint0() __attribute__ ((interrupt_handler));
void softint1() __attribute__ ((interrupt_handler));
void softint2() __attribute__ ((interrupt_handler));
void softint3() __attribute__ ((interrupt_handler));
void hardint0() __attribute__ ((interrupt_handler));
void hardint1() __attribute__ ((interrupt_handler));
void hardint2() __attribute__ ((interrupt_handler));
void hardint3() __attribute__ ((interrupt_handler));
void hardint4() __attribute__ ((interrupt_handler));
void hardint5() __attribute__ ((interrupt_handler));
void hardint6() __attribute__ ((interrupt_handler));
void hardint7() __attribute__ ((interrupt_handler));
void hardint8() __attribute__ ((interrupt_handler));
void hardint9() __attribute__ ((interrupt_handler));

// vector table

const unsigned long vector[] = {
      (unsigned long)boot,          // 0    0
      0,                            // 4    1
      0,                            // 8    2
      0,                            // 12   3
      (unsigned long)div0,          // 16   4
      0,                            // 20   5
      (unsigned long)unalign,       // 24   6
      (unsigned long)nmi,           // 28   7
      0,                            // 32   8
      0,                            // 36   9
      0,                            // 40   10
      0,                            // 44   11
      (unsigned long)softint0,      // 48   12
      (unsigned long)softint1,      // 52   13
      (unsigned long)softint2,      // 56   14
      (unsigned long)softint3,      // 60   15
      (unsigned long)hardint0,      // 64   16
      (unsigned long)hardint1,      // 68   17
      (unsigned long)hardint2,      // 72   18
      (unsigned long)hardint3,      // 76   19
      (unsigned long)hardint4,      // 80   20
      (unsigned long)hardint5,      // 84   21
      (unsigned long)hardint6,      // 88   22
      (unsigned long)hardint7,      // 92   23
      (unsigned long)hardint8,       // 96   24
      (unsigned long)hardint9        // 100  25
};

// interrupt routines

void div0()
    {
      int_num = 4;
    }

void unalign()
    {
      int_num = 6;
    }

void nmi()
    {
      int_num = 7;
    }

void softint0()
    {
      int_num = 12;
    }

void softint1()
    {
      int_num = 13;
    }

void softint2()
    {
      int_num = 14;
    }

void softint3()
    {
      int_num = 15;
    }

void hardint0()
    {
      int_num = 16;
    }

void hardint1()
    {
      int_num = 17;
    }

void hardint2()
    {
      int_num = 18;
    }

void hardint3()
    {
      int_num = 19;
    }

void hardint4()
    {
      int_num = 20;
    }

void hardint5()
    {
      int_num = 21;
    }
```

```
void hardint6()
    {
      int_num = 22;
    }

void hardint7()
    {
      int_num = 23;
    }

void hardint8()
    {
      int_num = 24;
    }

void hardint9()
    {
      int_num = 25;
    }
```

## ● Example for software interrupt handling

Example for software interrupt handling [gnu33\sample\int_c\main.c]

```
// main.c 2002.2.14
// boot and main program with C

volatile int int_num;

void boot()
    {
LOOP:
      asm("xld.w     %r8,0x0800");           // set SP
      asm("ld.w      %sp,%r8");
      asm("xld.w     %r8,0b10000");          // set PSR to interrupt enable
      asm("ld.w      %psr,%r8");
      asm("ld.w      %r15,0x0000");          // set Default data area pointer
      main();                                // call main
      goto LOOP;
    }

int main()
    {
      int j,k;

      asm("int 1");                          // software interrupt 1

      for (j = 0 ; ; j++)
          {
               k = int_num;
          }
    }
```

The asm("int 1"); routine is defined as software interrupt 1 in the interrupt vector table.

`(unsigned long)softint1,         // 52   13`   Definition of software interrupt 1

When this interrupt occurs, the program branches to the prototype-declared function in order to process the interrupt. In the case of the above example, the program, after executing int_num=13; returns to within the main(); function and enters an infinite loop.

## 2.3 C and Assembler Mixed Programming

Control can pass between C and assembler routines as desired, providing that rules for arguments, return values, and register content protection are observed.

### ● Creating an assembler routine called from C

gnu33\UTILITY\lib_src\ansilib\string\src\strcpy.s

```
;*****************************************************
; strcpy
;     string copy from src to dest until 0 terminate
;
; arguments : %r6:dest addr, %r7:src addr (0 terminate string)
; return    : %r4:dest addr
;*****************************************************

      .section .text
      .align   1
      .global  strcpy
      .type    strcpy,@function

strcpy:
      ld.w     %r4, %r6          ; return dest add

strcpy_loop:
      ld.ub    %r10, [%r7]+      ; copy src 1 byte to dest
      ld.b     [%r6]+, %r10
      cmp      %r10, 0           ; continue until 0 terminate
      jrne     strcpy_loop
      ret
```

This routine is called from a C routine as follows.
(Excerpt from gnu33\sample\ansilib\sansilib.c)

```
               |
#include <string.h>
               |
int main()
    {
               |
      char *pchMem;              /* for malloc, strcpy */
               |
      strcpy(pchMem, "This is strcpy test");
    }
```

The first and the second arguments are respectively placed in the R6 and the R7 registers when passed, and the return value is stored in the R4 register.

As in this example, arguments and return values must be exchanged using registers, as follows:
- The first to fourth arguments are placed in the R6 to the R9 registers when passed.
- In special cases, the preceding arguments and the fifth and subsequent arguments are placed in the stack when passed. (Refer to the compiled code.)
- The return value is stored in the R4 register when returned.

The limitations on register usage within the assembler routine called from a C routine are as follows:
- Before the R0 to R3 registers can be used, the contents must be saved to the stack using the pushn instruction. Also, the saved contents must be restored from the stack using the popn instruction.
- The R4 and R5 registers can be used without saving/restoring the contents until a returned value is set in the register before returning.
- The R6 to R9 registers can be used after the stored arguments are used. It is not necessary to restore the contents before returning.
- The R10 to R15 registers are reserved by the as assembler and ld linker for referencing symbols. Try to use these registers as little as possible.
- Passing arguments to the function that returns a structure data
  If the length of the structure data that is returned from a function is 8 bytes or less, the structure data is stored in the R4 and R5 registers used for storing returned values. In this case, the pointer to the structure that is normally sent as the 1st argument is not passed to the function.

For example, gnu33\UTILITY\lib_src\emulib\adddf3.s processes double-precision, floating-point additions. Since this routine uses all registers, the contents of the R0 to R3 registers are saved and restored before returning.

```
__adddf3:
      pushn    %r3      ; save register values
          |
      popn     %r3      ; restore register values
      ret
```

## ● Creating an assembler routine that calls a C function

C functions are compiled by the preceding rules. When creating an assembler routine that calls a C function, pay attention to the following:

### Rules for delivering arguments and return values

- The first to fourth arguments are placed in the R6 to R9 registers when passed.
- The R4 register is used to receive the return value.

### Register status at return

- The R0 to R3 registers hold the contents possessed when called.
- The R4 to R15 registers and other registers AHR, ALR, or PSR may have been modified.

## *2.4 Tools and Files for Assembly*

The user-created assembly source files are assembled using the following three software tools:

1. C compiler xgcc (specifying the -c -xassembler-with-cpp option)
2. Preprocessor cpp
3. Assembler as

* The assembly source files can be assembled by the assembler alone; in such a case, however, the preprocessor directives cannot be processed.

Example for assembly source (.s) created by user

```
; boot.s  2002.2.8
; boot program

#define SP_INI 0x0800           ; sp is in end of 2KB internal RAM        (1)
#define DP_INI 0x0000           ; default data area pointer %r15 is 0x0   (1)

      .text                                                               (2)
      .long    BOOT             ; BOOT VECTOR                             (2)
BOOT:
      xld.w    %r15,SP_INI                                                (3)
      ld.w     %sp,%r15         ; set SP
      ld.w     %r15,DP_INI      ; set default data area pointer
      xcall    main             ; goto main                               (3)
      xjp      BOOT             ; infinity loop                           (3)
```

(1) Quasi directives processed by cpp
(2) Directive commands processed by as
(3) Extended instructions processed by as

### ● Preprocessor instructions

The instructions beginning with "#" are quasi preprocessor directives, which provide additional functions, such as macro instructions, conditional assembly instructions, or symbol definitions of values and strings, which help create readable assembler code. These instructions are processed by cpp and expanded into basic instructions that can be assembled by as.

Preprocessor quasi directives [gnu33\sample\asm\as_withcpp.s]

```
/**********************************************************
      as instruction sample file with C preprocesser
      xgcc -c -xassembler-with-cpp as_withcpp.s
**********************************************************/

#include "test.h"

#define DATA1  0x1234               // define DATA1
#define DATA2  (DATA1 + 2) << 1     // define DATA2

BAR:                                /* start BAR function */
      xld.w    %r11,DATA1
      xld.w    %r10,DATA2
      sub      %r10,%r11
      xld.w    [BSS1],%r10
      ret

FOO:                                // start FOO function
      add      %r6,%r7
      ld.w     %r4,%r6
      ret

      .section .bss
BSS1:
      .zero 4
```

## ● Assembler directive commands

The assembler directive commands beginning with "." are primarily used to define data written into sections and ROM. The assembler directive commands are not processed until fed into as.

Assembler directive commands [gnu33\sample\asm\as_directive.s]

```
; as_directive.s        2002.3.11
; sample source for as directives

      .set     RAM1,0x0          ; set absolute data

      .text                      ; start text section
      .global  BOOT              ; BOOT become global symbol
      .long    BOOT              ; 32bit data
BOOT:                            ; label in code section
      ld.w     %r15,0
      xld.w    %r1,[DATA1]
      xld.w    [%r15+RAM1],%r1
      xld.ub   %r2,[DATA1+8]
      xld.w    [BSS1],%r2
      jp       BOOT
      .word    0x0000            ; same with nop

      .section .data
      .align   2                 ; align to 4 byte boundary
DATA1:                           ; label in data section
      .long    0x12345678        ; 32bit data
      .word    0x1234,0x5678     ; 16bit data
      .byte    0x90              ; 8bit data
      .ascii   "abc"             ; string data
      .space   4,0               ; 4bytes 0

      .section .bss
      .align   2                 ; align to 4 byte boundary
      .global  BSS1
BSS1:
      .zero    4                 ; 4 byte global bss data area
LBSS1:
      .zero    4                 ; 4 byte local bss data area
```

## ● Primary assembler instructions

When programming with the assembler, the programmer must understand how to write the following instructions.

- CPU core instructions (basic instructions)
- Macro instructions expanded by as (extended instructions)

Pooling all instructions of these two types produces a large number of available instructions, particularly an extensive list of instructions for as.

Until you are familiar with programming the S1C33, we recommend using the two types of extended instructions shown below and the primary basic instructions of the CPU core, and then gradually increasing the number of extended instructions according to the purposes.

### Two types of extended instructions

```
xld.w    %r8,0x12345678          ; Stores immediate value in register
xcall    sub                     ; Call to label
```

### Commonly used basic instructions

#### Arithmetic operation

```
add   %r1,%r2            ; Same as for sub and sbc
add   %r3,3
adc   %r5,%r3

cmp   %r7,%r9
cmp   %r15,-1

mlt.h %r9,%r8            ; unsigned mltu.h and mltu.w also available
mlt.w %r1,%r2            ; div is supported in subroutine form
```

#### Logical operation

```
and   %r2,%r1            ; Same as for or and xor
and   %r1,0b0111

not   %r2,%r1
not   %r1,-1
```

#### Shift

```
srl   %r10,5             ; Same as for sll, sra, sla, rr, and rl
srl   %r9,%r5
```

#### Register copy

```
ld.b  %r2,%r3            ; Same as for ld.ub, ld.h, ld,uh, and ld,w

ld.w  %r8,%alr           ; Same as for sp, ahr, alr, and psr
ld.w  %sp,%r9
```

#### Memory access

```
ld.b  %r9,[%r9]          ; Same as for ld.ub, ld.h, ld,uh, and ld,w
ld.b  %r15,[%r0]+

ld.b  [%r3],%r2          ; Same as for ld.h, and ld.w
ld.b  [%r4]+,%r0

btst  [%r9],0x1          ; Same as for bset, bclr, and bnot
```

#### Branch

```
jrgt  SYM                ; Same as for jrXX, jp, jrXX.d, and jp.d
```

#### Return

```
ret
ret.d
```

#### Interrupt

```
reti
int   3
```

#### Extended instruction

```
ext   0x123
```

#### Other

```
pushn %r15
popn  %r0
mac   %r12
nop
halt
slp
```

## ● Basic instructions

Basic instructions refer to the S1C33000 instruction set, which are assembled into machine codes by as. Write the core CPU mnemonics directly as is. For operands that specify addresses with immediate data, you may write a predefined label by itself, or in combination with displacement, symbol mask or pseudo-operand.

Example:
```
jrgt  LABEL             ; Specify label
call  LABEL+0x10        ; Specify label + displacement
ext   LABEL@h           ; Specify label + symbol mask
ext   LABEL@m
ld.w  %r9,LABEL@l       ; =ld.w %r9,LABEL
ext   doff_hi(FOO)      ; Specify pseudo-operand
ext   doff_lo(FOO)
ld.w  %r0,[%r15]        ; =ld.w %r0,[%r15 + (FOO address - __dp)]
```

The following lists the basic instructions. The instructions in bold can be written only in basic instructions, while the others can be written in the extended instructions.

Basic instruction list [gnu33\sample\asm\as_inst.s]

```
; as_inst.s    2002.2.8
; sample source for as instructions

; arithmetic operations
      add      %r1,%r2
      add      %r3,3
      add      %sp,0x123
      adc      %r5,%r3
      sub      %r1,%r2
      sub      %r3,3
      sub      %sp,0x123
      sbc      %r5,%r3
      cmp      %r7,%r9
      cmp      %r15,-1
      mlt.h    %r9,%r8
      mltu.h   %r7,%r4
      mlt.w    %r1,%r2
      mltu.w   %r5,%r1
      div0s    %r1
      div0u    %r2
      div2s    %r3
      div3s

; logical operations
      and      %r2,%r1
      and      %r1,0b0111
      or       %r2,%r1
      or       %r1,11
      xor      %r2,%r1
      xor      %r1,0x11
      not      %r2,%r1
      not      %r1,-1

; shift & rotation operations
      srl      %r10,5
      srl      %r9,%r5
      sll      %r10,5
      sll      %r9,%r5
      sra      %r10,5
      sra      %r9,%r5
      sla      %r10,5
      sla      %r9,%r5
      rr       %r10,5
      rr       %r9,%r5
      rl       %r10,5
      rl       %r9,%r5
```

```
; etc
      pushn    %r15
      popn     %r0
      mac      %r13
      nop
      halt
      slp
      scan0    %r1,%r2
      scan1    %r3,%r4
      swap     %r5,%r6
      mirror   %r7,%r7

; bit operations
      btst     [%r9],0x1
      bset     [%r0],7
      bclr     [%r15],0b1
      bnot     [%r10],5

; ext operations
      ext      0x123
      ext      SYM@ah
      ext      SYM+0x56@ah
      ext      SYM@al
      ext      SYM+0x56@al
      ext      SYM@h
      ext      SYM+0x56@h
      ext      SYM@m
      ext      SYM+0x56@m
      ext      SYM@rh
      ext      SYM@rm

; load operations
      ld.b     %r2,%r3
      ld.b     %r9,[%r9]
      ld.b     %r15,[%r0]+
      ld.b     %r6,[%sp+8]
      ld.b     [%r3],%r2
      ld.b     [%r4]+,%r0
      ld.b     [%sp+0x10],%r11
      ld.ub    %r2,%r3
      ld.ub    %r9,[%r9]
      ld.ub    %r15,[%r0]+
      ld.ub    %r6,[%sp+8]
      ld.h     %r2,%r3
      ld.h     %r9,[%r9]
      ld.h     %r15,[%r0]+
      ld.h     %r6,[%sp+8]
      ld.h     [%r3],%r2
      ld.h     [%r4]+,%r0
      ld.h     [%sp+0x10],%r11
      ld.uh    %r2,%r3
      ld.uh    %r9,[%r9]
      ld.uh    %r15,[%r0]+
      ld.uh    %r6,[%sp+8]
      ld.w     %r2,%r3
      ld.w     %r8,%alr
      ld.w     %sp,%r9
      ld.w     %r9,[%r9]
      ld.w     %r15,[%r0]+
      ld.w     %r6,[%sp+8]
      ld.w     [%r3],%r2
      ld.w     [%r4]+,%r0
      ld.w     [%sp+0x10],%r11
      ld.w     %r9,SYM@l

; branch operations
      jrgt     -1
      jrgt     SYM
      jrgt     SYM@rl
      jrgt.d   2
      jrgt.d   SYM
      jrgt.d   SYM@rl
      jrge     -1
      jrge     SYM
      jrge     SYM@rl
      jrge.d   2
      jrge.d   SYM
      jrge.d   SYM@rl
      jrlt     -1
      jrlt     SYM
      jrlt     SYM@rl
      jrlt.d   2
      jrlt.d   SYM
      jrlt.d   SYM@rl
      jrle     -1
      jrle     SYM
      jrle     SYM@rl
      jrle.d   2
      jrle.d   SYM
      jrle.d   SYM@rl
      jrugt    -1
      jrugt    SYM
      jrugt    SYM@rl
      jrugt.d  2
      jrugt.d  SYM
      jrugt.d  SYM@rl
      jruge    -1
      jruge    SYM
      jruge    SYM@rl
      jruge.d  2
      jruge.d  SYM
      jruge.d  SYM@rl
      jrult    -1
      jrult    SYM
      jrult    SYM@rl
      jrult.d  2
      jrult.d  SYM
      jrult.d  SYM@rl
      jrule    -1
      jrule    SYM
      jrule    SYM@rl
      jrule.d  2
      jrule.d  SYM
      jrule.d  SYM@rl
      jreq     -1
      jreq     SYM
      jreq     SYM@rl
      jreq.d   2
      jreq.d   SYM
      jreq.d   SYM@rl
      jrne     -1
      jrne     SYM
      jrne     SYM@rl
      jrne.d   2
      jrne.d   SYM
      jrne.d   SYM@rl
      call     -1
      call     SYM
      call     SYM@rl
      call     %r5
      call.d   2
      call.d   SYM
      call.d   SYM@rl
      call.d   %r8
      jp       -1
      jp       SYM
      jp       SYM@rl
      jp       %r5
      jp.d     2
      jp.d     SYM
      jp.d     SYM@rl
      jp.d     %r8
      ret
      ret.d
      reti
      retd
      int      3
      brk
```

## ● Assembly source level debug

Normally, when programs are to be debugged at the C source level, if the compile option -gstabs is added, directives in stab format are added to the elf file as debug information, allowing the program to be debugged at the source level.

The following describes how to debug assembler sources at the source level.

### Using the --gstabs option of the as assembler

When the --gstabs option is specified at start up of the as assembler, line debug information is added to the object file created. This information can be confirmed using the -g option of the objdump tool.

Example:

```
(sample.s)
      1:       .text
      2:       add      %r0,1
      3:       ext      0x100
      4:       ld.w     %r1,0x10

>as -o sample.o sample.s --gstabs
>ld -o sample.elf sample.o
>objdump -g sample.elf

sample.elf:     file format elf32-c33

sample.s:
/* file sample.s line 2 addr 0xc00000 */
/* file sample.s line 3 addr 0xc00002 */
/* file sample.s line 4 addr 0xc00004 */
```

### Using the --gstabs option of the as assembler after executing the cpp preprocessor

Assembly sources that contain preprocessor instructions, such as "#include <filename>", must be processed using the cpp preprocessor before assembling with the as assembler.

Example: Source expanded by cpp (sample.ps)

```
(inc1.h)
      1:       .set  DATA1, 0x01
      2:       .set  DATA2, 0x02
      3:       .set  DATA3, 0x03
      4:       .set  DATA4, 0x04
      5:       .set  DATA5, 0x05
      6:       .set  DATA6, 0x06

(sample.s)
      1:       #include  "inc1.h"
      2:                .text
      3:                add      %r0,1
      4:                ext      0x100
      5:                ld.w     %r1,0x10

>cpp sample.s > sample.ps

(sample.ps)
      1:       # 1 "sample.s"
      2:       # 1 "inc1.h" 1
      3:       .set  DATA1, 0x01
      4:       .set  DATA2, 0x02
      5:       .set  DATA3, 0x03
      6:       .set  DATA4, 0x04
      7:       .set  DATA5, 0x05
      8:       .set  DATA6, 0x06
      9:       # 1 "sample.s" 2
      10:
      11:      .text
      12:      add      %r0,1
      13:      ext      0x100
      14:      ld.w     %r1,0x10

>as -o sample.o sample.ps --gstabs
>ld -o sample.elf sample.o
>objdump -g sample.elf

sample.elf:     file format elf32-c33

sample.s:
```

```
/* file sample.s line 3 addr 0xc00000 */
/* file sample.s line 4 addr 0xc00002 */
/* file sample.s line 5 addr 0xc00004 */
```

**Using the -xassembler-with-cpp and --gstabs options in the xgcc command line**

```
>xgcc -c -xassembler-with-cpp -Wa,--gstabs sample.s
>ld -o sample.elf sample.o
>objdump -g sample.elf

sample.elf:     file format elf32-c33

sample.s:
/* file sample.s line 3 addr 0xc00000 */
/* file sample.s line 4 addr 0xc00002 */
/* file sample.s line 5 addr 0xc00004 */
```

The processing in the previous example using cpp and as can be executed simply by using these options.

## ● make file

A standard make file generated by the gwb33 work bench is shown below. Make files can be edited using the Make file editor in gwb33 or a general-purpose editor.

| | |
|---|---|
| Make file name: | sample.mak |
| Assembler source: | boot.s |
| C source: | main.c, sys.c |

In the above case, the make file is created in the same manner as the one shown below.

make file [sample.mak]

```
# make file made by GWB33

# make file made by gnu make

# macro definitions for target file

TARGET= sample

# macro definitions for tools & dir

TOOL_DIR = C:/GNU33
CC= $(TOOL_DIR)/xgcc
AS= $(TOOL_DIR)/xgcc
LD= $(TOOL_DIR)/ld
RM= $(TOOL_DIR)/rm
LIB_DIR= $(TOOL_DIR)/lib
SRC_DIR= .

# macro definitions for tool flags

CFLAGS= -B$(TOOL_DIR)/ -c -gstabs -O -mgda=0 -mdp=1 -mlong-calls -I$(TOOL_DIR)/
        include -fno-builtin
ASFLAGS= -B$(TOOL_DIR)/ -c -xassembler-with-cpp -Wa,--gstabs
LDFLAGS= -T $(TARGET).lds -Map $(TARGET).map -N

# macro definitions for object files

OBJS= boot.o \
      main.o \
      sys.o  \

OBJLDS=

# macro definitions for library files

LIBS= $(LIB_DIR)/libc.a $(LIB_DIR)/libgcc.a

# dependency list start

### src difinition start
```

```
SRC1_DIR= .
### src difinition end

$(TARGET).elf : $(OBJS) $(TARGET).mak $(TARGET).lds
      $(LD) $(LDFLAGS) -o $@ $(OBJS) $(OBJLDS) $(LIBS)

## boot.s
boot.o : $(SRC1_DIR)/boot.s
      $(AS) $(ASFLAGS) -o boot.o $(SRC1_DIR)/boot.s

## main.c
main.o : $(SRC1_DIR)/main.c
      $(CC) $(CFLAGS) $(SRC1_DIR)/main.c

## sys.c
sys.o : $(SRC1_DIR)/sys.c
      $(CC) $(CFLAGS) $(SRC1_DIR)/sys.c

# dependency list end

# clean files except source

clean:
      $(RM) -f $(OBJS) $(TARGET).elf $(TARGET).map
```

## ● Sharing of header files in assembler and C

The S5U1C33001C allows header files (*.h) to be shared in the assembly and C sources. Both the assembly and C header files can be referenced by using the preprocessor directive #include "filename.h" before a line in which the preprocessor definition is used. When assembling assembly sources, always be sure to put them through the C compiler xgcc and specify the -xassembler-with-cpp option if the preprocessor functions are to be used. If an attempt is made to assemble them by as alone, an error may occur. Under no circumstances may the assembler-inherent functions be used in C sources by defining them using the #define directive. Nor may the C language-inherent functions be used in assembly sources by defining them using #define.

Bad example 1:

(header1.h)
```
#define REF_SYMBOL  .extern symbol            ← This is a function inherent in the assembler.
```

(source.c)
```
    REF_SYMBOL;                               ← A syntax error will occur.
    int main( void ) {
    }
```

Bad example 2:

(header1.h)
```
#define SUCCESS_FLAG(x)  ( x>= 0 )            ← This is a function inherent in C.
```

(source.s)
```
    cmp SUCCESS_FLAG(%r0)                     ← A syntax error will occur.
```

Do not write a preprocessor-dependent branch such as #if, #ifdef, or #ifndef in assembly sources. The source and the actual assembler lines may be misplaced with respect to each other's position when they are displayed in the debugger. Furthermore, when header files are to be referenced by #include in assembly sources, do not use #include in the header files to be referenced. In this case as well, the source and the actual assembler lines may be misplaced when they are displayed.

Bad example:

(header1.h)
```
#include "header2.h"
        |
```
(src.s)
```
#include "header1.h"
        |
```
Header references are nested.

Good example:

(header1.h)
```
/* #include "header2.h" */
        |
```

(src.s)
```
#include "header2.h"
#include "header1.h"
        |
```

Shown below is an example in which headers are shared. The source for this example is included in \gnu33\sample\shareh. For details on the source, refer to this file. In this example, TASK_X, which is defined in a common header, is used in the assembly source when registering tasks. The program here uses a macro replacement function for the constants that can be used in common in the assembly and C sources.

The header file shown below can be used in common in the assembly and C sources. The #define preprocessor directive is used to define only the values that will be determined when the preprocessor is executed. To define the values that will be determined by an external file or a link, the program must be written in accordance with the respective rules of C or assembler.
(shareh.h)
```
    /* Macros in this header are used in both C and assembler sources. */

    #define TASK_MASK                0x00ff
    #define TASK_DISNABLE            0x0000
    #define TASK_ENABLE              0x0001
    #define TASK_DEFAULT             TASK_ENABLE

    #define TASK_USINGMASK           0xff00
    #define TASK_N( x )              ( 0x0100 << x )
    #define TASK_0                   TASK_N( 0 )
    #define TASK_1                   TASK_N( 1 )
    #define TASK_2                   TASK_N( 2 )
    #define TASK_3                   TASK_N( 3 )
    #define TASK_4                   TASK_N( 4 )
    #define TASK_5                   TASK_N( 5 )
    #define TASK_6                   TASK_N( 6 )
    #define TASK_7                   TASK_N( 7 )
    #define MAX_TASK_NUM             8

    #define TASK_0_7                 ( TASK_0 + TASK_1 + TASK_2 + TASK_3 \
                                       TASK_4 + TASK_5 + TASK_6 + TASK_7 )
    #define TASK_NOT0                TASK_0_7 - TASK_0
    #define TASK_0_3                 TASK_0 * 0xf

    /*******************************************************************/
    /* when defined, application go to start routine if exit.          */
    /*******************************************************************/
    #define GOSTART_IFAPPEND

    /*******************************************************************/
    /* if you change max task number, enable below line.               */
    /*******************************************************************/
    //#define OVERRIDE_MAXTASK    8
    /* new MAX_TASK_NUM ( max task number is 8. ) */
    #define OVERRIDE_MAX_TASK_NUM  8

    /*******************************************************************/
    /* if you don't use any tasks, enable below line.                  */
    /*******************************************************************/
    //#define NO_TASK

    /*******************************************************************/
    /* psr default setting                                             */
    /*******************************************************************/
    /* This configuration psr is set at the top of program. */
    /* if you don't permit interruption, comment here.*/
    #define IE_BIT  0x10
    /* set default interrput-level( In many case, below should be 0). */
    #define DEFAULT_INTERRUPT_LEBEL      0

    #define DEFAULT_PSR     ( IE_BIT | ( DEFAULT_INTERRUPT_LEBEL << 8 )   )

    /* dummy definition */
    #define MACRO_DAMMY
    #undef MACRO_DAMMY
```

In the file shown below, definitions that can be referenced only by the assembler are written. To use preprocessor definitions in the assembler, replacement by a string or numeral definition is effective.

(onlys.s)

```
    #define GETTASK_H            ext doff_hi( ulTaskManage )
    #define GETTASK_L            ext doff_hi( ulTaskManage )
    #define GETTASK( register )  ld.w register,[%r15]

    #define PUTTASK_H            ext doff_hi( ulTaskManage )
    #define PUTTASK_L            ext doff_hi( ulTaskManage )
    #define PUTTASK( register )  ld.w [%r15],register
```

Shown below is an example in which preprocessor definitions are used in the assembly source.

(asm.s)

```
    /* Nesting header files is not supported in assembler sources. */
    #include "shareh.h"
    #include "onlys.h"
              |
            xld.w     %r0,DEFAULT_PSR               ← Uses a preprocessor definition
                               /* Set IE, IL. See shareh.h About DEFAULT_PSR. */

    init_task:
            xld.w     %r6,TASK_0                    ← Uses a preprocessor definition
            xld.w     %r7,DO_INT
            call      add_task
            ret
              |
    SOFT_INT:
            pushn     %r14

            /* delete old task0 */
            xld.w     %r6,TASK_0                    ← Uses a preprocessor definition
            call      delete_task
            /* add dummy task0 */
            xld.w     %r6,TASK_0                    ← Uses a preprocessor definition
            xld.w     %r7,vfnSetDummy
            call      add_task

            /* add exit task */
            xld.w     %r6,TASK_7                    ← Uses a preprocessor definition
            xld.w     %r7,EXIT_MAIN
            call      add_task

            popn      %r14
            reti
              |
    /* exit all program */
    /* --- no arguments --- */
    EXIT_MAIN:
            /* read ulTaskManage */
            GETTASK_H                               ← Uses assembler-inherent
            GETTASK_L                               ← definitions
            GETTASK( %r4 )                          ←

            /* clear TASK_MASK part, and set TASK_DISABLE */
            xld.w     %r5,TASK_MASK
            not       %r5,%r5
            and       %r4,%r5

            /* write ulTaskManage */
            PUTTASK_H                               ← Uses assembler-inherent
            PUTTASK_L                               ← definitions
            PUTTASK( %r4 )                          ←
            ret
```

Shown below is an example of a header file that can only be used in C source files. The program shown here uses preprocessor definitions and a C syntax do {...} while(1) to create a quasi-local variable. Furthermore, an artifice is incorporated at the beginning of the file to ensure that no errors will be assumed even when the header file is referenced doubly. This approach is very effective in cases in which the mutual relationship between header files is complicated.

(csrc.h)

```
    #ifndef _ONLYC_H_                                /* _ONLYC_H_ */

    #define _ONLYC_H_

    #include "shareh.h"
```

```
/* macro definition changing a task bit to a task number. */
/* This definition can be used in only C-source files. */
#define TASK_NUM(  task_bit, returned_tasknum )  \
          do  { \
              unsigned long ulBuf; \
              \
              ulBuf = task_bit; \
              returned_tasknum = 0; \
              while ( (ulBuf & 0x100) == 0x0 ) { \
                  returned_tasknum++; \
                  ulBuf >>= 1; \
              }; \
          } while ( 0 );

/* #if, #ifdef or other branch prepro-instructions are
*    used in only C sources. */
#if defined( OVERRIDE_MAX_TASK_NUM )
#undef MAX_TASK_NUM
#define MAX_TASK_NUM       OVERRIDE_MAX_TASK_NUM
#endif

#ifdef NO_TASK
/* disable all tasks. */
#undef  TASK_DEFAULT
#define TASK_DEFAULT    TASK_DISNABLE
#endif

#endif                                             /* _ONLYC_H_ */
```

Shown below is an example of a C source file that uses a preprocessor definition.

(csrc.c)

```
#include "onlyc.h"
          |
int main_loop( void )
    {

    while ( 1 ) {
        volatile unsigned long *ulpTask = &ulTaskManage;
        unsigned long          ulTaskBuf;
        unsigned int iTaskBit;
        int iTaskNum ;

        if ( ( *ulpTask & TASK_MASK ) == TASK_ENABLE ) {
            ulTaskBuf = *ulpTask;
            for ( iTaskNum = 0, iTaskBit = TASK_0 ;
                iTaskNum < MAX_TASK_NUM ; iTaskNum++, iTaskBit <<= 1 ) {
                if ( (ulTaskBuf & iTaskBit) != 0 ) {
                    /* Execute task if registered. */
                    if ( fnpTask[ iTaskNum ] != (void *)0x0 ) {
                        fnpTask[ iTaskNum ]();
                    }
                }
            }
        } else {
            /* illeagal interruption occured */
            break;
        }
    }

#if !defined( GOSTART_IFAPPEND )
ENDLESS_LOOP:
    goto ENDLESS_LOOP
#else
    return 0;
#endif

}

void add_task( unsigned long ulTaskFlag, void *fpFunc )
{

    int iNum = 0;

    TASK_NUM(  ulTaskFlag, iNum )

    fnpTask[ iNum ] = fpFunc;
    ulTaskManage = ulTaskManage | ulTaskFlag ;
}
          |
```

## 2.5 Data Areas and Data-Area Pointers

The S5U1C33001C has multiple available data-area pointers that hold the base addresses of memory areas referred to as "data area". (These data-area pointers are assigned the CPU registers.) The data-area pointers enable the global variables, pre-valued variables, and constant data located in a data area to be accessed efficiently at high speed as a result of the use of smaller offset addresses than those used to access the entire memory. It is possible to specify which data is to be located in which data area through the use of an intra-source definition and a linker script setting.

For a data area to be accessed using a data-area pointer, the following operation is required:

1. Set a data-area pointer at the beginning of the program.
2. Specify a data area in the program in which variables are to be located.
3. Set compiler options associated with the data-area pointer.
4. Define sections in a linker script (if necessary).
5. Define the data-area pointer (the base address of the data area) as a symbol when linking the object files.

### 2.5.1 Types of Data Areas

The S5U1C33001C supports the following five data areas:

- Default data area
- G data area
- S data area
- T data area
- Z data area

The section attributes located in each data area and the CPU registers used as data-area pointers, data pointer symbols, and the like are listed in Table 2.5.1.1.

*Table 2.5.1.1 Data area*

| Data area | Attribute of located section | | | Register used | Directive used to access | Data-area-pointer symbol | 32-bit address |
|---|---|---|---|---|---|---|---|
| | Variable | Pre-valued variable | Constant | | | | |
| Default | .bss | .data | .rodata | (STD) R15<br>(ADV) DP<br>∗1 | (STD) doff_hi<br>(STD) doff_lo<br>(ADV) dpoff_h<br>(ADV) dpoff_m<br>(ADV) dpoff_l | __dp | Acceptable only for ADV |
| G | .gbss | .gdata | .rodata<br>∗3 | R15–R12<br>∗2 | goff_lo | __gdp | Not accepted |
| Z | .zbss | .zdata | .rozdata | R14 | zoff_hi<br>zoff_lo | __zdp | Not accepted |
| T | .tbss | .tdata | .rotdata | R13 | toff_hi<br>toff_lo | __tdp | Not accepted |
| S | .sbss | .sdata | .rosdata | R12 | soff_hi<br>soff_lo | __sdp | Not accepted |

∗1: If the -mc33adv option (compile in advanced macro mode) is used in compiling the source files, the pointer for the default data area will have its register assignment changed.

∗2: The register used for the G data area can be changed using the compiler option -mgdp=XX (XX: dp, sdp, tdp, or zdp).

∗3: All of the constant data in the G data area will be located in the .rodata section.

### *2.5.2 Sections*

One data area consists of multiple sections. Each section has an attribute specific to it.

**• .text attribute**

Programs are stored in this section. This attribute does not belong to a data area. The entity of this section is held in ROM. It can be transferred to RAM before being executed.

**• .bss, .gbss, .zbss, .tbss, and .sbss attributes**

Global variables and static-declared variables are stored in these sections. They are located in RAM, and do not have their entities in ROM.

| | |
|---|---|
| .bss | (located in the default data area) |
| .gbss | (located in the G data area) |
| .zbss | (located in the Z data area) |
| .tbss | (located in the T data area) |
| .sbss | (located in the S data area) |

**• .data, .gdata, .zdata, .tdata, and .sdata attributes**

Pre-valued variables are stored in these sections. They have their entities (initial values) in ROM, and can be transferred to RAM for use as pre-valued variables. The data areas in which they will be located are the same as those for .bss to .sbss.

**• .rodata, .rozdata, .rotdata, and .rosdata attributes**

Constants are stored in these sections. They have their entities in ROM. The data areas in which they will be located are the same as those for .bss to .sbss. However, the constants specified to be located in a G data area are included in the .rodata section, and will be located in the default data area.

**• Other attributes**

Any attribute can be set.

The attribute of each section should be specified in a linker script file (.lds).
Example: Definition of a section named .bss that has the attribute .bss

```
.bss 0x00000000 : {                  ← The .bss here is the name of the section to be defined.
      __START_bss = . ;
      *(.bss) ;                      ← Specifies the .bss attribute.
      __END_bss = . ;
}
```

### *2.5.3 Data-Area Pointers*

The xgcc compiler generates code for the addresses of global symbols (variables, pre-valued data, and constants) to be accessed, in the manner shown below.

Symbol address = data-area-pointer value + symbol value (offset value)

The use of data-area pointers eliminates the need to specify a 32-bit address for each memory location to be accessed. As memory locations can be accessed using only the offset value (26 or 13 bits) of a data-area pointer, the number of memory access instructions is reduced, as is the access time.
For example, the following in a C source

```
a = 1;
```

will be converted into the assembler code given below.

```
xld.w %r4,1               (1)
ext   doff_hi( a )        (2-1)
ext   doff_lo( a )        (2-2)
ld.w  [ %r15 ], %r4       (2-3)
```

This assembler code performs the following operation:

| | |
|---|---|
| (1) | Produces the value to be written to the variable 'a' in the register R4. |
| (2-1 through 2-3) | Writes the value that was set in R4 in (1) to the address of the variable 'a' (R15 + symbol offset value). Here, R15 plays the role of a data-area pointer (R15 is used as the default data-area pointer), and the "offset value for symbol" is added to the data-area pointer by the two extended instructions (2-1) and (2-2). |

The linker acquires the offset address for 'a' by means of (a - __dp), as it resolves the symbols in (2-1) and (2-2). The __dp is a symbol for the default data-area pointer, the value (address) of which is normally specified in a linker script file (with the same value also set in R15 by a program).
Bits 25–13 of the acquired offset address are applied to the code for (2-1), and bits 12–0 are applied to the code for (2-2).

In the above example, although two ext instructions are used to add the offset value for the symbol, if the distance between the data-area pointer and the symbol is short (8K bytes or less), (2-1) is unnecessary and the number of instructions can be reduced by one.

Presented above is an example in which the default data-area pointer __dp (R15) is used. The CPU registers used as symbols for other data-area pointers are listed below.

| | Symbol | Register | |
|---|---|---|---|
| Default data-area pointer: | __dp | R15 | |
| G data-area pointer: | __gdp | R15–R12 | (Shared with one of the other data areas) |
| Z data-area pointer: | __zdp | R14 | |
| T data-area pointer: | __tdp | R13 | |
| S data-area pointer: | __sdp | R12 | |

The data-area pointers must be set in the above registers after a program is booted. In no case will these registers be changed by a compiler. (The registers for unused data-area pointers may be used as scratch registers by specifying the compiler option -mdp.)
Example: To use the default, G, Z, and T data areas in a standard macro model

```
/*    boot.s    */

      .global BOOT
      .align 2
      .text
BOOT:
      xld.w      %r4, 0x800
      ld.w       %sp, %r4

      xld.w      %r15, __dp       /* set default data area pointer */
      xld.w      %r14, __zdp      /* set default z area pointer    */
      xld.w      %r13, __tdp      /* set default t area pointer    */
        :
```

The values for the data-area-pointer symbols should be set in a linker by one of the following methods:

1. Setting data-area pointers in the command line of the linker
   Example: To set __dp, __gdp, and __zdp using the -defsym option
   ```
   DOS>ld.exe -Map test.map -o test.elf boot.o -defsym __dp=0x0 -defsym __gdp=0x0
       -defsym __zdp=0x10000 -N
   ```

2. Specifying data-area pointers in a linker script file
   Example: To set __dp, __gdp, and __zdp in a linker script file
   ```
   DOS>ld.exe -T test.lds -Map test.map -o test.elf boot.o
   ```

   (Data-area-pointer specification part in test.lds)
   ```
                           :
           SECTIONS
           {
               /* data pointer symbol By GWB33 */
               __dp = 0x00000000;
               __gdp = 0x00000000;
               __zdp = 0x00010000;

               /* section information By GWB33 */
               . = 0x0;
                           :
   ```

∗ Data-area pointers can also be set using the linker script editor of the work bench gwb33.

Always be sure to define __dp, as it is used normally.

### *2.5.4 Specifying Compiler Options*

Before data areas can be used, the compiler options inherent in the S5U1C33001C must be set appropriately. The compiler options associated with data areas are listed below.

-mdp, -mgda, -mgdp, -mezda, -metda, -mesda

The -mdp option specifies the data area to be used.

| | |
|---|---|
| -mdp=1 | Only the default data area will be used. |
| -mdp=2 | The default and the Z data areas will be used. |
| -mdp=3 | The default and the Z and T data areas will be used. |
| -mdp=4 | The default and the Z, T, and S data areas will be used. |

The above four are the only combinations that can be specified by -mdp. No other combinations, e.g., use of the T and S data areas without use of the Z data area, can be specified. To use the S data area, for example, the Z and T areas must be used.
It is possible to use only the default and one other data area, such as Z, by specifying -mdp=4. In such a case, care should be taken to ensure that data will not be located in the T and S data areas by a program. However, the data areas must be used in the order of Z, T, and S, without skipping the intermediate areas.

The -mgda option specifies the size of the data to be located in the G data area. To locate a variable 4 bytes or less in size in the G data area, for example, specify -mgda=4. If -mgda=0 is specified, the G data area will not be used. The G data area is located at the beginning (8KB) of other data areas. -mgdp is the option for specifying that data area. For example, specifying -mgdp=zdp causes the G data area to be located at the beginning of the Z data area, with the result that the same data-area pointer as for the Z data area, R14, is used for the area. By default, -mgdp=dp is assumed, so that the G data area will be located at the beginning of the default data area. For details on the G data area, refer to Section 2.5.7, "G Data Area".

The options -mezda, -metda, and -mesda are used to expand the sizes of the Z, T, and S data areas, respectively, to 64MB. When these options are not specified, each area is 8KB in size and can be accessed using two instructions (ext + access instruction). When one of these options is specified, data access to the area is expanded to three instructions (ext + ext + access instruction). Use these options when it is necessary to access a memory area greater than 8KB in size using a data pointer. Note that the default data-area pointer is fixed for access of up to 64MB and cannot be changed to 8KB.

These compiler options must be specified the same way in all source files used. If they are specified differently between source files, register destruction or erratic memory access may occur, or an error may occur during linking.
Particularly in the creation of a library, it is recommended that only the default data area be used as much as possible, with the following options specified:
-mdp=4, -mgda=0

This method of creating a library is not desirable from a performance standpoint, as all of the registers R15 through R12 are reserved for use as data-area pointers (the number of those usable as scratch registers for the compiler is reduced), but because these registers will not be destroyed, the library can also be linked to modules that use other data areas.
When the ANSI library provided for the S5U1C33001C is used, a performance improvement can be expected as a result of recompiling the library using settings adjusted to suit the operating environment.

#### Supplementary explanation of the -mdp option

*Table 2.5.4.1 Registers used by each -mdp option*

| Register | -mdp=1 | -mdp=2 | -mdp=3 | -mdp=4 | -mdp=5 | -mdp=6 |
|---|---|---|---|---|---|---|
| R15 | __dp | __dp | __dp | __dp | __dp | __dp |
| R14 | Scratch register | __zdp | __zdp | __zdp | __zdp | __zdp |
| R13 | Scratch register | Scratch register | __tdp | __tdp | __tdp | __tdp |
| R12 | Scratch register | Scratch register | Scratch register | __sdp | __sdp | __sdp |
| R11 | Scratch register | Scratch register | Scratch register | Scratch register | Protection register* | Protection register* |
| R10 | Scratch register | Scratch register | Scratch register | Scratch register | Scratch register | Protection register* |

∗ xgcc uses the stack in place of this register.

Although the effective numbers usable for the -mdp option are shown here as 1 to 4, the compiler will actually accept -mdp=5 and -mdp=6 as well, without causing an error. However, -mdp=5 and -mdp=6 have no effect as options for specifying the data-area-pointer numbers. As the number of registers used as data pointers is limited to a maximum of 4 from R15 to R12, the number of data pointers is not changed by specifying -mdp=5 or greater. When -mdp=4, the compiler uses R11 and R10 as scratch registers. When -mdp=5 or -mdp=6, R11 or R11 and R10, respectively, become protected registers (those that the compiler does not use). These registers can be freely used from the assembler with no need to save values. It should be noted that, when -mdp=5 or 6 is specified, the register variables that would otherwise be assigned to R10 and R11 are assigned to the stack instead, with a resulting decrease in the execution speed. As assembler-only registers such as these may be utilized to good effect in, for example, compiler tuning in the future, they are left intact.
It is recommended that -mdp be set in the range of 1 to 4.

### 2.5.5 *Method for Locating Data in the Data Areas*

To locate variables and the like in a specific data area, use the method described below to define data.

<Data type>  <Symbol>   __attribute__((Xda));

(Xda= zda, tda, sda)

Example:

```
   char c_default;                   // put to default
   char c_z __attribute__((zda));    // put to Z
   char c_t __attribute__((tda));    // put to T
   char c_s __attribute__((sda));    // put to S
```

∗ In this example, -mgda=0 is assumed (G data area unused).

Global variables without initial values, pre-valued variables, and constants are located in sections .Xbss, .Xdata, and .roXdata, respectively. The data defined without data-area declarations is located in the default data area.

When variables and the like are located in the G data area, the size of the data to be located can be specified using the compiler option -mgda. Data no larger than the size specified by the -mgda option will be located in the G data area.
Example: When -mgda=2 is specified

```
   char c_data;                      // put to G
   unsigned short us_data;           // put to G
   long us_data;                     // put to default
```

∗ In this example, -mgdp=dp is assumed.

Constants are located in the default data area (.rodata section) regardless of what area is specified by the -mgdp option.

If a variable specified to be located in the G data area by the -mgda option is accompanied by __attribute__((Xda)) declaration, it is located preferentially in the Xda area, overriding the specification.

In the xgcc compiler, referencing variables or constants in other sources using an extern declaration requires caution. If the content of the extern declaration of a variable is not the same as that defined in the referenced file, the variable will inadvertently be handled as another variable present in other than the data area in which the variable to be referenced exists. When referencing variables by extern, make sure their data types and data-area declarations are exactly the same as those of the original variable.
Example 1: Improper data-area declaration

file1.c:
```
   char        c_sdata __attribute__((sda));
```
file2.c:
```
   extern char c_sdata;
```

In this case, as the variable c_sdata in file2.c is regarded as a variable in the default or the G data area, the variable defined in file1.c cannot be accessed. It is necessary that __attribute__((sda)) be declared in file2.c as well.

Example 2: Improper specification of the data size (when the compiler option -mgda=4 is specified)

```
file1.c:
   char       c_data[ 0x100 ];
file2.c:
   extern char c_data[];
```

Although the variable c_data in file1.c is a variable in the default data area, it is regarded in file2.c as a variable of 4 bytes or less in size that is present in the G data area. The same variable declaration must be entered in file2.c as well. The correct extern declaration is shown below.

```
file1.c:
   char       c_data[ 0x100 ];
file2.c:
   extern char c_data[ 0x100 ];
```

In order for the correct data area to be accessed, make sure the content of the extern declaration is the same as that of the variable in the referenced file.

## 2.5.6 Method for Setting Data-Area Pointers

### ● Basic settings

Data-area pointers should normally be set to the lowermost addresses in the respective data areas. In such a case, it is necessary that sections of a given data area be located within 64M bytes from the data-area-pointer value. However, for the Z, T, and S data areas, each of which is within 64M bytes in size, their data sizes must be expanded using the compiler options -mezda, -metda, and -mesda, respectively.
Only for systems using a CPU that supports advanced macro, the entire 32-bit space can be used as the default data area.

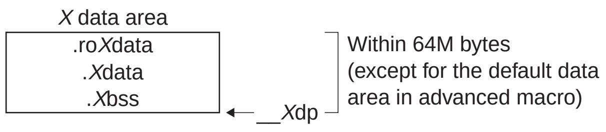


*Fig. 2.5.6.1 Maximum size of a data area*

The following explains the basic method for setting data-area pointers. Assuming that the read-only sections (such as the .text and .rodata sections) that are normally placed in ROM comprise the 'code section', and that the sections that are normally placed in RAM (such as the .bss and .data sections) comprise the 'data section'.

**In cases where the 'code section' and the 'data section' are both placed within 64MB from the default data-area pointer (__dp)**

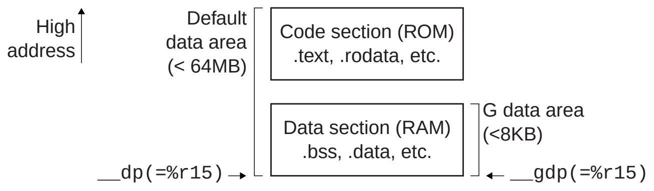


*Fig. 2.5.6.2 Location example 1*

If the 'data section' is to be located at lower addresses than the 'code section', set the lowermost address of the 'data section' in __dp. Set -mgdp=dp, so that the G data-area pointer can be shared with %r15.

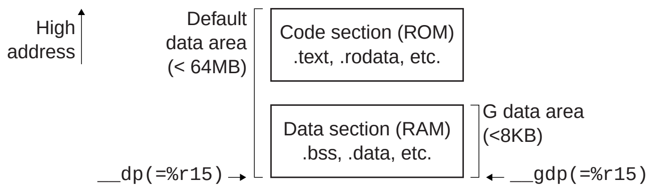
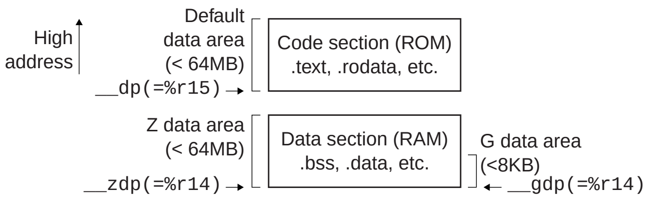


*Fig. 2.5.6.3 Location example 2*

If the 'code section' is to be located at lower addresses than the 'data section', set the lowermost address of the 'code section' in __dp. Set -mgdp=zdp, so that %r14 is assigned to the G data-area pointer, and set the lowermost address of the 'data section'.

**In cases neither the 'code section' nor the 'data section' can be placed within 64MB from the default data-area pointer (__dp)**

(such as when the data section is placed in area 17 while the code section is placed in area 18)


High address
Default data area (< 64MB)
__dp(=%r15)
Code section (ROM) .text, .rodata, etc.
Z data area (< 64MB)
__zdp(=%r14)
Data section (RAM) .bss, .data, etc.
G data area (<8KB)
__gdp(=%r14)

Set the lowermost address of the 'code section' in __dp. Set -mgdp=zdp, so that %r14 is assigned to the G data-area pointer, and set the lowermost address of the 'data section'.

*Fig. 2.5.6.4 Location example 3*

Because the G data area is only 8K bytes, if all data cannot be placed in the G data area, allocate the excess data to the S, T, or Z data areas by specifying an attribute (e.g., int __attribute__((sda)) foo;). In this case, however, because the S, T, and Z data areas use %r12, %r13, and %r14, respectively, data pointers need to be set not to use the G data area (-mgda=0).

The three sections .bss, .data, and .rodata are used in virtually all programs. Make sure that all of these sections will be located in the default data area within 64M bytes from the default data-area pointer __dp as much as possible.
The following shows an example that is normally used. In this example, ROM and RAM are assumed to be allocated to addresses beginning with 0xc00000 and those beginning with 0x0, respectively.

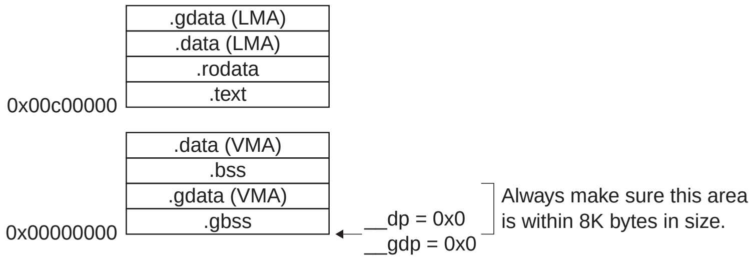


*Fig. 2.5.6.5 Section location and data-area-pointer setup example 1*

For systems that contain one each of ROM and RAM, section locations similar to this example are a standard configuration. If all of the data that must be accessed quickly cannot be located in 8K bytes of the G data area, prepare a Z data area after the .data section (VMA) or .gdata section (LMA), and make sure the data in that area will be accessed via a two-instruction expansion. Executing a program in RAM is another effective means of achieving high-speed processing.

## ● Adding a data-area pointer

For systems having a memory device other than that shown in the above configuration, set one of the S, T, or Z data areas. The example shown below assumes a system having a large-capacity RAM beginning with the address 0x04000000 added to the example configuration shown above.

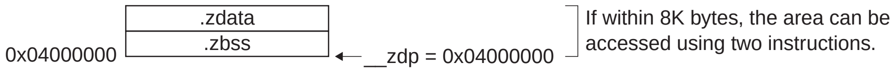
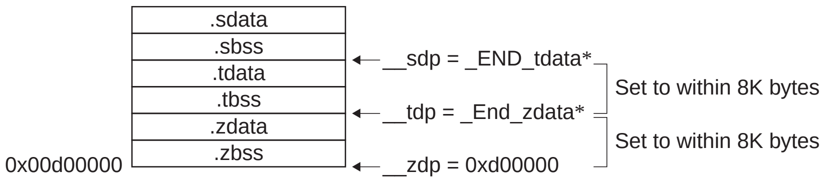

.zdata
.zbss
0x04000000
__zdp = 0x04000000
If within 8K bytes, the area can be accessed using two instructions.

*Fig. 2.5.6.6 Section location and data-area-pointer setup example 2*

This example uses a Z data area and sets the data-area pointer __zdp at the address 0x04000000 (start address of the device). If __zdp is set to any value greater than 0x04000000 here, areas from 0x04000000 up to the set address are handled as unused areas. Although symbols or variables can be set at addresses lower than the data-area pointer, those symbols cannot be accessed from the C source. A warning may be generated during linking. These symbols can be accessed from the assembler source, however.

If the .zbss and .zdata sections exceed 8K bytes in size, specify the -mezda option during compilation to extend the area size to 64M bytes. However, as three instructions are used to access the Z data area each time (two instructions for a size of 8K bytes), the expansion of area sizes is undesirable in terms of code size and access speed. If an S or T data area is usable here, set those data areas after setting the Z data area to increase the number of fast-accessible areas.



∗ In gwb33, symbols cannot be used to set the data-area pointer values. When gwb33 is used to create a linker script file, avoid setting those data-area pointers that cannot be set in that way. Instead, after outputting a linker script file, use an editor or the like to add settings for the desired data pointers.

*Fig. 2.5.6.7 Section location and data-area-pointer setup example 3*

In the case of this example, the data in the Z and T data areas can be accessed quickly via two-instruction expansion. However, be aware that the number of scratch registers usable for the compiler decreases as the number of data areas used increases (in this example, the CPU registers R14 through R12 cannot be used as scratch registers), it may cause a decrease in the overall speed of the program.

### ● Accessing constants

Constants such as those shown in the example below are placed in the .rodata section of the default data area, a section that is located in ROM.

Example:

```
sprintf( szBuf, "Hello, World!!\n" );
```

The character array "Hello, World!!\n" here is located in the .rodata section.

Though constants may not have been specifically defined, it is possible that a .rodata section exists as in this example. As the default data area is accessed using three instructions, a performance improvement can be expected as a result of defining the frequently referenced constants so that they are located in another 8K-byte data area.

Example:

```
const char sz_s_msg1[16] __attribute__((sda)) = "Hello, World!!\n";
sprintf( szBuf, sz_s_msg1 );
```

This example assumes that the S data area can be accessed using a two-instruction expansion (-mesda not used). If the S data area is accessed using three instructions, a speed improvement cannot be expected as long as the number of wait states required for memory access remain the same.

## 2.5.7 G Data Area

The G data area is a special data area that differs from the default, S, T, and Z data areas. Its size is fixed at 8K bytes, enabling it to be accessed quickly using two instructions. Furthermore, as it does not have an inherent data-area pointer, one of the other data-area pointers is used. Therefore, this area shares part of other data areas. When the device is used in advanced macro mode, the G data area can be set independently of other data areas.

For data to be placed in the G data area, no declarations such as those necessary for the S, T, and Z data areas are required. Data with no __attribute__ declaration is placed in the G data area if its size is equal to or smaller than the size specified using the -mgda option, not in the default data area.

Example: When -mgda=2 is specified (all in the example are global variables)

```
char  c_data;          ← Located in the G data area
short s_data;          ← Located in the G data area
int   i_data;          ← Located in the default data area
```

When -mgda=2 is specified, variables of less than 2 bytes in size are located in the G data area.
When -mgda=0 is specified, the G data area is unused.

Normally, sections of the G data area should be located at the beginning of RAM, followed by sections of a data area with which the data-area pointer is shared.
In the example below, the default data-area pointer is used as the G data-area pointer.

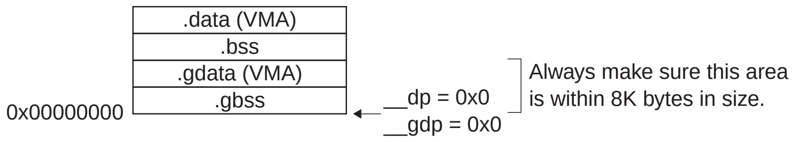


*Fig. 2.5.7.1 G data area*

When using the G data area, be sure to specify the -mgda and -mgdp options during compilation, and also confirm that the value of the data-area pointer symbol __gdp in a linker script file is the same as that of the specified data-area-pointer symbol.

- -mgdp option of the compiler
  To use the default data-area pointer (R15), -mgdp=dp (default setting)
  To use the Z data-area pointer (R14), -mgdp=zdp
  To use the T data-area pointer (R13), -mgdp=tdp
  To use the S data-area pointer (R12), -mgdp=sdp

- Settings in a linker script file
  To use the default data-area pointer, __gdp=__dp*
  To use the Z data-area pointer, __gdp=__zdp
  To use the T data-area pointer, __gdp=__tdp
  To use the S data-area pointer, __gdp=__sdp

  Always be sure to specify the data-area-pointer symbol to be used, or exactly the same numeric value.

  * When the advanced macro is used, this setting does not have to be __gdp=__dp.

The G data area is a data area that is used to store variables with or without initial values, and is effective only for RAM-type devices. Local static variables are also a candidate for being located in the G data area. Constant data cannot be located in the G data area.

### G data-area pointer in advanced macro mode

The advanced macro CPU has an available DP register that is used as the default data-area pointer. As this enables R15 to be assigned to the G data-area pointer, it is possible to set an independent data area that does not use the pointer for one of the other data areas.
The size of the G data area (maximum 8K bytes) and the conditions of data location specified by the -mgda option are the same as when the standard macro described above is used.
To use this setting, specify the compiler option for the advanced macro CPU, -mc33adv.
The -mgdp option can be omitted, as it is used by default (-mgdp=dp). In a linker script file, set the same data-area start address as that of R15 in __gdp.

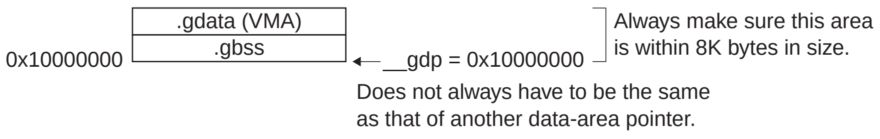

.gdata (VMA)
.gbss
0x10000000
__gdp = 0x10000000
Always make sure this area is within 8K bytes in size.
Does not always have to be the same as that of another data-area pointer.

*Fig. 2.5.7.2 G data area in advanced macro mode*

## 2.6 C and Code Optimization

This section explains how to optimize the instruction code generated by the C compiler, using main.c in gnu33\sample\ccode as an example.
The original source is shown below.

Assembly source boot routine [ccode\boot.s]

```
; boot.s  2002.02.04
; boot program

      .text
      .align   2
      .global  boot
      .long    boot
boot:
      xld.w    %r15,0x0800
      ld.w     %sp,%r15          ; set SP
      xld.w    %r15,__dp         ; set default data area pointer
      xcall    main              ; goto main
      xjp      boot              ; infinity loop
```

C source main program [ccode\main.c]

```
/* main.c 1999.7.28 */
/* sample program for optimize*/

struct ST gst;
int a;
struct ST {
      int s1;
      int s2;
};

main()
    {
      int b;
      struct ST st;
      int ar[10];

      a = 1;
      b = 2;

      st.s1 = 3;
      ar[3] = 4;

      sub1(a, &b);
      sub2();

      gst.s2 = 5;
      sub3(&st, ar);
    }

sub1(a,b)
    int a;
    int *b;
    {
      *b = a;
    }

sub2()
    {
      volatile char *vp;

      vp = (volatile char *)0x40000;
      *vp = 2;

      *(volatile char *)(0x48000) |= 0x1;
    }

sub3(st, ar)
    struct ST *st;
    int ar[];
    {
      st->s2 = 4;
      ar[5]=5;
    }
```

The code generated through the compilation of this default make file is shown below.

```
>objdump -S sample.elf > dis_elf.txt
```

Code derived by compiling [ccode\dis_elf.txt]

```
sample.elf:     file format elf32-c33

Disassembly of section .text:

00c00000 <__START_text>:
  c00000: 0004  nop                     ***
  c00002: 00c0  pops    %psr            ***

00c00004 <boot>:
      .align 2
      .global boot
      .long    boot
boot:
      xld.w    %r15,0x0800
  c00004: c020  ext     0x20
  c00006: 6c0f  ld.w    %r15,0x0        xld.w     %r15,0x800
      ld.w     %sp,%r15          ; set SP
  c00008: a0f1  ld.w    %sp,%r15        ld.w     %sp,%r15
      xld.w    %r15,__dp         ; set default data area pointer
  c0000a: c000  ext     0x0
  c0000c: c000  ext     0x0
  c0000e: 6c0f  ld.w    %r15,0x0        xld.w     %r15,0x0
      xcall    main              ; goto main
  c00010: c000  ext     0x0
  c00012: c000  ext     0x0
  c00014: 1c04  call    0x4             xcall     0x8        (0x00C0001C)
      xjp      boot              ; infinity loop
  c00016: dff8  ext     0x1ff8
  c00018: dfff  ext     0x1fff
  c0001a: 1ef5  jp      0xf5            xjp       0xffffffea (0x00C00004)

00c0001c <main>:
      int s2;
};

         |

sub3(st, ar)
    struct ST *st;
    int ar[];
    {
      st->s2 = 4;
  c00070: 6c44  ld.w    %r4,0x4         ld.w     %r4,0x4
  c00072: c004  ext     0x4
  c00074: 3c64  ld.w    [%r6],%r4       xld.w     [%r6+0x4],%r4

00c00076 <.LM23>:
      ar[5]=5;
  c00076: 6c54  ld.w    %r4,0x5         ld.w     %r4,0x5
  c00078: c014  ext     0x14
  c0007a: 3c74  ld.w    [%r7],%r4       xld.w     [%r7+0x14],%r4

00c0007c <.LM24>:
    }
  c0007c: 0640  ret                     ret
```

## ● About external variables and auto variables

The following section explains how external variables and auto variables are accessed.

```
      a = 1;                                                        ← Accesses to external variable
  c0001e: 6c16  ld.w    %r6,0x1         ld.w     %r6,0x1
  c00020: c000  ext     0x0
  c00022: c008  ext     0x8
  c00024: 3cf6  ld.w    [%r15],%r6      xld.w     [%r15+0x8],%r6

00c00026 <.LM4>:
      b = 2;                                                        ← Accesses to auto variable
  c00026: 6c24  ld.w    %r4,0x2         ld.w     %r4,0x2
  c00028: 5ca4  ld.w    [%sp+0xa],%r4   ld.w     [%sp+0xa],%r4
```

'a' is one of external variables (those with absolute addresses, which here include constants in ROM and static declared variables, in addition to variables in RAM), while 'b' is one of auto variables (variables placed in the stack).

Normally, an external variable is accessed following the procedure
1) Set 32-bit value (variable's address) in R15 (data pointer)
2) Access memory based on R15

Thus, four instructions are required.

Because auto variables are accessed following the procedure below
1) Access the location indicated by SP + offset

an auto variable in the stack area may be accessed with one instruction, if the offset is 63 bytes or less in byte access, 126 bytes or less in half-word access, or 252 bytes or less in word access. It may be accessed with two instructions even if the offset exceeds it. Relatively small number of auto variables are placed in registers automatically, resulting in even more efficient processing. Since they are already placed in registers, this is the case of "access with zero instructions".

For the following reasons, we recommend assigning variables used temporarily in a routine to auto variables whenever possible.
• The number of instructions required for access is small, as described above, and the processing speed is fast.
• Because auto variables are placed temporarily in the stack, RAM does not need to be occupied at all times, conserving RAM use.
• Absence of register assignments and unnecessary accesses make it easier to reap the benefits of optimization by the C compiler.

Excessive use of auto variables has the following disadvantage:
• The practice increases stack size, making it difficult to predict the upper limit.

The stack size can be checked with a debugger, as follows.
1) Allocate a slightly larger stack area.
2) Fill the stack with (as an example) 5555.
3) Execute the application.
4) Finally, display the stack area and check the maximum range of stack used (the range where 5555 are changed).

## ● About volatile variables

To reduce code size and increase processing speed, recent C compilers have been designed whenever possible to minimize loads / stores to memory and to recycle values placed in the registers. Conversely, a description of memory access in C does not guarantee that memory is accessed at that point. This presents problems for statements that access I / O registers. To resolve this problem, ANSI defines a type of variable known as "volatile". Use this type of variable to access I / O registers.

```
sub2()
    {
      volatile char *vp;

      vp = (volatile char *)0x40000;
  c0005e: c000  ext     0x0
  c00060: d000  ext     0x1000
  c00062: 6c05  ld.w    %r5,0x0         xld.w     %r5,0x40000

00c00064 <.LM18>:
      *vp = 2;
  c00064: 6c24  ld.w    %r4,0x2         ld.w     %r4,0x2
  c00066: 3454  ld.b    [%r5],%r4       ld.b     [%r5],%r4     ← Access

00c00068 <.LM19>:

      *(volatile char *)(0x48000) |= 0x1;
  c00068: c200  ext     0x200
  c0006a: 6005  add     %r5,0x0         xadd      %r5,0x8000
  c0006c: b050  bset    [%r5],0x0       bset     [%r5],0x0     ← bset access

00c0006e <.LBE3>:
    }
  c0006e: 0640  ret                     ret
```

The variable "vp" is declared as a volatile type, and the address 0x40000 is set with 2 written to it. This ensures a write to memory.

Additionally, 0x1 is OR written to address 0x48000. Here, the immediate value 0x48000 is cast for handling as an address pointer. Using the volatile byte type to set or clear a bit generates the instructions bset and bclr, enabling processing with one instruction where three instructions may otherwise be required.

## ● About pointer variables

Access to a location pointed to by a pointer variable is processed with one instruction.

```
sub1(a,b)
    int a;
    int *b;
    {
      *b = a;
  c0005a: 3c76  ld.w    [%r7],%r6       ld.w     [%r7],%r6     ← Access by one instruction

00c0005c <.LM14>:
    }
  c0005c: 0640  ret                     ret
```

## ● About structure variables and arrays

Basically external or auto variables, structure variables and arrays are accessed in the same way as the external and auto variables previously discussed.

```
main()
    {
  c0001c: 840d  sub     %sp,0xd         sub      %sp,0xd
               :
               :
      st.s1 = 3;
  c0002a: 6c34  ld.w    %r4,0x3         ld.w     %r4,0x3
  c0002c: 5cb4  ld.w    [%sp+0xb],%r4   ld.w     [%sp+0xb],%r4 ← Accesses auto variable

00c0002e <.LM6>:
      ar[3] = 4;
  c0002e: 6c44  ld.w    %r4,0x4         ld.w     %r4,0x4
  c00030: 5c34  ld.w    [%sp+0x3],%r4   ld.w     [%sp+0x3],%r4 ← Accesses auto variable
               :
               :
      sub2();
  c0003c: c000  ext     0x0
  c0003e: c000  ext     0x0
  c00040: 1c0f  call    0xf             xcall     0x1e       (0x00C0005E)

00c00042 <.LM9>:

      gst.s2 = 5;
  c00042: 6c54  ld.w    %r4,0x5         ld.w     %r4,0x5
  c00044: c000  ext     0x0
  c00046: c004  ext     0x4
  c00048: 3cf4  ld.w    [%r15],%r4      xld.w     [%r15+0x4],%r4   ← Accesses external variable
```

Before performing an access, the C compiler converts each element of a structure or array into an offset relative to the SP when the element is an auto variable, or into an absolute address when the element is an external variable. Structures and arrays are thus handled in exactly the same way as ordinary auto and external variables.

## ● About pointer type structures and arrays

```
sub3(st, ar)
    struct ST *st;
    int ar[];
    {
      st->s2 = 4;
  c00070: 6c44  ld.w    %r4,0x4         ld.w     %r4,0x4
  c00072: c004  ext     0x4                                          ← Access as offset
  c00074: 3c64  ld.w    [%r6],%r4       xld.w     [%r6+0x4],%r4      ← Two instructions

00c00076 <.LM23>:
      ar[5]=5;
  c00076: 6c54  ld.w    %r4,0x5         ld.w     %r4,0x5
  c00078: c014  ext     0x14                                         ← Access as offset
  c0007a: 3c74  ld.w    [%r7],%r4       xld.w     [%r7+0x14],%r4     ← Two instructions

00c0007c <.LM24>:
    }
  c0007c: 0640  ret                     ret
```

When the pointer for an external variable structure or array is used as shown above, each element of the structure or array may be accessed with two instructions. (This is true for up to 4KB of access area, with a maximum offset of 13 bits. Larger areas require three instructions.) This technique effectively provides access to large external variable areas.

## ● Precautions to be taken when global symbols with the same name are defined

1) Even when two or more global symbols with the same name are defined in one C source file, the compiler does not assume an error or issue a warning. They are regarded as one symbol definition when processed by the compiler. If it is necessary to display a warning for such symbol definitions, specify the "-Wredundant-decls" option before compiling.

2) If global symbols with the same name but without an extern declaration are defined in different C source files, the linker does not assume an error for duplicate definitions. The symbols are regarded as the same address when processed by the linker.
Example:
(src1.c)
```
int      iSym1;
iSym1 = 1;
```
(src2.c)
```
int      iSym1;
iSym1 = 2;
```
Although the actual code is linked normally, the symbol iSym1 is recognized as existing in two instances in the .bss section, and a double sized space is reserved in the .bss section.
iSym1 = 4 bytes, so 4 bytes × 2 = 8 bytes in total

This causes an unused area (in this case, 4 bytes) to occur. To avoid this, always be sure to write extern declarations when external symbols are to be referenced.
Example:
(src1.c)
```
int      iSym1;
iSym1 = 1;
```
(src2.c)
```
extern   int      iSym1;
iSym1 = 2;
```

3) If constant symbols with the same name are defined in different C source files and a reference is made to one of those symbols, a value defined in the same source file is referenced.
Example:
(src1.c)
```
const sym1 = 1;
int i;

i = sym1;         /* Value 1 is assigned to i */
```
(src2.c)
```
const sym1 = 2;
int k;

k = sym1;         /* Value 2 is assigned to k */
```
Because coding such as this may cause a problem or failure, avoid definitions of the same symbol name as much as possible.

4) Symbols with exactly the same name may be defined in one file without causing a problem. For example, when two instances of a symbol with the same name are defined, two variable areas are reserved in memory. Note, however, that although one of the variables can be accessed normally, the other variable cannot be accessed at all. As this means that memory space is used wastefully, avoid defining symbols with the same name as much as possible.
Example:
```
int  siData1;

int  siData1;

sub() {
         siData1 = 1;
};
```
Although siData1 on one side can be accessed normally, siData1 on the other side cannot be accessed, resulting in wasteful use of .bss.

## ● Conclusion

The following lists recommendations for C code and code optimization in order of importance.

1) Use auto variables (variables in the stack) unless external variables (those with absolute addresses) are unavoidable.

2) Write external variables as structures or arrays, and access them as offset from the beginning pointer. This is generally effective for address ranges up to 4KB.

Whenever possible, use -O for the GCC33 optimize switch. Specifying -O2 or -O3 only results in special optimizing processing, without improving results.

The C compiler optimizes code generation according to one of the specified switches -O, -O2, or -O3. When generating code, the xgcc C compiler optimizes it by placing emphasis on code efficiency and speed (mainly code efficiency). The greater the value of -O, the higher the code efficiency. However, there is a greater possibility of causing a problem, such as absence of some debugging information in the output. If optimization cannot be executed normally, reduce the value of optimization. Normally, -O should be specified. The basic make file generated by the gwb33 work bench specifies -O option when invoking the xgcc C compiler.
When an optimization is specified, the xgcc C compiler reuses the value loaded from the memory to the register to reduce memory read / write operations. So, sometimes the memory may not be accessed. To avoid this situation, take measures as shown below.

- Declare variables with "volatile". Example) volatile char IO_port1;
- Do not specify the optimization.
- Use "-fvolatile". Pointers are accessed as volatile objects.
  Use "-fvolatile-global". External variables are all accessed as volatile objects.

## *2.7 Mapping by Linker*

Sections are mapped into memory based on a linker script file.
Use the -T command-line option of the ld linker to specify the linker script file.
Shown below is an example for a general script file using only the default data-area pointer, where xgcc option -mgdp=dp.

### ● Memory mapping

Define the data-area pointer at address 0x0, __dp = 0x0

Internal RAM

.gbss: Located beginning at 0x0
.gdata: Located beginning at the last address of .gbss (Load area follows .rodata)
.bss: Located beginning at the last address of .gdata
.data: Located beginning at the last address of .bss (Load area follows .gdata load area)

External ROM

.text in test2.o, test3.o: Located beginning at 0x600000

External ROM

.text (other than test2.o, test3.o): Located beginning at 0xc00000
.rodata: Located beginning at the last address of .text
.gdata load area (LMA): Located beginning at the last address of .rodata
Initial values must be copied to __start_gdata when executed
.data load area (LMA): Located beginning at the last address of .gdata
Initial values must be copied to __start_data when executed

Here, the boot vector is assumed to be present at the beginning of the .text section in boot.o.

### ● Linker script file

The linker script file should be created in the same manner as the one shown below.

Linker script file

```
OUTPUT_FORMAT("elf32-c33", "elf32-c33",
              "elf32-c33")
OUTPUT_ARCH(c33)
SEARCH_DIR(.);
SECTIONS
{
      __dp = 0x0;

      . = 0x0;
      .gbss   : { *(.gbss) }
      .gdata  :
               AT (ADDR (.text) + SIZEOF (.text) + SIZEOF (.rodata))
                        { __start_gdata = . ; *(.gdata); __end_gdata = . ; }
      .bss    : { *(.bss) }
      .data   :
               AT (LOADADDR (.gdata) + SIZEOF (.gdata))
               { __start_data = . ; *(.data); __end_data = . ; }

      . = 0x600000;
      outputa : {
               test3.o (.text)
               test2.o (.text)
      }

      . = 0xc00000;
      .text   : {
               boot.o(.text)                      ← (1)
               main.o(.text)
               C:/gnu33/lib/libc.a(.text)         ← (2)
               C:/gnu33/lib/libgcc.a(.text);
      }
      .rodata : { *(.rodata) }
}
```

(1) Explicitly specify that boot.o with a boot vector be located at the beginning of the section.

(2) If "*" cannot be specified as a file location in the section, the library will be forcibly added to any section. As it is possible that this added library will overlap another section, be sure to explicitly specify the location of the library as much as possible.

*Note: If a file name is specified without specifying "*" for the file location in the section, the sections for which "*" is specified thereafter become such that the last file specified is located at the beginning of the section.*

## ● TEXT, DATA, and BSS sections

Contents written in C and assembly sources are ultimately categorized into three sections.

| | |
|---|---|
| TEXT section | This section stores program code and ROM data. |
| DATA section | This section stores R/W'able data with initial values. |
| BSS section | This section stores R/W'able data without initial values. |

Example:

```
int a;                    ← Placed in the BSS section
int b=1;                  ← Placed in the DATA section
const int c=2;            ← Placed in the TEXT section

main()                    ← Program is placed in the TEXT section
    {
      a=b=c;
    }
```

Compiling the above results in the following.

```
      .global b
      .section .data
.stabs "b:G(0,1)",32,0,0,0
      .align 2
      .type     b,@object
      .size     b,4
b:
      .long 1                          ← b is data in the DATA section
      .global c
      .section .rodata
.stabs "c:G(0,1)",32,0,0,0
      .align 2
      .type     c,@object
      .size     c,4
c:
      .long 2                          ← c is data in the TEXT section
      .section .text
      .align 1
.stabs "main:F(0,1)",36,0,0,main
      .global main
      .type     main,@function
main:
.stabn 68,0,8,.LM1-main
.LM1:
.stabn 68,0,9,.LM2-main
.LM2:
      xld.w    %r4,2    ;0x2           ← All instructions are placed in the TEXT section
      ext doff_hi(b)
      ext doff_lo(b)
      ld.w     [%r15],%r4
      ext doff_hi(a)
      ext doff_lo(a)
      ld.w     [%r15],%r4
.stabn 68,0,10,.LM3-main
.LM3:
      ret
.Lfe1:
      .size     main,.Lfe1-main
.stabs "a:G(0,1)",32,0,0,0
      .global a
```

```
        .section .bss
        .align 2
        .type     a,@object
        .size     a,4
a:
        .zero    4                          ← a is placed in the BSS section
        .text
        .stabs "",100,0,0,Letext
Letext:
        .ident   "GCC: (GNU) 2.95.2 19991024 (release)"
```

Note that the classification of and directive commands for TEXT, DATA, and BSS incorporate UNIX concepts.
Support for DATA sections (R/W'able variables with initial values) varies by specific vendor-supplied development tool. Since some development tools do not support DATA sections, avoid using this section when creating a new source. For better portability, define data as BSS section variables and initialize them in the program as necessary.
When using C sources already developed on a PC, the DATA sections in the source may be left intact. When handling DATA sections as R/W'able data in a built-in system, you need to write the data into ROM and expand into RAM when booting. Some real-world examples are provided further below.

## ● Method for using the DATA section and caching the program to internal RAM

The DATA section is a R/W'able variable area with initial values. (For more information on each section, see "● TEXT, DATA, and BSS sections" above.) To use the DATA section, the following three conditions must be met.

1) The initial values of variables are written into ROM.
2) The data in ROM is expanded into RAM.
3) Program operation is based on the expanded into RAM.

By following a similar method, the program can be copied into the internal RAM for execution at high speed. If the program exists in external ROM or flash memory, one to two wait states may be incurred for access. However, when it is copied into the internal RAM, the program can be executed with zero wait states.
This procedure is illustrated using gnu33\sample\section.

### Method for specifying a linker command file

Example: Excerpt from section\section.lds

```
        .bss 0x00000000 :
          {
           __START_bss = . ;
           main.o(.bss) ;
           __END_bss = . ;
          }

        .data __END_bss : AT( __END_rodata )
          {
           __START_data = . ;         ← Section symbol indicating RAM area for data expansion
           *(.data) ;
           __END_data = . ;
          }
        __START_data_lma = LOADADDR( .data );   ← Section symbol indicating the storage area in ROM

        .cache __END_data : AT( __START_data_lma + SIZEOF( .data ) )
          {
           __START_cache = . ;        ← Section symbol indicating the program cache area in RAM
           cache.o(.text) ;
           __END_cache = . ;
          }
        __START_cache_lma = LOADADDR( .cache );← Section symbol indicating the storage area in ROM

        Share1 __END_cache :
          {
           __START_Share1 = . ;
           share1.o(.bss) ;
           __END_Share1 = . ;
          }
```

```
      Share2 __END_cache :
        {
         __START_Share2 = . ;
         share2.o(.bss) ;
         __END_Share2 = . ;
        }

      .text 0x00c00000 :
        {
         __START_text = . ;
         boot.o(.text)
         main.o(.text)
         c:/gnu33/lib/libgcc.a(.text) ;
         __END_text = . ;
        }

      .rodata __END_text :
        {
         __START_rodata = . ;
         *(.rodata) ;
         __END_rodata = . ;
        }
```

The linked map information can be confirmed using objdump -h.

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .bss          00000008  00000000  00000000  00001000  2**2
                  ALLOC
  1 .data         00000004  00000008  00c000d4  00003008  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .cache        0000003e  0000000c  00c000d8  0000300c  2**1
                  CONTENTS, ALLOC, LOAD, CODE
  3 Share1        00000192  0000004a  0000004a  00001042  2**2
                  ALLOC
  4 Share2        00000192  0000004a  0000004a  00001eb0  2**2
                  ALLOC
  5 .text         000000ce  00c00000  00c00000  00002000  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  6 .rodata       00000006  00c000ce  00c000ce  000020ce  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  7 .stab         00000618  00c000d4  00c000d4  0000304c  2**2
                  CONTENTS, READONLY, DEBUGGING
  8 .comment      000000be  00c00a78  00c00a78  00003664  2**0
                  CONTENTS, READONLY
  9 .stabstr      0000038c  00c006ec  00c006ec  00003722  2**0
                  CONTENTS, READONLY, DEBUGGING
```

### Transfer when booting

Transfer the data into RAM using the section symbols defined in the linker script file before starting the program, as shown below.

Example: Excerpt from section\boot.s

```
      .text
      .long BOOT                             // BOOT VECTOR

BOOT:
      xld.w    %r15,0x800
      ld.w     %sp,%r15                      // set SP 0x800
      xld.w    %r15, __dp                    // set data pointer
      // if you use Advanced Macro CPU, please use below source.
/*    xld.w    %r4, __dp                     // set data pointer
      ld.w     %dp,%r4                       // set data pointer        */

      // ROM has data at end of rodata; copy it
      xld.w    %r6, __START_data_lma         // .data LMA
      xld.w    %r7, __START_data             // .data VMA
      xld.w    %r8, __END_data               // end of .data VMA
      call     HCOPY_LOOP

      // transfer function cache_exec's program code to RAM
      xld.w    %r6, __START_cache_lma        // .cash LMA
      xld.w    %r7, __START_cache            // .cash VMA
      xld.w    %r8, __END_cache        // .cash VMA
      call     HCOPY_LOOP

      // start main

      xcall    main                    // goto main
      jp       BOOT                    // infinity loop

      ; copy %r6 addr to %r7 addr until %r8 addr

HCOPY_LOOP:
      ld.b     %r4,[%r6]+              // read byte from src addr
      ld.b     [%r7]+,%r4              // write byte to dest addr
      cmp      %r7,%r8                 //
      jrult    HCOPY_LOOP
      ret
```

**Writing in C**

When written in C, code similar to that shown below results.

```
extern char __START_cache_lma;
extern char __START_cache;
extern char __END_cache;

char *src = &__START_cache_lma
char *dst = &__START_cache;

while(dst < &__END_cache)
      *dst++ = *src++;
```

*Note: If the symbols generated by a linker script are referenced in C code, a linker error may result, depending on the symbol value. When the above "char *dst=&__START_cache;" is compiled, code similar to that shown below is generated.*

```
ext   doff_hi(__START_cache)
ext   doff_lo(__START_cache)
add   %r4,%r15
```

*Although no problems are encountered if the offset from the symbol (address) is within 26 bits in length, if the range is exceeded, the problem may occur that the address cannot be set correctly, which may result in a linker error. This restriction is specific to the S5U1C33001C, which supports data areas, and does not cause any problem in the GNU i386 compiler or the like.*

## ● Specifying a library

To specify the ANSI library (lib / libc.a) or the emulation library (lib / libgcc.a) provided for the S5U1C33001C in a linker script file, write a script like that shown below.

```
         :
         :
.text 0xc01000 :
{
   boot.o(.text)
   main.o(.text)
   C:/gnu33/lib/libc.a(.text)
   C:/gnu33/lib/libgcc.a(.text)
}
```

Make sure the library drive name and directory written here are exactly the same as the content specified in the make file. The upper / lower cases must also be the same.

## ● Precautions regarding section-name definitions in a linker script file

In a linker script file, always be sure to define the sections ".text" and ".data". If these two section names do not exist in the linked elf file, the debugger gdb will be unable to load that file.

Example:

```
TEST1 0xc00000 : { *.o (.text) }
TEST2 0xc01000 : { *.o (.data) }
```

For a definition like this, the ".text" and ".data" sections will not be output to the elf file.

## ● Examples of Linkage

Example 1) when the default data area only is used

The following is a linker script for minimum configuration:

Sample linker script file: sample\ldscript\default\default.lds

```
OUTPUT_FORMAT("elf32-c33", "elf32-c33",
              "elf32-c33")
OUTPUT_ARCH(c33)
SEARCH_DIR(.);
SECTIONS
{
      /* data pointer symbol By GWB33 */
      __dp = 0x0;                                      ... 1

      /* section information By GWB33 */
      . = 0x0;

      .bss 0x00000000 :                                ... 2
        {
         __START_bss = . ;
         *(.bss) ;
         __END_bss = . ;
        }

      .data __END_bss : AT( __END_rodata )             ... 3
        {
         __START_data = . ;
         *(.data) ;
         __END_data = . ;
        }
      __START_data_lma = LOADADDR( .data );

      .text 0x00c00000 :                               ... 4
        {
         __START_text = . ;
         *(.text) ;
         __END_text = . ;
        }

      .rodata __END_text :                             ... 5
        {
         __START_rodata = . ;
         *(.rodata) ;
         __END_rodata = . ;
        }

}
```

1. The default data-area pointer (__dp) is set to 0x0.
2. All .bss sections in the input files are located beginning with address 0x0 as a .bss section. These sections do not have an actual code, so it is not necessary to specify the load memory address.
3. The memory space immediately following the .bss section are allocated to the .data sections in the input files. The initial data (actual code) is located following the .rodata section. The LMA is specified by the AT statement. __END_rodata is the symbol defined in the .rodata command and it indicates the location counter value immediately following the .rodata section. __START_data and __END_data are defined to refer the start and end virtual memory addresses (VMA) of the .data section. They can be used to copy data from LMA to VMA in the initialize routine. For details on copying, refer to "Transfer when booting" in the paragraph "● Method for using the DATA section and caching the program to internal RAM".
4. All .text sections in the input files are located beginning with address 0xc00000.
5. All .rodata sections in the input files are located immediately following the .text section. __END_rodata is defined for specifying the LMA of the .data section.

Figure 2.7.1 shows the memory map configured with this sample script.

```
ld -o sample.elf file1.o file2.o -T default.lds
```

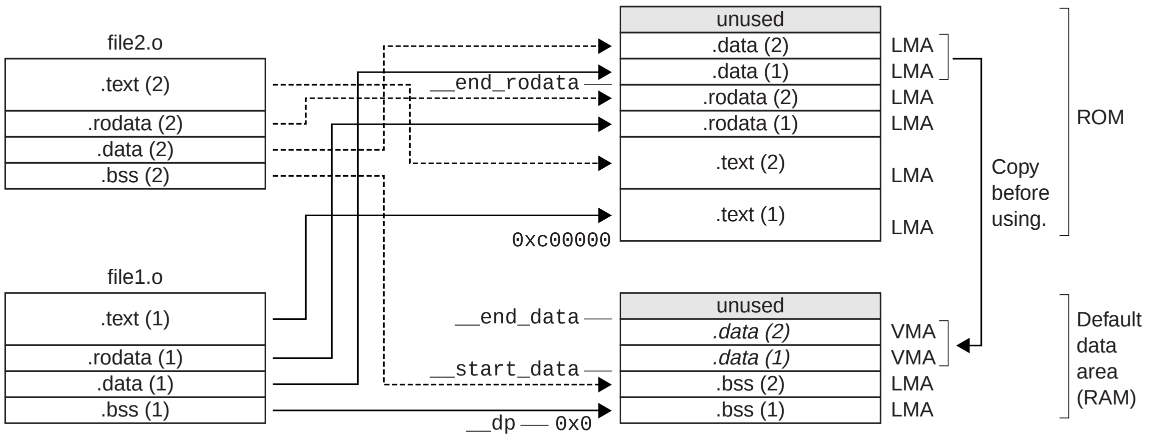


*Fig. 2.7.1 Memory map (Example 1)*

Example 2) when all data areas are used

The following is a sample linker script when all the data areas are used:

Sample linker script file: sample\ldscript\all_area\allarea.lds

```
OUOUTPUT_FORMAT("elf32-c33", "elf32-c33",
                "elf32-c33")
OUTPUT_ARCH(c33)
SEARCH_DIR(.);
SECTIONS
{

    __dp  = 0x0;
    __gdp = 0x0;
    __sdp = 0x1000;
    __tdp = 0x2000;
    __zdp = 0x3000;

      . = 0x0;
      .gbss : { *(.gbss) }
      .gdata  :
        AT (__load_gdata)
        { __start_gdata = . ; *(.gdata); __end_gdata = . ; }

      .bss : { *(.bss) }
      .data  :
        AT (__load_data)
        { __start_data = . ; *(.data); __end_data = . ; }

      . = 0x1000;
      .sbss : { *(.sbss) }
      .sdata  :
        AT (__load_sdata)
        { __start_sdata = . ; *(.sdata); __end_sdata = . ; }

      . = 0x2000;
      .tbss : { *(.tbss) }
      .tdata  :
        AT (__load_tdata)
        { __start_tdata = . ; *(.tdata); __end_tdata = . ; }

      . = 0x3000;
      .zbss : { *(.zbss) }
      .zdata  :
        AT (__load_zdata)
        { __start_zdata = . ; *(.zdata); __end_zdata = . ; }

      . = 0xc00000;
      .text    : { *(.text)   }
      .rodata  : { *(rodata)  }
      .rosdata : { *(rosdata) }
      .rotdata : { *(rotdata) }
      .rozdata : { *(rozdata) }

      __load_data = . ;
```

```
        __load_gdata = . + SIZEOF(.data) ;
        __load_sdata = . + SIZEOF(.data) + SIZEOF(.gdata) ;
        __load_tdata = . + SIZEOF(.data) + SIZEOF(.gdata) + SIZEOF(.sdata);
        __load_zdata = . + SIZEOF(.data) + SIZEOF(.gdata) + SIZEOF(.sdata)
                         + SIZEOF(.tdata);
}
```

Setting of each data area is the same as that of the default data area in Example 1 except for the section name and defined symbols.
The section map is shown in Figure 2.7.2.

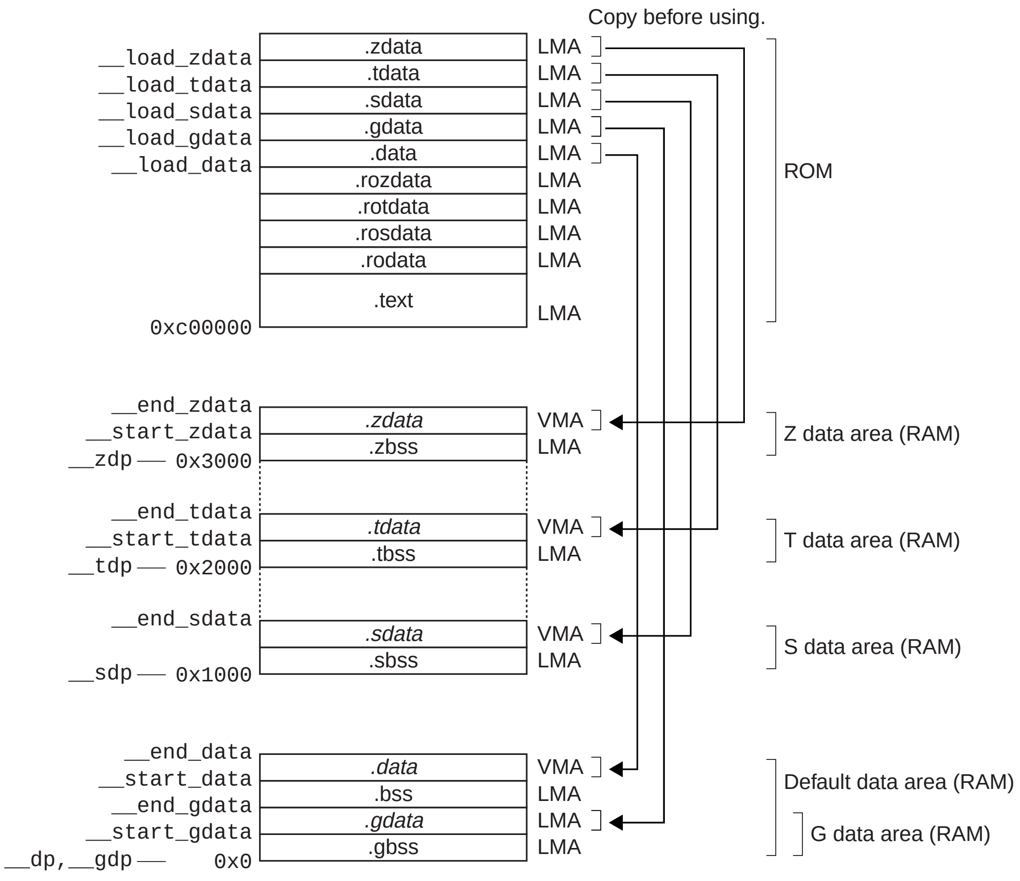


*Fig. 2.7.2 Memory map (Example 2)*

Example 3) when virtual and shared sections are used

The following is a sample linker script when virtual and shared sections are used:

A linker script that uses virtual and shared sections

```
OUTPUT_FORMAT("elf32-c33", "elf32-c33",
              "elf32-c33")
OUTPUT_ARCH(c33)
SEARCH_DIR(.);
SECTIONS
{
      /* data pointer symbol By GWB33 */
      __dp = 0x0;

      /* section information By GWB33 */
      . = 0x0;

      .bss 0x00000000 :
        {
         __START_bss = . ;
         *(.bss) ;
         __END_bss = . ;
        }

      .data __END_bss : AT( __END_rodata )
        {
         __START_data = . ;
         *(.data) ;
         __END_data = . ;
        }
      __START_data_lma = LOADADDR( .data );

      .text_foo1 __END_data : AT( __START_data_lma+SIZEOF( .data )
        {
         __START_text_foo1 = . ;
         foo1.o(.text) ;
         __END_text_foo1 = . ;
        }
      __START_text_foo1_lma = LOADADDR( .text_foo1 );

      .text_foo2 __END_data : AT( __START_text_foo1_lma+SIZEOF( .text_foo1 )
        {
         __START_text_foo2 = . ;
         foo2.o(.text) ;
         __END_text_foo2 = . ;
        }
      __START_text_foo2_lma = LOADADDR( .text_foo2 );

      .text_foo3 __END_data : AT( __START_text_foo2_lma+SIZEOF( .text_foo2 )
        {
         __START_text_foo3 = . ;
         foo3.o(.text) ;
         __END_text_foo3 = . ;
        }
      __START_text_foo3_lma = LOADADDR( .text_foo3 );

      .text 0x00600000 :
        {
         __START_text = . ;
         *(.text) ;
         __END_text = . ;
        }

      .rodata __END_text :
        {
         __START_rodata = . ;
         *(.rodata) ;
         __END_rodata = . ;
        }
}
```

The section map is shown in Figure 2.7.3

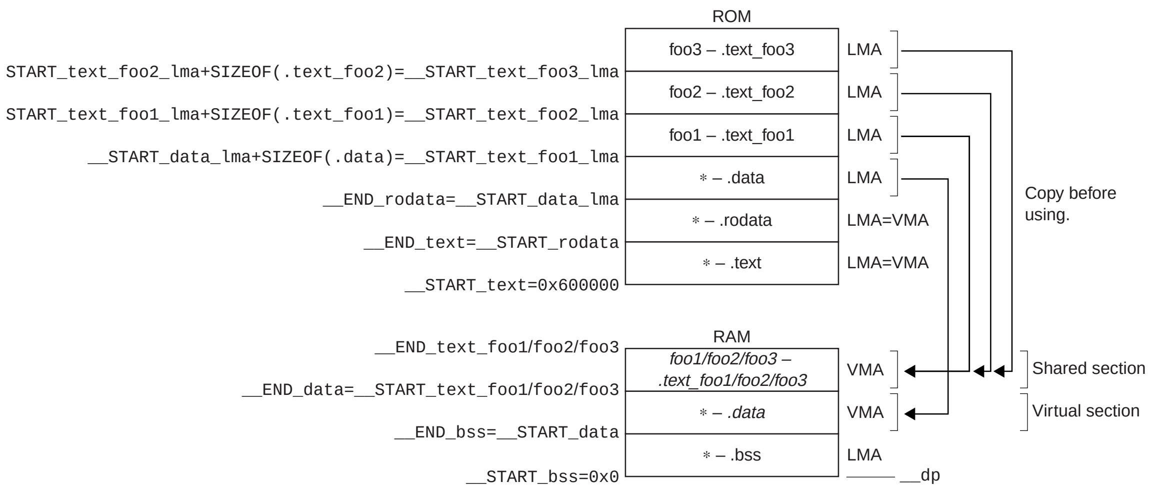

ROM
foo3 – .text_foo3
foo2 – .text_foo2
foo1 – .text_foo1
* – .data
* – .rodata
* – .text
LMA
LMA
LMA
LMA
LMA=VMA
LMA=VMA
Copy before using.
START_text_foo2_lma+SIZEOF(.text_foo2)=__START_text_foo3_lma
START_text_foo1_lma+SIZEOF(.text_foo1)=__START_text_foo2_lma
__START_data_lma+SIZEOF(.data)=__START_text_foo1_lma
__END_rodata=__START_data_lma
__END_text=__START_rodata
__START_text=0x600000
RAM
foo1/foo2/foo3 – .text_foo1/foo2/foo3
* – .data
* – .bss
VMA
VMA
LMA
__dp
Shared section
Virtual section
__END_text_foo1/foo2/foo3
__END_data=__START_text_foo1/foo2/foo3
__END_bss=__START_data
__START_bss=0x0

*Fig. 2.7.3 Memory map (Example 3)*

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. .text sections in other files than these three files are located in the .text section beginning with 0x600000 and are executed at the stored address in the ROM.

## 2.8 Libraries

This C compiler package contains the following three types of libraries:

| | |
|---|---|
| libc.a: | ANSI library |
| libgcc.a: | Emulation library |
| libgccP.a: | Emulation library for high-precision arithmetical operations |

The libraries are included in the \gnu33\lib directory.
The emulation libraries libgcc.a and libgccP.a are the standard libraries called by the xgcc C compiler, so as a rule be sure to link them.
The ANSI library libc.a may be linked as necessary.

### 2.8.1 ANSI Library (libc.a)

The ANSI library is created observing the following rules with respect to usage and protection. When programming to call the libraries, take these rules into consideration.

Registers used in the library

- The registers R0 to R11 are used.
- The registers R12 to R15 are not used.
- The registers R0 to R3 are protected by saving to the stack before execution of a function and by restoring from the stack after completion of the function.

Data areas used in the library

- The variables and other data referenced in the library are located in the default data area. In the library, R15 is referenced as the default data-area pointer. For this reason, the R15 register must be initialized before using the library.
- In the library, only the default data area is used. The other data areas (G, S, T, and Z) are unused.

### 2.8.2 Emulation Libraries (libgcc.a, libgccP.a)

The emulation libraries are created observing the following rules with respect to usage and protection. When programming to call the libraries, take these rules into consideration.

Registers used in the libraries

- The registers R0 to R14 are used.
- The register R15 is not used.
- The registers R0 to R3 are protected by saving to the stack before execution of a function and by restoring from the stack after completion of the function. In libraries using R10 through R14 as well, these registers are protected by saving to the stack.

Data areas used in the libraries

No data areas are used in the emulation libraries.

### 2.8.3 Notes on Using Libraries in Advanced Macros

When compiling for the advanced macro (-mc33adv option is used), make sure R15 and DP are initialized with the same value.
The application program references symbols in the default data area using DP, while the ANSI library uses R15, not DP. Therefore, DP and R15 must hold the same value in order to maintain consistency.
This is because the same libraries (libgcc.a, libc.a) are used even in the advanced macro. (No libraries exclusively for the advanced macro are available.)

### 2.8.4 Interrupt Mask Cycles in Emulation Libraries

In the emulation libraries libgcc.a and libgccP.a, some functions use the registers provided for data areas (R12–R15). If an interrupt occurs during the execution of one of these functions, the program becomes unable to access the data area normally in an interrupt handler routine. Therefore, interrupts are masked during function execution.
The following shows the functions for which interrupts are masked, and the number of interrupt mask cycles.

| Function name | Number of mask cycles |
|---|---|
| __mulsf3 | 163 |
| __muldf3 | 184 (180) |
| __floatsisf | 76 |
| __floatsidf | 81 |
| __extendsfdf2 | 63 |
| __divsf3 | 158 |
| __divdf3 | 208 (220) |
| __adddf3 | 163 (177) |
| __subdf3 | 163 (177) |

Numbers in ( ) indicate the number of mask cycles for the libgccP.a functions.

As interrupts are masked in the emulation library, if an interrupt occurs during execution of the library, the execution speed of the program may decrease. The corrective measure to be taken in such a case is described below.

(1) From the source in /utility/lib_src/emulib, to recreate the library, delete the lines in which interrupts are masked.
(2) In the interrupt handler routine, add a process to reset the data pointers, as shown below.

| | |
|---|---|
| At the start of a function: | Push R12–R15 onto the stack |
| | Reset the data-area pointers (R12–R15). |
| | : |
| | Access the data area |
| | : |
| At the end of the function: | Pop R12–R15 off the stack. |

However, because the amount of code increases before and after the interrupt handling, the processing speed of the interrupt handler itself will decrease. Please examine the performance of the program through comparison with a case in which an existing library is used.

### 2.8.5 Precautions to Be Taken When Adding a Library

To add a library other than the ANSI and emulation libraries when linking the program, specify the files passed to the linker ld in the order shown below.

```
ld.exe  (program.o)  (Added library)  (ANSI library)  (Emulation library)
```

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.lds -o withmylib.elf boot.o libc.a libgcc.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.lds -o withmylib.elf boot.o mylib.a libc.a libgcc.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.lds -o withmylib.elf boot.o lib3.a lib2.a lib1.a libc.a libgcc.a
```

The make files output by the work bench gwb33 have a LIBS macro for specifying a library. Libraries can also be added by editing this macro.
Example: To add c:\myapp\lib\mylib.a

```
LIBS= $(LIB_DIR)/libc.a $(LIB_DIR)/libgcc.a
                                ↓
LIBS= c:/myapp/lib/mylib.a $(LIB_DIR)/libc.a $(LIB_DIR)/libgcc.a
```

*Note: Make sure "\" is converted to "/". TOOL_DIR and LIB_DIR are the only macro names that the work bench can recognize and replace in the LIBS macro. When using a work bench linker script, do not use any other macros in the LIBS macro.*

If the LIBS macro has been altered, synchronism between the make files and the linker script files (*.lds) will be lost. Therefore, always be sure to perform the updating operation from the linker script editor in the work bench gwb33. To perform this update, call up the linker script editor and click on OK. The associated files (lds, ldt) will thereby be overwritten. Be aware that the contents of linker script files that have been manually altered will be lost.

## *2.9 Differences between the S5U1C33001C and the S5U1C33000C*

The S5U1C33001C has a GNU-based common interface, allowing users to develop programs with greater flexibility and efficiency than with the conventional development environment, the S5U1C33000C.
The GNU-based common interface allows the assembler or linker development procedures in other development environments to be used directly as they are. Development in Linux is also possible. (Although the Linux version is outside the scope of the guarantee with regard to its operation, the same sources as in the Windows version are used to compile tools.)
The S5U1C33001C can handle the instructions available exclusively for the advanced macro CPU, making it capable of powerful assembler programming. The advanced macro CPU can, in effect, access 4G bytes of memory space using the same number of instructions as for the code generated by the S5U1C33000C.
In the S5U1C33000C, memory areas only up to 64M bytes from the data-area pointer could be accessed, whereas in the S5U1C33001C, because multiple data-area pointers have been introduced, even the standard macro CPU can access a maximum of $64 \times 4 = 256$M bytes of space.

### ● Enhanced functions of the S5U1C33001C

#### Compiler

- Many and varied user requirements can be satisfied as a result of the data-area pointers added.
  - In the S5U1C33001C, data-area pointers from %r10 to %r15 can be used for -mdp=6 (maximum), or data-area pointers from %r12 to %r15 can be used for -mdp=4 (recommended, default). (In the S5U1C33000C, only %r8 could be used through specification of the -gp option.)
  - The data-area pointer extension options make the respective areas accessible in up to 64MB (except for __gdp). At a maximum, 64MB × 4 = 256MB of memory space can be accessed (conventionally limited to 64MB). In advanced macro mode, any area in the entire 32-bit address space (4GB) can be accessed.
  - Programs can be compiled with high code efficiency close to that of two-pass make in the S5U1C33000C, if so specified using an option.
- High-speed dispatch for ROS is made possible by changing register assignments.
- The optimization routine has been enhanced (by upgrading the development base gcc version).
- Diverted use of many GNU free sources is possible.
- Interrupt handlers can be written in C. Routines for saving registers when an interrupt has occurred are also automatically generated.

#### Binutils

- GNU-based binutils can be used.
  - A number of optional functions intrinsic to GNU, such as the assembler, linker, make, objdump, and objcopy, can be used. (For the S5U1C33000C, dedicated tools such as Hex33 (objcopy), dis33 (objdump), and dmp33 (objdump) are used.)
- Simple extended instructions have been adopted.
  - The codes output by the compiler operate at high speed.
  - The correspondence between the assembler and source is easily understood.
  - Functions equivalent to those of the linker command ver.3 of the S5U1C33000C can be specified in a linker script of the S5U1C33001C. Furthermore, a wide variety of functions are available for use.

#### Work bench

- The work bench enables the visual allocation of files.

#### Debugger

- Compatible with the S5U1C33000H (ICD ver.2.0)
  - The software / hardware breaks, execution trace, and data breaks supported by db33 are fully supported.
  - Commands in c33 XXX format are partly compatible with conventional db33 commands.

- Fully compatible with the S5U1C33001H (ICD ver.3.0)
  - The target program can be downloaded at high speed through USB communication.
  - Area breaks are usable (advanced macro CPU only).
  - Bus breaks and bus traces are usable (advanced macro CPU only).
  - Compatible with flash writer mode of the S5U1C33001H (ICD ver.3.0). (The debugger can be used singly as a flash writer, with no need for any other tool.)
  - The amount of information handled in a trace has been doubled.
  - If the target is the advanced macro, the area break, bus break, and bus trace functions can be used.
- A simulator compatible with both standard and advanced macro modes is included.
- Symbol watch that makes use of expressions can be used.
  - It has been made possible to display the contents of structure members, reference memory symbols using numeric expressions, and watch local variables, all of which were impossible with db33.
- The line of debugging-use step commands has been expanded.
  - The newly introduced command "finish" enables the program to be stepped through until the end of a function.
- The newly adopted "elf" format facilitates the extraction of debugging information.
  - Due to the adoption of the "elf" file format, information can easily be acquired from binary data (through the use of objdump.exe).
- An interface common to GNU gdb has been adopted.
  - The debugger can be used by anyone who does not have sufficient knowledge of the S5U1C33001C tool.
- A simple program can be created in a command file.
- Text mode is included (outside the scope of the guarantee).
  - The --nw option enables debugging in CUI mode.
- The S5U1C33104H (ICE33) is not supported.
- Easy source-level debugging
  - Programs transferred into the internal RAM can also be debugged easily.

### Tool general

- The software (GNU tools) has been updated periodically and debugged by many programmers, making it easy to maintain.
- A number of types of embedded-use software have been developed based on GNU tools.
  - Toppers, TINET, Linux, Husion TCP / IP, etc.
- The tools can be used to some extent by anyone who does not have knowledge of the S5U1C33000C tool, provided that he or she is familiar with GNU tools.
  - The Q&A on GNU tools available at Websites can be used for reference.
- All sources are supplied. They can be customized to suit the user application (outside the scope of the support and guarantee).

*Table 2.9.1 Compatibility table of tools*

| Function | S1C33 Family software tools | | S1C33 Family hardware tools | | | |
|---|---|---|---|---|---|---|
| | S5U1C33000C | S5U1C33001C | S5U1C33000H ver.1 | S5U1C33000H ver.2 | S5U1C33000H ver.3 | S5U1C330M1D1 |
| Usefulness for standard core | ○ | ○ | ○ | ○ | ○ | ○ |
| Usefulness for advanced core | × | ○ | × | × | ○ | × |
| Serial connection | ○ | ○ | ○ | ○ | × | ○ |
| Parallel connection | ○ | ○ | ○ | ○ | × | × |
| USB connection | × | ○ | × | × | ○ | × |
| CPU clock upper limit | | | 50 MHz | 60 MHz | 60 MHz | 60 MHz |
| Trace function of debugger | | | ○ | ○ | ○ | × |
| Bus break function of debugger | | ○ | × | × | ○ | × |
| Bus trace function of debugger | | ○ | × | × | ○ *1 | × |
| Area break function of debugger | | ○ | × | × | ○ *1 | × |
| Forcible break function of debugger | | | ○ | ○ | ○ *1 | × |
| Debug monitor (S5U1C331M2S) | | | × | × | × | ○ *2 |
| Reset function of hardware tool | | | × | × | ○ | ○ |
| S5U1C33000H(ICD33 ver.2) | ○ | ○ | | | | |
| S5U1C33001H(ICD33 ver.3) | × | ○ | | | | |
| S5U1C330M1D1 | ○ | ○ | | | | |

○: Compatible ×: Not compatible
*1 Possible only for the advanced core chips
*2 The S5U1C330M1D1 requires a dedicated debug monitor (S5U1C331M2S).

The S5U1C33001C provides higher performance by up to approximately 25% compared to the S5U1C33000C. The following shows the execution time required for each tool when the source shown below is executed after being compiled by each tool.
S5U1C33000C: 18926.80 µs
S5U1C33001C: 14255.55 µs

The execution time required by the S5U1C33001C is approximately 3/4 that of the S5U1C33000C, as shown above, indicating that the execution speed is approximately 25% higher. Furthermore, the code size for the S5U1C33001C is also slightly smaller.
S5U1C33000C: 274 bytes (after two-pass make)
S5U1C33001C: 268 bytes

### Conditions

| | |
|---|---|
| CPU clock: | 20 MHz |
| External RAM access: | One wait state |
| Compilation: | For the S5U1C33000C, the source was compiled by two-pass make<br>For the S5U1C33001C, the option "-O2 -mdp=1, -mgda=8192, -fno-builtin -mno-memory" was specified |

### The source file used for evaluation

```
#define MAT_ARG1        3
#define MAT_ARG2        3

#define MAT_ARRAYSIZE   50
struct stMatrix {
    long          lData[ MAT_ARG1 ][ MAT_ARG2 ];
};

// proto-types
void vfnInitMatrix( struct stMatrix *stClear );
void vfnMuliMatrix( struct stMatrix *stRet,
            struct stMatrix *stLeft,
            struct stMatrix *stRight );

struct stMatrix stTestData1[ MAT_ARRAYSIZE ], stTestData2[ MAT_ARRAYSIZE ];
struct stMatrix stRet[ MAT_ARRAYSIZE ];

int main()
    {
        int nCnt, nCnt2, nCnt3;

        // initialize matrix
        for( nCnt = 0 ; nCnt < MAT_ARRAYSIZE ; nCnt++ ) {
            vfnInitMatrix( &stTestData2[ nCnt ] );
        }

        // set each randam value
        for( nCnt = 0 ; nCnt < MAT_ARRAYSIZE ; nCnt++ ) {

            for( nCnt2 = 0 ; nCnt2< MAT_ARG1; nCnt2++ ) {
                for( nCnt3 = 0 ; nCnt3 < MAT_ARG2; nCnt3++ ) {
                    stTestData1[ nCnt ].lData[ nCnt2 ][ nCnt3 ] = nCnt + nCnt2 + nCnt3;
                }
            }
        }

        // mutiply
        for( nCnt = 0 ; nCnt < MAT_ARRAYSIZE ; nCnt++ ) {
            vfnMuliMatrix( &stRet[ nCnt ],
                     &stTestData1[ nCnt ], &stTestData2[ nCnt ]  );
        }

    return 0;
}

void vfnInitMatrix( struct stMatrix *stClear )
{
    int nCnt, nCnt2;
    long lBuf;

    for( nCnt = 0 ; nCnt< MAT_ARG1; nCnt++ ) {
        for( nCnt2 = 0 ; nCnt2 < MAT_ARG2; nCnt2++ ) {
            if ( nCnt == nCnt2 )
                lBuf = 1;
            else
                lBuf = 0;
            stClear->lData[ nCnt ][ nCnt2 ] = lBuf;
        }
    }
}
void vfnMuliMatrix( struct stMatrix *stRet,
            struct stMatrix *stLeft,
            struct stMatrix *stRight )
{
    int nCnt, nCnt2, nCnt3;
    long lBuf;
    // multiple each matrix
    for( nCnt = 0; nCnt < MAT_ARG1 ; nCnt++ ) {         //y
        for( nCnt2 = 0; nCnt2 < MAT_ARG2 ; nCnt2++ ) {      //x
            lBuf= 0;
            for ( nCnt3 = 0 ; nCnt3 < MAT_ARG1 ; nCnt3++ ) {
                lBuf +=  ( stLeft->lData[ nCnt ][ nCnt3 ] )
                        *
                         ( stLeft->lData[ nCnt3 ][ nCnt2 ] );
            }
            stRet->lData[ nCnt ][ nCnt2 ] = lBuf;
        }
    }
}
```

The above content is merely an example. A significant performance improvement over the S5U1C33000C cannot be expected in all cases.

## *2.10 Transporting the S5U1C33000C Assets*

Following is a description of how to transport the S5U1C33000C assets to S5U1C33001C, using several examples by way of explanation.
The differences between the S5U1C33000C and S5U1C33001C packages are summarized in Table 2.10.1.

*Table 2.10.1 Differences between the S5U1C33000C and S5U1C33001C packages*

| Tool | S5U1C33000C | S5U1C33001C |
|---|---|---|
| make | **make.exe** (S1C33 original) | **make.exe** (GNU)<br>Environment path must be set before all functions can be used |
| Compiler | **gcc33.exe** (GNU ANSI C) | **xgcc.exe** (GNU ANSI C)<br>• Directives for S1C33<br>• Data areas supported<br>• Advanced macro instructions supported |
| Preprocessor | **pp33.exe** (S1C33 original) | **cpp.exe** (GNU) |
| Assembler | **as33.exe** (S1C33 original)<br>• Global pointer supported (GP)<br>• S1C33 original assembler directives | **as.exe** (GNU)<br>• Data-area-compatible pseudo-operand<br>• Directives for S1C33<br>• Advanced macro instructions supported<br>• Expansion of extended instructions |
| Extended instructions | **ext33.exe** (S1C33 original)<br>• Expansion of extended instructions<br>• Two-pass make | Instruction extender disused by reviewing the extended instructions (two-pass make also disused) |
| Linker | **lk33.exe** (S1C33 original) | **ld.exe** (GNU)<br>• Data area supported |
| Linker mapping | Command file (`cm`) | Linker script file (`lds`) |
| ANSI library | `io.lib`, `lib.lib`, `math.lib`, `string.lib`, `ctype.lib` | `libc.a` |
| Emulation library | `fp.lib`, `idiv.lib`, `fpp.lib` | `libgcc.a`, `libgccP.a` |
| Librarian | **lib33.exe** (S1C33 original) | **ar.exe** (GNU) |
| Debugger | **db33.exe** (S1C33 original)<br>SIM<br>ICE33<br>ICD33 Ver. 2.x<br>MON33<br>MEM33 | **gdb.exe** (S1C33 original)<br>SIM<br>ICD33 Ver. 2.x, Ver. 3, or later<br>MON33 |
| HEX converter | **hex33.exe** (S1C33 original) | **objcopy.exe** (GNU)<br>objcopy.exe usable for multiple purposes |
| Disassemble | **dis33.exe** (S1C33 original) | **objdump.exe** (GNU)<br>Possible with objdump -S -d |
| Dump tool | **dmp33.exe** (S1C33 original) | **objdump.exe** (GNU)<br>Possible by specifying the objdump option |
| Replacement tool | **sed.exe** | **sed.exe** |
| Work bench | **wb33.exe** (S1C33 original) | **gwb33.exe** (S1C33 original)<br>Linker input file can be edited |

The following lists the files that need to be modified when transported.
• Makefiles (*.mak)
• Initialize processing
• C source files (*.c)
• Assembler source files (*.s)
• Linker command files (*.cm)
• Debugger parameter files (*.par)

### 2.10.1 Transporting Makefiles (*.mak)

The locations to be modified in the makefiles are the lines in which tools are mentioned, as almost all tools are different with S5U1C33001C.

| | |
|---|---|
| • Directory path description | \ → / (slash) |
| • Tool directory | C:\cc33 → C:/gnu33 |
| • Compiler name | gcc33 → xgcc |
| • Assembler name | as33 → as |
| • Linker name | lk33 → ld |
| • Librarian name | lib33 → ar |
| • make | Delete two-path makes, if any. |
| • Delete pp33 and ext33. | |
| • Option flags in each tool | Modify to S5U1C33001C specifications. |

#### ● Example of modifying make file

Makefile for S5U1C33000C

```
TOOL_DIR = C:\cc33
GCC33 = $(TOOL_DIR)\gcc33
PP33  = $(TOOL_DIR)\pp33
EXT33 = $(TOOL_DIR)\ext33
AS33  = $(TOOL_DIR)\as33
LK33  = $(TOOL_DIR)\lk33
LIB33 = $(TOOL_DIR)\lib33
MAKE  = $(TOOL_DIR)\make
SRC_DIR =

# macro definitions for tool flags

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

# dependency list

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

boot.ms : $(SRC_DIR)boot.s
      $(PP33) $(PP33_FLAG) $(SRC_DIR)boot.s
      $(EXT33) $(EXT33_FLAG) boot.ps
boot.o : boot.ms
      $(AS33) $(AS33_FLAG) boot.ms

main.ms : $(SRC_DIR)main.c
      $(GCC33) $(GCC33_FLAG) $(SRC_DIR)main.c
      $(EXT33) $(EXT33_FLAG) main.ps
main.o : main.ms
      $(AS33) $(AS33_FLAG) main.ms

# clean files except source

clean:
      del *.srf
      del *.o
      del *.ms
      del *.ps
```

Makefile for S5U1C33001C

```
TOOL_DIR = C:/gnu33
CC = $(TOOL_DIR)/xgcc
AS = $(TOOL_DIR)/as
LD = $(TOOL_DIR)/ld
RM = $(TOOL_DIR)/rm
SRC_DIR =

# macro definitions for tool flags

CFLAGS= -B$(TOOL_DIR)/ -c -gstabs -O -mgda=0 -mdp=1 -mlong-calls -I$(TOOL_DIR)/
        include -fno-builtin
ASFLAGS = -B$(TOOL_DIR)/ -xassembler-with-cpp -Wa,--gstabs
LDFLAGS = -Tsample.lds -N -Map test.map

# macro definitions for library files

LIBS = $(LIB_DIR)/libc.a $(LIB_DIR)/libgcc.a

# dependency list

test.elf : $(OBJS)
      $(LD) $(LFLAGS) -o test.elf $(OBJS) $(LIBS)

boot.o : $(SRC_DIR)boot.s
      $(AS) $(AFLAGS) -o boot.o boot.s

main.o : $(SRC_DIR)main.c
      $(CC) $(CFLAGS) main.c

# clean files except source

clean:
      $(RM) *.o
      $(RM) *.elf
```

### 2.10.2 Initialize Processing

The instructions in the boot processing that sets R8 (global pointer) need to be changed to data-area-pointer settings. Normally, set R15 as the address for the default data-area pointer.
To use G, S, T, or Z data area, also set R12, R13, or R14 accordingly.
Furthermore, the data-area pointers may be initialized using data area symbol names, as shown below.

```
xld.w %r12,__sdp         ;set S data area pointer
xld.w %r13,__tdp         ;set T data area pointer
xld.w %r14,__zdp         ;set Z data area pointer
xld.w %r15,__dp          ;set default data area pointer
```

### 2.10.3 Transporting C Source Files (*.c)

Because both gcc33 (S5U1C33000C) and xgcc (S5U1C33001C) are compliant with ANSI C, their basic grammars are the same, except that xgcc has several added command-line options and reserved words.

- Global variables and the like in sizes less than 4 bytes are located in the G data area by default. Although the G data area has its upper-limit size fixed at 8KB, since the G data area can be accessed using only two instructions (compared to three instructions for the default data area), it is advisable that the largest possible number of variables be located in the G data area, in order to reduce code size. For this purpose, set the sizes of data to be placed in the G data area using the -mgda option.
  Example: Locate the int variable bar in the G data area
  ```
        int bar;
        - - - - - - - -
        >xgcc -S -O -mgda=4 sample.c
  ```
- To change the data areas, use the reserved word __attribute__.
  Example: Locate the int variable bar in the S data area
  ```
        int __attribute__((sda)) bar;
  ```

- Interrupt handler functions
  1) In S5U1C33000C, C interrupt handler functions are created using the interrupt filter utility (c33\utility\filter_int). In S5U1C33001C, interrupt handler functions can be implemented by declaring a function prototype with __attribute__ ((interrupt_handler)).

     Example: If you want the function foo to be an interrupt function, declare the following function prototype.

     ```
     void foo(void) __attribute__ ((interrupt_handler));
     ```
  2) The "asm("reti");" line within the interrupt function can be deleted by declaring the function with __attribute__ ((interrupt_handler)) the same way as in 1).

## *2.10.4 Transporting the Assembler Source Files (*.s)*

### ● Registers

The differences between the ways in which registers are used in S5U1C33000C and S5U1C33001C are summarized in Table 2.10.4.1.
The assembler source lines must be corrected to take these differences into account. However, if corrections are made simply by changing register numbers (for "pushn %rs", "popn %rs", and so on) unintended effects will result. Therefore, caution must be used when making corrections.

*Table 2.10.4.1 Method of using general-purpose registers*

| Register | S5U1C33000C | S5U1C33001C |
|---|---|---|
| R0<br>R1<br>R2<br>R3 | Registers that need have to their values saved when calling a function | Registers that need have to their values saved when calling a function |
| R4<br>R5 | Scratch registers | Register for storing returned values<br>Register for storing returned values |
| R6<br>R7 | | Register for passing argument (1st word)<br>Register for passing argument (2nd word) |
| R8 | Global pointer | Register for passing argument (3rd word) |
| R9 | For ext33 use | Register for passing argument (4th word) |
| R10<br>R11 | Register for storing returned values<br>Register for storing returned values | Scratch register/unused<br>Scratch register/unused |
| R12<br>R13<br>R14<br>R15 | Register for passing argument (1st word)<br>Register for passing argument (2nd word)<br>Register for passing argument (3rd word)<br>Register for passing argument (4th word) | S data-area pointer register or scratch register<br>T data-area pointer register or scratch register<br>Z data-area pointer register or scratch register<br>Default data-area pointer register |

### ● Comments

When using cpp, change ";" with "//" or "/* ... */". An error results if a comment beginning with ";" contains a quote only.

Source for S5U1C33000C

```
.code
.ascii "ABCD"                 ← OK
ld.w   %r0,1  ;'              ← Error occurs in cpp
ld.w   %r1,2  ;'A'            ← OK
ld.w   %r2,3  ;"ABC"          ← OK
ld.w   %r3,4  ;"              ← Error occurs in cpp
```

Source for S5U1C33001C

```
.text
.ascii "ABCD"                 ← OK
ld.w   %r0,1  //'             ← OK
ld.w   %r1,2  //'A'           ← OK
ld.w   %r2,3  /*"ABC"*/       ← OK
ld.w   %r3,4  /*"    */       ← OK
```

## ● Section definition directives

The section definition directives are compared between as33 and as in Table 2.10.4.2.

*Table 2.10.4.2 Section definition directives*

| **as33** (S5U1C33000C) | **as** (S5U1C33001C) |
|---|---|
| `.code` | `.text` |
| `.data` | `.data` |
| `.comm` | `.global` + `.bss`, `.xbss` (*x* = g, s, t, z) |
| `.lcomm` | `.bss`, `.xbss` (*x* = g, s, t, z) |

### 1) .code

Change to the .text section.

### 2) .data

Can be used as is.

### 3) .comm

For global specification in as, use the .global and .bss (.*X*bss) directives.
Example:
**as33** (S5U1C33000C)

```
      .comm    symbol 4
```

**as** (S5U1C33001C)

```
      .section .bss
      .global  symbol
      .align   2
   symbol:
      .skip    4
```

Pay attention to the alignment of symbols when changing .comm directives:
In as33, symbols are located at a boundary address according to the data size.
In as, the symbol alignment must be specified using the .align directive like the example above.
Although it can be changed to the directive command of as ".comm symbol, size" without causing a problem in transporting of the source file, keep the following points in mind when making the change.

Differences in alignment by symbol size

**as33** (S5U1C33000C)

| Size | Alignment (in byte units) |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 or more | 4 |

**as** (S5U1C33001C)

| Size | Alignment (in byte units) |
|---|---|
| 1 | 1 |
| 2–3 | 2 |
| 4–7 | 4 |
| 8–15 | 8 |
| 1 or more | 16 |

Order in which symbols are located after linking
In as33, symbols are located in memory in the order in which they are written in the assembly source, whereas in as they are not always located in the order of writing.
Example: When a .comm section is located beginning with the address 0x10000

(Assembly source)

```
   .comm    data1 4
   .comm    data2 4
   .comm    data3 4
```

(Memory map after linking)

**as33** (S5U1C33000C)

```
0x10000: data1
0x10004: data2
0x10008: data3
```

**as** (S5U1C33001C)

```
0x10000: data3                ← Not in the order
0x10004: data1                ← in which they are
0x10008: data2                ← written in the source file
```

For the programs that were created assuming that symbols will be located in memory in the order in which they are written in the source, be sure to declare the .bss section in the changed source file as described above, to ensure that it will be transported properly.

### 4) .lcomm

For local specification in as, use the .bss (.*X*bss) directive.

Example:

**as33** (S5U1C33000C)

```
   .lcomm   symbol 4
```

**as** (S5U1C33001C)

```
   .section .bss
   .align   2
symbol:
   .skip    4
```

Pay attention to the alignment of symbols same as the .comm directive.

## ● Data definition directives

The data definition directives must be changed, as some of them differ in size between as33 and as. These size differences are as shown in Table 2.10.4.3.

*Table 2.10.4.3 Data definition directives*

| **as33** (S5U1C33000C) | **as** (S5U1C33001C) | Size |
|---|---|---|
| .byte | .byte | 1 byte |
| .half | .short | 2 bytes |
| | .hword | 2 bytes |
| | .word | 2 bytes |
| .word | .int | 4 bytes |
| | .long | 4 bytes |

Modification example:

**as33** (S5U1C33000C)

```
   .code
   .word    boot              ;vector address
boot:
   xld.w    %r15,0x0800
   ld.w     %sp,%r15
```

**as** (S5U1C33001C)

```
   .text
   .long    boot              ;vector address
boot:
   xld.w    %r15,0x0800
   ld.w     %sp,%r15
```

Although the .word directive in as33 is 4 bytes long, in as it is 2 bytes long. Change .word to .int or .long.

## ● Debug directives

Of the as33 debug directives, .endfile, .loc, and .def have no effect in as. Thus, be sure to delete them. To add debugging information, reassemble the source files using the --gstabs option of the as assembler.

## ● Extended instructions

Not all of the extended instructions of ext33 (S5U1C33000C) are supported in as. The codes of unusable extended instructions must be regenerated by using the basic instructions and usable extended instructions in combination.

*Table 2.10.4.4 Table of differences between ext33 and as extended instructions (standard macro)*

| **ext33** (S5U1C33000C) | | **as** (S5U1C33001C) | |
|---|---|---|---|
| xadd | %rd,%rd,imm32 | xadd | %rd,imm32 |
| xsub | %rd,%rd,imm32 | xsub | %rd,imm32 |
| xadd | %sp,%sp,imm32 | – | |
| xsub | %sp,%sp,imm32 | – | |
| xadd | %rd,%rs,imm32 | – | |
| xsub | %rd,%rs,imm32 | – | |
| xadd | %rd,%sp,imm32 | – | |
| xsub | %rd,%sp,imm32 | – | |
| xadd | %rd,%rd,%sp | – | |
| xsub | %rd,%rd,%sp | – | |
| xadd | %sp,%sp,%rs | – | |
| xsub | %sp,%sp,%rs | – | |
| xcmp | %rd,sign32 | xcmp | %rd,sign32 |
| xcmp | %rd,%sp | – | |
| xcmp | %sp,%rs | – | |
| xand | %rd,%rd,sign32 | xand | %rd,sign32 |
| xoor | %rd,%rd,sign32 | xoor | %rd,sign32 |
| xxor | %rd,%rd,sign32 | xxor | %rd,sign32 |
| xand | %rd,%rs,sign32 | – | |
| xoor | %rd,%rs,sign32 | – | |
| xxor | %rd,%rs,sign32 | – | |
| xnot | %rd,sign32 | xnot | %rd,sign32 |
| xsrl | %rd,%rs | – | |
| xsll | %rd,%rs | – | |
| xsra | %rd,%rs | – | |
| xsla | %rd,%rs | – | |
| xrr | %rd,%rs | – | |
| xrl | %rd,%rs | – | |
| xsrl | %rd,imm5 | xsrl | %rd,imm5 |
| xsll | %rd,imm5 | xsll | %rd,imm5 |
| xsra | %rd,imm5 | xsra | %rd,imm5 |
| xsla | %rd,imm5 | xsla | %rd,imm5 |
| xrr | %rd,imm5 | xrr | %rd,imm5 |
| xrl | %rd,imm5 | xrl | %rd,imm5 |
| xld.b | %rd,[%sp+imm32] | xld.b | %rd,[%sp+imm32] |
| xld.ub | %rd,[%sp+imm32] | xld.ub | %rd,[%sp+imm32] |
| xld.h | %rd,[%sp+imm32] | xld.h | %rd,[%sp+imm32] |
| xld.uh | %rd,[%sp+imm32] | xld.uh | %rd,[%sp+imm32] |
| xld.w | %rd,[%sp+imm32] | xld.w | %rd,[%sp+imm32] |
| xld.b | [%sp+imm32],%rs | xld.b | [%sp+imm32],%rs |
| xld.h | [%sp+imm32],%rs | xld.h | [%sp+imm32],%rs |
| xld.w | [%sp+imm32],%rs | xld.w | [%sp+imm32],%rs |
| xld.b | %rd,[symbol±imm32] | xld.b | %rd,[symbol+imm26] |
| xld.ub | %rd,[symbol±imm32] | xld.ub | %rd,[symbol+imm26] |
| xld.h | %rd,[symbol±imm32] | xld.h | %rd,[symbol+imm26] |
| xld.uh | %rd,[symbol±imm32] | xld.uh | %rd,[symbol+imm26] |
| xld.w | %rd,[symbol±imm32] | xld.w | %rd,[symbol+imm26] |
| xld.b | [symbol±imm32],%rs | xld.b | [symbol+imm26],%rs |
| xld.h | [symbol±imm32],%rs | xld.h | [symbol+imm26],%rs |
| xld.w | [symbol±imm32],%rs | xld.w | [symbol+imm26],%rs |
| xld.w | [symbol±imm32],%sp | – | |

| **ext33** (S5U1C33000C) | **as** (S5U1C33001C) |
|---|---|
| `xld.b    %rd,[imm32]` | – |
| `xld.ub   %rd,[imm32]` | – |
| `xld.h    %rd,[imm32]` | – |
| `xld.uh   %rd,[imm32]` | – |
| `xld.w    %rd,[imm32]` | – |
| `xld.b    [imm32],%rs` | – |
| `xld.h    [imm32],%rs` | – |
| `xld.w    [imm32],%rs` | – |
| `xld.w    [imm32],%sp` | – |
| `xld.b    %rd,[%rb+symbol±imm32]` | – |
| `xld.ub   %rd,[%rb+symbol±imm32]` | – |
| `xld.h    %rd,[%rb+symbol±imm32]` | – |
| `xld.uh   %rd,[%rb+symbol±imm32]` | – |
| `xld.w    %rd,[%rb+symbol±imm32]` | – |
| `xld.b    [%rb+symbol±imm32],%rs` | – |
| `xld.h    [%rb+symbol±imm32],%rs` | – |
| `xld.w    [%rb+symbol±imm32],%rs` | – |
| `xld.w    [%rb+symbol±imm32],%sp` | – |
| `xld.b    %rd,[%rb+imm32]` | `xld.b    %rd,[%rb+imm26]` |
| `xld.ub   %rd,[%rb+imm32]` | `xld.ub   %rd,[%rb+imm26]` |
| `xld.h    %rd,[%rb+imm32]` | `xld.h    %rd,[%rb+imm26]` |
| `xld.uh   %rd,[%rb+imm32]` | `xld.uh   %rd,[%rb+imm26]` |
| `xld.w    %rd,[%rb+imm32]` | `xld.w    %rd,[%rb+imm26]` |
| `xld.b    [%rb+imm32],%rs` | `xld.b    [%rb+imm26],%rs` |
| `xld.h    [%rb+imm32],%rs` | `xld.h    [%rb+imm26],%rs` |
| `xld.w    [%rb+imm32],%rs` | `xld.w    [%rb+imm26],%rs` |
| `xld.w    [%rb+imm32],%sp` | – |
| `xld.w    %rd,symbol±imm32` | `xld.w    %rd,symbol±imm32` |
| `xld.w    %rd,sign32` | `xld.w    %rd,sign32` |
| `xbtst    [symbol±imm32],imm3` | `xbtst    [symbol+imm26],imm3` |
| `xbclr    [symbol±imm32],imm3` | `xbclr    [symbol+imm26],imm3` |
| `xbset    [symbol±imm32],imm3` | `xbset    [symbol+imm26],imm3` |
| `xbnot    [symbol±imm32],imm3` | `xbnot    [symbol+imm26],imm3` |
| `xbtst    [imm32],imm3` | – |
| `xbclr    [imm32],imm3` | – |
| `xbset    [imm32],imm3` | – |
| `xbnot    [imm32],imm3` | – |
| `xbtst    [%rb+symbol±imm32],imm3` | – |
| `xbclr    [%rb+symbol±imm32],imm3` | – |
| `xbset    [%rb+symbol±imm32],imm3` | – |
| `xbnot    [%rb+symbol±imm32],imm3` | – |
| `xbtst    [%rb+imm32],imm3` | `xbtst    [%rb+imm26],imm3` |
| `xbclr    [%rb+imm32],imm3` | `xbclr    [%rb+imm26],imm3` |
| `xbset    [%rb+imm32],imm3` | `xbset    [%rb+imm26],imm3` |
| `xbnot    [%rb+imm32],imm3` | `xbnot    [%rb+imm26],imm3` |
| `xbtst    [%sp+imm32],imm3` | – |
| `xbclr    [%sp+imm32],imm3` | – |
| `xbset    [%sp+imm32],imm3` | – |
| `xbnot    [%sp+imm32],imm3` | – |
| – | `scall    label+imm22` |
| – | `scall.d  label+imm22` |
| – | `sjp      label+imm22` |
| – | `sjp.d    label+imm22` |
| – | `sjreq    label+imm22` |
| – | `sjreq.d  label+imm22` |
| – | `sjrne    label+imm22` |
| – | `sjrne.d  label+imm22` |
| – | `sjrgt    label+imm22` |
| – | `sjrgt.d  label+imm22` |
| – | `sjrge    label+imm22` |
| – | `sjrge.d  label+imm22` |
| – | `sjrlt    label+imm22` |
| – | `sjrlt.d  label+imm22` |

| **ext33** (S5U1C33000C) | **as** (S5U1C33001C) |
|---|---|
| – | `sjrle    label+imm22` |
| – | `sjrle.d  label+imm22` |
| – | `sjrugt   label+imm22` |
| – | `sjrugt.d label+imm22` |
| – | `sjruge   label+imm22` |
| – | `sjruge.d label+imm22` |
| – | `sjrult   label+imm22` |
| – | `sjrult.d label+imm22` |
| – | `sjrule   label+imm22` |
| – | `sjrule.d label+imm22` |
| – | `scall    sign22` |
| – | `scall.d  sign22` |
| – | `sjp      sign22` |
| – | `sjp.d    sign22` |
| – | `sjreq    sign22` |
| – | `sjreq.d  sign22` |
| – | `sjrne    sign22` |
| – | `sjrne.d  sign22` |
| – | `sjrgt    sign22` |
| – | `sjrgt.d  sign22` |
| – | `sjrge    sign22` |
| – | `sjrge.d  sign22` |
| – | `sjrlt    sign22` |
| – | `sjrlt.d  sign22` |
| – | `sjrle    sign22` |
| – | `sjrle.d  sign22` |
| – | `sjrugt   sign22` |
| – | `sjrugt.d sign22` |
| – | `sjruge   sign22` |
| – | `sjruge.d sign22` |
| – | `sjrult   sign22` |
| – | `sjrult.d sign22` |
| – | `sjrule   sign22` |
| – | `sjrule.d sign22` |
| `xcall    label+imm32` | `xcall    label+imm32` |
| `xcall.d  label+imm32` | `xcall.d  label+imm32` |
| `xjp      label+imm32` | `xjp      label+imm32` |
| `xjp.d    label+imm32` | `xjp.d    label+imm32` |
| `xjreq    label+imm32` | `xjreq    label+imm32` |
| `xjreq.d  label+imm32` | `xjreq.d  label+imm32` |
| `xjrne    label+imm32` | `xjrne    label+imm32` |
| `xjrne.d  label+imm32` | `xjrne.d  label+imm32` |
| `xjrgt    label+imm32` | `xjrgt    label+imm32` |
| `xjrgt.d  label+imm32` | `xjrgt.d  label+imm32` |
| `xjrge    label+imm32` | `xjrge    label+imm32` |
| `xjrge.d  label+imm32` | `xjrge.d  label+imm32` |
| `xjrlt    label+imm32` | `xjrlt    label+imm32` |
| `xjrlt.d  label+imm32` | `xjrlt.d  label+imm32` |
| `xjrle    label+imm32` | `xjrle    label+imm32` |
| `xjrle.d  label+imm32` | `xjrle.d  label+imm32` |
| `xjrugt   label+imm32` | `xjrugt   label+imm32` |
| `xjrugt.d label+imm32` | `xjrugt.d label+imm32` |
| `xjruge   label+imm32` | `xjruge   label+imm32` |
| `xjruge.d label+imm32` | `xjruge.d label+imm32` |
| `xjrult   label+imm32` | `xjrult   label+imm32` |
| `xjrult.d label+imm32` | `xjrult.d label+imm32` |
| `xjrule   label+imm32` | `xjrule   label+imm32` |
| `xjrule.d label+imm32` | `xjrule.d label+imm32` |
| `xcall    sign32` | `xcall    sign32` |
| `xcall.d  sign32` | `xcall.d  sign32` |
| `xjp      sign32` | `xjp      sign32` |
| `xjp.d    sign32` | `xjp.d    sign32` |
| `xjreq    sign32` | `xjreq    sign32` |

| **ext33** (S5U1C33000C) | **as** (S5U1C33001C) |
|---|---|
| `xjreq.d  sign32` | `xjreq.d  sign32` |
| `xjrne    sign32` | `xjrne    sign32` |
| `xjrne.d  sign32` | `xjrne.d  sign32` |
| `xjrgt    sign32` | `xjrgt    sign32` |
| `xjrgt.d  sign32` | `xjrgt.d  sign32` |
| `xjrge    sign32` | `xjrge    sign32` |
| `xjrge.d  sign32` | `xjrge.d  sign32` |
| `xjrlt    sign32` | `xjrlt    sign32` |
| `xjrlt.d  sign32` | `xjrlt.d  sign32` |
| `xjrle    sign32` | `xjrle    sign32` |
| `xjrle.d  sign32` | `xjrle.d  sign32` |
| `xjrugt   sign32` | `xjrugt   sign32` |
| `xjrugt.d sign32` | `xjrugt.d sign32` |
| `xjruge   sign32` | `xjruge   sign32` |
| `xjruge.d sign32` | `xjruge.d sign32` |
| `xjrult   sign32` | `xjrult   sign32` |
| `xjrult.d sign32` | `xjrult.d sign32` |
| `xjrule   sign32` | `xjrule   sign32` |
| `xjrule.d sign32` | `xjrule.d sign32` |

*Table 2.10.4.5 Table of differences between ext33 and as extended instructions (advanced macro)*

| **ext33** (S5U1C33000C) | **as** (S5U1C33001C) |
|---|---|
| – | `ald.b    %rd,[symbol+imm19]` |
| – | `ald.ub   %rd,[symbol+imm19]` |
| – | `ald.h    %rd,[symbol+imm19]` |
| – | `ald.uh   %rd,[symbol+imm19]` |
| – | `ald.w    %rd,[symbol+imm19]` |
| – | `ald.b    [symbol+imm19],%rs` |
| – | `ald.h    [symbol+imm19],%rs` |
| – | `ald.w    [symbol+imm19],%rs` |
| – | `xld.b    %rd,[%dp+imm32]` |
| – | `xld.ub   %rd,[%dp+imm32]` |
| – | `xld.h    %rd,[%dp+imm32]` |
| – | `xld.uh   %rd,[%dp+imm32]` |
| – | `xld.w    %rd,[%dp+imm32]` |
| – | `xld.b    [%dp+imm32],%rs` |
| – | `xld.h    [%dp+imm32],%rs` |
| – | `xld.w    [%dp+imm32],%rs` |

### *2.10.5 Transporting Linker Command Files (*.cm)*

To map the sections of objects you have created into actual memory by S5U1C33001C, create a linker script file; when executing the ld linker and specify the -T option.
Although in linker command files (*.cm) both the location addresses of sections and the link files are specified, these specifications in ld are entered separately. Specifically, the location addresses are specified in a linker script file, and the link files are specified in the command line when linking.
Furthermore, the S5U1C33000C libraries must be replaced with the S5U1C33001C libraries.

*Table 2.10.5.1 Libraries*

| **as33** (S5U1C33000C) | **as** (S5U1C33001C) |
|---|---|
| `io.lib`, `lib.lib`, `math.lib`, `string.lib`, `ctype.lib` | `libc.a` |
| `fp.lib`, `idiv.lib` | `libgcc.a` |
| `fpp.lib`, `idiv.lib` | `libgccP.a` |

## ● Modification example

Linker command file sample.cm for lk33 (S5U1C33000C)

```
;Map set
-code 0x0601000           ; set relative code section start address
-bss  0x0000400           ; set relative bss  section start address

-code 0x0600000 {boot.o}  ; set code sections to absolute address

;Library path
-l ..\..\lib

;Executable file
-o ansilib.srf

;Object files
boot.o
main.o
sys.o
lib.o

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

Linker script file sample.lds for ld (S5U1C33001C)

```
OUTPUT_FORMAT("elf32-c33", "elf32-c33",
              "elf32-c33")
OUTPUT_ARCH(c33)
SEARCH_DIR(.);
SECTIONS
{
      __dp = 0x400;

      . = 0x400;
      .bss      : { *(.bss) }
      .data     : { *(.data) }

      . = 0x600000;
      .bootsec  : { boot.o (.text) }

      . = 0x0601000;
      .text     : { *(.text) }

}
```

Command line of the ld linker

```
ld -o sample.elf boot.o main.o sys.o lib.o ../lib/libc.a ../lib/libgcc.a -Tsample.lds
```

## ● Example of changing lk33 Ver.3.0 linker command files

Linker command file sample.cm for lk33 (S5U1C33000C)

```
-v3

-defaddr IN_RAM=0x0
-defaddr EXT_ROM=0xc00000

-codeblock OBJ1
{
      sample2.o
      sample3.o
}

-ucodeblock OBJ2           ... Define shared blocks
{                              The CODE sections in sample1.o and sample4.o are set in the same address
      sample1.o
      sample4.o
}

-addr IN_RAM
{
      DEFAULT_BSS
      @DEFAULT_DATA
      @OBJ1                ... Locate "OBJ1" as a virtual block
      @OBJ2                ... Locate "OBJ2" as a virtual and shared block
}

-addr EXT_ROM
{
      DEFAULT_CODE
      DEFAULT_DATA
      OBJ1
      OBJ2
}
sample1.o
sample2.o
sample3.o
sample4.o
sample5.o
sample.lib
```

Linker script file sample.lds for ld (S5U1C33001C)

```
OUTPUT_FORMAT("elf32-c33", "elf32-c33",
              "elf32-c33")
OUTPUT_ARCH(c33)
SEARCH_DIR(.);
SECTIONS
{
      /* data pointer symbol By GWB33 */
      __dp = 0x0;

      /* section information By GWB33 */
      . = 0x0;

      .bss 0x00000000 :
        {
         __START_bss = . ;
         *(.bss) ;
         __END_bss = . ;
        }

      .text 0x00c00000 :
        {
         __START_text = . ;
         sample5.o(.text) ;
         C:/GNU33/lib/sample.a(.text);
         __END_text = . ;
        }

      .rodata __END_text :
        {
         __START_rodata = . ;
```

```
          __END_rodata = . ;
         }

       .data __END_bss : AT( __END_rodata )
         {
          __START_data = . ;
          *(.data) ;
          __END_data = . ;
         }
       __START_data_lma = LOADADDR( .data );

       .OBJ1 __END_data : AT( __START_data_lma+SIZEOF( .data ))
         {
          __START_OBJ1 = . ;
          sample2.o(.text)
          sample3.o(.text) ;
          __END_OBJ1 = . ;
         }
       __START_OBJ1_lma = LOADADDR( .OBJ1 );

       .OBJ2_1 __END_OBJ1 : AT( __START_OBJ1_lma+SIZEOF( .OBJ1 ))
         {
          __START_OBJ2_1 = . ;
          sample1.o(.text) ;
          __END_OBJ2_1 = . ;
         }
       __START_OBJ2_1_lma = LOADADDR( .OBJ2_1 );

       .OBJ2_2 __END_OBJ1 : AT( __START_OBJ2_1_lma+SIZEOF( .OBJ2_1 ))
         {
          __START_OBJ2_2 = . ;
          sample4.o(.text) ;
          __END_OBJ2_2 = . ;
         }
       __START_OBJ2_2_lma = LOADADDR( .OBJ2_2 );

}
```

By specifying multiple object files in one section as for the definition of OBJ1, it is possible to obtain the same effect as for block definitions in lk33.

Result of sample.lds linked by the linker ld (objdump -h sample.elf)

```
block.elf:     file format elf32-c33

Sections:
Idx Name       Size      VMA       LMA       File off  Algn
  0 .bss       00000014  00000000  00000000  00001000  2**2
               ALLOC
  1 .text      00000006  00c00000  00c00000  00001000  2**1
               CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .rodata    00000005  00c00006  00c00006  00001006  2**0
               CONTENTS, ALLOC, LOAD, DATA
  3 .data      00000014  00000014  00c0000b  00001014  2**2
               CONTENTS, ALLOC, LOAD, DATA
  4 .OBJ1      0000000c  00000028  00c0001f  00001028  2**1
               CONTENTS, ALLOC, LOAD, CODE
  5 .OBJ2_1    00000006  00000034  00c0002b  00001034  2**1
               CONTENTS, ALLOC, LOAD, CODE
  6 .OBJ2_2    00000006  00000034  00c00031  00002034  2**1
               CONTENTS, ALLOC, LOAD, CODE
  7 .stab      00000714  0000003c  0000003c  0000203c  2**2
               CONTENTS, READONLY, DEBUGGING
  8 .comment   000000be  00000b66  00000b66  00002750  2**0
               CONTENTS, READONLY
  9 .stabstr   00000416  00000750  00000750  0000280e  2**0
               CONTENTS, READONLY, DEBUGGING
```

VMAs for OBJ2_1 and OBJ2_2 are allocated to the same address (0x00000034).

### *2.10.6 Transporting Debugger Parameter Files (*.par)*

The gdb debugger (S5U1C33001C) requires a parameter file when it starts up, similar to the db33 debugger (S5U1C33000C). Because gwb33 (S5U1C33001C) comes with a parameter file generator (as does wb33 (S5U1C33000C)), regenerate the *.par files using this function.

Click the [PAR edit] button on the gwb33 dialog box to bring up a dialog box in which to create a parameter file.

### *2.10.7 Differences in Structure between srf33 Object Files (S5U1C33000C) and elf Object Files (S5U1C33001C)*

The following shows the differences in structure between the srf33 format files generated by the assembler as33 and linker lk33 included with the S5U1C33000C package, and the elf format files generated by the assembler as and linker ld included with the S5U1C33001C package.

**● Structure of the srf33 format file**

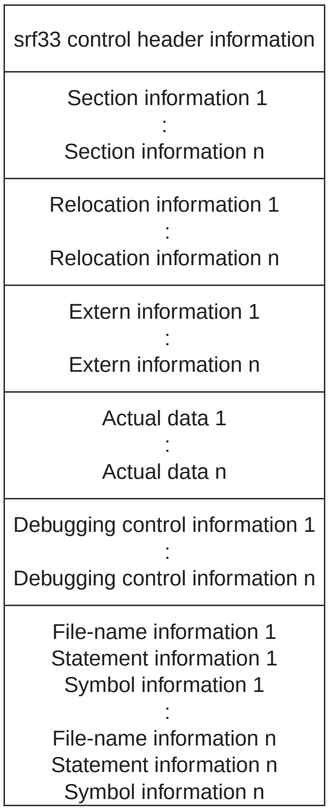


*Fig 2.10.7.1 Layout in srf file*

The section information is respectively linked with the relocation information, extern information, and actual data.
For details on the srf33 object files, refer to the Appendix in the "S5U1C33000C Manual".

## ● Structure of elf format file

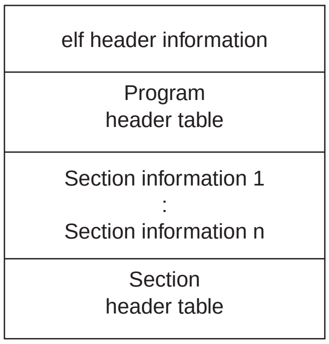


*Fig. 2.10.7.2 Layout in elf file*

The elf header information includes the address size, type of architecture, file version, entry point, and other information.
The assembler and linker handle the elf file as a collection of logical sections written in the section header table. By contrast, the system loader handles the elf file as a collection of segments written in the program header table.

For details on the elf object files, refer to Websites and related documents.

The content and structure of the header information are incompatible with those of srf33 files, as shown above.

## *2.11 Precautions on Use of the S5U1C33001C Tool*

### ● New-line character in Cygwin

When the S5U1C33001C is used on a personal computer in which Cygwin has been installed, care must be taken, as the tool may not operate normally depending on the new-line character settings. The new-line character settings recommended for the S5U1C33001C are CR + LF (same as in Windows). New-line character settings other than this should be changed by setting up Cygwin.

### ● About the partial-offset display of extended assembler instructions

The extended instructions of the assembler, as in the case of the S5U1C33001C, include the description "SYMBOL+imm26" in their operands. This imm26 can actually be assembled as a 32-bit value. However, the actual offset value that is set during linking of the source files must satisfy the conditional expression given below.

imm26 + [offset value for SYMBOL] < 0x4000000

(The offset value for SYMBOL is the distance from the data-area pointer to SYMBOL.)

Example:
```
xld.b  [SYMBOL+imm26],rd
```

This instruction is expanded into the following format.
```
ext    (SYMBOL+imm26)@ah...A   = ext ((SYMBOL+imm26) - __dp)[25:13]
ext    (SYMBOL+imm26)@al...B   = ext ((SYMBOL+imm26) - __dp)[12:0]
xld.b  [%r15], %rd
```

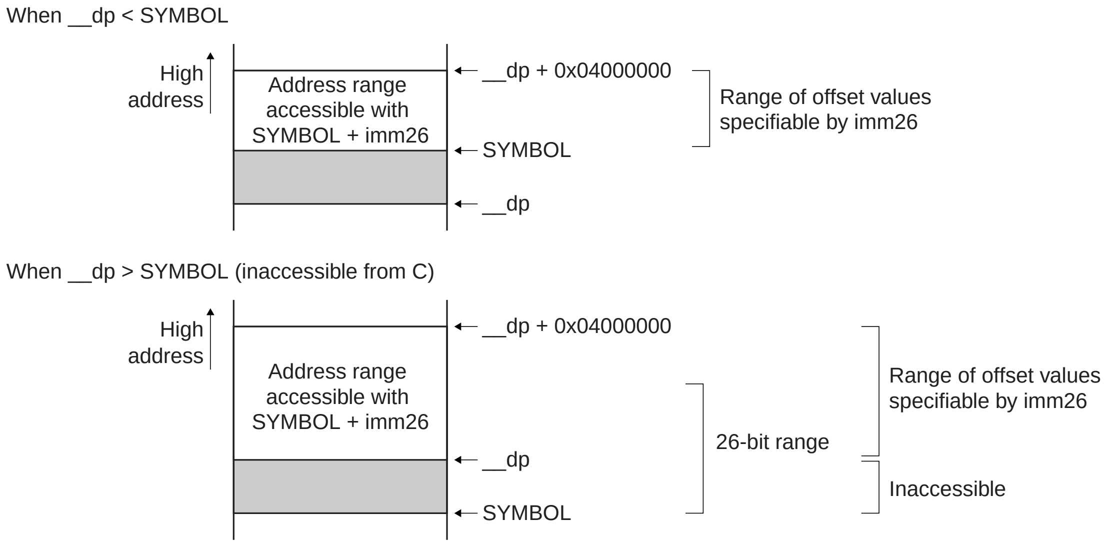

When __dp < SYMBOL
High address
Address range accessible with SYMBOL + imm26
__dp + 0x04000000
SYMBOL
__dp
Range of offset values specifiable by imm26
When __dp > SYMBOL (inaccessible from C)
High address
Address range accessible with SYMBOL + imm26
__dp + 0x04000000
__dp
SYMBOL
26-bit range
Range of offset values specifiable by imm26
Inaccessible

*Fig. 2.11.1 Specifiable range of SYMBOL + imm26*

The upper limit of the accessible range of the above instruction is __dp + 0x04000000, as shown above. Therefore, when __dp < SYMBOL, as is normally the case, values from 0 to (__dp + 04000000) - SYMBOL can be specified for imm26. When __dp > SYMBOL, offset values exceeding the 26-bit range can also be specified. However, SYMBOL-based addresses cannot be accessed from C sources. These addresses can be accessed from the assembly sources, provided that SYMBOL + imm26 is in the range of __dp to (__dp + 04000000).

## ● About the SHARE file for GUI debugging use

A group of files present in the \gnu33\utility\share folder are the files written in TCL / TK language, which are used in GUI mode of the debugger gdb. As these are text files, they can be altered by an editor or the like. However, care must be taken, as the debugger gdb may become inoperable depending on the altered content. If it is necessary to alter the content, be sure to gain a full understanding of the detailed specifications before actually making a change.

Some of these files have been customized from insight gdb for use with the S5U1C33001C manufactured by Epson. The \gnu33\utility\share\modify folder includes a file indicating the customized file names and the contents of modification.

# 3 Programming the S1C33 Peripheral Functions

This chapter describes some basic methods for programming the peripheral functions of the S1C33 chip.

Note: *Unless otherwise noted, the peripheral functions and following example code apply to the S1C33209. Functionality or control register addresses may differ, depending on the specific microcomputer.*

## 3.1 Setting Up BCU

The following code demonstrates how to set up SRAM (same as for ROM and flash) and DRAM. This is a BCU setup example in cases where the S1C33209, both core and bus, operates at 25 MHz and has SRAM and DRAM connected to areas 10 and 13, respectively.

BCU setup example

```
void setbcu()
    {
      volatile short *ps0;
      volatile char *pc0;

// set bcu

      ps0 = (short *)0x48126;    // area 9-10 1 wait
      *ps0 = 0x01;

      ps0 = (short *)0x48122;    // area 13    dram                              (1)
      *ps0 = 0x82;               // area 14    2 wait

      pc0 = (char *)0x4014d;     // pre-scaler fpr 8bit TM0                      (2)
      *pc0 = 0x09;               //    1/4
      pc0 = (char *)0x40161;     // 8bit TM0 reload
      *pc0 = 0x7e;               //    20us in 25MHz
      pc0 = (char *)0x40160;     // 8bit TM0
      *pc0 = 0x3;                //    start

      ps0 = (short *)0x4812e;                                                    (3)
      *ps0 = 0x06e0;             //  fast page, col=9bit, refresh enable, CBR,
      ps0 = (short *)0x48130;
      *ps0 = 0x208;              //  ras1/cas2, precharge1, cefunc=01
    }
```

### ● Settings for SRAM, ROM, and flash

Settings for SRAM, ROM, and flash can be made for each area below using BCU registers at addresses 0x48120 to 0x4812B.

**Setup areas**

18–17, 16–15, 14–13, 12–11, 10–9, 8–7, 6, 5–4

**Setup contents**

a) Device size: 8 or 16 bits
   (Area 6 switches between 8 and 16 bits, depending on address.)
b) Number of wait cycles: 0 to 7 cycles
   (During writes, wait cycles of 1 or more are assumed, even if you set 0 here.)
c) Output disable delay time: 0.5 to 3.5 cycles
   (These wait cycles are inserted when accessing locations across area boundaries.)

In this example, areas 9–10 are set for device size = 16 bits, wait cycle = 1, and output disable delay time = 0.5 cycles.

```
ps0 = (short *)0x48126;          // area 9-10 1 wait
*ps0 = 0x01;
```

While this presents no problems when two x8 type SRAMs are used for the 16-bit width, the external interface method (0x4812E•D3) must be set to #BSL in 1 when using x16 type SRAM. Two types cannot coexist. This is detailed in "Connecting x16 type SRAM" in Chapter 4, "The Basic S1C33 Chip Board Circuit".

## ● DRAM settings

Areas 14, 13, 8, and 7 can be set for DRAM.

### (1) Selecting DRAM

Set the DRAM select bit to 1 for the area using DRAM. In this example, area 13 is set as 16-bit wide DRAM. Area 14 can be used as 2-wait cycle, 16-bit wide SRAM, etc.

```
   ps0 = (short *)0x48122;          // area 13    dram
   *ps0 = 0x82;                     // area 14    2 wait
```

### (2) DRAM refresh settings using 8-bit timer 0

In this example, the clock input prescaler for 8-bit timer 0 is set to 1/4 mode. As a result, 8-bit timer 0 is clocked with 25 MHz divided by 4. Additionally, 0x7e is set as the timer reload value. Because the timer input clock is thus divided by 125 (0x7e + 1), the refresh cycle is 20 µs, equal to the original operating clock (25 MHz) divided by 500.

```
   pc0 = (char *)0x4014d;           // pre-scaler fpr 8bit TM0
   *pc0 = 0x09;                     //    1/4
   pc0 = (char *)0x40161;           // 8bit TM0 reload
   *pc0 = 0x7e;                     //    20us in 25MHz
   pc0 = (char *)0x40160;           // 8bit TM0
   *pc0 = 0x3;                      //    start
```

### (3) DRAM parameter settings

Finally, perform detailed DRAM setup. Note that the following settings are reflected in all connected DRAMs, even when DRAMs are connected to multiple areas.
At address 0x4812E, you can select
1. EDO / fast page mode
2. Column size 8 (8–11 bits)
3. Refresh enable / disable
4. Self / CBR refresh
5. Refresh RPC delay (1, 2)
6. Refresh RAS pulse width (2–5)

Additionally, at address 0x48130, select
7. Successive RAS mode
8. Number of RAS precharges
9. Number of CAS cycles
10. Number of RAS cycles

In this example, settings are made for fast page mode, CBR refresh, RAS = 1 cycle, CAS = 2 cycles, and precharge = 1 cycle.

```
   ps0 = (short *)0x4812e;
   *ps0 = 0x06e0;                   //  fast page, col=9bit, refresh enable, CBR,
   ps0 = (short *)0x48130;
   *ps0 = 0x208;                    //  ras1/cas2, precharge1, cefunc=01
```

In addition, after powering on, DRAM may require some finite time or dummy cycles before becoming usable. Code needs to account for these requirements, in addition to the preceding example.

## ● BCLK, CEFUNC

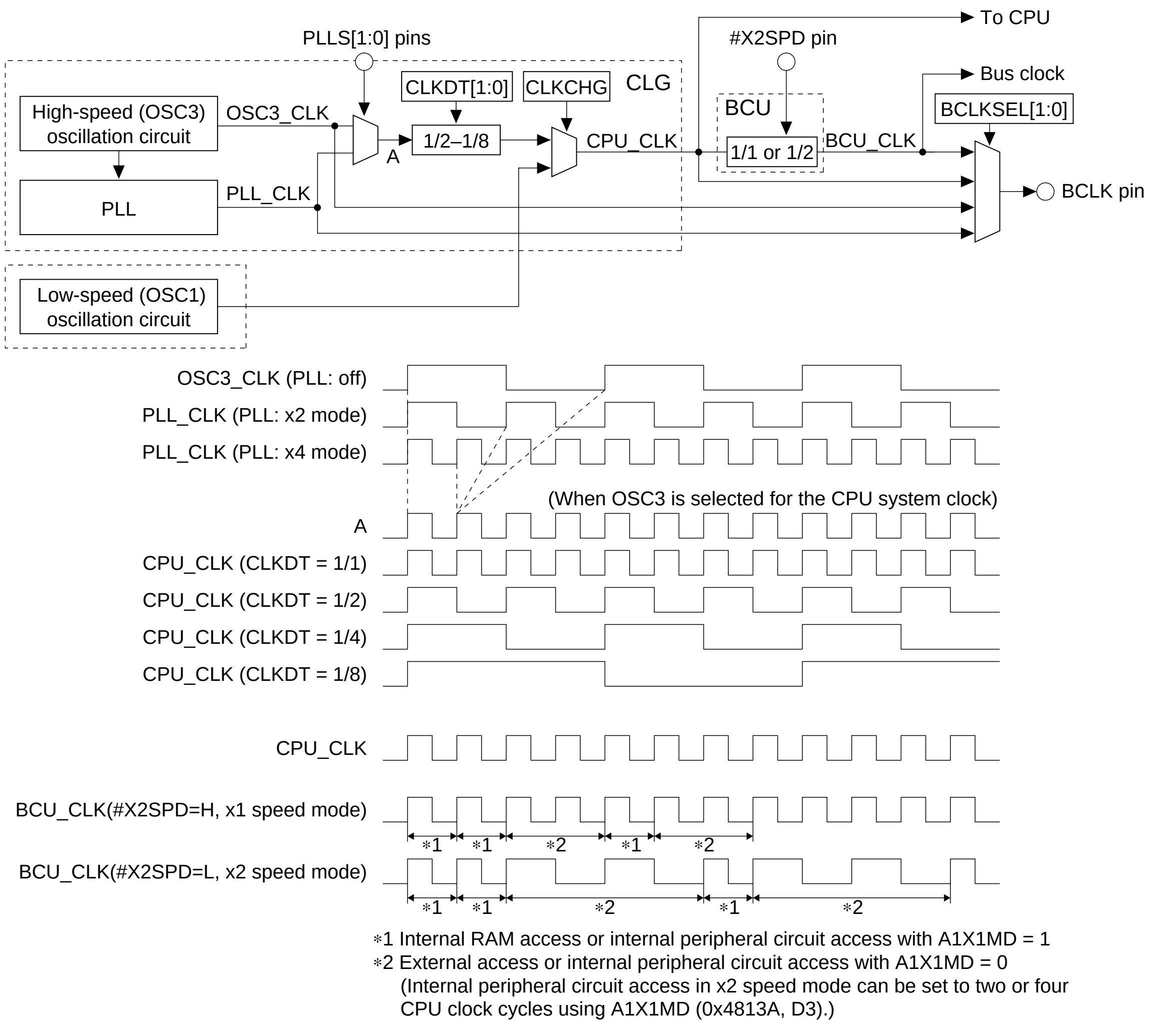
The control bits for setting up BCU for special purposes are available at addresses 0x4812E and 0x48130. Two frequently-used control bits are described below.

### BCLK (0x4812E•DF): BCLK output enable

Controls the clock output from the BCLK pin. By default, this is set at output (0). But since this output consumes several mA of current, set BCLK high (1), if not required.

To output, select from among PLL output clock, OSC3 clock, BCU clock, or CPU clock for the BCLK output clock, using BCLKSEL[1:0] (0x4813A • D[1:0]).

| BCLKSEL1 | BCLKSEL0 | Output clock |
|---|---|---|
| 1 | 1 | PLL_CLK (PLL output clock) |
| 1 | 0 | OSC3_CLK (OSC3 oscillation clock) |
| 0 | 1 | BCU_CLK (BCU operating clock) |
| 0 | 0 | CPU_CLK (CPU operating clock) |



To CPU
PLLS[1:0] pins
#X2SPD pin
Bus clock
CLKDT[1:0]
CLKCHG
CLG
BCU
BCLKSEL[1:0]
High-speed (OSC3) oscillation circuit
OSC3_CLK
A
1/2–1/8
CPU_CLK
1/1 or 1/2
BCU_CLK
PLL
PLL_CLK
BCLK pin
Low-speed (OSC1) oscillation circuit
OSC3_CLK (PLL: off)
PLL_CLK (PLL: x2 mode)
PLL_CLK (PLL: x4 mode)
(When OSC3 is selected for the CPU system clock)
A
CPU_CLK (CLKDT = 1/1)
CPU_CLK (CLKDT = 1/2)
CPU_CLK (CLKDT = 1/4)
CPU_CLK (CLKDT = 1/8)
CPU_CLK
BCU_CLK(#X2SPD=H, x1 speed mode)
∗1 ∗1 ∗2 ∗1 ∗2
BCU_CLK(#X2SPD=L, x2 speed mode)
∗1 ∗1 ∗2 ∗1 ∗2
∗1 Internal RAM access or internal peripheral circuit access with A1X1MD = 1
∗2 External access or internal peripheral circuit access with A1X1MD = 0 (Internal peripheral circuit access in x2 speed mode can be set to two or four CPU clock cycles using A1X1MD (0x4813A, D3).)

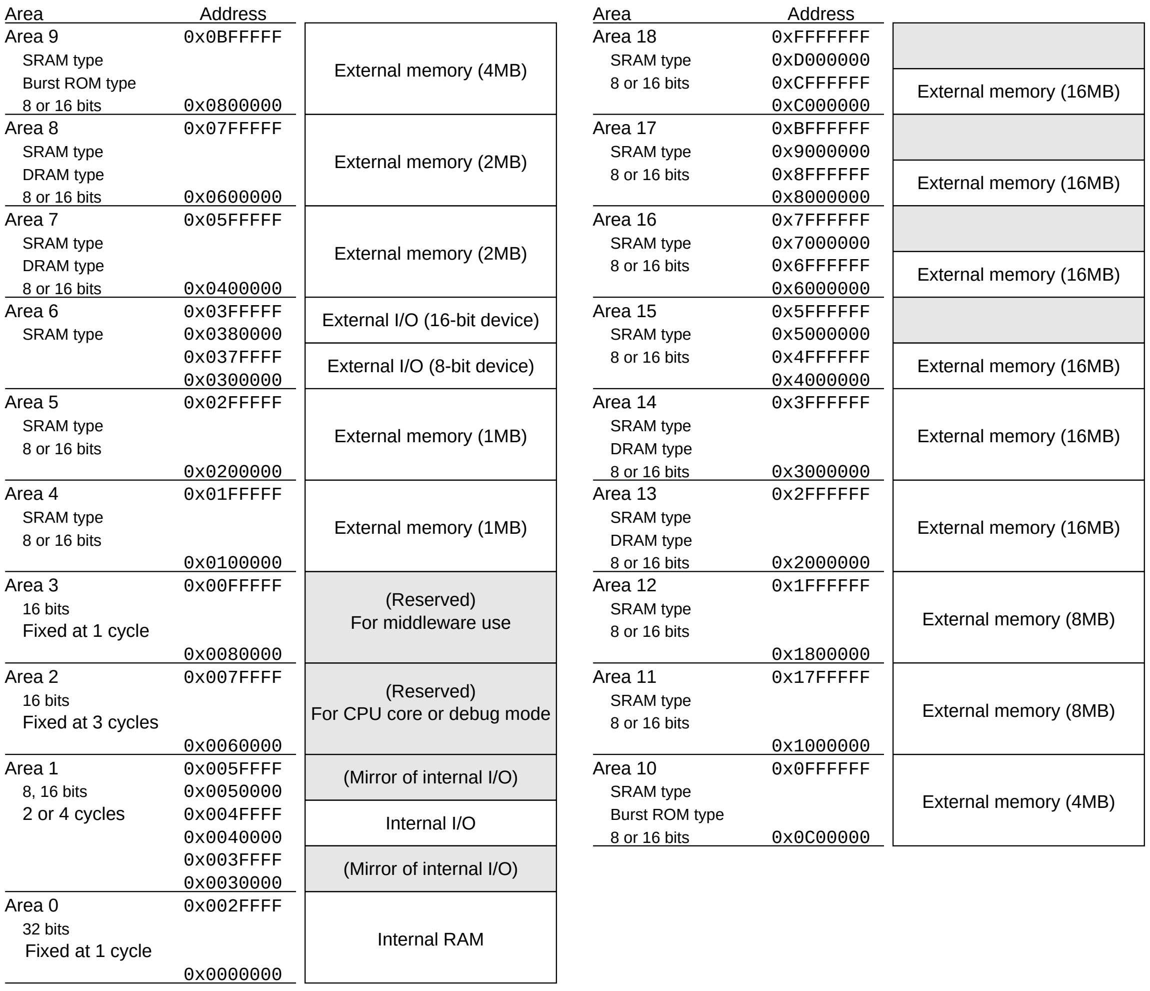
*Clock system (S1C33209)*

### CEFUNC[1:0] (0x48130•D[A:9]): #CE pin function selection

Because the S1C33209 has only 7 #CE pins, it is unable use the entire address space at the same time. Instead, it allows selection of the memory area to be used by setting CEFUNC.

| Pin | CEFUNC = "00" | CEFUNC = "01" | CEFUNC = "1x" |
|---|---|---|---|
| #CE4 | #CE4 | #CE11 | #CE11+#CE12 |
| #CE5 | #CE5 | #CE15 | #CE15+#CE16 |
| #CE6 | #CE6 | #CE6 | #CE7+#CE8 |
| #CE7/#RAS0 | #CE7/#RAS0 | #CE13/#RAS2 | #CE13/#RAS2 |
| #CE8/#RAS1 | #CE8/#RAS1 | #CE14/#RAS3 | #CE14/#RAS3 |
| #CE9 | #CE9 | #CE17 | #CE17+#CE18 |
| #CE10EX | #CE10EX | #CE10EX | #CE9+#CE10EX |

(Default: CEFUNC = "00")



*S1C33 address space*

| Area | Address | |
|---|---|---|
| Area 10 (#CE10) SRAM type, Burst ROM type, 8 or 16 bits | 0x0FFFFFF 0x0C00000 | External memory 6 (4MB) |
| Area 9 (#CE9) SRAM type, Burst ROM type, 8 or 16 bits | 0x0BFFFFF 0x0800000 | External memory 5 (4MB) |
| Area 8 (#CE8/#RAS1) SRAM type, DRAM type, 8 or 16 bits | 0x07FFFFF 0x0600000 | External memory 4 (2MB) |
| Area 7 (#CE7/#RAS0) SRAM type, DRAM type, 8 or 16 bits | 0x05FFFFF 0x0400000 | External memory 3 (2MB) |
| Area 6 (#CE6) SRAM type | 0x03FFFFF 0x0380000 | External I/O (16-bit device) |
| | 0x037FFFF 0x0300000 | External I/O (8-bit device) |
| Area 5 (#CE5) SRAM type, 8 or 16 bits | 0x02FFFFF 0x0200000 | External memory 2 (1MB) |
| Area 4 (#CE4) SRAM type, 8 or 16 bits | 0x01FFFFF 0x0100000 | External memory 1 (1MB) |

CEFUNC = "00"

| Area | Address | |
|---|---|---|
| Area 17 (#CE17) SRAM type, 8 or 16 bits | 0xBFFFFFF 0x9000000 | (Mirror of External memory 6) |
| | 0x8FFFFFF 0x8000000 | External memory 6 (16MB) |
| Area 15 (#CE15) SRAM type, 8 or 16 bits | 0x5FFFFFF 0x5000000 | (Mirror of External memory 5) |
| | 0x4FFFFFF 0x4000000 | External memory 5 (16MB) |
| Area 14 (#CE14/#RAS3) SRAM type, DRAM type, 8 or 16 bits | 0x3FFFFFF 0x3000000 | External memory 4 (16MB) |
| Area 13 (#CE13/#RAS2) SRAM type, DRAM type, 8 or 16 bits | 0x2FFFFFF 0x2000000 | External memory 3 (16MB) |
| Area 11 (#CE11) SRAM type, 8 or 16 bits | 0x17FFFFF 0x1000000 | External memory 2 (8MB) |
| Area 10 (#CE10) SRAM type, Burst ROM type, 8 or 16 bits | 0x0FFFFFF 0x0C00000 | External memory 1 (4MB) |
| Area 6 (#CE6) SRAM type | 0x03FFFFF 0x0380000 | External I/O (16-bit device) |
| | 0x037FFFF 0x0300000 | External I/O (8-bit device) |

CEFUNC = "01"

| Area | Address | |
|---|---|---|
| Area 17+18 (#CE17+18) SRAM type, 8 or 16 bits | 0xFFFFFFF 0xD000000 | (Mirror of External memory 7') |
| | 0xCFFFFFF 0xC000000 | External memory 7' (16MB) |
| | 0xBFFFFFF 0x9000000 | (Mirror of External memory 7) |
| | 0x8FFFFFF 0x8000000 | External memory 7 (16MB) |
| Areas 15–16 (#CE15+16) SRAM type, 8 or 16 bits | 0x7FFFFFF 0x7000000 | (Mirror of External memory 6') |
| | 0x6FFFFFF 0x6000000 | External memory 6' (16MB) |
| | 0x5FFFFFF 0x5000000 | (Mirror of External memory 6) |
| | 0x4FFFFFF 0x4000000 | External memory 6 (16MB) |
| Area 14 (#CE14/#RAS3) SRAM type, DRAM type, 8 or 16 bits | 0x3FFFFFF 0x3000000 | External memory 5 (16MB) |
| Area 13 (#CE13/#RAS2) SRAM type, DRAM type, 8 or 16 bits | 0x2FFFFFF 0x2000000 | External memory 4 (16MB) |
| Areas 11–12 (#CE11+12) SRAM type, 8 or 16 bits | 0x1FFFFFF 0x1000000 | External memory 3 (16MB) |
| Areas 9–10 (#CE9+10EX) SRAM type, Burst ROM type, 8 or 16 bits | 0x0FFFFFF 0x0800000 | External memory 2 (8MB) |
| Areas 7–8 (#CE7+8) SRAM type, 8 or 16 bits | 0x07FFFFF 0x0400000 | External memory 1 (4MB) |

CEFUNC = "10" or "11"

*Selection of external memory area*

## *3.2 Setting Up 8-bit Timer*

In general, four settings are required for peripheral functions.

**1. Prescaler setting**

The operating clock for each peripheral function is always frequency-divided by the prescaler before being fed into the peripheral function.

**2. Setting of the peripheral function itself**

Each peripheral function has registers to determine operating mode and to start or stop it.

**3. Interrupt controller setting (when using interrupts)**

Interrupt requests generated by each peripheral function are always fed into the interrupt controller before being sent to the CPU core.

**4. External pin setting (when using external pins)**

By default, external pins are set for general-purpose I/O ports or input ports. Before external pins can be used for peripheral functions, their functionality must be selected by setting up registers.

The following section describes a simple interrupt control program based on an 8-bit timer, using the sample from sample\icdtrc\ of S5U1C33001C.

### ● S5U1C33000H trace auxiliary interrupt program

This sample is an code example for reinforcing the S5U1C33000H trace function. The S5U1C33000H trace function displays the PC value by analyzing program flow from the PC value (as a starting point such as time at which the program begins running) and the debugger's disassembly information. However, the PC value starting point is not always known, especially in trace overwrite mode. Thus, this program periodically generates an interrupt using the 8-bit timer to confirm the PC value (since the absolute value of PC is output when executing reti), allowing continuation of PC analysis by S5U1C33000H using that PC value as a starting point.

Vector section

```
      .text
      .long BOOT                ; boot,rest VECTOR
      .long RESERVED            ; reserved 4
      .long RESERVED            ; reserved 8
      .long RESERVED            ; reserved 12
      .long EXP_DIV0            ; divided by 0 exception
      .long RESERVED            ; reserved 20
      .long EXP_UNADDR          ; address un-aligned exception
      .long NMI                 ; nmi
      .long RESERVED            ; reserved 32
      .long RESERVED            ; reserved 36
      .long RESERVED            ; reserved 40
      .long RESERVED            ; reserved 44

      .long SOFT_INT            ; software interrupt 0
      .long SOFT_INT            ; software interrupt 1
      .long SOFT_INT            ; software interrupt 2
      .long SOFT_INT            ; software interrupt 3

      .long HARD_INT            ; hardware interrupt  0
      .long HARD_INT            ; hardware interrupt  1
                                      |
      .long HARD_INT            ; hardware interrupt  35
      .long TIME_INT            ; hardware interrupt  36                 (1)
                                      ;set 8 bit timer ch0 interrupt vector
      .long HARD_INT            ; hardware interrupt  37
      .long HARD_INT            ; hardware interrupt  38
```

(1) Vector table setting

Register the interrupt routine in the vector table.

```
      .long HARD_INT            ; hardware interrupt  35
      .long TIME_INT            ; hardware interrupt  36
                                      ;set 8 bit timer ch0 interrupt vector
      .long HARD_INT            ; hardware interrupt  37
```

Initialization and interrupt service routine

```
.global INIT_8TIMER
INIT_8TIMER:

;interrupt disable
        ld.w    %r4,0x00                                            (1)
        ld.w    %psr,%r4        ;IE disnable
        xld.w   %r4,0x40160                                         (2)
        ld.w    %r5,0x00
        ld.b    [%r4],%r5       ;8timer0 disable in timer-reg
        xld.w   %r4,0x4014d                                         (3)
        ld.w    %r5,0x7
        ld.b    [%r4],%r5       ;8timer0 disable in pri-scaler

;8bit timer set
        xld.w   %r4,0x40161                                         (4)
        xld.w   %r5,0x26        ;interrupt every 10000 clocks
        ld.b    [%r4],%r5       ;8timer0 interval

        xld.w   %r4,0x40146                                         (5)
        ld.w    %r5,0x00
        ld.b    [%r4],%r5       ;8timer0 clock is divided clock

        xld.w   %r4,0x40269                                         (6)
        xld.w   %r5,0x03
        ld.b    [%r4],%r5       ;8timer0 interrupt priority level 3

        xld.w   %r4,0x40275                                         (7)
        xld.w   %r5,0x01
        ld.b    [%r4],%r5       ;8timer0 interrupt enable

        xld.w   %r4,0x40285                                         (8)
        xld.w   %r5,0x00
        ld.b    [%r4],%r5       ;8timer0 interrupt flag clear

        xld.w   %r4,0x4014d                                         (9)
        ld.w    %r5,0x0f
        ld.b    [%r4],%r5       ;8timer0 enable in pri-scaler

;8bit timer start
        ld.w    %r4,0x10                                            (10)
        ld.w    %psr,%r4        ;IE enable

        xld.w   %r4,0x40160                                         (11)
        xld.w   %r5,0x3
        ld.b    [%r4],%r5       ;clock out on,preset,start
; exit function
        ret

.global TIME_INT
TIME_INT:
        pushn   %r1
        xld.w   %r1,0x40285
        xld.w   %r0,0x01
        ld.b    [%r1],%r0       ;8timer5 interrupt flag reset
        popn    %r1
        reti
```

(1) Disabling interrupts

Disable the IE flag (to disable interrupts). Altering the interrupt settings while interrupts are enabled causes the program to operate erratically.

```
        ld.w    %r4,0x00
        ld.w    %psr,%r4        ;IE disnable
```

(2) Stopping the 8-bit timer/counter

Temporarily stop the timer from counting down before resetting it.

```
        xld.w   %r4,0x40160
        ld.w    %r5,0x00
        ld.b    [%r4],%r5       ;8timer0 disable in timer-reg
```

(3) Stopping prescaler clock supply to the 8-bit timer
Before altering the timer setting, to ensure safety, stop the clock supplied from the prescaler to the 8-bit timer.

```
     xld.w   %r4,0x4014d
     ld.w    %r5,0x7
     ld.b    [%r4],%r5       ;8timer0 disable in pri-scaler
```

(4) Setting the 8-bit timer
The reload data 0x26 is used to generate an interrupt every 10,069 clock periods of the prescaler's input clock (by default, OSC3 or PLL output).
(0x26 + 1) × 256 = 10,059 clock periods

With the CPU core operating at 20 MHz, an interrupt is generated every 0.5 ms.

```
     xld.w   %r4,0x40161
     xld.w   %r5,0x26        ;interrupt every 10000 clocks
     ld.b    [%r4],%r5       ;8timer0 interval
```

(5) Selecting the input clock for the 8-bit timer
Select a divided clock for input to the 8-bit timer. This is the setting associated with the prescaler.

```
     xld.w   %r4,0x40146
     ld.w    %r5,0x00
     ld.b    [%r4],%r5       ;8timer0 clock is divided clock
```

(6) Setting the interrupt priority level of the 8-bit timer
Set this interrupt priority level to 3.

```
     xld.w   %r4,0x40269
     xld.w   %r5,0x03
     ld.b    [%r4],%r5       ;8timer0 interrupt priority level 3
```

(7) Altering settings of the 8-bit-timer interrupt enable register
Enable the 8-bit-timer interrupt.

```
     xld.w   %r4,0x40275
     xld.w   %r5,0x01
     ld.b    [%r4],%r5       ;8timer0 interrupt enable
```

(8) Clearing the 8-bit-timer interrupt factor
Clear the 8-bit-timer interrupt factor flag.

```
     xld.w   %r4,0x40285
     xld.w   %r5,0x00
     ld.b    [%r4],%r5       ;8timer0 interrupt flag clear
```

(9) Reenabling clock supply from the prescaler
Reenable the prescaler to allow it to start supplying a clock to the 8-bit timer. At this point, the timer remains idle and does not count down.

```
     xld.w   %r4,0x4014d
     ld.w    %r5,0x0f
     ld.b    [%r4],%r5       ;8timer0 enable in pri-scaler
```

(10) Reenabling interrupts
Set the IE bit in the PSR to "1" in order to reenable the interrupts. Hereafter, interrupts from any source will be accepted.

```
     ld.w    %r4,0x10
     ld.w    %psr,%r4        ;IE enable
```

(11) Resetting the 8-bit timer to allow it to start counting down
Finally, reset the counter of the 8-bit timer and allow it start counting down. Although the timer/counter here is made to start counting at the time it is reset, the timer/counter can be reset first and then made to start counting without causing any problem.

```
     xld.w   %r4,0x40160
     xld.w   %r5,0x3
     ld.b    [%r4],%r5       ;clock out on,preset,start
```

## 3.3 Setting Up 16-bit Timer

Here, we will explain how to control 16-bit timer interrupts and PWM output, using the source code for S5U1C331M2S middleware as an example. Note that the S1C33104's 16-bit timer significantly differs in functionality from that of the S1C33209.

### ● Interrupt settings

The following describes the compare B interrupt of 16-bit timer 4.

#### Vector section

```
#define INT30  mdyInt            // mdy interrupt routine                (1)

      .text

      .long    RESET
      .long    RESERVED
      .long    RESERVED
      .long    RESERVED
      .long    ZERODIV
      .long    RESERVED
      .long    ADDRERR
      .long    NMI
      .long    RESERVED
      .long    RESERVED
      .long    RESERVED
      .long    RESERVED
      .long    SOFTINT0
      .long    SOFTINT1
      .long    SOFTINT2
      .long    SOFTINT3
      .long    INT0
      .long    INT1
               |
      .long    INT29
      .long    INT30                                                     (1)
      .long    INT31
               |
```

(1) Setting the interrupt vector
Register the interrupt routine mdyInt as the vector for INT30 (compare B interrupt of 16-bit timer 4).

#### Interrupt disable and PSR save/restore routine

```
      .section .bss
      .align   2
MDY_PSR :
      .zero    4

      .global mdyInt

mdyIntOff:                                                               (1)
      xld.w    %r10,MDY_PSR
      ld.w     %r11,%psr         // save %psr and IE disable
      ld.w     [%r10],%r11
      ld.w     %r10,0
      ld.w     %psr,%r10
      ret
      .global mdyIntOn
mdyIntOn:                                                                (2)
      xld.w    %r10,MDY_PSR
      ld.w     %r11,[%r10]
      ld.w     %psr,%r11         // restore %psr
      ret
```

(1) Disabling interrupts
Save the PSR contents and set the IE bit to 0 to disable interrupts.

(2) Enabling interrupts
Restore the contents of PSR saved in (1).

These settings are called from C.

## 16-bit timer setup section

```
//*********************************************************************
// void mdyTmOpen(unsigned short freq)
//    start timer 4, underflow interrupt with freq count
//    prescaler is 1/1024
//*********************************************************************

void mdyTmOpen(unsigned short freq)
    {
      unsigned char ucTmp;

// interrupt disable
      mdyIntOff();                                                                      (1)

// set TM4 prescaler to 1/1024, 0b00001110

      *(volatile unsigned char *)(0x4014b) = 0xe;                                       (2)

// set TM4 reload and compare data

      *(volatile unsigned short *)(0x481a2) = freq;      //set compare b                (3)
      *(volatile unsigned short *)(0x481a4) = 0x0;       //dummy data for up counter

// set TM4 control register
// fine mode off,compare buf off,reverse off,internal clock,clock out off,preset,stop
//    0x401a6 0010,

      *(volatile unsigned char *)(0x481a6) = 0x0;

// set TM4 match compare b come to cpu interrupt

      *(volatile unsigned char *)(0x40291) &= 0xBf;      //set timer 4 enable           (4)

// set TM4,interrupt priority level 3

      ucTmp = *(volatile unsigned char *)(0x40268);
      ucTmp = ucTmp & 0xf0;
      ucTmp = ucTmp | 0x3;
      *(volatile unsigned char *)(0x40268) = ucTmp;

// clear TM4 interrupt factor flags (write 1, and reset)

      *(volatile unsigned char *)(0x40284) &= 0x0C;

// set TM4 underflow interrupt enable

      *(volatile unsigned char *)(0x40274) |= 0x04;      //set timer 4 enable

// start TM4 counter

      *(volatile unsigned char *)(0x481a6) |= 0x01;                                     (5)

// interrupt enable
      mdyIntOn();                                                                       (6)
    }
```

(1) Disabling interrupts

Disable interrupts as a precautionary measure.

```
// interrupt disable
   mdyIntOff();
```

(2) Setting the prescaler

A divide-by-1024 clock from the prescaler is fed into timer 4 as its input clock.

```
// set TM4 prescaler to 1/1024, 0b00001110
   *(volatile unsigned char *)(0x4014b) = 0xe;
```

(3) Timer 4 cycle (compare B), compare A, and other settings

The compare B interrupt cycle of timer 4 is set to (freq + 1) × 1024 clock periods by the following settings:

```
// set TM4 reload and compare data
   *(volatile unsigned short *)(0x481a2) = freq;      //set compare b
   *(volatile unsigned short *)(0x481a4) = 0x0;       //dummy data for up counter
```

Set other parameters for timer 4.

```
// set TM4 control register
// fine mode off,compare buf off,reverse off,internal clock,clock out off, ...
// 0x401a6 0010,
   *(volatile unsigned char *)(0x481a6) = 0x0;
```

(4) Setting the interrupt controller

Set the interrupt controller so that the compare B interrupt of timer 4 is forwarded to the CPU as an immediate interrupt, not as an IDMA start request.

```
// set TM4 match compare b come to cpu interrupt
   *(volatile unsigned char *)(0x40291) &= 0xBf;      //set timer 4 enable
```

Set the interrupt priority to 3.

```
// set TM4,interrupt priority level 3
   ucTmp = *(volatile unsigned char *)(0x40268);
   ucTmp = ucTmp & 0xf0;
   ucTmp = ucTmp | 0x3;
   *(volatile unsigned char *)(0x40268) = ucTmp;
```

As a precaution, clear the interrupt factor flag.

```
// clear TM4 interrupt factor flags (write 1, and reset)
   *(volatile unsigned char *)(0x40284) &= 0x0C;
```

Enable the compare B interrupt.

```
// set TM4 underflow interrupt enable
   *(volatile unsigned char *)(0x40274) |= 0x04;      //set timer 4 enable
```

(5) Timer start

Let timer 4 begin counting.

```
// start TM4 counter
   *(volatile unsigned char *)(0x481a6) |= 0x01;
```

(6) Enabling interrupts

Reenable interrupts.

```
// interrupt enable
   mdyIntOn();
```

Note that a separate interrupt routine (mdyInt) needs to be written. Make sure that the interrupt factor flag is always cleared in the interrupt routine.

Example for clearing:

```
// clear TM4 interrupt factor flags (write H and reset)
      *(volatile unsigned char *)(0x40284) &= 0x0C;
```

This prevents the re-occurrence of the same interrupt when interrupts are enabled.

## ● PWM settings

The following section describes how to process PWM. The source code for S5U1C331M2S middleware is used as an example.

PWM initial settings

```
//*******************************************************************
// void mdyTm0Set (unsigned short count, unsigned short compare)
//*******************************************************************

static void mdyTm0Set (unsigned short count, unsigned short compare, int reverse)
    {
// interrupt disable
      mdyIntOff();                                                              (1)

// set P22 port to TM0

      *(volatile char *)(0x402d8) |= 0x04;                                      (2)

// set TM0 prescaler to 1/16, 0b0001011

      *(volatile unsigned char *)(0x40147) = 0x0b;                              (3)

// set TM0 reload and compare data

      *(volatile unsigned short *)(0x48182) = count;          //compare B       (4)
      *(volatile unsigned short *)(0x48180) = compare;        //compare A

// set TM0 control register
// fine mode off, compare buf, reverse, internal clock, clock out on, preset, stop
// internal clock, clock out on, preset, stop
//    0x4018e 0b00010100 or 0b00110100

      if (reverse==1){
               *(volatile unsigned char *)(0x48186) = 0x34;
      }
      else{
               *(volatile unsigned char *)(0x48186) = 0x24;
      }

// reset TM0 counter

      *(volatile unsigned char *)(0x48186) |= 0x02;                             (5)

// interrupt enable
      mdyIntOn();                                                               (6)
    }
```

(1) Disabling interrupts

Disable interrupts as a precautionary measure.

```
// interrupt disable
   mdyIntOff();
```

(2) Selecting port functions

Because the ports used for PWM (16-bit timer) output are set for general-purpose input/output ports by default, change their function to PWM output.

```
// set P22 port to TM0
   *(volatile char *)(0x402d8) |= 0x04;
```

(3) Setting the prescaler

A divide-by-16 clock from the prescaler is fed into timer 0 as its input clock.

```
// set TM0 prescaler to 1/16, 0b0001011
   *(volatile unsigned char *)(0x40147) = 0x0b;
```

(4) Setting timer 0

Start by setting up compare A and compare B registers. Compare B + 1 counts comprise one cycle. In normal mode, output starts from 0; in inverse mode, output starts from 1. Compare A + 1 counts select output between 0 and 1.

For example, when in normal mode compare B = 5 and compare A = 0, the output is 0 in the first clock period and 1 in the remaining other four clock periods. This is repeated.

```
// set TM0 reload and compare data
   *(volatile unsigned short *)(0x48182) = count;           //compare B
   *(volatile unsigned short *)(0x48180) = compare;         //compare A
```

Set other parameters for timer 0.

```
// set TM0 control register
// fine mode off, compare buf, reverse, internal clock, clock out on, preset, stop
// internal clock, clock out on, preset, stop
// 0x4018e 0b00010100 or 0b00110100
   if (reverse==1){
            *(volatile unsigned char *)(0x48186) = 0x34;
   }
   else{
            *(volatile unsigned char *)(0x48186) = 0x24;
   }
```

(5) Reset the counter for timer 0

Reset the counter for timer 0 to 0.

```
// reset TM0 counter
   *(volatile unsigned char *)(0x48186) |= 0x02;
```

(6) Enabling interrupts

Finish by reenabling interrupts.

```
// interrupt enable
   mdyIntOn();
```

### PWM start section

```
//*********************************************************************
// void mdyTm0Start ()
//*********************************************************************

static void mdyTm0Start ()
    {
// start TM0 counter

      *(volatile unsigned char *)(0x48186) |= 0x03;
    }
```

This function starts PWM.

### PWM change section

```
//*********************************************************************
// void mdyTm0Change (unsigned short count, unsigned short compare)
//*********************************************************************

static void mdyTm0Change (unsigned short count, unsigned short compare)
    {
// set TM0 reload and compare data
      *(volatile unsigned short *)(0x48182) = count;     // compare B
      *(volatile unsigned short *)(0x48180) = compare;   // compare A
    }
```

This function changes the cycles and duty of PWM waveform. In setting (4) of the mdyTm0set() function, the compare buffer (0x48186•D5 = 1) is enabled to allow compare A / B data to be written to the buffer asynchronously with the counter. The data once stored in the buffer is set in the compare A / B registers when the counter returns a 0 upon matching compare B. If the entire compare buffer is not being used, a single occurrence of compare A matching may be undetected unless synchronized since compare A / B data take effect when written.

## *3.4 Setting Up Serial Interface*

This section describes how to control asynchronous communications via a serial interface, using the source code for S5U1C331M2S middleware as an example.

### ● Asynchronous communications using an external clock

The following example is an assembly source excerpted from mon33g\src\m3s_sci.s. In this example, communications are controlled by polling rather than by using interrupts.

Initialize routine

```
#define        MON_VER  0x10          ;moinitor firm-ware version

#define        STDR     0x000401e0    ;transmit data register(ch0)
#define        SRDR     0x000401e1    ;receive data register(ch0)
#define        SSR      0x000401e2    ;serial status register(ch0)
#define        SCR      0x000401e3    ;serial control register(ch0)
#define        SIR      0x000401e4    ;IrDA control register(ch0)
#define        PIO_SET  0x07          ;port function register

#define        SIR_SET  0x0           ;SIR set(1/16 mode)
#define        SCR_SET  0x7           ;SCR set(#SCLK input 1.843MHz 115200bps)
#define        SCR_EN   0xc0          ;SCR enable
#define        PIO      0x000402d0    ;IO port (P port) register

      .text
;*******************************************************************************
;
;     void m_io_init()
;              serial port initial function
;
;*******************************************************************************
      .global  m_io_init
m_io_init:
      xld.w    %r1,SIR                                              (1)
      ld.w     %r0,SIR_SET            ;1/16 mode
      ld.b     [%r1],%r0              ;SIR set

      xld.w    %r1,SCR                                              (2)
      ld.w     %r0,SCR_SET
      ld.b     [%r1],%r0              ;SCR set(#SCLK input 1.843MHz)

      xld.w    %r1,PIO                                              (3)
      xld.w    %r0,PIO_SET
      ld.b     [%r1],%r0              ;IO port set

      xld.w    %r1,SCR                                              (4)
      xld.w    %r0,SCR_EN
      xoor     %r0,SCR_SET
      ld.b     [%r1],%r0              ;SCR set
      ret
```

(1) Selecting the division ratio

Set the division ratio of the sampling clock to 1/16.

```
      xld.w    %r1,SIR
      ld.w     %r0,SIR_SET      ;1/16 mode
      ld.b     [%r1],%r0        ;SIR set
```

(2) Setting transfer mode

Set transfer mode to asynchronous 8-bit mode, with one stop bit, no parity, and external clock for SCLK. For S5U1C331M2S communications, a 1.843 MHz external clock is fed from S5U1C330M1D1 (115,200 bps).

```
      xld.w    %r1,SCR
      ld.w     %r0,SCR_SET
      ld.b     [%r1],%r0        ;SCR set(#SCLK input 1.843MHz)
```

(3) Selecting input / output pin functions

Set the pins shared with I / O ports for serial interface mode.

```
      xld.w    %r1,PIO
      xld.w    %r0,PIO_SET
      ld.b     [%r1],%r0                ;IO port set
```

(4) Enabling transmit / receive

Enable transmit / receive operations.

```
      xld.w    %r1,SCR
      xld.w    %r0,SCR_EN
      xoor     %r0,SCR_SET
      ld.b     [%r1],%r0                ;SCR set
```

Transmit routine

```
;******************************************************************************
;
;     void m_snd_1byte( sdata )
;              1 byte send function
;                IN : uchar sdata (R6)  send data
;
;******************************************************************************
      .global  m_snd_1byte
m_snd_1byte:
      pushn    %r3                      ;save r3-r0
snd000:
      xld.w    %r0,SSR                  ;Address set
      xld.w    %r1,STDR

      xbtst    [%r0],0x1                ;TDBE1(bit1) == 0(full) ?           (1)
      jreq     snd000                   ;if full, jp snd000
      xld.b    [%r1],%r6                ;write data                         (2)
      popn     %r3                      ;restore r3-r0
      ret
```

(1) Checking the transmit buffer

Check the serial interface status register bits to determine if the transmit buffer is empty; wait until it is emptied.

```
   snd000:
      xbtst    [%r0],0x1                ;TDBE1(bit1) == 0(full) ?
      jreq     snd000                   ;if full, jp snd000
```

(2) Sending one byte of data

When the transmit buffer is empty, send one byte of data from R6.

```
      xld.b    [%r1],%r6                ;write data
```

Receive routine

```
;******************************************************************************
;
;     uchar m_rcv_1byte()
;              1 byte receive function
;                OUT : 0 receive OK
;                      1 receive ERROR (framing  err)
;                      2               (parity   err)
;                      3               (over run err)
;
;******************************************************************************
      .global  m_rcv_1byte
m_rcv_1byte:
      pushn    %r3                      ;save r3-r0
      xld.w    %r1,SSR                  ;Address set
      xld.w    %r2,SRDR
rcv000:
      xbtst    [%r1],0x0                ;RDBF1(bit0) == 0(empty) ?          (1)
      jreq     rcv000                   ;if empty, jp rcv000

      ld.w     %r4,0x0                                                      (2)
      xbtst    [%r1],0x4                ;FER1(bit4) == 0 ?
      jreq     rcv010
      xbclr    [%r1],0x4             ;FER1(bit4) 0 clear
      ld.w     %r4,0x1               ;return 1
rcv010:
      xbtst    [%r1],0x3             ;PER1(bit3) == 0 ?
      jreq     rcv020
      xbclr    [%r1],0x3             ;PER1(bit3) 0 clear
      ld.w     %r4,0x2               ;return 2
rcv020:
      xbtst    [%r1],0x2             ;OER1(bit2) == 0 ?
      jreq     rcv030
      xbclr    [%r1],0x2             ;OER1(bit2) 0 clear
      ld.w     %r4,0x3               ;return 3
rcv030:
      xld.b    %r0,[%r2]             ;read data                        (3)
      xld.w    %r1,m_rcv_data        ;read data set
      ld.b     [%r1],%r0
      popn     %r3                   ;restore r3-r0
      ret
```

(1) Wait for receive

Wait until receive data is placed in the buffer.

```
rcv000:
   xbtst    [%r1],0x0             ;RDBF1(bit0) == 0(empty) ?
   jreq     rcv000                ;if empty, jp rcv000
```

(2) Check for receive errors

Check for framing, parity, and overrun errors.

```
   ld.w     %r4,0x0
   xbtst    [%r1],0x4             ;FER1(bit4) == 0 ?
   jreq     rcv010
   xbclr    [%r1],0x4             ;FER1(bit4) 0 clear
   ld.w     %r4,0x1               ;return 1
rcv010:
   xbtst    [%r1],0x3             ;PER1(bit3) == 0 ?
   jreq     rcv020
   xbclr    [%r1],0x3             ;PER1(bit3) 0 clear
   ld.w     %r4,0x2               ;return 2
rcv020:
   xbtst    [%r1],0x2             ;OER1(bit2) == 0 ?
   jreq     rcv030
   xbclr    [%r1],0x2             ;OER1(bit2) 0 clear
   ld.w     %r4,0x3               ;return 3
```

(3) Reading out receive data

If no errors are found, read out one byte of receive data from the buffer and save it to RAM.

```
rcv030:
   xld.b    %r0,[%r2]             ;read data
   xld.w    %r1,m_rcv_data        ;read data set
   ld.b     [%r1],%r0
```

## ● Asynchronous communications using an internal clock

The transmit and receive sections are the same as with an external clock; only the initialize routine differs. Although this is a source for the S1C33104, it may be used in the same way as for the S1C33209, except that no pull-up processing is required.

Initialize routine

```
#define         MON_VER  0x11           ;moinitor firm-ware version
#define         P8TS3    0x0004014e     ;8bit timer3 clock rate register
#define         PT3      0x0004016c     ;8bit timer3 control register
#define         RLD3     0x0004016d     ;8bit timer3 reload data register
#define         STDR1    0x000401e5     ;transmit data register
#define         SRDR1    0x000401e6     ;receive data register
#define         SSR1     0x000401e7     ;serial status register
#define         SCR1     0x000401e8     ;serial control register
#define         SIR1     0x000401e9     ;IrDA control register
#define         PIO      0x000402d0     ;IO port (P port) register
#define         IOU      0x000402d3     ;IO port (P port) pull up register

      .text
;*****************************************************************************
;
;     void m_io_init()
;              serial port initial function
;
;*****************************************************************************
      .global  m_io_init
m_io_init:
      xld.w    %r1,SIR1                                           (1)
      ld.w     %r0,0x00
      ld.b     [%r1],%r0         ;SIR1 set
      xld.w    %r1,SCR1                                           (2)
      ld.w     %r0,0x03
      ld.b     [%r1],%r0         ;SCR1 set
      xld.w    %r1,PIO                                            (3)
      xld.w    %r0,0x30
      ld.b     [%r1],%r0         ;IO port set
      xld.w    %r1,IOU                                            (4)
      xld.w    %r0,0xff
      ld.b     [%r1],%r0         ;pull up set
      xld.w    %r1,P8TS3                                          (5)
      xld.w    %r0,0x80
      ld.b     [%r1],%r0         ;P8TS3 set
      xld.w    %r1,RLD3                                           (6)
;     xld.w    %r0,0x20          ;9600bps
      xld.w    %r0,0x7           ;38400bps
;     xld.w    %r0,0x3           ;38400bps(clock:10MHz)for debug
;     xld.w    %r0,0xf           ;19200bps
      ld.b     [%r1],%r0         ;RLD3 set
      xld.w    %r1,PT3                                            (7)
      ld.w     %r0,0x07
      ld.b     [%r1],%r0         ;PT3 set
      xld.w    %r1,SCR1                                           (8)
      xld.w    %r0,0xc3
      ld.b     [%r1],%r0         ;SCR1 set
      ret
```

(1) Selecting the division ratio

Set the division ratio of the sampling clock to 1/16.

```
      xld.w    %r1,SIR1
      ld.w     %r0,0x00
      ld.b     [%r1],%r0         ;SIR1 set
```

(2) Setting transfer mode

Set transfer mode to asynchronous 8-bit mode, with one stop bit, no parity, and internal clock (8-bit timer 3).

```
      xld.w    %r1,SCR1
      ld.w     %r0,0x03
      ld.b     [%r1],%r0         ;SCR1 set
```

(3) Selecting input / output pin functions
Set the pins shared with I / O ports for serial interface mode.

```
    xld.w    %r1,PIO
    xld.w    %r0,0x30
    ld.b     [%r1],%r0         ;IO port set
```

(4) Setting pull-ups (S1C33104)
Enable pull-ups for the serial interface input pins. This processing is used for the S1C33104 but is not required for the S1C33209. For real-world applications, we recommend connecting pull-up resistors external to the chip regardless of microcomputer type.

```
    xld.w    %r1,IOU
    xld.w    %r0,0xff
    ld.b     [%r1],%r0         ;pull up set
```

(5) Setting the prescaler
Set the prescaler's division ratio for 8-bit timer 3 to 1 / 2 of internal clock. For the S1C33209, you can also select 1 / 1 (0x40146 • D3 = 1).

```
    xld.w    %r1,P8TS3
    xld.w    %r0,0x80
    ld.b     [%r1],%r0         ;P8TS3 set
```

(6) Setting the 8-bit timer
Preset value 7 (7 + 1 = divide-by-8) in the 8-bit timer. Results for the Proto Board of Epson (operating clock = 20 MHz) are as follows.
20 MHz → divided by 2 by prescaler → divided by 8 by timer → divided by 2 by serial interface → divided by 16 for sampling use = 20,000,000 / 2 / 8 / 2 / 16 = 39,062 bps

This creates a +1.7% error with respect to 38,400 bps. An error of this magnitude will not affect the other side any significantly, no operational problems should result under normal conditions.

```
    xld.w    %r1,RLD3
    xld.w    %r0,0x7           ;38400bps
    ld.b     [%r1],%r0         ;RLD3 set
```

(7) Starting the 8-bit timer
Start the 8-bit timer.

```
    xld.w    %r1,PT3
    ld.w     %r0,0x07
    ld.b     [%r1],%r0         ;PT3 set
```

(8) Enabling transmit / receive
Enable transmit / receive operations.

```
    xld.w    %r1,SCR1
    xld.w    %r0,0xc3
    ld.b     [%r1],%r0         ;SCR1 set
```

## *3.5 Setting Up A/D Converter*

This section describes a software-triggered A/D conversion routine, using a sample excerpted from gnu33\sample\drv33209\demo_ad2\.

Vector table [vector.c]

```
extern void int_ad(void);                                                      (1)

/* vector table */
const unsigned long vector[] = {
      (unsigned long)boot,             // 0        0
                |
      (unsigned long)dummy,            // 248      62
      (unsigned long)dummy,            // 252      63
      (unsigned long)int_ad,           // 256      64                          (1)
      (unsigned long)dummy,            // 260      65
      (unsigned long)dummy,            // 264      66
      (unsigned long)dummy,            // 268      67
      (unsigned long)dummy,            // 272      68
      (unsigned long)dummy,            // 276      69
      (unsigned long)dummy,            // 280      70
      (unsigned long)dummy             // 284      71
};
```

(1) Setting the vector table
This sample generates an interrupt on completion of A/D conversion and acquires the A/D converted data in an interrupt routine. Register the start address of this interrupt routine in the vector table (at vector table start address + 0x100).

Initializing A/D converter [drv_ad2.c]

```
#include "..\include\ad.h"
#include "..\include\common.h"
#include "..\include\int.h"
#include "..\include\io.h"
#include "..\include\presc.h"

/* Prototype */
void init_ad(void);
unsigned short read_ad_data(void);
void int_ad(void);
extern void save_psr(void);
extern void restore_psr(void);

/********************************************************************************
 * init_ad
 *   Type :     void
 *   Ret val :  none
 *   Argument : void
 *   Function : Initialize A/D converter.
 ********************************************************************************/
void init_ad(void)
{
      /* Save PSR and disable all interrupt */
      save_psr();

      /* Set A/D converter port setting */
      *(volatile unsigned char *)IN_CFK6_ADDR = IN_CFK60_AD0;  // A/D ch.0 port  (1)

      /* SPT = A/D converter sampling time
           OSC3 = OSC3 clock (40MHz)
           PDR = Prescaler clock division (1/32)
           ST = A/D converter sampling time (9clock)
           TADC = A/D converter sampling and  convert time (10us)
              SPT = ST / (OSC3 x PDR)
                  = 9 / (40 x 1000000 x 1/32)
                  = 7.2us
              Must be SPT > TADC / 2 */
```

```
    /* Set A/D converter prescaler setting (CLK/32) */
    *(volatile unsigned char *)PRESC_PSAD_ADDR                              (2)
             = PRESC_PTONL_ON | PRESC_CLKDIVL_SEL4;
             // Set A/D converter prescaler (CLK/32)

    /* Set A/D converter status register */
    *(volatile unsigned char *)AD_CH_ADDR = AD_MS_NOR | AD_TS_SOFT;         (3)
             // A/D converter software trigger and normal mode
    *(volatile unsigned char *)AD_CS_ADDR = AD_CS_0 | AD_CE_0;              (3)
             // A/D converter start channel AD0 and A/D end channel AD0
    *(volatile unsigned char *)AD_OWE_ADDR                                  (3)
             =  AD_ADE_ENA | AD_ADST_STOP | AD_OWE_NOERR;
             // A/D converter enable, A/D converter stop,
             // A/D converter over write error clear
    *(volatile unsigned char *)AD_ST_ADDR = AD_ST_9;                        (2)
             // A/D converter sampling 9 clocks

    /* Set A/D converter interrupt CPU request on interrupt controller */
    *(volatile unsigned char *)INT_RS1_RADE_RP4_ADDR = INT_RIDMA_DIS;
             // IDMA request disable and CPU request enable                (4)

    /* Set A/D converter interrupt priority level 3 on interrupt controller */
    *(volatile unsigned char *)INT_PSIO1_PAD_ADDR = INT_PRIH_LVL3;          (4)

    /* Reset A/D converter interrupt factor flag on interrupt controller */
    *(volatile unsigned char *)INT_FADE_FCTM_FP4_ADDR = INT_FADE;           (4)
             // Reset A/D converter interrupt factor flag

    /* Set A/D converter interrupt enable on interrupt controller */
    *(volatile unsigned char *)INT_EADE_ECTM_EP4_ADDR = INT_EADE;           (4)
             // Set A/D converter interrupt enable

    /* Restore PSR */
    restore_psr();
}
```

A group of include files listed at the top of this routine is found in gnu33\sample\drv33209\include. Refer to each file for detailed information on the contents of definition.

(1) Setting the analog input pin
Set the A/D converter channel 0 input pin (which is shared with K60 general-purpose input port) for analog input. (By default, it is used as a K60 general-purpose input pin.)

```
/* Set A/D converter port setting */
*(volatile unsigned char *)IN_CFK6_ADDR = IN_CFK60_AD0;     // A/D ch.0 port
```

(2) Setting the prescaler and sampling time

```
/* SPT = A/D converter sampling time
     OSC3 = OSC3 clock (40MHz)
     PDR = Prescaler clock division (1/32)
     ST = A/D converter sampling time (9clock)
     TADC = A/D converter sampling and  convert time (10us)
          SPT = ST / (OSC3 x PDR)
              = 9 / (40 x 1000000 x 1/32)
              = 7.2us
        Must be SPT > TADC / 2 */
```

This comment demonstrates how the A/D converter input clock is calculated. First, set the prescaler's division ratio at which the A/D converter operating clock is generated from the system clock. Any multiple of 2 from 1/2 to 1/256 can be selected. Here, anticipating the use of a 40 MHz system clock, we set the prescaler's division ratio to 1/32.
Next, set the input sampling time to 9 A/D converter clock periods. This is the sample-and-hold time. This time must be equal to or greater than 1/2 (5 µs or more) of A/D conversion time $t_{ADC}$ (min. 10 µs). In this example, this is 7.2 µs. If 1/16 is selected for the prescaler, it is doubled to 3.6 µs. Although no operational problems will results with a sampling time of 5 µs or less, reduced sampling times may result in more frequent errors. Following a sample-and-hold, the A/D converter performs a successive comparison in approximately 10 clock periods and outputs a 10-bit A/D conversion result.

Set the prescaler division ratio for the A / D converter to 1 / 32. Set the sampling time for A / D conversion to 9 clock periods.

```
/* Set A/D converter prescaler setting (CLK/32) */
*(volatile unsigned char *)PRESC_PSAD_ADDR = PRESC_PTONL_ON | PRESC_CLKDIVL_SEL4;
   // Set A/D converter prescaler (CLK/32)
                              |
*(volatile unsigned char *)AD_ST_ADDR = AD_ST_9;
   // A/D converter sampling 9 clocks
```

(3) Setting the A / D converter

For conversion mode (continuous or normal), select normal. For trigger (external / K52, 8-bit timer 0, 16-bit timer 0, or software trigger), select software trigger.

```
/* Set A/D converter status register */
*(volatile unsigned char *)AD_CH_ADDR = AD_MS_NOR | AD_TS_SOFT;
   // A/D converter software trigger and normal mode
```

Set the conversion channel to channel 0.

```
*(volatile unsigned char *)AD_CS_ADDR = AD_CS_0 | AD_CE_0;
   // A/D converter start channel AD0 and A/D end channel AD0
```

Enable A / D conversion.

```
*(volatile unsigned char *)AD_OWE_ADDR
    =  AD_ADE_ENA | AD_ADST_STOP | AD_OWE_NOERR;
   // A/D converter enable, A/D converter stop, A/D .. over write error clear
```

(4) Setting interrupt

Using the interrupt controller, set the A / D conversion interrupt as an interrupt request to the CPU.

```
/* Set A/D converter interrupt CPU request on interrupt controller */
*(volatile unsigned char *)INT_RS1_RADE_RP4_ADDR = INT_RIDMA_DIS;
   // IDMA request disable and CPU request enable
```

Set the interrupt level to 3.

```
/* Set A/D converter interrupt priority level 3 on interrupt controller */
*(volatile unsigned char *)INT_PSIO1_PAD_ADDR = INT_PRIH_LVL3;
```

Clear the interrupt factor flag.

```
/* Reset A/D converter interrupt factor flag on interrupt controller */
*(volatile unsigned char *)INT_FADE_FCTM_FP4_ADDR = INT_FADE;
   // Reset A/D converter interrupt factor flag
```

Enable the interrupt.

```
/* Set A/D converter interrupt enable on interrupt controller */
*(volatile unsigned char *)INT_EADE_ECTM_EP4_ADDR = INT_EADE;
   // Set A/D converter interrupt enable
```

Interrupt processing [drv_ad2.c]

```
/******************************************************************************
 * read_ad_data
 *   Type :     unsigned short
 *   Ret val :  A/D converter data
 *   Argument : void
 *   Function : Read A/D converter data.
 ******************************************************************************/
unsigned short read_ad_data(void)
{
      return(*(volatile unsigned short *)AD_ADD_ADDR);   // A/D converter data   (2)
}

/******************************************************************************
 * int_ad
 *   Type :     void
 *   Ret val :  none
 *   Argument : void
 *   Function : A/D converter interrupt function.
 *                Read A/D converter status and A/D convert data.
 ******************************************************************************/
void int_ad(void)
{
      extern volatile unsigned short ad_data;// A/D data
      extern volatile int ad_int;            // A/D converter interrupt flag

      INT_BEGIN;                                                                  (1)
      ad_data = read_ad_data();  // Read A/D converter data                       (2)
      ad_int = TRUE;             // A/D converter interrupt flag on
      *(volatile unsigned char *)INT_FADE_FCTM_FP4_ADDR = INT_FADE;
                                 // Reset A/D converter interrupt factor flag    (3)
      INT_END;                                                                    (1)
}
```

When A / D conversion is complete, an A / D interrupt is generated and int_ad() is called.

(1) Saving / restoring registers

To save and restore registers at the beginning and end of the interrupt handling routine, we use INT_BEGIN and INT_END, defined in common.h.

```
#define INT_BEGIN    asm("pushn  %r15")
#define INT_END      asm("popn   %r15\n   reti")
```

(2) Reading out the conversion result

Call read_ad_data(), store the A / D conversion result in a variable, and set a flag to indicate that readout is complete.

```
ad_data = read_ad_data();     // Read A/D converter data
ad_int = TRUE;                // A/D converter interrupt flag on
```

(3) Resetting the interrupt factor flag

Clear the interrupt factor flag.

```
*(volatile unsigned char *)INT_FADE_FCTM_FP4_ADDR = INT_FADE;
   // Reset A/D converter interrupt factor flag
```

Application section [demo_ad2.c]

```
    |
unsigned short ad_data;
volatile int ad_int;             // A/D converter interrupt flag
    |
init_ad();                                                                  (1)
    |
for (i = 0; i < DATA_SIZE; i++) {
      ad_int = FALSE;
      /* A/D converter start by software trigger */
      *(volatile unsigned long*)AD_OWE_ADDR |= 0x02;                        (2)
                                 // Set A/D converter run bit (ADST[D1] = 1)

      for (;;) {
               if (ad_int == TRUE) {                                        (3)
                        write_str("    A/D AD0 data ... ");
                        write_hex(ad_data);
                        break;
               }
      }
}
```

This control program performs actual A / D conversion.

(1) Initializing
Call the previously mentioned init_ad() and initialize the A / D converter and interrupt settings.

(2) Starting A / D conversion
Start A / D conversion with a software trigger.
```
/* A/D converter start by software trigger */
*(volatile unsigned long*)AD_OWE_ADDR |= 0x02;
                    // Set A/D converter run bit (ADST[D1] = 1)
```

(3) Getting A / D conversion result
When A / D conversion is complete, the previously mentioned interrupt handling routine int_ad() is called. When processing is complete, the flag ad_int is set. Check this flag; if set to 1, read out the conversion result from the variable ad_data for display on the screen.
```
for (;;) {
   if (ad_int == TRUE) {
            write_str("       A/D AD0 data ... ");
            write_hex(ad_data);
            break;
   }
}
```

## *3.6 Setting Up IDMA*

An example of the transfer of data after invoking IDMA with a software trigger is described here using an excerpt from the sample in gnu33\sample\drv33209\idma.

Header files for IDMA table [\sample\include\idma.h]

```
        |
#define  IDMA_DCHN_ADDR    0x48204         // Address for IDMA channel number, IDMA
                                              start register
#define  IDMA_DMAEN_ADDR   0x48205         // Address for IDMA enable register
        |
/* IDMA control word bit field definition ... 1st word */
#define  IDMA_LNKEN_ENA    0x80000000      // Link enable
        |
/* IDMA control word bit field definition ... 2nd word */
#define  IDMA_DINTEN_ENA   0x80000000      // IDMA terminate interrupt enable
#define  IDMA_DINTEN_DIS   0x00000000      // IDMA terminate interrupt disable
#define  IDMA_DATSIZ_HW    0x40000000      // Data size .. half word
#define  IDMA_DATSIZ_BYTE  0x00000000      // Data size .. byte
#define  IDMA_SRINC_INC    0x30000000      // Increase address (return initial value
                                              when block transfer mode)
        |
/* IDMA control word bit field definition ... 3rd word */
#define  IDMA_DMOD_BLOCK   0x80000000      // Block transfer mode
        |
#define  IDMA_DSINC_INC    0x30000000      // Increase address (return initial value
                                              when block transfer mode)
        |
```

Setting up and starting IDMA [demo_idma.c]

```
        |
#include "..\include\common.h"
#include "..\include\idma.h"

/* Prototype */
int main(void);
void fill_mem1(unsigned long);
void fill_mem2(unsigned long);
void check_data(unsigned long , unsigned long);
extern void write_str(char *);
extern void init_idma(unsigned long, unsigned char);
extern void write_idma_info(unsigned long *, unsigned long, unsigned long, unsigned
long);

/* Global variable define */
volatile int idmaint_flg;              // IDMA interrupt flag

/* IDMA source and destination address */
#define  IDMA_SRC_START0   0x6c0000    // IDMA ch.0 source start address
#define  IDMA_SRC_START1   0x6d0000    // IDMA ch.1 source start address
#define  IDMA_DST_START0   0x6e0000    // IDMA ch.0 destination start address
#define  IDMA_DST_START1   0x6f0000    // IDMA ch.1 destination start address

/* IDMA start channel, link channel, transfer block size, transfer counter */
#define  IDMA_ST_CH0       0x0         // IDMA start ch.0
#define  IDMA_LINK         0x1000000   // IDMA transfer link ch.1 (1st word:D30-D24)
#define  IDMA_CNT          0x100       // IDMA transfer count (1st word:D23-D8)
#define  IDMA_BSIZE        0x80        // IDMA block size (1st word:D7-D0)

struct {
         unsigned long data[3];
} dma_control[128];                                                            (1)

/*******************************************************************************
 * main
 *   Type :     void
 *   Ret val :  none
 *   Argument : void
 *   Function : IDMA demonstration program.
 ******************************************************************************/
```

```
int main(void)
{
   unsigned long first_wd, second_wd, third_wd;
   unsigned char start_ch;

   write_str("*** IDMA demonstration ***\n");
   write_str("\n");

   /* Fill memory */
   write_str("*** Filling memory area(address 0x6c0000 - 0x6c00ff) ***\n");
   write_str("    Pattern  ... 0x00 0x01 .... 0xfe 0xff\n");
   fill_mem1(IDMA_SRC_START0);
   write_str("*** Filling memory area(address 0x6e0000 - 0x6e00ff) ***\n");
   write_str("    Pattern  ... 0xff 0xfe .... 0x01 0x00\n");
   fill_mem2(IDMA_SRC_START1);

   /* Disable IDMA transfer */
   *(volatile unsigned char*)IDMA_DMAEN_ADDR &= 0xfe;

   /* Initialize IDMA */
   write_str("*** Initialize IDMA following setting ***\n");
   write_str("\n");
   write_str("  IDMA ch.0 setting\n");
   write_str("       LINK enable ... Next channel is 1\n");
   write_str("       Transfer 1 time\n");
   write_str("       128 half word transfer\n");
   write_str("       Interrupt enable\n");
   write_str("       Half word data size\n");
   write_str("       Source and destination address increment\n");
   write_str("       Block transfer mode\n");
   /* Set IDMA ch.0 */                                                    (2)
   first_wd = IDMA_LNKEN_ENA | IDMA_LINK | IDMA_CNT | IDMA_BSIZE;
   second_wd = IDMA_DINTEN_ENA | IDMA_DATSIZ_HW| IDMA_SRINC_INC | IDMA_SRC_START0;
   third_wd = IDMA_DMOD_BLOCK | IDMA_DSINC_INC | IDMA_DST_START0;
   write_idma_info((unsigned long *)(&dma_control[0]), first_wd, second_wd, third_wd);

   write_str("\n");
   write_str("  IDMA ch.1 setting\n");
   write_str("       LINK disable\n");
   write_str("       Transfer 1 time\n");
   write_str("       128 half word transfer\n");
   write_str("       Interrupt enable\n");
   write_str("       Half word data size\n");
   write_str("       Source and destination address increment\n");
   write_str("       Block transfer mode\n");
   /* Set IDMA ch.1 */                                                    (3)
   first_wd = IDMA_LNKEN_DIS | IDMA_CNT | IDMA_BSIZE;
   second_wd = IDMA_DINTEN_ENA |IDMA_DATSIZ_HW | IDMA_SRINC_INC | IDMA_SRC_START1;
   third_wd = IDMA_DMOD_BLOCK | IDMA_DSINC_INC | IDMA_DST_START1;
   write_idma_info((unsigned long *)(&dma_control[1]), first_wd, second_wd, third_wd);

   /* Intialize IDMA control information and start channel */
   start_ch = IDMA_ST_CH0;
   init_idma((unsigned long)dma_control, start_ch);                       (1)
                                    // Set IDMA control information and start ch.0

   /* Initialize IDMA interrupt flag */
   idmaint_flg = FALSE;

   /* Enable IDMA transfer */
   *(volatile unsigned char*)IDMA_DMAEN_ADDR |= 0x01;

   /* Start IDMA transfer */
   write_str("\n");
   write_str("*** IDMA transfer starts by software trigger ***\n");
   *(volatile unsigned char *)IDMA_DCHN_ADDR |= 0x80;                     (4)

   while (1) {
           if (idmaint_flg == TRUE) {
                   break;
           }
   }

   /* Disable IDMA transfer */
   *(volatile unsigned char *)IDMA_DMAEN_ADDR &= 0xfe;   // Disable IDMA transfer.
```

```
    /* Checking IDMA data */
    write_str("\n");
    check_data(IDMA_SRC_START0, IDMA_DST_START0);
    write_str("\n");
    check_data(IDMA_SRC_START1, IDMA_DST_START1);

    write_str("\n");
    write_str("*** IDMA demonstration finish ***\n");
}
        |
```

(1) Reserving a control information area

Reserve a 128-channel control information area in RAM using the structure shown below.

```
struct {
    unsigned long data[3];
} dma_control[128];
            :
init_idma((unsigned long)dma_control, start_ch); // Set IDMA control information
```

The address of this structure is set in the IDMA base address register (0x48200, 0x48202) by init_idma() in drv_idma.c.

```
void init_idma(unsigned long addr, unsigned char ch)
{
            :
    /* Set IDMA control information address */
    *(volatile unsigned long *)IDMA_DBASEL_ADDR = addr;
            :
```

In this sample, the two channels ch0 and ch1 are set as shown below.

(2) Settings of IDMA ch0

The IDMA ch0 transfers data from the address 0x6c0000–0x6c00ff to the address 0x6e0000–0x6e00ff. The data size and the transfer count are set to 16 bits and 128 times, respectively, with data copied from the above address in one transfer. Furthermore, the two channels are linked so that ch1 is started upon completion of a transfer on ch0.

```
/* Set IDMA ch.0 */
first_wd = IDMA_LNKEN_ENA | IDMA_LINK | IDMA_CNT | IDMA_BSIZE;
second_wd = IDMA_DINTEN_ENA | IDMA_DATSIZ_HW| IDMA_SRINC_INC | IDMA_SRC_START0;
third_wd = IDMA_DMOD_BLOCK | IDMA_DSINC_INC | IDMA_DST_START0;
write_idma_info((unsigned long *)(&dma_control[0]), first_wd, second_wd, third_wd);
```

write_idma_info() is included in drv_idma.c.

```
void write_idma_info(unsigned long *addrptr, unsigned long word1,
         unsigned long word2, unsigned long word3)
{
         /* Set IDMA control information */
         *addrptr = word1;                // Write 1st word
         *(addrptr + 1) = word2;          // Write 2nd word
         *(addrptr + 2) = word3;          // Write 3rd word
}
```

(3) Settings of IDMA ch1

The IDMA ch1 transfers data from the address 0x6d0000–0x6d00ff to the address 0x6f0000–0x6f00ff. The data size and the transfer count are set to 16 bits and 128 times, respectively, with data copied from the above address in one transfer. No link channels are set that will be started upon completion of a transfer on ch1.

```
/* Set IDMA ch.1 */
first_wd = IDMA_LNKEN_DIS | IDMA_CNT | IDMA_BSIZE;
second_wd = IDMA_DINTEN_ENA |IDMA_DATSIZ_HW | IDMA_SRINC_INC | IDMA_SRC_START1;
third_wd = IDMA_DMOD_BLOCK | IDMA_DSINC_INC | IDMA_DST_START1;
write_idma_info((unsigned long *)(&dma_control[1]), first_wd, second_wd, third_wd);
```

(4) Executing a transfer

Write 0x80 to the IDMA start register (0x48204) in order to start ch0 using a software trigger.

```
*(volatile unsigned char *)IDMA_DCHN_ADDR |= 0x80;
```

Data transfer is performed exactly as in the case of the control information that has been set for ch0, upon completion of which ch1 is started by linkage, with data transferred in the same way.

## *3.7 Setting Up HSDMA*

An example of the transfer of data after HSDMA is invoked using a software trigger is described here using an excerpt from the sample in gnu33\sample\drv33209\hsdma.

Header files for HSDMA table [\sample\include\hsdma.h]

```
                |
#define  HSDMA_HSDMA_ADDR   0x4029a    // Address for HSDMA software trigger register
                |
#define  HSDMA_HSD1_SOFT    0x0000     // HSDMA ch.1 software trigger
                |
#define  HSDMA_HST1         0x02       // HSDMA ch.1 software trigger
                |
#define  HSDMA_DUAL_DUAL    0x80000000 // HSDMA dual mode
                |
#define  HSDMA_DINTEN_ENA   0x80000000 // HSDMA interrupt enable
                |
#define  HSDMA_DATSIZE_HALF 0x40000000 // HSDMA half-word
                |
#define  HSDMA_DMOD_BLK     0x80000000 // HSDMA transfer mode
                |
#define  HSDMA_INC_INIT     0x20000000 // HSDMA address control Inc.(init)
                |
```

Setting up and starting HSDMA [demo_hsdma.c]

```
                |
#include "..\include\common.h"
#include "..\include\hsdma.h"

/* Prototype */
int main(void);
void fill_mem(unsigned long);
void check_data(unsigned long , unsigned long);
extern void write_str(char *);
extern void init_hsdma(unsigned char, unsigned char, unsigned long, unsigned long,
unsigned long);

/* Global variable define */
volatile int hdmaint_flg;              // High-speed DMA interrupt flag

/* HSDMA source and destination address */
#define     HSDMA_SRC_START   0x6c0000 // HSDMA ch.1 source start address
#define     HSDMA_DST_START   0x6d0000 // HSDMA ch.1 destination start address

/* HSDMA channel */
#define     HSDMA_CH1         1        // HSDMA ch.1

/******************************************************************************
 * main
 *   Type :     void
 *   Ret val :  none
 *   Argument : void
 *   Function : High-speed DMA demonstration program.
 ******************************************************************************/
int main(void)
{
   unsigned char trig;
   unsigned long mode, src, dst;

   write_str("*** High-speed DMA demonstration ***\n");
   write_str("\n");

   /* Fill memory */
   write_str("*** Filling memory area(address 0x6c0000 - 0x6c00ff) ***\n");
   write_str("    Pattern  ... 0x00 0x01 .... 0xfe 0xff\n");
   fill_mem(HSDMA_SRC_START);

   /* Initialize HSDMA */
   write_str("*** Initialize HSDMA following setting ***\n");
   write_str("  HSDMA ch.1 setting\n");
   write_str("       Transfer 1 time\n");
   write_str("       128 half word transfer\n");
   write_str("       Interrupt enable\n");
   write_str("       Half word data size\n");
   write_str("       Source and destination address increment\n");
   write_str("       Block transfer mode\n");

   /* Disable HSDMA transfer */                                              (1)
   *(volatile unsigned char *)HSDMA_HS1EN_ADDR &= 0xfe;  // Disable HSDMA transfer.

   /* HSDMA trigger mode */                                                  (1)
   trig = HSDMA_HSD1_SOFT;                      // HSDMA ch.1 software trigger
   /* HSDMA mode and transfer count */                                       (1)
   mode = HSDMA_DUAL_DUAL | (1 << 8) | 0x80;    // HSDMA dual mode and transfer count
1, block size 0x80
   /* HSDMA source address */                                                (1)
   src = HSDMA_DINTEN_ENA | HSDMA_DATSIZE_HALF | HSDMA_INC_INIT | HSDMA_SRC_START;
         // HSDMA interrupt enable, half word size, source address increment
(init.), source address 0x6c0000
   /* HSDMA destination address */
   dst = HSDMA_DMOD_BLK | HSDMA_INC_INIT | HSDMA_DST_START;                  (1)
         // Destination address increment (init.), destination address 0x6d0000
   init_hsdma(HSDMA_CH1, trig, mode, src, dst);                              (1)

   /* Initialize HSDMA interrupt flag */
   hdmaint_flg = FALSE;

   /* Enable IDMA transfer */                                                (2)
   *(volatile unsigned char *)HSDMA_HS1EN_ADDR |= 0x01;  // Enable HSDMA transfer.

   /* Start HSDMA transfer */
   write_str("\n");
   write_str("*** HSDMA transfer starts by software trigger ***\n");
   *(volatile unsigned char *)HSDMA_HSDMA_ADDR |= HSDMA_HST1;                (2)

   while (1) {
         if (hdmaint_flg == TRUE) {
               break;
         }
   }

   /* Disable HSDMA transfer */
   *(volatile unsigned char *)HSDMA_HS1EN_ADDR &= 0xfe;  // Disable HSDMA transfer.

   /* Checking HSDMA data */
   write_str("\n");
   check_data(HSDMA_SRC_START, HSDMA_DST_START);

   write_str("\n");
   write_str("*** HSDMA demonstration finish ***\n");
}
              |
```

(1) Setting the HSDMA

Always confirm that HSDMA is disabled before setting HSDMA. If set while operating, the register may be read or written to incorrectly.

`*(volatile unsigned char *)HSDMA_HS1EN_ADDR &= 0xfe;  // Disable HSDMA transfer.`

Set up HSDMA ch1 as shown below.

| | |
|---|---|
| • Trigger | Software trigger |
| • Address mode | Dual address mode |
| • Transfer mode | Block transfer mode |
| • Data size | Half word |
| • Source and destination address control | Increment (including initialization) |
| • Transfer counter | 128 |
| • Source address | 0x6c0000 |
| • Destination address | 0x6d0000 |

```
   /* HSDMA trigger mode */
   trig = HSDMA_HSD1_SOFT;                      // HSDMA ch.1 software trigger
   /* HSDMA mode and transfer count */
   mode = HSDMA_DUAL_DUAL | (1 << 8) | 0x80;    // HSDMA dual mode and transfer
count 1, block size 0x80
   /* HSDMA source address */
   src = HSDMA_DINTEN_ENA | HSDMA_DATSIZE_HALF | HSDMA_INC_INIT | HSDMA_SRC_START;
            // HSDMA interrupt enable, half word size, source address increment
(init.), source address 0x6c0000
   /* HSDMA destination address */
   dst = HSDMA_DMOD_BLK | HSDMA_INC_INIT | HSDMA_DST_START;
            // Destination address increment (init.), destination address 0x6d0000
   init_hsdma(HSDMA_CH1, trig, mode, src, dst);
```

(2) Executing a transfer

Enable HSDMA after all settings have been made.

```
/* Enable IDMA transfer */
*(volatile unsigned char *)HSDMA_HS1EN_ADDR |= 0x01;  // Enable HSDMA transfer.
```

Set the ch1 software trigger bit (D1) in the software trigger register (0x4029a) to "1", in order to start HSDMA ch1.

```
/* Start HSDMA transfer */
write_str("\n");
write_str("*** HSDMA transfer starts by software trigger ***\n");
*(volatile unsigned char *)HSDMA_HSDMA_ADDR |= HSDMA_HST1;
```

## *3.8 Clock Settings*

The S1C33209 includes a clock timer capable of counting up to 64K days in units of 1/128 seconds. Here, we will explain how to generate an alarm interrupt exactly one minute later using this clock timer. The example program used here can be found in gnu33\sample\drv33209\ct.

### ● The one-minute alarm interrupt

Vector table [vector.c]

```
/* vector table */
const unsigned long vector[] = {
      (unsigned long)boot,             // 0        0
               |
               |
      (unsigned long)dummy,            // 252      63
      (unsigned long)dummy,            // 256      64
      (unsigned long)int_ct,           // 260      65                      (1)
      (unsigned long)dummy,            // 264      66
               |
};
```

(1) Setting the vector table
   Register the interrupt handling routine int_c as the clock timer interrupt vector.

Initial settings [drv_ct.c]

```
#include "..\include\common.h"
#include "..\include\ct.h"
#include "..\include\int.h"

/* Prototype */
void init_ct(void);
void int_ct(void);
extern void save_psr(void);
extern void restore_psr(void);

/********************************************************************************
 * init_ct
 *   Type :     void
 *   Ret val :  none
 *   Argument : void
 *   Function : Initialize clock timer to use real time clock.
 ********************************************************************************/
void init_ct(void)
{
      /* Save PSR and disable all interrupt */
      save_psr();                                                          (1)

      /* Set clock timer interrupt disable on interrupt controller */
      *(volatile unsigned char *)INT_EADE_ECTM_EP4_ADDR = INT_ENABLE_DIS;
               // Set clock timer interrupt disable

      /* Stop clock timer */
      *(volatile unsigned char *)CT_TCRUN_ADDR &= 0xfe;                    (2)

      /* Reset clock timer */
      *(volatile unsigned char *)CT_TCRUN_ADDR |= CT_TCRST_RST;

      /* Set clock timer data (1999.01.01 21:05) */
      *(volatile unsigned char *)CT_TCHD_ADDR = 0x05;                      (3)
               // Minute data (5 minutes)
      *(volatile unsigned char *)CT_TCDD_ADDR = 0x15;
               // Hour data (21 hours)
      *(volatile unsigned char *)CT_TCNDL_ADDR = 0xd7;
               // Year-month-day low byte data (3287 days)
      *(volatile unsigned char *)CT_TCNDH_ADDR = 0x0c;
               // Year-month-day high byte data (3287 days)
```

```
        /* Set clock timer comparison data */
        *(volatile unsigned char *)CT_TCCH_ADDR = 0x06;                            (4)
                // Minute comparison data (6 minutes)
        *(volatile unsigned char *)CT_TCCD_ADDR = 0x0;
                // Hour comparison data (0 hour)
        *(volatile unsigned char *)CT_TCCN_ADDR = 0x0;
                // Day comparison data (0 day)

        /* Set clock timer interrupt factor control flag */
        *(volatile unsigned char *)CT_TCAF_ADDR                                    (5)
                = CT_TCISE_NONE | CT_TCASE_M | CT_TCIF_RST | CT_TCAF_RST;

        /* Set clock timer interrupt priority level 3 on interrupt controller */
        *(volatile unsigned char *)INT_PCTM_ADDR = INT_PRIL_LVL3;                  (6)

        /* Reset clock timer interrupt factor flag on interrupt controller */
        *(volatile unsigned char *)INT_FADE_FCTM_FP4_ADDR = INT_FCTM;
                // Reset clock timer interrupt factor flag

        /* Set clock timer interrupt enable on interrupt controller */
        *(volatile unsigned char *)INT_EADE_ECTM_EP4_ADDR = INT_ECTM;
                // Set clock timer interrupt enable

        /* Restore PSR */
        restore_psr();                                                             (7)
}
```

(1) Disabling interrupts

Save PSR and mask interrupts with IE.

```
/* Save PSR and disable all interrupt */
save_psr();
```

Using the interrupt controller, disable the clock timer interrupt.

```
/* Set clock timer interrupt disable on interrupt controller */
*(volatile unsigned char *)INT_EADE_ECTM_EP4_ADDR = INT_ENABLE_DIS;
   // Set clock timer interrupt disable
```

(2) Resetting the clock timer

After stopping the clock timer, reset the counter.

```
/* Stop clock timer */
*(volatile unsigned char *)CT_TCRUN_ADDR &= 0xfe;

/* Reset clock timer */
*(volatile unsigned char *)CT_TCRUN_ADDR |= CT_TCRST_RST;
```

(3) Setting the date and time

Set the date and time to 21:05, January 1, 1999. The 3287 days set in the day counter are calculated using January 1, 1990 as the starting point.

```
/* Set clock timer data (1999.01.01 21:05) */
*(volatile unsigned char *)CT_TCHD_ADDR = 0x05;
   // Minute data (5 minutes)
*(volatile unsigned char *)CT_TCDD_ADDR = 0x15;
   // Hour data (21 hours)
*(volatile unsigned char *)CT_TCNDL_ADDR = 0xd7;
   // Year-month-day low byte data (3287 days)
*(volatile unsigned char *)CT_TCNDH_ADDR = 0x0c;
   // Year-month-day high byte data (3287 days)
```

(4) Setting an alarm

Here, we set 6 minutes as comparison data and set the alarm interrupt to occur in one minute.

```
/* Set clock timer comparison data */
*(volatile unsigned char *)CT_TCCH_ADDR = 0x06;
   // Minute comparison data (6 minutes)
*(volatile unsigned char *)CT_TCCD_ADDR = 0x0;
   // Hour comparison data (0 hour)
*(volatile unsigned char *)CT_TCCN_ADDR = 0x0;
   // Day comparison data (0 day)
```

(5) Settings for alarm interrupt

Enable only the minutes alarm interrupt. Clear the interrupt fuctor generation and alarm fuctor generation flags.

```
/* Set clock timer interrupt factor control flag */
*(volatile unsigned char *)CT_TCAF_ADDR
   = CT_TCISE_NONE | CT_TCASE_M | CT_TCIF_RST | CT_TCAF_RST;
```

These steps set the internal functions of the clock timer, not the interrupt controller. This control register must always be reset before use, since its initial value cannot be guaranteed.

(6) Setting the interrupt controller

Set the interrupt level to 3.

```
/* Set clock timer interrupt priority level 3 on interrupt controller */
*(volatile unsigned char *)INT_PCTM_ADDR = INT_PRIL_LVL3;
```

Clear the clock timer interrupt factor flag.

```
/* Reset clock timer interrupt factor flag on interrupt controller */
*(volatile unsigned char *)INT_FADE_FCTM_FP4_ADDR = INT_FCTM;
   // Reset clock timer interrupt factor flag
```

Enable the clock timer interrupt.

```
/* Set clock timer interrupt enable on interrupt controller */
*(volatile unsigned char *)INT_EADE_ECTM_EP4_ADDR = INT_ECTM;
   // Set clock timer interrupt enable
```

Note that the clock timer interrupt has no IDMA request flag and can function only as an interrupt to the CPU.

(7) Return processing

Restore PSR and enable interrupts.

```
/* Restore PSR */
restore_psr();
```

## Interrupt processing [drv_ct.c]

```
/*******************************************************************************
 * int_ct
 *   Type :     void
 *   Ret val :  none
 *   Argument : void
 *   Function : Clock timer interrupt function.
 *******************************************************************************/
void int_ct(void)
{
      extern volatile int ctint_flg;

      INT_BEGIN;                                                           (1)
      ctint_flg = TRUE; // Clock timer interrupt flag on                  (2)
      *(volatile unsigned char *)INT_FADE_FCTM_FP4_ADDR = INT_FCTM;        (3)
               // Reset clock timer interrupt factor flag
      INT_END;                                                             (1)
}
```

(1) Saving and restoring registers

Use INT_BEGIN and INT_END (defined in common.h) to save and restore registers at the beginning and end of the interrupt handling routine.

```
#define INT_BEGIN    asm("pushn %r15")
#define INT_END      asm("popn  %r15\n  reti")
```

(2) Setting an interrupt-generated confirmation flag

Set a flag notifying the host routine that an interrupt has been generated.

```
ctint_flg = TRUE;                   // Clock timer interrupt flag on
```

(3) Resetting the cause of the interrupt flag

Clear the interrupt factor flag.

```
*(volatile unsigned char *)INT_FADE_FCTM_FP4_ADDR = INT_FCTM;
   // Reset clock timer interrupt factor flag
```

Application section [demo_ct.c]

```
          |
      /* Initialize clock timer */
      write_str("*** Initialize clock timer and start ***\n");
      write_str("     Today date and time (1999.01.01 21:05)\n");
      write_str("     Set minute alarm interrupt enable (6 minutes)\n");
      init_ct();                                                      (1)

      /* Run clock timer */
      write_str("*** Run clock timer ***\n");
      *(volatile unsigned char *)CT_TCRUN_ADDR |= 0x01;               (2)

      /* Initialize clock timer interrupt flag */
      ctint_flg = FALSE;

      write_str("*** Wait 1 minute ***\n");
      write_str("\n");

      while (1) {                                                     (3)
              if (ctint_flg == TRUE) {
                      break;
              }
      }

      /* Stop clock timer */
      write_str("*** Stop clock timer ***\n");
      *(volatile unsigned char *)CT_TCRUN_ADDR &= 0xfe;               (4)
          |
```

(1) Initial settings
Call the above-mentioned init_ct() and initialize the clock timer and interrupt settings.

(2) Starting the clock timer
Start the clock timer and clear the interrupt-generated confirmation flag.

```
*(volatile unsigned char *)CT_TCRUN_ADDR |= 0x01;

/* Initialize clock timer interrupt flag */
ctint_flg = FALSE;
```

(3) Wait for alarm interrupt
When the alarm time arrives, the above-mentioned interrupt handling routine int_ct() is called. When processing is complete, the flag ctint_flg is set. Loop the program until this flag is set to 1. An alarm interrupt occurs one minute after the clock timer starts.

```
while (1) {
   if (ctint_flg == TRUE) {
            break;
   }
}
```

(4) Stopping the clock timer
After the interrupt occurs, stop the clock timer.

```
*(volatile unsigned char *)CT_TCRUN_ADDR &= 0xfe;
```

## 3.9 SLEEP

This section explains the processing preceding SLEEP mode entry, and how to exit SLEEP using the alarm function. The explanation uses an example file found in gnu33\sample\drv33209\osc of S5U1C33000C ver.3 or later.

Main routine [demo_osc.c]

```
int main(void)
{
      unsigned char     pwr;     /* Power control register data */
      unsigned char     clk;     /* Clock control register data */

      write_str("*** OSC demonstration ***\n");
      write_str("\n");

      /* OSC3 high-speed mode */
      write_str("*** OSC3 high-speed mode ***\n");
      write_str("     System clock select 1/1, Prescaler output ON, CPU clock OSC3,
               OSC3 ON, OSC1 ON\n");
      write_str("     HALT clock option OFF, OSC3-stabilize waiting function ON\n");
      write_str("     OSC1 external output control OFF\n");
      pwr = OSC_CLKDT_11 | OSC_PSCON_ON | OSC_CLKCHG_OSC3 | OSC_SOSC3_ON |       (1)
               OSC_SOSC1_ON;
      clk = OSC_HALT2OP_OFF | OSC_8T1ON_ON | OSC_PF1ON_OFF;
      set_OSC(pwr, clk);

      /* If you use sleep mode, you set OSC3-stabilize waiting function on
               and run 8-bit timer 1 */
      /* Initialize 8-bit timer */
      write_str("*** Initialize 8-bit timer ***\n");
      write_str("     8-bit timer 1 ... CLK/4096\n");
      write_str("     8-bit timer 1 reload data
               ... 0x62 (10ms on OSC3 clock 40MHz)\n");
      init_8timer1();                                                           (2)

      /* Initialize clock timer */
      write_str("*** Initialize clock timer and start ***\n");
      write_str("     Today data and time (1999.01.01 21:05)\n");
      write_str("     Set minute alarm interrupt enable (6 minutes)\n");
      init_ct();                                                                (3)

      /* Run clock timer */
      write_str("*** Run clock timer ***\n");
      *(volatile unsigned char *)CT_TCRUN_ADDR |= 0x01;                         (4)

      write_str("*** Wait 1 minute ***\n");

      write_str("*** Sleep mode ***\n");
      write_str("\n");

      /*Run 8-bit timer 1 */
      run_8timer(T8P_PTRUN1_ADDR);                                              (5)

      /* Sleep */
      asm("slp");                                                               (6)

      /* Stop 8-bit timer 1 */
      write_str("*** Return to OSC3 high-speed mode from sleep mode ***\n");

      write_str("\n");
      write_str("*** OSC demonstration finish ***\n");
}
```

(1) Setting the oscillator circuit

Call set_osc() and set the following.

```
pwr = OSC_CLKDT_11 | OSC_PSCON_ON | OSC_CLKCHG_OSC3 | OSC_SOSC3_ON | OSC_SOSC1_ON;
clk = OSC_HALT2OP_OFF | OSC_8T1ON_ON | OSC_PF1ON_OFF;
set_OSC(pwr, clk);
```

| | |
|---|---|
| OSC_CLKDT_11 | System clock division ratio = 1/8 |
| OSC_PSCON_ON | Prescaler ON |
| OSC_CLKCHG_OSC3 | CPU operating clock = OSC3 |
| OSC_SOSC3_ON | High-speed (OSC3) oscillation ON |
| OSC_SOSC1_ON | Low-speed (OSC1) oscillation ON |
| OSC_HALT2OP_OFF | HALT2 mode OFF |
| OSC_8T1ON_ON | High-speed (OSC3) oscillation wait after SLEEP exit function ON |
| OSC_PF1ON_OFF | OSC1 clock external output OFF * |

* OSC1 clock external output is disabled when the internally-wired clock from OSC1 block to FOSC pin is disabled, reducing current consumption during SLEEP mode to a minimum. If necessary, the OSC1 clock may be output even during SLEEP mode. In this case, the P14/DCLK/FOSC1 pin must be set for OSC1 clock output (FOSC1).

(2) Initializing 8-bit timer 1

Because the high-speed (OSC3) oscillation wait function is used after exiting SLEEP, set the wait time in 8-bit timer 1. This processing is performed in init_8timer1().

(3) Setting the clock timer

Call init_ct() and set the clock timer to generate an alarm interrupt one minute after starting. For more information on processing by init_ct(), see Section 3.8, "Clock Settings".

(4) Starting the clock timer

Start the clock timer.

```
*(volatile unsigned char *)CT_TCRUN_ADDR |= 0x01;
```

(5) Starting 8-bit timer 1

Call run_8timer() and start 8-bit timer 1.

(drv_8timer.c)

```
void run_8timer(unsigned long reg)
{
   *(volatile unsigned char *)reg |= 0x01;
}
```

(6) SLEEP

Execute the SLP instruction to enter SLEEP mode. The high-speed (OSC3) oscillator circuit stops.

```
asm("slp");
```

(7) Exiting SLEEP

Even during SLEEP mode, the clock timer is paced by the low-speed (OSC1) oscillation circuit to allow the processor to be roused from SLEEP mode when the set alarm interrupt occurs. The high-speed (OSC3) oscillation circuit begins operating upon exiting SLEEP, but program execution can restart only after an oscillation stabilization wait interval (10 ms) elapses. This is set in 8-bit timer 1.

With this program running on a S5U1C33208D3, current consumption was measured at 65 mA for normal operations and 35 mA during SLEEP — a savings of about 30 mA.

### Setting the oscillation circuit [drv_osc.c]

```
void set_osc(unsigned char pwr, unsigned char clk)
{
      /* Before power control register write access,
               set power control register protect flag write enable */
      *(volatile unsigned char *)OSC_CLGP_ADDR = OSC_CLGP_ENA;             (1)

      /* Set power control register */
      *(volatile unsigned char *)OSC_SOSC_ADDR = pwr;

      /* Before clock control register write access,
               set power control register protect flag write enable */
      *(volatile unsigned char *)OSC_CLGP_ADDR = OSC_CLGP_ENA;             (2)

      /* Set clock control register */
      *(volatile unsigned char *)OSC_PF1ON_ADDR = clk;
}
```

(1) Setting the power control register

Remove write protection for the power control register (0x40180). Write the set value passed from main() into this register.

```
/* Before power control register write access,
   set power control register protect flag write enable */
*(volatile unsigned char *)OSC_CLGP_ADDR = OSC_CLGP_ENA;

/* Set power control register */
*(volatile unsigned char *)OSC_SOSC_ADDR = pwr;
```

(2) Setting the clock option register

Remove write protection for the clock option register (0x40190). Write the set value passed from main() into this register.

```
/* Before clock control register write access,
   set power control register protect flag write enable */
*(volatile unsigned char *)OSC_CLGP_ADDR = OSC_CLGP_ENA;

/* Set clock control register */
*(volatile unsigned char *)OSC_PF1ON_ADDR = clk;
```

### Setting 8-bit timer 1 [drv_8timer.c]

```
void init_8timer1(void)
{
      /* Save PSR and disable all interrupt */
      save_psr();                                                          (1)

      /* Set 8bit timer1 prescaler */
      *(volatile unsigned char *)PRESC_P8TS0_P8TS1_ADDR                    (2)
               |= (PRESC_PTONH_ON | PRESC_CLKDIVH_SEL7);
               // Set 8bit timer1 prescaler (CLK/4096)

      /* Set 8bit timer1 reload data */
      *(volatile unsigned char *)T8P_RLD1_ADDR = 0x62;                     (3)
               // Set reload data (0x62 ... 10ms on OSC3 clock 40MHz)

      /* Set 8bit timer1 clock output off, preset and timer stop */
      *(volatile unsigned char *)T8P_PTRUN1_ADDR
               = T8P_PTOUT_OFF | T8P_PSET_ON | T8P_PTRUN_STOP;

      /* Set 8bit timer1 interrupt CPU request on interrupt controller */
      *(volatile unsigned char *)INT_R16T5_R8TU_RS0_ADDR = INT_RIDMA_DIS;  (4)
               // IDMA request disable and CPU request enable

      /* Set 8bit timer1 interrupt priority level 3 on interrupt controller */
      *(volatile unsigned char *)INT_P8TM_PSIO0_ADDR = INT_PRIL_LVL3;

      /* Reset 8bit timer1 interrupt factor flag on interrupt controller */
      *(volatile unsigned char *)INT_F8TU_ADDR = INT_F8TU1;
               // Reset 8bit timer1 underflow interrupt factor flag
```

```
        /* Set 8bit timer1 interrupt disable on interrupt controller */
        *(volatile unsigned char *)INT_E8TU_ADDR &=~INT_E8TU1;
                 // Set 8bit timer1 underflow interrupt disable

        /* Restore PSR */
        restore_psr();                                                              (5)
}
```

(1) Disabling interrupts

Save PSR and mask interrupts with IE.

```
/* Save PSR and disable all interrupt */
save_psr();
```

(2) Setting the prescaler

Set the prescaler division ratio to 1 / 4096.

```
/* Set 8bit timer1 prescaler */
*(volatile unsigned char *)PRESC_P8TS0_P8TS1_ADDR
   |= (PRESC_PTONH_ON | PRESC_CLKDIVH_SEL7);
   // Set 8bit timer1 prescaler (CLK/4096)
```

(3) Setting 8-bit timer

Set 0x62 as the reload data. This value generates an OSC3 oscillation stabilization wait time of about 10 ms when the CPU operates at 40 MHz.

25 µs (=1 / 40 MHz) × 4096 × (0x62 + 1) = approx. 10 ms

```
/* Set 8bit timer1 reload data */
*(volatile unsigned char *)T8P_RLD1_ADDR = 0x62;
   // Set reload data (0x62 ... 10ms on OSC3 clock 40MHz)
```

Preset the above reload data in the counter. Do not start the timer yet.

```
/* Set 8bit timer1 clock output off, preset and timer stop */
*(volatile unsigned char *)T8P_PTRUN1_ADDR
   = T8P_PTOUT_OFF | T8P_PSET_ON | T8P_PTRUN_STOP;
```

(4) Setting the interrupt controller

Disable IDMA start with an 8-bit timer 1 interrupt.

```
/* Set 8bit timer1 interrupt CPU request on interrupt controller */
*(volatile unsigned char *)INT_R16T5_R8TU_RS0_ADDR = INT_RIDMA_DIS;
   // IDMA request disable and CPU request enable
```

Set the 8-bit timer interrupt priority level to 3.

```
/* Set 8bit timer1 interrupt priority level 3 on interrupt controller */
*(volatile unsigned char *)INT_P8TM_PSIO0_ADDR = INT_PRIL_LVL3;
```

Reset the 8-bit timer 1 interrupt factor flag.

```
/* Reset 8bit timer1 interrupt factor flag on interrupt controller */
*(volatile unsigned char *)INT_F8TU_ADDR = INT_F8TU1;
   // Reset 8bit timer1 underflow interrupt factor flag
```

Leave the 8-bit timer 1 interrupt disabled.

```
/* Set 8bit timer1 interrupt disable on interrupt controller */
*(volatile unsigned char *)INT_E8TU_ADDR &=~INT_E8TU1;
   // Set 8bit timer1 underflow interrupt disable
```

(5) Return processing

Restore PSR and enable interrupts.

```
/* Restore PSR */
restore_psr();
```

## *3.10 SDRAM Controller*

At present, the SDRAM controller is incorporated into the following types of devices:
S1C33L03, S1C33205

The following shows examples of the initialization program for using SDRAM.

### Example of initialization routine for 4M words × 16 bits × 4 banks (32MB) of SDRAM

```
;*****************************************************************************
;
;   Copyright (C) SEIKO EPSON CORP. 2002
;   All rights Reserved
;
;   File name : SDRAM_led.s
;
;    Revision  :
;       2002/10/01  M.Toki start
;       2003/04/18  CH.Yoon Port to GNU
;*****************************************************************************
    .text
    .long   START

START:

INIT_SDRAM_32MB:
;;;------------------------- SDRAM access configuration -----------------------------------
;;;************************************************************
;;;***************** C33 macro setting part ***********************
;;;************************************************************
;;; set CEFUNC to use #CE13/14 (upper area)
    xld.w %r0,0x48131
    bset [%r0],0x1
;;; set area 6,13,14 to internal access
    xld.w %r0,0x48132
    xld.w %r1,0x2200
    ld.h [%r0],%r1
;;; area 6 -> output disable 0.5, wait 2
    xld.w %r0,0x4812A
    xld.w %r1,0x0237
    ld.h [%r0],%r1
;;; available #WAIT
    xld.w %r0,0x04812E
    bset [%r0],0x0
;;; area 13,14 -> 8bit device, output disable 2.5, wait 0
    xld.w %r0,0x048122
    xld.w %r1,0x30
    ld.h [%r0],%r1

;;;************************************************************
;;;************** SDRAM Controller REG setting part ***************
;;;************************************************************
;;;------------------------------------------------
;;;  area13 0x2000000 - 0x2FFFFFF(16MB)
;;;  area14 0x3000000 - 0x3FFFFFF(16MB)
;;;------------------------------------------------
;///////////////////////////////////////////
;;; SDRAM area configuration register
    xld.w %r0,0x39FFC0 ;
    xld.w %r1,0xc8     ; set area13&14 to SDRAM area, #SDCE0(#CE13) available
    ld.b [%r0],%r1     ; (32MB area available)
;///////////////////////////////////////////
;;; SDRAM control register
;;; xld.w %r0,0x39FFC1 ;
;;; xld.w %r1,0xff     ; SDRAM self-refresh -> disable, initial sequence ->PRE REF MRS
;;; ld.b [%r0],%r1     ; Little endian
;///////////////////////////////////////////
;;; SDRAM address configuration register
    xld.w %r0,0x39FFC2 ;
    xld.w %r1,0x2a     ; col 512 / row 8K / bank 4 -> 256Mb[32MB] available
    ld.b [%r0],%r1     ;
;///////////////////////////////////////////
```

```
;;; SDRAM mode set-up register
    xld.w %r0,0x39FFC3 ;
    xld.w %r1,0x40     ; 2 CAS Latency ,burst length = 1
    ld.b [%r0],%r1     ;
;///////////////////////////////////////////
;;; SDRAM timing set-up register 1
    xld.w %r0,0x39FFC4 ;
    xld.w %r1,0x4A     ; Tras=2,Trp=1,Trc=2          ... Recommended setting to operate with 25 MHz clock in x1 speed mode
    ld.b [%r0],%r1     ;
;///////////////////////////////////////////
;;; SDRAM timing set-up register 2
    xld.w %r0,0x39FFC5 ;
    xld.w %r1,0x48     ; Trcd=1,Trsc=2,Trrd=1
    ld.b [%r0],%r1     ;
;///////////////////////////////////////////
;;; SDRAM auto refresh count low-order register
;;; xld.w %r0,0x39FFC6 ;
;;; xld.w %r1,0xff     ;
;;; ld.b [%r0],%r1     ;
;///////////////////////////////////////////
;;; SDRAM auto refresh count high-order register
    xld.w %r0,0x39FFC7 ;
    xld.w %r1,0x00     ;
    ld.b [%r0],%r1     ;
;///////////////////////////////////////////
;;; SDRAM self refresh count register
;;; xld.w %r0,0x39FFC8 ;
;;; xld.w %r1,0x0f     ;
;;; ld.b [%r0],%r1     ;
;///////////////////////////////////////////
;;; SDRAM advanced control register
    xld.w %r0,0x39FFC9 ;
    xld.w %r1,0x20     ; data width -> 16bit, bank interleave -> on
    ld.b [%r0],%r1     ;

;;;****************************************************************
;;;***************** SDRAM controller power up ********************
;;;****************************************************************
    xld.w %r0,0x39FFC1 ; SDRAM control register
    xld.w %r1,0x39FFCA ; SDRAM status register
    xld.w %r2,0x0
    xld.w %r3,0x10

;;; enable SDRAM signal
    bset [%r0],0x7     ; set SDRENA[D7/0x39FFC1]

SDRAM_SIGNAL_EN:
    add %r2,0x1        ; SDRAM signal enable waiting loop
    cmp %r2,%r3
    xjrne SDRAM_SIGNAL_EN

;;; SDRAM power up
    bset [%r0],0x6     ; set SDRINI[D6/0x39FFC1]

POWER_UP:
    btst [%r1],0x7     ; SDRAM power-up waiting loop
    xjrne POWER_UP

;;;-------------------- end of SDRAM access configuration ------------------------

;;;********************************************************************************
;;;************************* Program translating **********************************
;;;********************************************************************************
; Transfer data from 0x00c00400 to 0x00001000
    xld.w  %r9,0x2000000    ; r9 = 0x2000000 : destination address
    xld.w  %r8,0x0001000    ; r8 = 0x0001000 : source      address
    xld.w  %r7,0x0000000    ; reference data ( "nop,nop" of end -> 0x00000000)

DATA_TRANSFER:
    ld.w   %r6,[%r8]+       ; data transfer FROM iRAM to SDRAM
    ld.w  [%r9]+,%r6        ;
    cmp    %r6,%r7             ; compare the instruction code with 0x00000000(nop,nop)
    xjrne  DATA_TRANSFER       ; if instruction code = "nop,nop", exit this loop
;;;*******************************************************************************
    xld.w   %r9,0x2000000      ; r9 = 0x2000000 :
    jp      %r9                ; jp to "LED on/off loop" of SDRAM
;;;------------------------- end of program translating ---------------------------


;******************************************************************************
;
;    Target source program    - LED ON/OFF - program
;
;******************************************************************************
main:
    xld.w   %r10,0x402d2
    xld.w   %r12,0x08
    ld.b    [%r10],%r12        ; P7 I/O port INPUT mode
    xld.w   %r10,0x402d1
    xld.w   %r12,0x00
    ld.b    [%r10],%r12        ; LED ON

    xld.w   %r13,0x100000      ; wait counter
loop:
    sub     r13,0x1
    xjrgt   loop

    xld.w   %r12,0x08
    ld.b    [%r10],%r12        ; LED OFF

    xld.w   %r13,0x100000      ; wait counter
loop2:                         ; loop for wait
    sub     %r13,0x1
    xjrgt   loop2

    xld.w   %r12,0x00
    ld.b    [%r10],%r12        ; LED ON

    xld.w   %r13,0x100000      ; wait counter
loop3:                         ; loop for wait
    sub     %r13,0x1
    xjrgt   loop3

    xjp     loop               ; jp to "loop"
;******************************************************************************
    nop                        ; program end indicator
    nop                        ;
    nop                        ;
    nop                        ;
```

## Example of initialization routine for 2M words × 16 bits × 4 banks (16MB) of SDRAM

```
INIT_SDRAM_16MB:
;;;----------------------- SDRAM access configuration -----------------------------------
;;;*****************************************************************
;;;**************** C33 macro setting part ***********************
;;;*****************************************************************

;;; set CEFUNC to use #CE13/14 (upper area)
        xld.w   %r0,0x48131
        bset    [%r0],0x1

;;; set area 6,13,14 to internal access
        xld.w   %r0,0x48132
        xld.w   %r1,0x2200
        ld.h    [%r0],%r1

;;; area 6 -> output disable 0.5, wait 2
        xld.w   %r0,0x4812A
        xld.w   %r1,0x0237
        ld.h    [%r0],%r1

;;; available #WAIT
        xld.w   %r0,0x04812E
        bset    [%r0],0x0

;;; area 13,14 -> 16bit device, output disable 2.5, wait 0
        xld.w   %r0,0x048122
        xld.w   %r1,0x30
        ld.h    [%r0],%r1

;;;*****************************************************************
;;;************** SDRAM Controller REG setting part ***************
;;;*****************************************************************
;;;-----------------------------------------------
;;;area13       0x2000000 - 0x2FFFFFF(16MB)
;;;area14       0x3000000 - 0x3FFFFFF(16MB)
;;;-----------------------------------------------
;///////////////////////////////////////////
;;; SDRAM area configuration register
        xld.w   %r0,0x39FFC0     ;
        xld.w   %r1,0x88         ; set area13 to SDRAM area, #SDCE0(#CE13) available
        ld.b    [%r0],%r1        ; (16MB area available)
;///////////////////////////////////////////
;;; SDRAM control register
;;;     xld.w   %r0,0x39FFC1     ;
;;;     xld.w   %r1,0xff         ; SDRAM self-refresh -> disable, initial sequence ->PRE REF MRS
;;;     ld.b    [%r0],%r1        ; Little endian
;///////////////////////////////////////////
;;; SDRAM address configuration register
        xld.w   %r0,0x39FFC2     ;
        xld.w   %r1,0x26         ; col 512 / row 4K / bank 4 -> 128Mb[16MB] available
        ld.b    [%r0],%r1        ;
;///////////////////////////////////////////
;;; SDRAM mode set-up register
        xld.w   %r0,0x39FFC3     ;
        xld.w   %r1,0x40         ; 2 CAS Latency ,burst length = 1
        ld.b    [%r0],%r1        ;
;///////////////////////////////////////////
;;; SDRAM timing set-up register 1
        xld.w   %r0,0x39FFC4     ;
        xld.w   %r1,0x4A         ; Tras=2,Trp=1,Trc=2 ... Recommended setting to operate with 25 MHz clock in x1 speed mode
        ld.b    [%r0],%r1        ;
;///////////////////////////////////////////
;;; SDRAM timing set-up register 2
        xld.w   %r0,0x39FFC5     ;
        xld.w   %r1,0x48         ; Trcd=1,Trsc=2,Trrd=1
        ld.b    [%r0],%r1        ;
;///////////////////////////////////////////
;;; SDRAM auto refresh count low-order  register
;;;     xld.w   %r0,0x39FFC6     ;
;;;     xld.w   %r1,0xff         ;
;;;     ld.b    [%r0],%r1        ;
;///////////////////////////////////////////
;;; SDRAM auto refresh count high-order register
        xld.w   %r0,0x39FFC7     ;
        xld.w   %r1,0x00         ;
        ld.b    [%r0],%r1        ;
;///////////////////////////////////////////
;;; SDRAM self refresh count register
;;;     xld.w   %r0,0x39FFC8     ;
;;;     xld.w   %r1,0x0f         ;
;;;     ld.b    [%r0],%r1        ;
;///////////////////////////////////////////
;;; SDRAM advanced control register
        xld.w   %r0,0x39FFC9     ;
        xld.w   %r1,0x20         ; data width -> 16bit, bank interleave -> on
        ld.b    [%r0],%r1        ;

;;;*****************************************************************
;;;***************** SDRAM controller power up ********************
;;;*****************************************************************
        xld.w   %r0,0x39FFC1     ; SDRAM control register
        xld.w   %r1,0x39FFCA     ; SDRAM status  register
        xld.w   %r2,0x0
        xld.w   %r3,0x10

;;; enable SDRAM signal
        bset    [%r0],0x7        ; set SDRENA[D7/0x39FFC1]
SDRAM_SIGNAL_EN:
        add     %r2,0x1          ; SDRAM signal enable waiting loop
        cmp     %r2,%r3
        jrne    SDRAM_SIGNAL_EN

;;; SDRAM power up
        bset    [%r0],0x6        ; set SDRINI[D6/0x39FFC1]
POWER_UP:
        btst    [%r1],0x7        ; SDRAM power-up waiting loop
        jrne    POWER_UP

;;;------------------------ end of SDRAM access configuration ---------------------------
ret
```

The SDRAM can be accessed after executing the above program.

## ● Precautions

(1) Make sure that two wait cycles are inserted when accessing area 6, where the SDRAM controller is allocated. With any other number of specified wait cycles, data may not be written normally to the SDRAM control registers.

(2) Set the area used for an SDRAM for internal access (A8IO (DA / 0x48132) = "1" or A14IO (DD / 0x48132) = "1").

(3) Before entering HALT2 or SLEEP mode, be sure to place the SDRAM in self-refresh mode, because the SDRAM cannot be auto-refreshed while in those modes. In that case, confirm that SDRSRM (D6 / 0x39FFCA) = "0" (i.e., that the SDRAM is in self-refresh mode) before executing the HALT or SLP instruction.
If an access to the SDRAM occurs while being self-refreshed, the SDRAM is taken out of self-refresh mode; thus always make sure the SDRAM check and the HALT / SLP instruction execution are performed from devices other than the SDRAM.

(4) Do not access addresses 0x39FFCB to 0x39FFCD, as the user program will not be able to control the CPU.

(5) If the program accesses an area out of the address range set using the address setting register (0x39FFC2), an unintended area is accessed and the stored data may be overwritten. Therefore, do not access an area out of the set range.

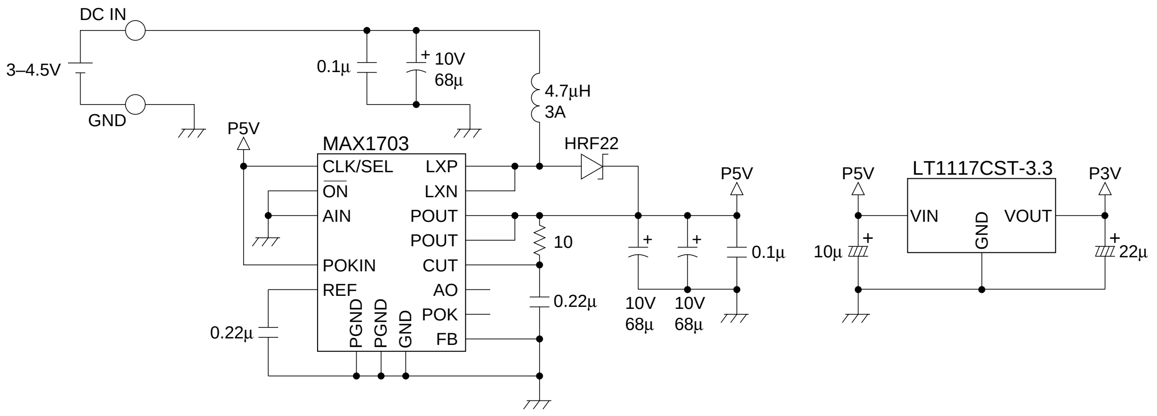
# 4 THE BASIC S1C33 CHIP BOARD CIRCUIT

This chapter explains the basic circuit design of the S1C33209.

## 4.1 Power Supply

Here, we'll explain the power supply based on a DC-DC converter, using the S5U1C33209D1 circuit as an example.

### ● DC-DC converter



This power supply circuit steps up the 3 to 4.5 V input voltage with a switching regulator to generate a 5 V power supply for the external I/O and memory block, as well as for the analog block. This 5 V power supply is stepped down with a linear regulator to generate a 3.3 V power supply for the CPU core. Because the S1C33209's CPU core operates at 3.3 V, two such power supplies are required if the external interface operates with 5 V.

Select capacitors carefully when using a switching-mode power supply as in the S5U1C33209D1. A 68 µF decoupling capacitor is positioned between the battery and coil. Due to the large rush current flowing here, large ESR (equivalent internal resistance) results in power dissipation and abbreviated battery life. For example, battery life can vary as much as 1.5 times between the OS capacitor used in the S5U1C33209D1 and an ordinary electrolytic capacitor. The capacitors located after the coil do not significantly affect battery life. If noise is a consideration, choose capacitors with low ESRs. The capacitor is used to maintain as consistent a post-coil voltage as possible, and its change voltage (ripple) increases proportionately with ESR. For S5U1C33209D1, using an electrolytic capacitor produces sufficient noise in audio output to render audio quality unusable. Use the OS capacitor to reduce relative noise levels to about 1/10, levels at which noise is generally not a problem.
In digital circuits, differences between capacitors produces only slight differences in noise margins. But such differences are significant in analog circuits. In analog circuits, for increased safety, avoid using a switching-mode power supply if possible.

## ● Decoupling capacitors

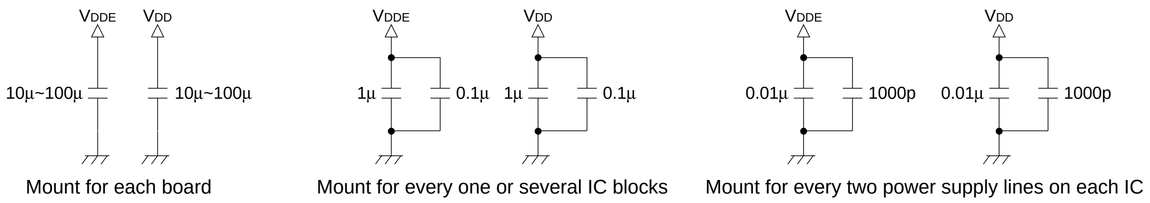
Use the following four methods for noise abatement between power supply and ground lines.

1) Use a circuit board comprised of four or more layers, and provide full-surface GND and full-surface VDD layers.

2) Attach a 100 µF electrolytic capacitor per circuit board. For small circuit boards, attach a 10 µF tantalum capacitor.

3) Attach a 1 µF + 0.1 µF laminated ceramic capacitor to the CPU and to the memory block.

4) Attach a 0.01 µF + 1000 pF chip-form laminated ceramic capacitor to each IC, positioning it as close to the power supply pins as possible.


VDDE
VDD
10µ~100µ
10µ~100µ
Mount for each board
VDDE
VDD
1µ
0.1µ
1µ
0.1µ
Mount for every one or several IC blocks
VDDE
VDD
0.01µ
1000p
0.01µ
1000p
Mount for every two power supply lines on each IC

The capacitors in 2) to 4) above cover the following frequency ranges:

| | |
|---|---|
| 100 µF: | Absorbs AC components in frequencies below several 100 kHz. |
| 1 µF: | Absorbs AC components in frequencies from several 100 kHz to several MHz. |
| 0.1 µF: | Absorbs AC components in frequencies from several MHz to about 20 MHz. |
| 0.01 µF: | Absorbs AC components in frequencies from 10 MHz to about 50 MHz. |
| 1000 pF: | Absorbs AC components in frequencies from several 10 MHz to about 100 MHz. |

Omission of any of these capacitors results in difficulty absorbing noise in that frequency range. For example, in the common arrangement of 0.1 µF per IC, noise for frequencies around 10 MHz is absorbed relatively efficiently, but noise cannot be absorbed in frequencies above 10 MHz. S1C33209 circuit boards operating at frequencies above 40 MHz are subject to noise at frequencies approaching 100 MHz or even higher. This noise can only be absorbed with a capacitor of about 1000 pF. Additionally, since inductance resulting from extended wiring lowers the upper absorption limit of the 1000 pF capacitor, be sure to mount it at the closest position possible to the pin, second only to the PLL capacitor described further below. Failure to do so will lower the actual upper limit of the absorption range below 100 MHz.

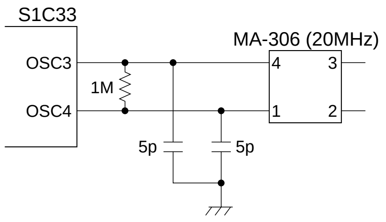
When using double-sided circuit boards, reinforce GND as much as possible to ensure equivalent GND potentials at each location. To prevent voltage fluctuations, use a decoupling capacitor to reinforce power supply lines on each block.

## 4.2 Oscillation Circuit

The following section discusses oscillation circuits, referring to the S5U1C33209D1 and S5U1C33208E1 as examples.

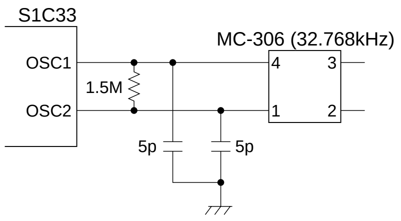
### ● 20 MHz resonator

This example applies to the S5U1C33209D1 when the high-speed (OSC3) oscillation circuit is comprised of a crystal resonator, a resistor, and capacitors.
For more information on resistor and capacitor values, see the documentation supplied with your crystal resonator.



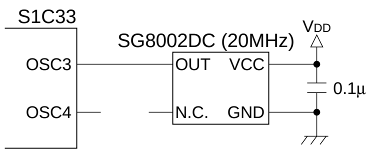
### ● 32 kHz resonator

This example applies to the S5U1C33209D1 when the 32 kHz, low-speed (OSC1) oscillation circuit is comprised of a crystal resonator, a resistor, and capacitors.
For more information on resistor and capacitor values, see the documentation supplied with your crystal resonator.



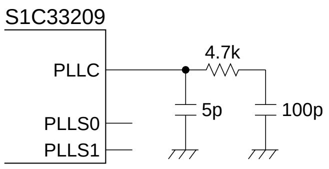
### ● 20 MHz oscillator

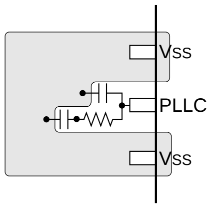
This example applies to the S5U1C33208E1 when the oscillator's output clock is fed to the OSC3 pin. Make sure the voltage level of the input clock is the same as that of the operating clock (VDD) of the CPU core (e.g. 3.3 V). The same applies when an external clock is fed to the OSC1 pin.



### ● PLL, core clock, and bus clock

Related pins are PLLC and PLLS[1:0].





The PLLC must have the shortest wiring pattern of all other pins. To prevent crosstalk from other signal lines, it should also be enclosed with the largest GND pattern possible. Poor noise characteristics on the PLLC line will result in increased jitter, or adversely affect the clock's duty ratio.

Select a high-speed operating clock for the S1C33209 from the following three options by processing the PLLS0 and PLLS1 pins.

| | |
|---|---|
| PLLS1 = 0, PLLS0 = 0: | The OSC3 clock is used directly as is. (Because PLL is unused, current consumption slightly lowers.) |
| PLLS1 = 1, PLLS0 = 1: | A 2-times OSC3 clock is selected. (10–20 MHz clock input for OSC3) |
| PLLS1 = 0, PLLS0 = 1: | A 4-times OSC3 clock is selected. |

This clock is fed into the CPU core in the chip. The #X2SPD pin is used to determine the bus operating clock.
#X2SPD = 1: The bus operates with the same clock as the core.
#X2SPD = 0: The bus operates at half the frequency of the core clock.

Combination example:

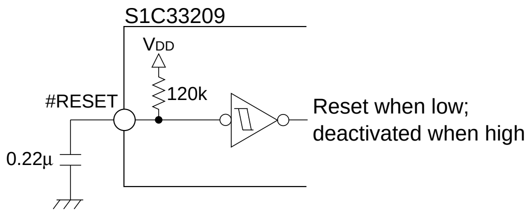
| OSC3 | PLLS1 | PLLS0 | #X2SPD | Core clock | Bus clock |
|---|---|---|---|---|---|
| 20 MHz | 0 | 0 | 1 | 20 MHz | 20 MHz |
| 20 MHz | 1 | 1 | 0 | 40 MHz | 20 MHz |
| 15 MHz | 0 | 1 | 0 | 60 MHz | 30 MHz |

## *4.3 Reset Circuit*

This section describes a simple RC reset circuit as well as a more sophisticated circuit with a reset IC capable of power supply voltage detection.

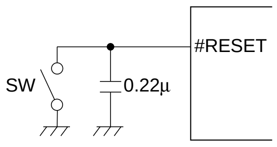
### ● Reset by an RC network

The S1C33209's reset input pin consists of a Schmitt trigger circuit with a pull-up resistor of about 120 kΩ. A simple reset circuit can be configured just by connecting an off-chip capacitor of about 0.22 µF. The 0.22 µF capacitor may be laminated ceramic or an electrolytic capacitor.



This comprises an RC time constant of about 15 to 20 ms from power-on to VDD/2. This circuit has the simplest structure. But because the reset input is only 120 kΩ pull-up and because reset is recognized at a rising edge, it is also susceptible to noise. Make sure the capacitor is connected to the reset pin at the shortest distance possible, within design constraints.

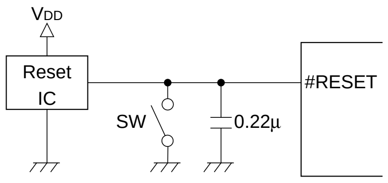
When using the S5U1C33000H for debugging, we recommend attaching a reset switch to your system. When you encounter difficulty connecting the S5U1C33000H and target, this lets you hold down the reset switch while turning on the S5U1C33000H, then release the reset switch, ensuring that the S5U1C33000H and target are connected. For development purposes, only a switch needs to be inserted between the capacitor and VSS.



### ● Reset circuit using a reset IC

The reset IC used in the S5U1C33T01D1 (PST572 made by Mitsumi) is a three-terminal type connecting VDD and GND. When VDD is below the rated level, it drives VOUT low; when VDD is above the rated level, it puts VOUT in a high-impedance state.

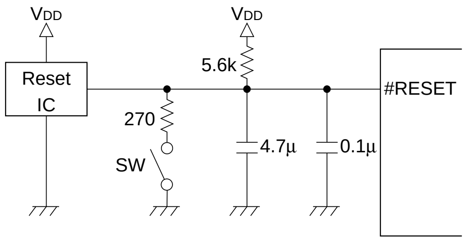
Adding this IC to the above reset circuit results in the following (excerpt from the S5U1C33T01D1 circuit):



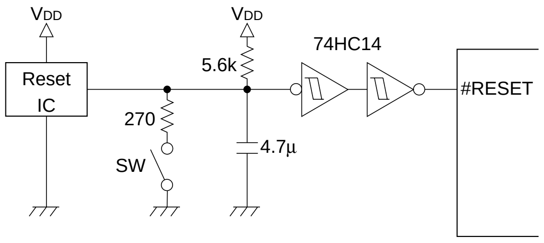
## ● Protecting reset against noise

If the reset circuit described above is routed around apart from the IC, it becomes susceptible to crosstalk. In such cases, take the following protective measures.

1) Reduce the pull-up resistance.
2) Attach a decoupling capacitor on the pin side.
3) Enclose with a GND pattern to protect against crosstalk.
4) Drive reset high / low with low impedance using logic.



In this example of noise protection, the reset line is pulled high with external 5.6 kΩ. The switch also has 270 Ω connected in series, thereby limiting the current flowing into it. A 0.1 µF decoupling capacitor is inserted on the reset pin side to reduce high-frequency noise, which can easily ride on the line due to crosstalk.



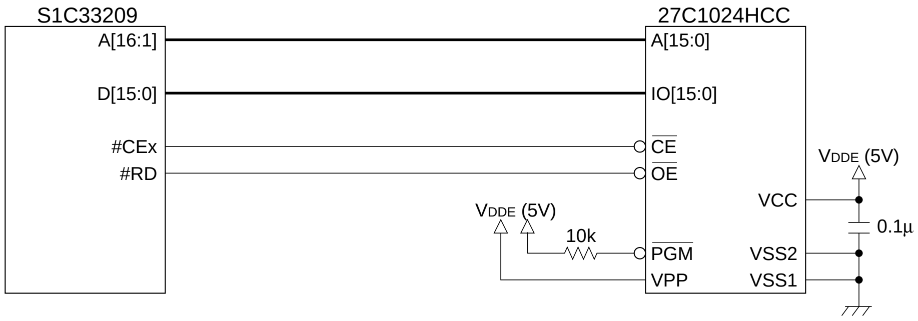
In this example of noise protection, a 74HC14 (Schmitt type inverter) is inserted to drive reset with logic. This renders the reset circuit significantly resistant to noise.

In addition to reset, all edge-activated ports such as NMI and input interrupts require caution regarding erratic device behavior induced by noise. Make the wiring as short as possible, particularly for inputs whose high / low levels are regulated using pull-up / pull-down resistors. Implement protective measures, such as the ones described above. Use of pull-up / pull-down resistors of about 100 kΩ makes it crucial that the line and pin be connected by the shortest distance. Even for pull-up / pull-down resistors of 10 kΩ or less, avoid extending wiring unnecessarily. Check with an oscilloscope to confirm absence from crosstalk.

## 4.4 Connecting ROM

Using the S5U1C33209D1 and S5U1C33000H as examples, a ROM connection diagram is shown below.

### ● Connecting x16 ROM



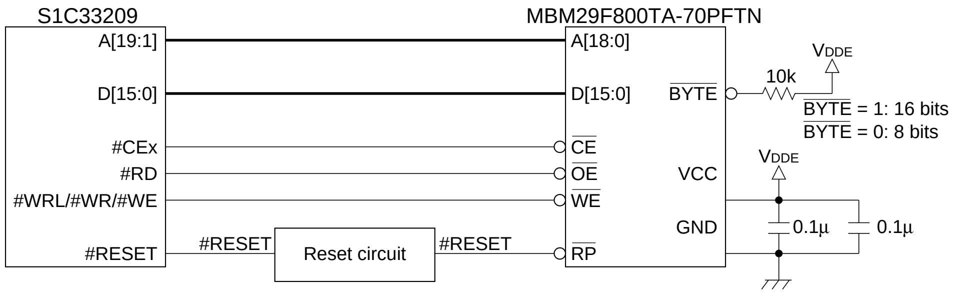
The S5U1C33209D1 has a 1M-bit EPROM packaged in a 44-pin PLCC. The S1C33209 I/O and this ROM both operate at 5 V.
When the bus clock speed is 20 MHz, the ROM access time requirements are as follows:
For 2-cycle read with one wait state (bus cycle = 100 ns), ROM access time of 80 ns (or 75 ns for 3.3 V) or greater
For 3-cycle read with two wait states (bus cycle = 150 ns), ROM access time of 130 ns (or 125 ns for 3.3 V) or greater

## 4.5 Connecting Flash Memory

Using the S5U1C33209D1 as an example, the following is a diagram of a x16-type flash memory connection.

### ● Connecting x16 flash memory

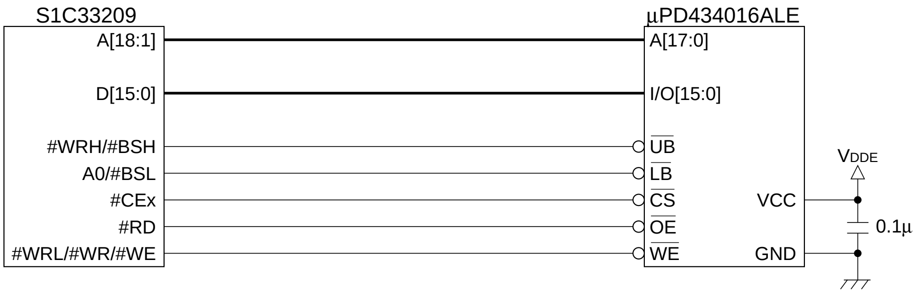
An 8M-bit flash memory in a 48-pin TSOP package is connected directly to the chip. A x8/x16 dual-type flash memory labeled "29F800" is used in a x16 configuration.



## 4.6 Connecting SRAM

### ● Connecting x16 SRAM

In the example shown below, one 4M-bit, x16 SRAM is connected to the chip.

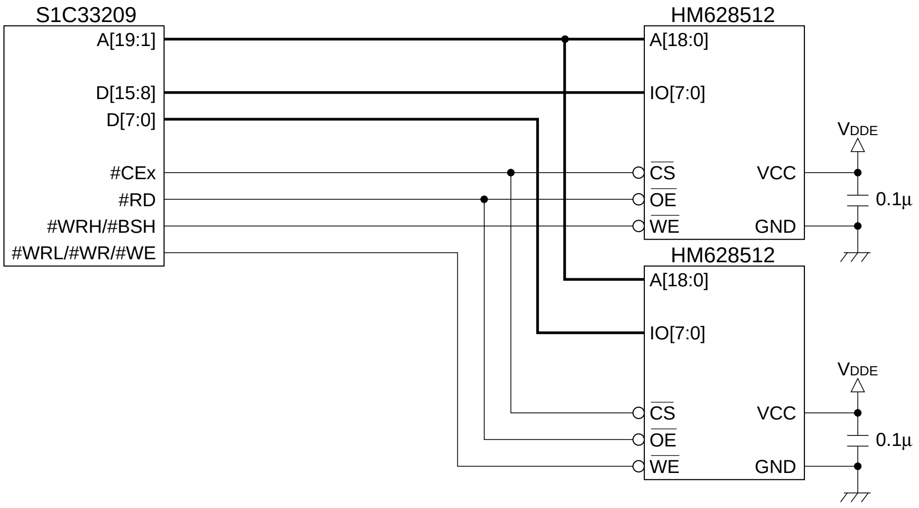


This type of RAM cannot be accessed with a default BCU setting. If BCU is changed to BSL mode, the RAM becomes operational with the wiring shown above. BSL mode is selected by setting D3 at 0x4812E to 1. Setting D3 = 0 selects regular A0 mode.

*Note: In the S1C33209 and S1C33L01, BSL mode cannot be used in combination with S5U1C33000H debugging. Use the S5U1C331M2S for debugging.*

### ● Connecting two x8 SRAMs

When two SRAM units are required, we recommend using two x8 type units, since they are easily connected, without requiring external logic. In the example shown below, two 4M-bit, x8 SRAMs are connected to the S5U1C33209D1.



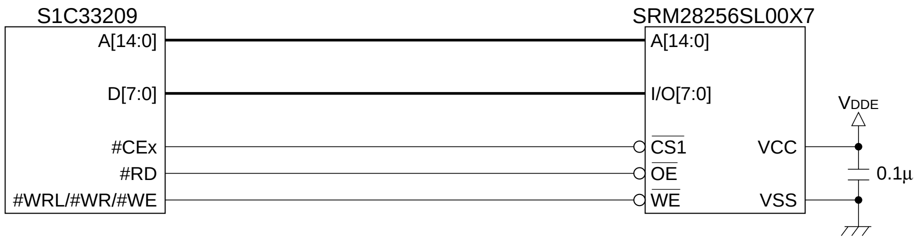
The address, #CE, and #RD outputs may be connected directly to the chip, although the 2-device connection may increase their load capacitance.

At a bus clock speed of 20 MHz, RAM access time requirements are as follows:
For 2 cycles with one wait state (bus cycle = 100 ns), RAM access time of 80 ns (or 75 ns for 3.3 V) or greater
For 3 cycles with two wait states (bus cycle = 150 ns), RAM access time of 130 ns (or 125 ns for 3.3 V) or greater

The access time for SRAM mounted on the S5U1C33209D1 (operating at 20 MHz) is 55 ns in one wait state.

### ● Connecting one x8 SRAM

This example illustrates the connection of a single 256K-bit, x8 SRAM.

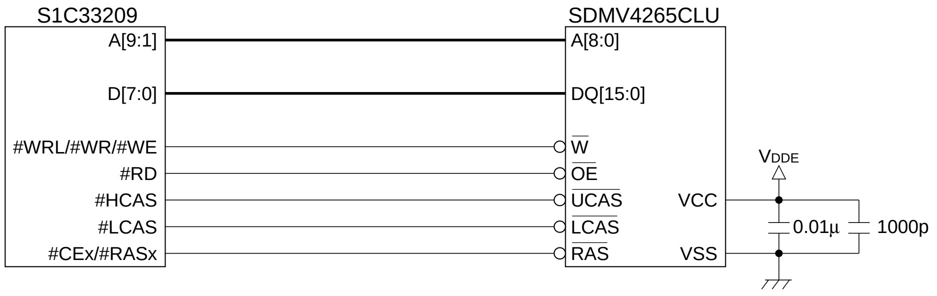


By default, the BCU is set to 16-bit size. Change its setting to 8-bit and set each area's setup register D6 or DE bit to 1.

## *4.7 Connecting DRAM*

Using the S5U1C33L01D1 as an example, the following shows a DRAM connection diagram. Note that a DRAM pattern is prepared on the S5U1C33L01D1, but that no DRAMs are yet mounted.

### ● Connecting 4M-bit, x16 DRAM

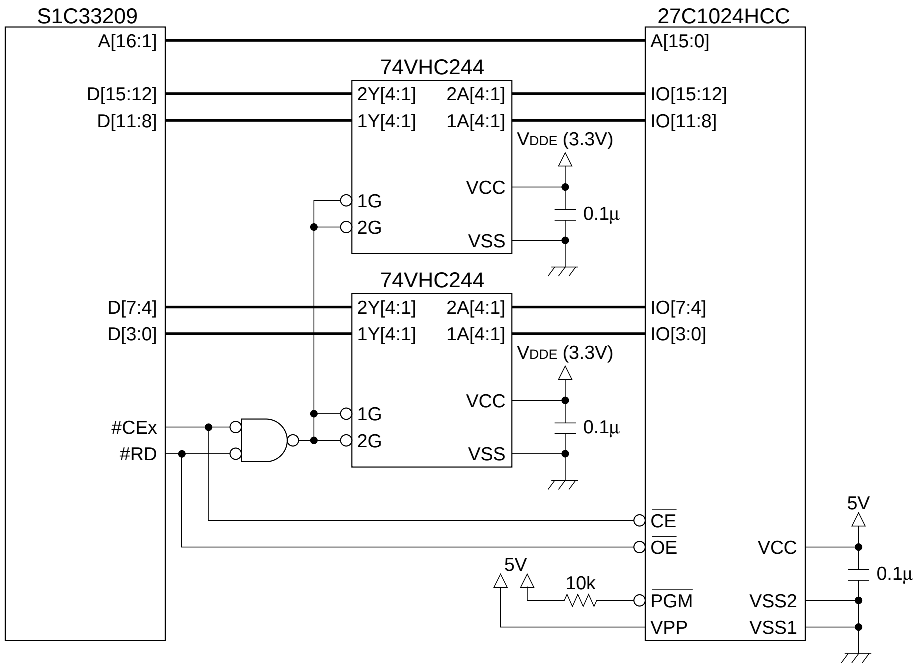


For more information on BCU settings, see Section 3.1, "Setting Up BCU".

## *4.8 Connecting 5 V ROM and 3.3 V Bus*

### ● Method for connecting a 5 V ROM to a 3.3 V bus

The S1C33209 bus is not 5 V-tolerant. If another 3.3 V memory device is connected, current will also flow into that memory. Connecting a 5 V device to the 3.3 V I / O S1C33 chip requires a buffer to absorb the potential difference.



In this example, two pieces of the 74VHC244 convert 5 V ROM output data to 3.3 V during a ROM read. The buffer operates only when the ROM is selected. This is used in the S5U1C33000H (the CPU, however, is the S1C33104).
VHC-type ICs tolerate 5 V input and receive 5 V signals even when operating at a power supply voltage of 3.3 V. Many low-voltage CMOS ICs exhibit this voltage-tolerant feature.
Although the address, #RD, and #CE signals fed to the ROM are at 3.3 V, they can be entered directly only if the ROM is TTL-level compatible (high at 2.0 V or above, low at 0.8 V or below).
If the 16244 is used for the buffer IC in place of the 244, one IC may be sufficient. The signal connected to the G pin on the buffer is an AND'd product of #CE and #RD. Data is output from Y only during ROM reads. Swapping out the buffer IC for a 245 or 16245 and connecting #CE to the G pin and #RD to the DIR pin creates a bi-directional buffer, in which case the AND logic shown above is unnecessary. A bi-directional buffer also permits use of ROM emulation memory, like the MEM33DIP42.

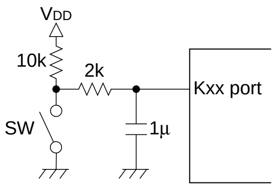
## 4.9 Ports

### ● Processing unused I/O (P) ports

By default, the unused I/O ports are set for input. Connect unused ports to $V_{DD}$ or $V_{SS}$, or switch them for output immediately after booting. Take care that ports connected to $V_{DD}$ or $V_{SS}$ are never set for output.

### ● Eliminating chattering on input (K, P) ports

Except for K60–K67, the K and P ports are Schmitt inputs, as is the reset pin. To simply eliminate several ms of chattering on 2-level switch inputs, configure a circuit like the one shown below.



In this example, no internal pull-ups are used. Turn-off from 0 to $V_{DD}$ constitutes a rise time of about 10 ms, eliminating several ms of chattering. Turn-on from $V_{DD}$ to 0 constitutes a fall time of about 2 ms. You can also reduce current drain at switch-on time by using a larger R. However, since this results in vulnerability to noise, route the wiring carefully.

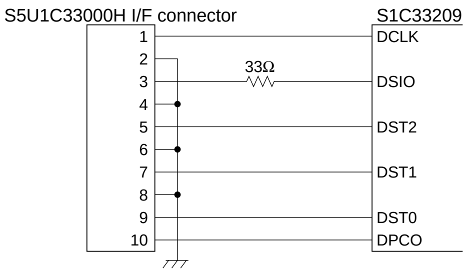
For pins which are not Schmitt inputs, use a 74HC14 or equivalent to eliminate chattering. To determine if a particular pin is a Schmitt input, see the user's manual supplied with each IC. (For the S1C33209 Technical Manual, see Appendix B, "Pin Characteristics".)

## *4.10 Connections for Debugging*

Using the S5U1C33209D1 as an example, this section explains how to connect the S5U1C33000H and the S5U1C330M1D1 for S5U1C331M2S.

### ● Connecting the S5U1C33000H

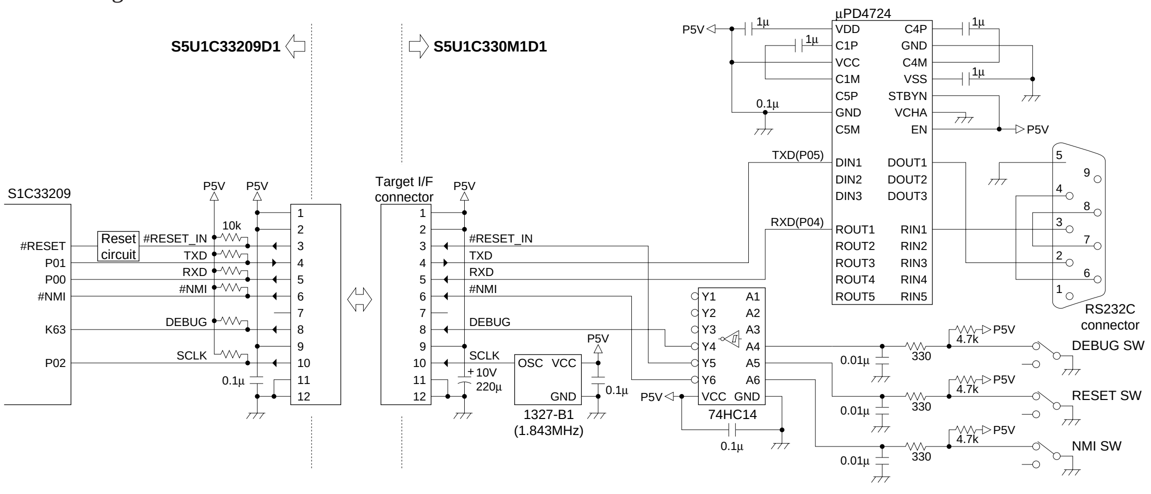
The S1C33209 has six dedicated pins to which a debugger can be connected, including DCLK, DSIO, DST2, DST1, DST0, and DPCO.
Add a 33 Ω resistor in series to DSIO. Use a total of 10 lines to connect to the S5U1C33000H, including four additional GND lines.



If the above pattern cannot be laid out on the circuit board, use aerial wiring to connect, without inserting the 33 Ω resistor. The S5U1C33000H will function with this connection.
You can also disable the PC trace function with the S5U1C33000H (by pushing the rightmost DIP switch down) and connect to the S5U1C33000H with only a total of four lines consisting of DCLK, DSIO, DST2, and one GND line. Except for PC trace, this allows all debug functions in ICD mode to be used without problems.
Make sure the above wiring length is 5 cm or less. In particular, the 33 Ω resistor for DSIO must be located as close to the 33 chip as possible. DSIO is the only input pin and is pulled high with internal 120 kΩ. A low pulse on this input places the device in debug mode. To prevent erratic DSIO behavior, if you are not debugging, leave the 33 Ω resistor out and minimize the pattern length of DSIO, or connect it high to 3.3 V (the core's VDD voltage) to prevent including noise.

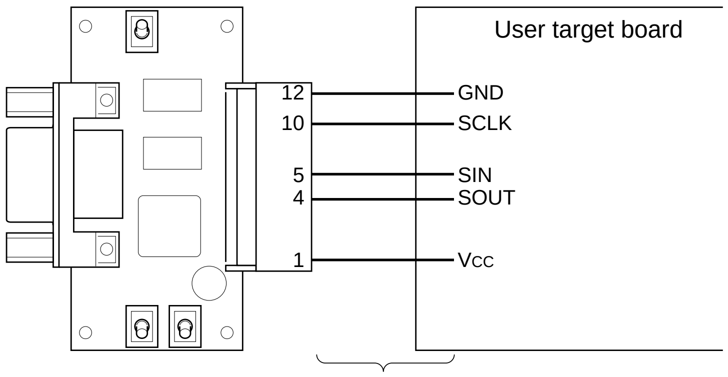
### ● Connection with the S5U1C330M1D1

S5U1C331M2S uses the following resources: 10K bytes of ROM, 4K bytes of RAM, and one serial interface channel.
The diagram shown below depicts a S5U1C330M1D1 circuit diagram and an interface component on the target board.



The S5U1C33209D1 is connected to the S1C33 to allow use of all S5U1C330M1D1 functions. Of these, three lines - RESET input, NMI input, and the debug switch for input port connection - are used for the sake of convenience rather than necessity.

Only essential pins need to be connected, as shown below.



Signal lines must be less than 10 cm in length.

There are five essential pins: SCLK, SIN, SOUT, GND, and $V_{DD}$. $V_{DD}$ is 5 V for the S5U1C330M1D1, and 3.3 V for the S5U1C330M1D2.

# 5 Speaker Output and External Analog Circuit using Fine PWM

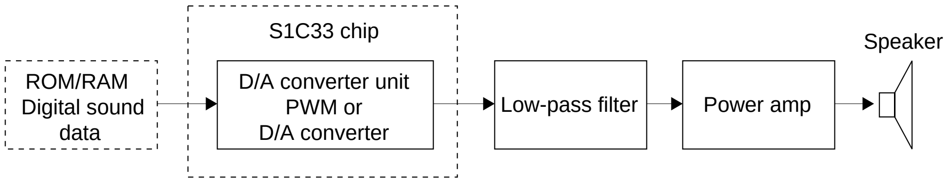
## 5.1 General Sound Output Circuits Based on Microcomputer

Sound (music) output to speakers using a microcomputer requires the following three general components.

1) D / A converter unit
   Converts digital sound data into analog form.

2) Low-pass filter unit
   Eliminates quantization noise from the D / A converted analog sound, smoothing it into a continuous analog waveform.

3) Power amp and speaker unit
   Amplifies the low-pass filtered analog waveform and drives the speaker.


S1C33 chip
Speaker
ROM/RAM Digital sound data
D/A converter unit PWM or D/A converter
Low-pass filter
Power amp

Here, we'll explain a general method for building a relatively low-cost sound output system, using a single power supply, as well as the structure of each block, sampling frequencies, and output accuracy vs. quality.

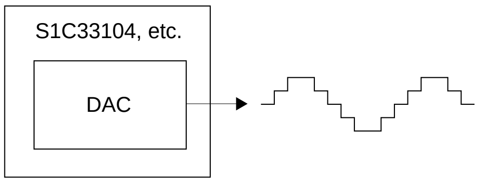
### 5.1.1 D/A Converter Unit

Digital sound data is generally converted into analog data using the following three methods:
1) Conversion by DAC
2) Conversion by resistor ladder
3) Conversion by PWM

Each method is explained below.

#### ● Conversion by DAC

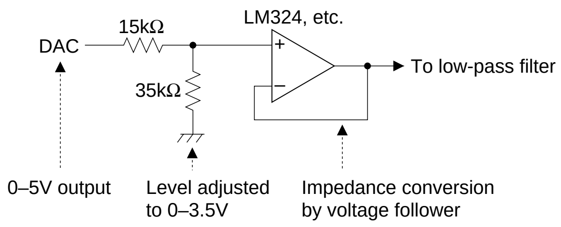
This method uses the DAC incorporated in a microcomputer to output sound.



The DAC built into a microcomputer generally is a R-2R resistor ladder-type with 8- to 10-bit resolution. For higher accuracy, prepare a dedicated off-chip DAC. A 12-bit R-2R type DAC is commonly used for sound output; 14–20-bit delta-sigma type DACs are often used for audio.

When using a DAC, pay attention to its output impedance. If the DAC produces low-impedance output (if capable of 5–10 mA output) using the op amp in the latter stage of R-2R, it can be received directly by the low-pass filter unit in the next stage.

High-impedance output may require a voltage follower to lower impedance.



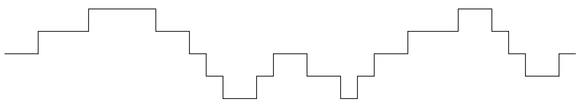
The input voltage is limited by the op amp used. Because an inexpensive CMOS-type op amp (e.g. LM324) is used in this example, the input voltage is divided by resistors to adjust it into the range 0 to 3.5 V. In this case, since a current flows into GND through 15 kΩ + 35 kΩ resistors, care must be taken that it does not exceed the rated output current of the DAC.
The following provides a rough guide to the op amp's input voltage range relative to the power supply voltage.

1) For ordinary bipolar type and FET type (e.g. RC4558 operating with positive/negative dual-power supplies)
   Positive power supply voltage - 1 to 1.5 V to negative power supply voltage + 1 to 1.5 V

2) For CMOS types (e.g. LM324 operating with single power supply)
   Positive power supply voltage - 1 to 1.5 V to GND + several mV to several 10 mV (almost GND)

This also applies to output voltages. A rail-to-rail type capable of full swing relative to the power supply is also available. While output rail-to-rail is relatively inexpensive, input/output rail-to-rail is too costly for low-cost systems.

DAC output is an analog waveform with quantization noise riding on it, as shown below.

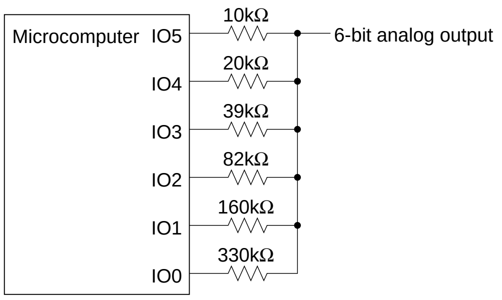


For output with 8 kHz of sampling frequency, for example, write digital data every 1/8000 seconds into the DAC in software. Because the output does not change states until the next data write, the output is in noncontiguous staircase form. This is quantization noise, which degrades audio quality centering around the same frequency as sampling. A low-pass filter in the next stage is required to eliminate this noise. Audio quality depends heavily on low-pass filter performance characteristics.

While the S1C33104 chip's internal 8-bit DAC may be used for audio output, 8-bit resolution is generally considered inadequate for audio quality. The S1C33209 uses a 16-bit timer and the PWM method described further below to provide high resolution, from 10 bits to a maximum of 15 bits.

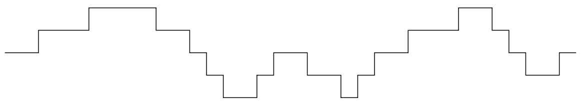
## ● Conversion by resistor ladder

This configures a simplified version of DAC by connecting external resistors to a microcomputer's I/O ports. This method is used specifically for microcomputers lacking a DAC, but may be used for all types of microcomputers.



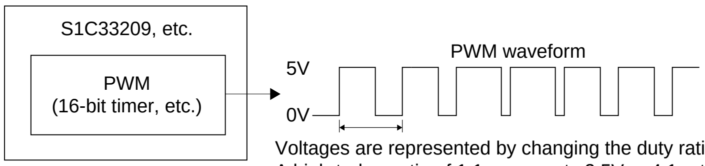
Resistor values selected from the E24 series
10 kΩ and 20 kΩ have a 1% error; others are 5% accurate.

The resistors used for higher-order bits must provide better accuracy. Resistors with 1% accuracy are limited to 6-bit resolution. Even those with 0.5% accuracy are generally limited to 7 bits. But since relative accuracy is important, we can obtain a resolution of 8 bits (more or less) by using a R-2R resistor ladder (with resistors integrated into a single component, using the R-2R method, e.g. resistor arrays from BI Technologies in the U.S.). In most cases, the R-2R method D / A with 12 bits or more produces the desired resolution by trimming output internally. But because this method creates high output impedance, a voltage follower is required for low impedance conversion before the output can be fed to the low-pass filter unit.
The waveform itself is of the same staircase form containing quantization noise as for the DAC described above.



## ● Conversion by PWM

Instead of outputting analog voltages, this method represents voltages by changing the duty ratio (the ratio of 1 to 0 pulse widths) of a digital waveform. PWM outputs differ markedly from DAC output waveforms. But after smoothing with a low-pass filter, we obtain a staircase analog waveform containing quantization noise, as with DAC output waveforms. Furthermore, since the audio portion of PWM has the same spectrum as that of DAC output, both are perceived as identical to human ears.



Voltages are represented by changing the duty ratio in this one cycle.
A high-to-low ratio of 1:1 represents 2.5V, a 4:1 ratio represents 4V, and a 2:3 ratio represents 2V.

In this case, PWM cycles (carrier frequency) must be greater than the D / A conversion cycles (band to be reproduced). For example, we use a carrier frequency of 80 kHz or higher for sound reproduction. When passed through a low-pass filter that cuts frequencies above 20 kHz, we obtain the same staircase analog waveform obtained from the DAC described above.

The PWM waveform has a broad noise spectrum centering around the carrier frequency, say 80 kHz. This frequency band significantly exceeds the audio frequency, so that even when this PWM waveform is output directly to speakers, it has no perceptible effect on sound for human ears. We can safely convert PWM waveforms into continuous analog waveforms using a low-pass filter before speaker reproduction. Since the low-pass filter used in the next stage to cut quantization noise can also be used for this purpose, a low-pass filter is not required for this D / A converter unit. But not all noise concentrates around 80 kHz, and traces of PWM noise are found even in the audible frequency range. These noise components can be reduced by using a higher carrier frequency — 160 to 320 kHz — but, in practice, the 80 kHz carrier presents no problems for 10-bit D / A conversion. In addition, since the output impedance is regulated to low impedance by I / O pads for PWM use, no impedance conversion by a voltage follower is required.

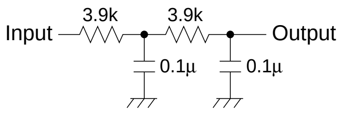
The accuracy of the PWM method is determined by the resolution of the pulse width. To realize 8-bit accuracy, one cycle must be $256 \times 80$ kHz, requiring a 20 MHz reference clock. The audio output library for the S1C33209 realizes 10-bit accuracy with PWM, providing high audio quality comparable to that of a 10-bit DAC. Normally, 10-bit accuracy requires $1024 \times 80$ kHz = 80 MHz clock, but the S1C33209 obtains the same effects with a 40 MHz clock, thanks to PWM technology.

For years, the PWM method been known as a D / A conversion method that features high differentiation accuracy. But due to its need for a high-frequency reference clock, the method has not always been practical for the voice band. A variation of this method has been used for voice applications as a delta-sigma type DAC in which PWM is converted into PDM (Pulse Density Modulation), which is then subjected to digital signal processing in the time-base direction to improve S / N ratios. This high-resolution PWM is a Seiko Epson exclusive technology, in which pulse width is controlled in units of half-clock periods. Combined with a S1C33 chip capable of operating at 40 MHz or better, this technology has made possible significant advances — now outpacing DAC — for PWM, which was formerly regarded as impractical for audio output use.

### *5.1.2 Low-pass Filter Unit*

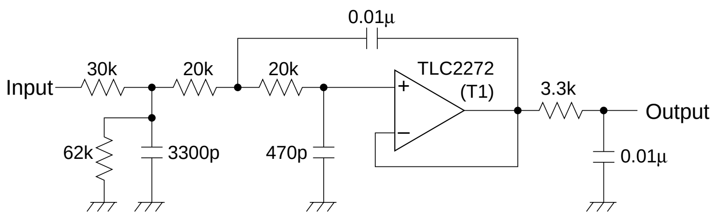
The quantization noise generated during D / A conversion degrades audio quality. To the human ear, this noise appears as roughness in the sound, with shrill high tones. Resolving this problem requires careful design of the low-pass filter unit to eliminate quantization noise. Costs must also be considered.
For low-cost systems, we recommend second-order to fourth-order low-pass filters. Set the cutoff frequency to about 1 / 2.5 of the sampling frequency (for fourth-order filters) or 1 / 3 (for third-order filters), or 1 / 3.5 to 1 / 4 (for second-order filters). Attenuate higher frequencies. At higher cutoff frequencies, quantization noise centering around the sampling frequency becomes conspicuous, degrading audio quality. Due to their low attenuation, the safe course is to avoid first-order low-pass filters. Note that depending on usage conditions (for example, when you want artificially emphasized high tones to be heard clearly against background noise), quantization noise is sometimes generated intentionally.

#### ● Example of a second-order filter



Low-pass filters, each consisting of R and C, are combined to form a second-order filter. This example is designed for 8 kHz output, with a cutoff frequency slightly lower than 2 kHz. This enables configuration of an inexpensive filter with two resistors and two capacitors. However, attenuation near the cutoff frequency is moderate, resulting in slightly degraded audio quality - a tinkling, metallic sound.

#### ● Example of a fourth-order filter



This configures the third-order active filter with one op amp, followed by an additional first-order low-pass filter consisting of R and C, together forming the fourth-order filter. Providing good attenuation characteristics, this filter is adequate for acceptable audio quality in low-cost systems. This example is designed for 8 kHz output, with a cutoff frequency of approximately 3 kHz. Additionally, 30 kΩ and 62 kΩ inserted at the input narrow the input voltage range by a factor of 0.67 to prevent saturating the op amp input.

#### ● Oversampling

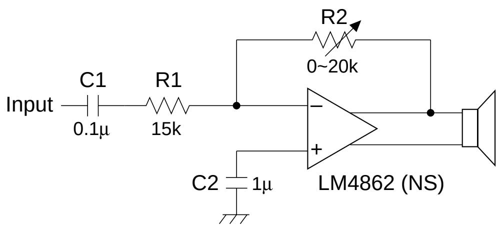
In Seiko Epson's speech and music middleware (e.g. S5U1C330V1S, S5U1C330T1S, and S5U1C330S1S), x2 oversampling technology is used in audio output to significantly reduce software quantization noise, reducing the load on the low-pass filter unit. The result is such that even when using filters above the fourth-order, no differences in audio quality can be detected by ear. Without oversampling, the fourth-order shown above is inadequate. The commonly used fifth and higher-order Chebyshev filters are structurally complex and expensive.

Also see Section 5.4, "Examples of Audio Output Analog Circuits".

### *5.1.3 Power Amp and Speaker Unit*

We describe two examples here. In one, we use a dedicated differential-type power amp to produce a large sound volume. In the second, we use transistors to realize moderate sound volume at low cost.

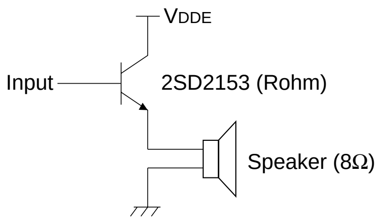
#### ● Example of a power amp



Capacitor C1 at the input configures the first-order high-pass filter to cut D.C components. The cutoff frequency determined by C1 and R1 is normally around 50 Hz. With lower cutoff frequencies, the popping tone heard when sound is first produced becomes conspicuous. The input impedance here must be several times higher than the output impedance of the low-pass filter unit. If the relative magnitudes of the impedances are the same or in reverse relationship to this requirement, filter characteristics may be altered due to mutual interference of low- and high-pass filters.

The differential amp is used to drive the speaker. Audio volume is determined by R2/R1.

#### ● Driving the speaker with a transistor



The speaker is driven here by an emitter-follower. For such applications, select a transistor with large hfe (500 or greater; Darlington is unusable due to its narrow voltage range). For current amplification, the impedance in the D/A converter and the low-pass filter units must be matched to this.

Also see Section 5.4, "Examples of Audio Output Analog Circuits".

## *5.2 About Sampling Frequency and Bit Precision vs. Audio Quality*

### ● Sampling frequency

Higher sampling frequencies generate more high tones, with better fidelity to natural sound. A sampling frequency of at least 2 kHz or higher is required. At lower sampling frequencies, sound becomes unclear, making speech difficult to make out. Audio quality increases as sampling frequencies increase to 4 kHz, 8 kHz, and 16 kHz. Of these frequencies, 8 kHz was adopted for telephone communications. For this reason, 8 kHz is used in countless products. The sampling frequencies above 16 kHz are 22 kHz and 32 kHz, frequencies with which sound may be reproduced close to 10 kHz and 15 kHz, respectively. But due to the relative insensitivity of human hearing to the higher frequencies and the limited performance expected of low-cost systems, no further increase in audio quality is to be expected. Higher sampling frequencies include the 44.1 kHz CD and 48 kHz DAT classes.

As sampling frequencies increase, so does data volume. As a rough guide, we recommend the following sampling frequencies for low-cost systems:

| | |
|---|---|
| Human voice: | 8 kHz (when data size concerns have priority) |
| | 16 kHz (when audio quality has priority) |
| Music: | 22.05 kHz |
| | 32 kHz (high-end audio quality) |

### ● Bit precision

S / N ratios change significantly according to the number of bits used in the D / A converter unit. Roughly speaking, when the number of bits increases by one, the S / N ratio increases by 6 dB. The approximate relationship between the number of bits and audio quality is given below.

(1) 8 kHz sampling

| | |
|---|---|
| 1 to 3 bits: | Sound is hidden behind noise, so that the speech is difficult to make out (the signals are perceived as human voice signals, but nothing further can be perceived). |
| 4 to 5 bits: | Although speech can be understood, noise levels are significant and obtrusive. |
| 6 to 7 bits: | Sound quality is clearer, but irritating noise levels persist. |
| 8 to 9 bits: | Results are useable for real-world applications, with perceptible noise levels, which are not disagreeable to the ear. |
| 10 bits or more: | No noise can be perceived, even when heard in a quiet environment. |

A precision of at least 8 bits is desirable. If resources permit, consider using 10 bits.

(2) 22 kHz or higher sampling

As sampling frequencies increase, quantization noise becomes more conspicuous to the ear.

| | |
|---|---|
| 8 to 9 bits: | Even in somewhat noisy rooms, noise remains perceptible. |
| 10 to 11 bits: | Under normal conditions, noise cannot be detected. |
| 12 bits or more: | No noise can be detected, even when heard in a quiet environment. |

A precision of at least 10 bits is desirable. If resources permit, consider using 12 bits.

(3) 16 kHz sampling

Audio quality is almost midway between 8 kHz and 22 kHz. A precision of at least 9 bits is desirable. If resources permit, consider using 11 bits.

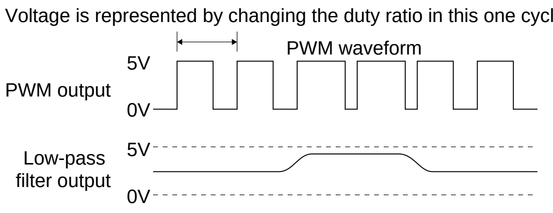
The S1C33104 by itself is capable only of 8-bit output, using its internal 8-bit DAC. The S1C33209 can produce 8 to 32 kHz, 10 to 15-bit output thanks to its PWM, providing ample capabilities for most applications.

## *5.3 10-bit D/A Conversion by PWM*

The S1C33209 is able to realize high-resolution audio output, from 10 bits up to 15 bits, thanks to its high-resolution PWM technology. This section will first describe 10-bit output with high-resolution PWM, then discuss 15-bit output. (See Section 5.6, "15-bit D / A Conversion by PWM".)

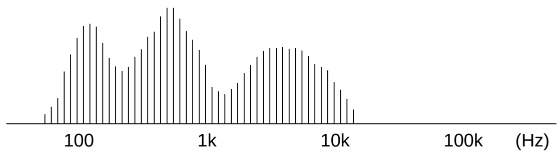
### ● Differences between PWM and DAC

As previously described, PWM uses the duty ratio to represent voltages, and its waveform differs markedly to the eye. However, when PWM components are removed by a low-pass filter, the resulting waveform closely resembles DAC output waveforms.

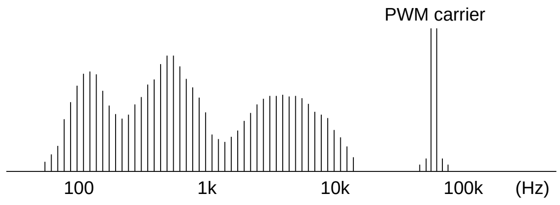

Voltage is represented by changing the duty ratio in this one cycle.
PWM waveform
5V
PWM output
0V
5V
Low-pass filter output
0V

The human ear perceives frequency spectrum as sound rather than waveforms.

Spectrum of DAC output



Spectrum of PWM output

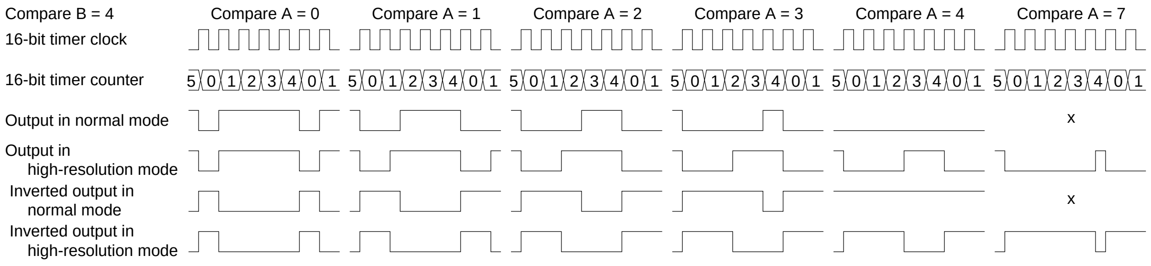


PWM output has significant power near the carrier frequency, but in the same spectrum as that of DAC output in the audible frequency range. Thus, although the output waveforms of PWM and DAC are quite different, both PWM and DAC outputs are perceived as identical by human ears. Because the PWM carrier noise disappears when processed by the low-pass filter unit, eliminating quantization noise, the spectra of both waveforms ultimately match.

### ● About high-resolution PWM mode

The accuracy of PWM output depends on how elaborately the duty ratio of output waveform can be controlled. Obtaining 8-bit accuracy using a constant cycle of 80 kHz requires: 80 kHz × 256 clock periods = 20 MHz clock, which indicates that pulse width must be controlled in units of 0.05 µs. The PWM available with the audio output middleware for the S1C33209 is 10-bit accurate, so that control of one clock width requires 80 kHz × 1024 = 80 MHz clock. The S1C33209 drives the 16-bit timer for PWM use with a 40 MHz clock, and controls output pulse width in units of half-clock periods. Combined, this results in 80 MHz equivalent PWM output.

PWM output in high-resolution mode



## ● PWM programming using high-resolution mode

In this section, we'll discuss how to produce PWM output in high-resolution mode, using gnu33\sample\drv33208\pwm as an example.

High-resolution PWM control (Excerpt from drv_pwm.c)

```
void init_16timer1(unsigned short compareA, unsigned short compareB)
{
      /* Save PSR and disable all interrupt */
      save_psr();

      /* Set 16bit timer1 prescaler */
      *(volatile unsigned char *)PRESC_P16TS1_ADDR                           (1)
               = PRESC_PTONL_ON | PRESC_CLKDIVL_SEL0;
               // Set 16bit timer1 prescaler (CLK/1)

      /* Set 16bit timer1 TM1 port enable */
      *(volatile unsigned char *)IO_CFP2_ADDR |= IO_CFP23_TM1;               (2)

      /* Set 16bit timer1 comparison match A data */
      *(volatile unsigned short *)T16P_CR1A_ADDR = compareA;                 (3)

      /* Set 16bit timer1 comparison match B data */
      *(volatile unsigned short *)T16P_CR1B_ADDR = compareB;                 (3)

      /* Set 16bit timer1 mode, fine mode, comparison buffer enable, output normal */
      *(volatile unsigned char *)T16P_PRUN1_ADDR = T16P_SELFM_FM | T16P_SELCRB_ENA
               | T16P_OUTINV_NOR | T16P_CKSL_INT | T16P_PTM_ON | T16P_PSET_OFF
               | T16P_PRUN_RUN;                                              (4)

      /* Restore PSR */
      restore_psr();
}

void  set_16timer1(unsigned short compareA)
{
      /* Set 16bit timer1 comparison match A data */
      *(volatile unsigned short *)T16P_CR1A_ADDR = compareA;
}
```

### Initializing the PWM timer (16-bit timer channel 1)

(1) Setting the prescaler

Feed the clock directly to 16-bit timer 1 without dividing it by the prescaler.

```
/* Set 16bit timer1 prescaler */
*(volatile unsigned char *)PRESC_P16TS1_ADDR
   = PRESC_PTONL_ON | PRESC_CLKDIVL_SEL0;
   // Set 16bit timer1 prescaler (CLK/1)
```

(2) Switching over port functions

Switch the functions of pins shared with I/O ports for PWM output.

```
/* Set 16bit timer1 TM1 port enable */
*(volatile unsigned char *)IO_CFP2_ADDR |= IO_CFP23_TM1;
```

(3) Setting compare data

Set the compare A data (pulse rise timing) for 16-bit timer 1.

```
/* Set 16bit timer1 comparison match A data */
*(volatile unsigned short *)T16P_CR1A_ADDR = compareA;
```

Set the compare B data (cycle) for 16-bit timer 1.

```
/* Set 16bit timer1 comparison match B data */
*(volatile unsigned short *)T16P_CR1B_ADDR = compareB;
```

(4) Setting 16-bit timer 1 mode and starting

Set the timer's operational mode and allow PWM output to start.

```
/* Set 16bit timer1 mode, fine mode, comparison buffer enable, output normal */
*(volatile unsigned char *)T16P_PRUN1_ADDR = T16P_SELFM_FM | T16P_SELCRB_ENA
   | T16P_OUTINV_NOR | T16P_CKSL_INT | T16P_PTM_ON | T16P_PSET_OFF | T16P_PRUN_RUN;
```

The following settings are made here:
- Select high-resolution mode (to produce high-resolution PWM output)
- Enable the compare data buffer (to set duty change data asynchronously)
- Select non-inverted output (each cycle begins with 0)
- Select the internal clock (prescaler output clock)
- Turn timer output on (outputs PWM waveform)

When the timer starts, the output waveform begins with 0. When the counter matches compare A, it goes high (= 1); when the counter matches compare B, it goes low (= 0). These ascending and descending transitions comprise one cycle, which is determined by the set value of compare B. Unless the compare A register is changed at this point, the same waveform is output in the next cycle.

### Changing the duty ratio

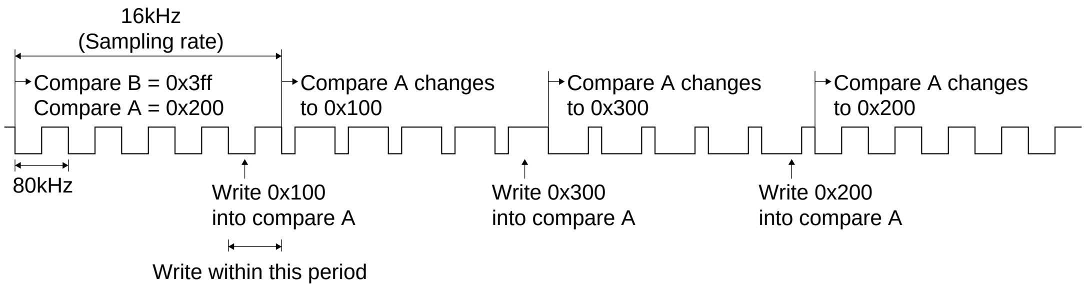
Because the compare data buffer is enabled in (4), compare A data can be written to asynchronously with a count operation.

```
void  set_16timer1(unsigned short compareA)
{
      /* Set 16bit timer1 comparison match A data */
      *(volatile unsigned short *)T16P_CR1A_ADDR = compareA;
}
```

When compare A is rewritten by this function, a new duty ratio takes effect, beginning with the next cycle. Because the output waveform at the time of the write is unaffected, the waveform can be changed smoothly.

### Compare data

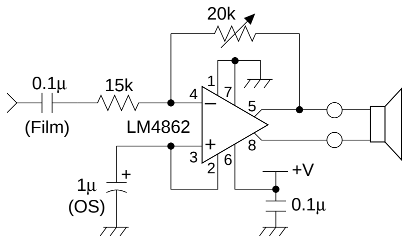
For audio output, set compare B to 80 kHz or higher in terms of cycle and write the data to be D/A converted directly into the compare A data buffer asynchronously every sampling period (8 to 32 kHz).



## 5.4 Examples of Audio Output Analog Circuits

### ● Power amp

Shown below is the power amp circuit mounted in the S5U1C330A1D1.



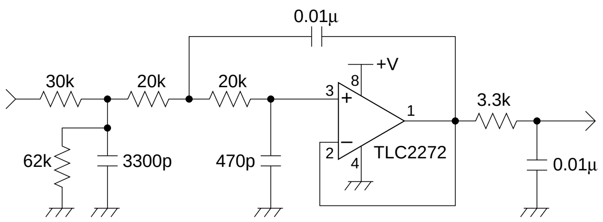
Film capacitors are better than ceramic capacitors as capacitors for signal reception. Inexpensive polyethylene film capacitors may be used without problems. Because ceramic capacitors exhibit minute hysteresis, use of this capacitor type in a circuit in which signal passes directly may result in signal distortion. An OS capacitor is most suitable for the 1 µF capacitor used for AC coupling apart from GND. Although electrolytic capacitors may be used, they affect audio quality, if only slightly. Carbon film type resistors with 5% accuracy should serve adequately.

The types of speakers generally used for audio applications are 4 Ω to 8 Ω. Commonly used for portable equipment are 8 Ω speakers; even smaller equipment uses speakers above 8 Ω (e.g. 24 Ω).

### ● Low-pass filter configured with an op amp

The following shows examples of 8, 16, and 22.05 kHz sampling low-pass filters configured with one op amp. All are audio quality-prioritized, fourth-order low-pass filters mounted in the S5U1C330A1D1 and S5U1C330A2D1.

#### Fourth-order low-pass filter for 8 kHz sampling (S5U1C330A1D1)



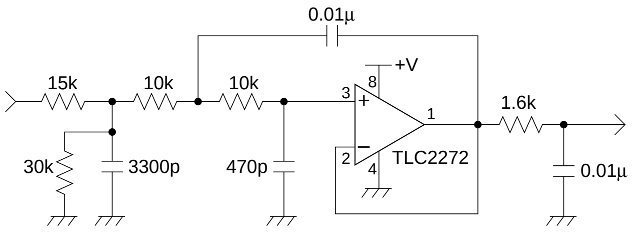
To eliminate 8 kHz sampling quantization noise, choose a cutoff frequency in the range 3.5 kHz to 2.7 kHz. In this filter, the cutoff frequency is set to 3.0 kHz. As the cutoff frequency rises, quantization noise becomes audible at around 3.5 kHz (when using x2 oversampling). The first dividing resistor lowers the 5 V input to a little above 3 V, matching it to the op amp's rated input voltage (0 to about 3.5 V). The op amp is the third-order filter, and the RC network following it is the first-order filter. Together, they comprise the fourth-order low-pass filter.

Here, use carbon film resistors with 5% accuracy or better. Metal film resistors are ideal, but the difference is relatively insignificant, unless minute signals are being handled.

Capacitor selection requires care. When using laminated ceramic capacitors, select a B-characteristic type that guarantees accuracy of ±10% or better (at worst, ±20%) within the operating temperature range. Do not use capacitors with +80% -40% Z accuracy. Be particularly leery of inexpensive 0.01 µF capacitors, since most are Z-accurate. Low-pass filter characteristics deteriorate with lower accuracy. Although film capacitors are suitable for analog circuits, they are not always ideal for low-cost audio output.

For the op amp, choose a CMOS-type single-power supply with an input voltage range of 0 to 3.5 V. An inexpensive op amp is fine. The same applies for DAC output.

### Fourth-order low-pass filter for 16 kHz sampling (S5U1C330A2D1)

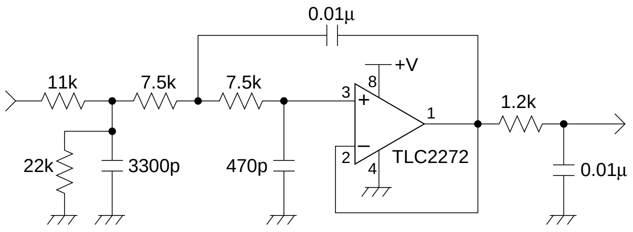


Configured in the same way as the 8 kHz sampling circuit, this filter has a cutoff frequency set to 6.1 kHz. If all resistor values are halved without changing capacitor values, the cutoff frequency doubles while the characteristic curve remains unchanged. The same is true when all capacitor values are halved without changing resistor values. However, because the capacitors are primarily of the E6 series and the range of capacitance values is relatively narrow, E24 series-based resistors are to be preferred.

E6 series: Six discrete values–10, 15, 22, 33, 47, and 68 (every 1.5-fold)
E24 series: 24 discrete values– 10, 11, 12, 13, 15, 16, 18, 20, 22, 24, 27, 30, 33, 36, 39, 43, 47, 51, 56, 62, 68, 75, 82, and 91 (every 1.1-fold)

Although more minute choices are available for some components, it is safer to design with the above-valued resistors, which are relatively easy to obtain.

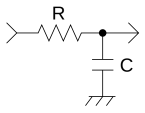
### Fourth-order low-pass filter for 22.05 kHz sampling (S5U1C330A2D1)



This circuit is the same as those described above, with the 8 kHz sampling resistance values replaced by 8/22 values. The cutoff frequency is 8.3 kHz.

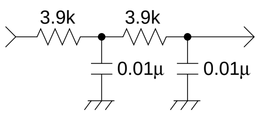
## ● Low-pass filter comprised of an RC network

The first-order low-pass filter consisting of R and C is configured as shown below. Its cutoff frequency is obtained by calculating $1/(2\pi \times R \times C)$.



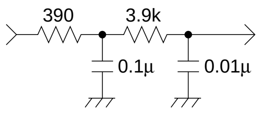
The attenuation factor is 6 dB/oct. When frequency doubles, the waveform is halved. For this reason, audible quantization noise cannot be entirely eliminated. Thus, two such filters are used, with one placed above the other. The configuration creates an effective low-pass filter for cost-priority systems. The resistors and capacitors used in this RC low-pass filter also require caution with regard to usage, just as for op amp based fourth-order filters. Again, we recommend avoiding Z-accuracy capacitors.

### Second-order RC low-pass filter for 8 kHz sampling



This configuration comprises a low-pass filter whose cutoff frequency is 2 kHz. However, because the preceding and following RC networks have the same impedance, the roll-off near the cutoff frequency is moderated by interference.

Shown below is a circuit with this part improved (used in the S5U1C330A1D1).



Because the impedance in the following stage differs by a factor of 10 from the preceding stage, the attenuation characteristics near the cutoff frequency are quite sharp. However, since the resistance in the preceding stage is small, a current of about 2 mA (when operating at 5 V) flows into it from the S1C33209 chip. When the resistance is 3.9 kΩ, this current is around 0.2 mA. Note that the PWM output characteristics are slightly bowl-shaped, a shape determined by the resistance value in the preceding stage. For example, the output characteristics are bowl-shaped by about 40 mV for 390 Ω, and by about 4 mV for 3.9 kΩ. This affects the distortion factor slightly.

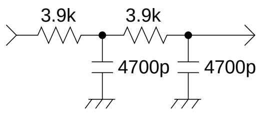
When connecting to the DAC of the S1C33104, change the 390 Ω resistor in the preceding stage to 150 Ω, since the DAC's output section contains an internal resistor of approximately 250 Ω in series.

The impedance in the following stage must be lower than that of the power amp's high-pass filter. To prevent impedance interference, this impedance value must be 1/4 or less — preferably 1/10 or less — that of the latter. A resistance value of 3.9 kΩ was determined, assuming a power amp input impedance of 15 kΩ or greater. Because large impedances greater than 1/4 of the power amp value affect the characteristics of both, overall design considerations must also account for the design of the power amp.

Of the two circuits above, we recommend the first example (3.9 kΩ + 0.01 µF stacked two-high). If a greater emphasis on high tones is desired, try changing 0.01 µF to 6800 pF. Note that quantization noise will increase.

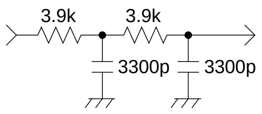
When using a two-high stack of RC networks, take care that the impedance of the following stage is never lower than that of the preceding stage. Characteristics may otherwise become degraded to the point of unusability.

## RC low-pass filter for 16 kHz sampling



With this circuit, the 0.01 µF capacitor for 8 kHz sampling is nearly halved to 4700 pF. The cutoff frequency is approximately 4 kHz. If a greater emphasis on high tones is desired, change 4700 pF to 3300 pF. Note that quantization noise will increase.

## RC low-pass filter for 22.05 kHz sampling

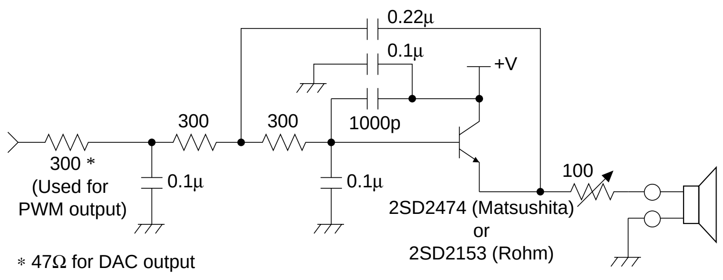


With this circuit, the 0.01 µF capacitor for 8 kHz sampling is nearly divided by 3 to 3300 pF. The cutoff frequency is approximately 6 kHz. If a greater emphasis on high tones is desired, change 3300 pF to 2200 pF. Note that quantization noise will increase.

## ● Driving the speaker with a transistor

When using transistors to drive the speaker, design the low-pass filter and power amp unit side by side. The third-order low-pass filter is adopted here.

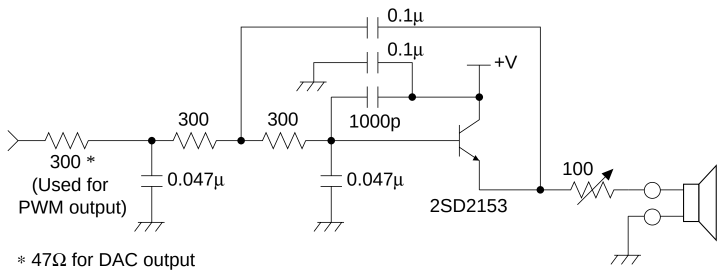
### Transistor amp circuit for 8 kHz sampling



Choose a transistor of 500 or larger hfe (current amplification factor). Because the current is amplified, the low-pass filter unit must have low impedance. An impedance of about 1 kΩ from the D / A converter unit to the transistor results in a good balance. Larger impedances values rapidly reduce sound volume, so that a small increase in impedance will result in a dramatic drop in sound volume. Conversely, smaller impedances make circuit design difficult, including selection of capacitor capacitance current values. Nor will this noticeably raise sound volumes. In the above example, the cutoff frequency is approximately 2.5 kHz.

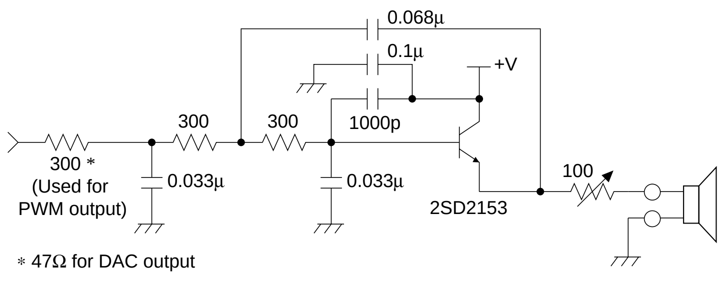
When entering from the DAC of the S1C33104, change the 300 Ω resistor in the preceding stage to 47 Ω (300 Ω minus the DAC's internal resistor of about 250 Ω). Note that the 0.1 µF capacitor connected to +V is used to decouple the power supply, and that the 1000 pF is used to prevent oscillation. Without these capacitors, the transistor output may oscillate. The 100 Ω variable resistor in front of the speaker is used to control the volume.

### Transistor amp circuit for 16 kHz sampling



The low-pass filter's cutoff frequency is about 5 kHz.

### Transistor amp circuit for 22.05 kHz sampling

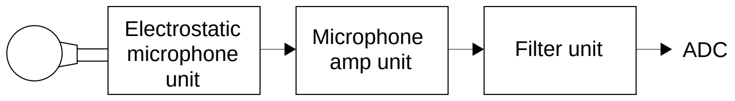


The low-pass filter cutoff frequency is about 8 kHz.

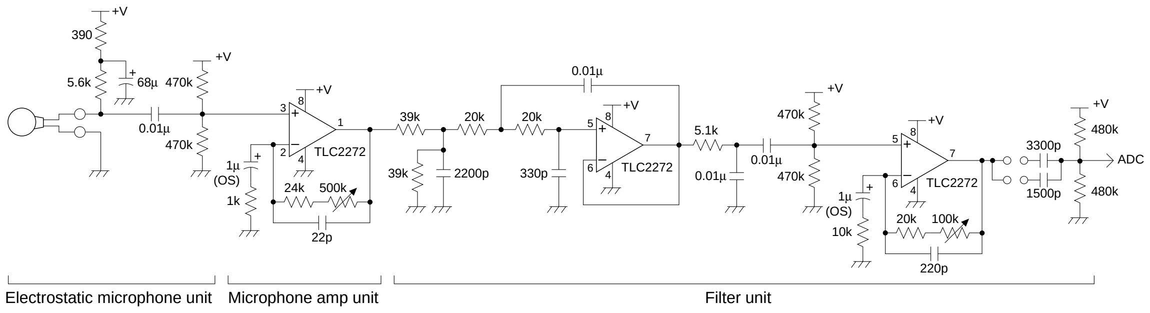
The low-pass filters used here can may be used in combination with the S1C33209 PWM or S1C33104 DAC.

## *5.5 Example of a Sound Input Analog Circuit*

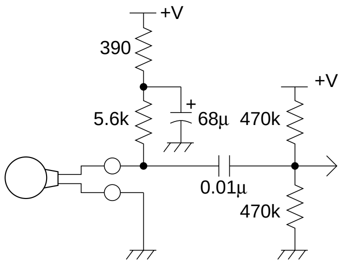
This section explains how to enter sound using an A / D converter.



Although the specific configuration of the sound input circuit depends on the input source, we'll examine it separately in the blocks shown above (configuration of the S5U1C330A1D1 circuit).



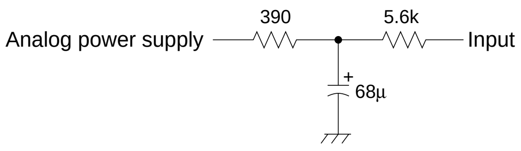
### ● Electrostatic microphone unit



#### Electrostatic microphone and AC coupling

The manufacturer's original recommendation for the 5.6 kΩ resistor inserted in the line-feed power to the electrostatic microphone was originally 1.5 kΩ. This is because the potential difference here constitutes the input signal level; we therefore increased the resistor value to reduce the burden on the microphone amp in the next stage, producing a 3.7-fold gain. This also reduces current consumption. However, an excessively large resistor value reduces current more than necessary, destabilizing the electrostatic microphone itself. The feasible limit may be around 4 times the original value. We use metal film resistors here, since minute signals of a magnitude less than mV are being handled.

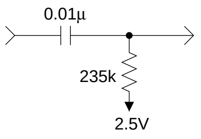
The noise appearing here, including power supply noise, is amplified in direct proportion to the amount of gain here and in the next stage. Thus, noise must be smaller here than at any other point in the circuit. To this end, the analog power supply has a first-order low-pass filter with a cutoff frequency of 5 Hz comprised of 390 Ω and 68 µF, which cuts voice band noise over a wide frequency range.



For 68 µF, an electrolytic capacitor may be used without problems.

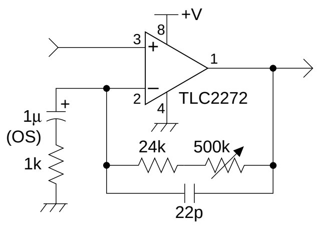
The 0.01 µF capacitor and 470 kΩ resistors, one to GND and one to the power supply, are used for AC coupling to 1 / 2 power supply voltage, and to cut the DC component as a first-order high-pass filter. The cutoff frequency is approximately 70 Hz, below which frequencies are attenuated.

### High-pass filter equivalent circuit



For resistors used for AC coupling, select ones providing 1% accuracy or better. Unless the exact middle point is set here, large-scale amplification by the microphone amp may cause the signal to exceed the VDD–GND range, producing clipping. Depending on the amplification factor, an accuracy of 0.5% may be required. Because minute signals pass through the high-pass filtering capacitor, use a film capacitor (polyester). Ceramic or other types of capacitors may degrade audio quality.

## ● Microphone amp unit



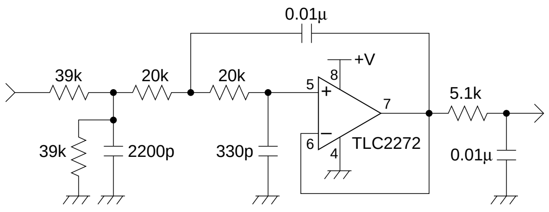
The gain for this AC amplifier may be adjusted in the range of 24-fold to 524-fold using a variable resistor. Combined with the 3.7-fold gain in the electrostatic microphone unit, this amounts to a gain of 90-fold to 2,000-fold. However, because 524-fold is used for experimental purposes, the amp as installed in actual products may need to be configured in two stages, or receive other consideration. Note that with the same gain, noise is smaller for amplification in one stage than for amplification in two stages.

Adjust the gain in the range 24 k / 1 k = 24-fold to (24 k + 500 k) / 1 k = 524-fold using the 500 kΩ variable resistor. This variable resistor may be preselected from the readily-available values 1, 2, or 5. The 22 pF capacitor connected in parallel with 24 kΩ and 500 kΩ is a low-pass filter that lowers the gain in highs. However, to prevent oscillation of the op amp, its cutoff frequency is high, varying with the variable resistor value. Such feedback loop low-pass filters do little to prevent oscillation. It is better to lower the gain with the RC low-pass filter at the input, since the cutoff frequency in this case is fixed and high oscillation prevention effects are already present. But because the input stage is already AC-coupled, we gave up the idea of using an RC low-pass filter.

The 1 kΩ and 1 µF comprise the first-order high-pass filter with cutoff of 150 Hz. For low-cost systems discussed in this manual, 50 or 60 Hz — including ham noise and low frequencies — results in various problems. Along with AC coupling in the preceding stage, this filter reduces these noise sources to a minimum. The remaining noise is eliminated by a filter in the following stage.

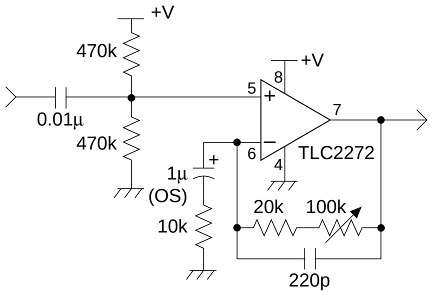
## ● Filter unit

### Fourth-order low-pass filter



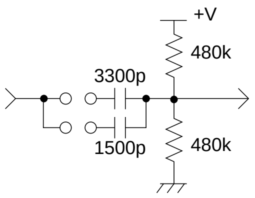
Mounted on the S5U1C330A1D1 board is a microphone low-pass filter with 3.5 kHz cutoff, as shown above. This filter cuts unwanted high-frequency components, improving perceived sound quality. The effect is not dramatic, and the filter may be omitted. Here, the amplitude is halved with a dividing resistor, as matched to the op amp. This is divided by considering the gain of the AC amp in the next stage.

### AC amp



This circuit is a 2-fold to 20-fold AC amp. The 0.01 μF and 470 kΩ comprise the first-order high-pass filter with 70 Hz cutoff, and the 10 kΩ and 1 μF comprise a 15 Hz, first-order high-pass filter, while the 20 kΩ + 100 kΩ (20–120 kΩ) and 220 pF comprise a 50 kHz–10 kHz first-order low-pass filter. If amplification up to high frequencies is desired, reduce the 220 pF. The cutoff frequency increases in inverse proportion to this capacitance.

### High-pass filter

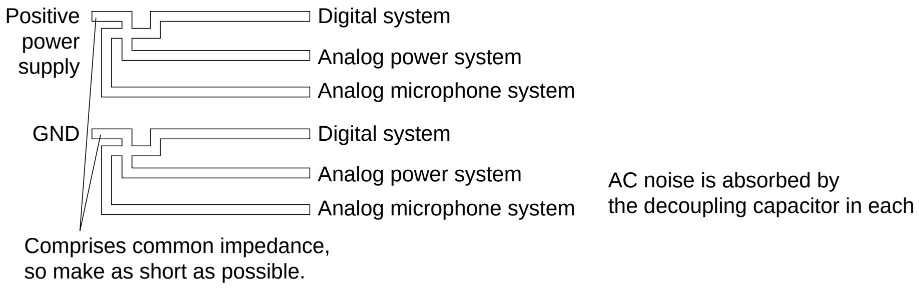


Here, a high-pass filter is used for AC coupling to 1/2 power supply voltage and to cut low tones that adversely affect sound compression. The relationship between capacitor capacitances and cutoff frequencies is shown below.
4800 pF: 250 Hz cutoff
3300 pF: 300 Hz cutoff
1500 pF: 500 Hz cutoff

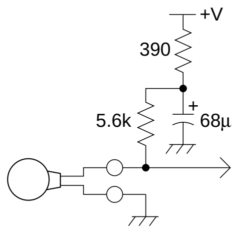
Although the default capacitance for the S5U1C330A1D1 is 4800 pF, other capacitances may be tried, depending on the usage environment. For example, the VSX sound compression included in the S5U1C330V1S sound compression/expansion middleware may yield better results at 500 Hz, since it is susceptible to DC noise.

## ● About the analog power supply

Using the same power supply in both analog and digital systems leaves systems susceptible to noise and other problems. Use dedicated batteries and linear regulators in the analog system, separate from the digital system. Dividing the analog power supply between heavy load blocks (e.g. speaker) and minute voltage blocks (microphone) will prove more effective. The use of multiple regulators is ideal. A simpler alternative, one-point grounding (connecting to GND at one point centering around the power supply), helps eliminate common impedance, which is also beneficial.



Microcomputer programs cause loads to fluctuate periodically, which as power supply fluctuations affect microphone input. To absorb these fluctuations, separate the regulator. Or better, insert a low-pass filter with several Hz to 10 Hz cutoff in the power supply for the electrostatic microphone, as with the S5U1C330A1D1.



Due to their noise, even linear regulators (especially of the low-drop type) affect microphone input. For the sake of safety, we strongly recommend attaching this low-pass filter to the microphone input circuit.

For switching-mode power supplies as used in the S5U1C33209D1, use an OS capacitor with low-ESR or an SP cap for the output capacitor to minimize ripples. Never use electrolytic capacitors; they increase noise. In S5U1C33209D1 + S5U1C330A1D1 systems, noise is suppressed with only the low-pass filter for the microphone power supply, based on various characteristics measurements. However, this solution is imperfect. The AC coupling part and op amp power supply issue remain to be resolved. We recommend using linear regulators, which are less problematic than switching regulators. When using switching regulators, be sure to verify usefulness with the actual product, and take various noise preventive measures.

## *5.6 15-bit D/A Conversion by PWM*

The S1C33209 is able to support 8 kHz to 48 kHz sampling frequencies up to 15-bit precision, thanks to Seiko Epson's exclusive hybrid PWM technology. This makes possible high audio quality approaching CD quality, at extremely low cost.

The hybrid PWM technology is implemented by a combination of the following three techniques:

(1) High-resolution PWM
By controlling PWM output in units of half-clock periods as described in Section 5.3, this technique can produce speech / music output of up to 10-bit precision in a single channel.

(2) Dual PWM
Through a synthesis of two channels of high-resolution PWM, this technique can produce speech / music output with a precision of up to 15 bits.

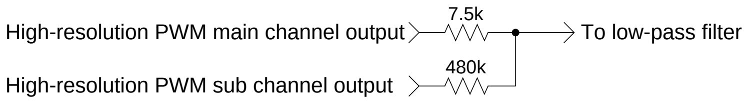
(3) Soft adjust PWM
During PWM output, this technique deploys corrective software processing to produce high-accuracy output, with a linearity error as small as 0.01%.

This section discusses dual PWM and soft-adjust PWM.

### ● Dual PWM

#### Basic principle

Dual PWM is a technique used to extend bit precision by forwarding the same output data from two channels in high-resolution PWM mode, then synthesizing them with external resistors. We recommend synthesizing the main and sub channels at a ratio of 1 to 64, and directly synthesizing raw PWM waveforms before passing them through the low-pass filter.


7.5k
High-resolution PWM main channel output
To low-pass filter
480k
High-resolution PWM sub channel output

480 / 7.5 = 64.0

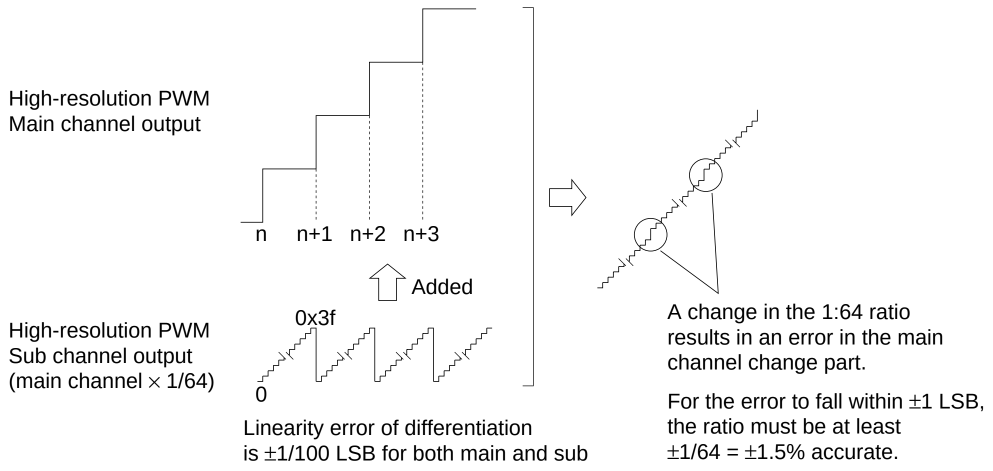
High-resolution PWM provides extremely high differentiation accuracy, with an error of 1 / 100 LSB or less when actually measured. (Use PLL at x2 or better. Using x1 OSC3 directly as is destroys the duty ratio, making it impossible to obtain this level of differentiation accuracy. For 1-channel high-resolution PWM, x1 may be used without problems.)

By adding the sub PWM divided exactly by 64 to the main PWM, we can add a precision of 6 bits to the bit precision of the main channel alone. For the main channel, use a carrier frequency of 160 kHz or higher for noise reduction (320 kHz is the upper limit; do not use any carrier frequency higher than that). As a result, the main channel is 9 bits precise (when operating at 40 MHz or better). Adding 6 sub-channel bits improves overall precision to 15 bits.

#### Resistance accuracy

The accuracy of resistors configuring the 1:64 ratio affects the accuracy of the D / A conversion. If the resistors are exactly 480.0 kΩ and 7.5 kΩ, no problem arise. However, for reasons involving manufacturing cost, the resistors used in mass production have ±1% or ±0.5% errors. In addition, 480 kΩ resistors are difficult to obtain; it is not available in the E24 series. Two resistors, 470 kΩ + 10 kΩ, may be substituted. Most affected by this error is the change part of the main channel. If the sub channel is exactly 1 / 64 of the main channel, the sub channel changes from 0x3f to 0x0 in the main channel's change part. An error in the combined resistance causes this relative position to drift. The differential error in only this part is as follows:

| | |
|---|---|
| Resistor with 0.1% error: | 15 bits ±1 LSB or less |
| Resistor with 0.5% error: | 14 bits ±1 LSB or less |
| Resistor with 1% error: | 13 bits ±0.7 LSB or less |

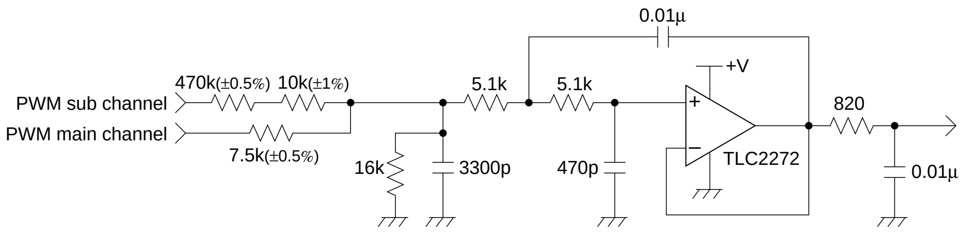

High-resolution PWM
Main channel output
n
n+1
n+2
n+3
Added
High-resolution PWM
Sub channel output
(main channel × 1/64)
0x3f
0
Linearity error of differentiation
is ±1/100 LSB for both main and sub
A change in the 1:64 ratio
results in an error in the main
channel change part.
For the error to fall within ±1 LSB,
the ratio must be at least
±1/64 = ±1.5% accurate.

Since a differentiation accuracy of 15 bits ±0.5 LSB more or less applies to 63/64 patterns in which the sub channel changes to other values, audio quality is not degraded as much by the error. Nevertheless, we recommend using resistors with small error values, about 0.5% accuracy, if possible. At worst, try using resistors with 1% error. Do not use resistors with 5% error values.
The two to three resistors used to combine resistance are the only resistors requiring high accuracy. Resistors with 5% error or so may be used for the low-pass filter in the following stage.

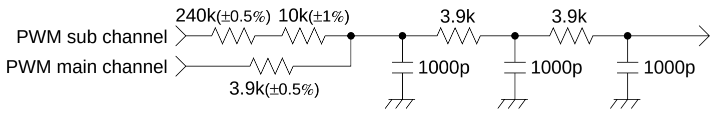
## ● Circuit example (S5U1C330A3D1)

### Low-pass filter for 32 kHz or higher sampling

Fourth-order op amp block



Third-order RC network



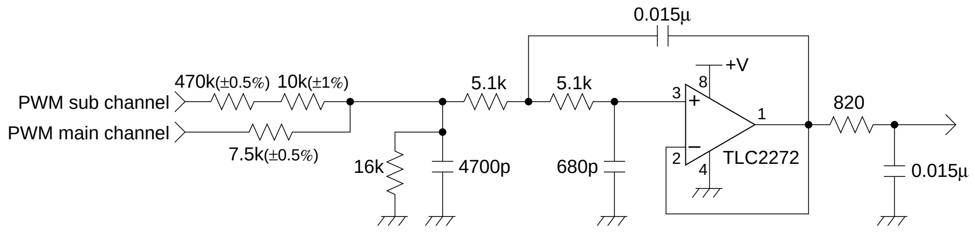
Before the ordinary low-pass filter, add the first-stage synthesizing resistors and connect two-channel PWM outputs. Make sure the ratio of the synthesizing resistors is as close to 64.0-fold as possible (by calculation, within ±0.2% error, from 63.87-fold to 64.13-fold). Use resistance values in the E24 series that are readily available. For difficult to obtain resistance values, use two resistors in pairs as an alternative. Use high-accuracy (0.5% to 1%) resistors for the synthesizing resistors. The resistance values in the above example fall within ±0.2%, as follows:
480 k / 7.5 k = 64.0 (480k = 470 k + 10 k)
250 k / 3.9 k = 64.10 (250k = 240 k + 10 k)

With an emphasis on the attenuation factor, the RC filter is stacked three-high. Although the difference is infinitesimal for 32 kHz sampling, a fourth-order filter using an op amp is more effective.

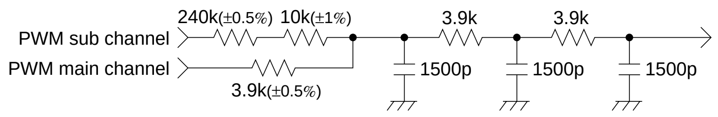
For the circuits shown below, capacitor values have been changed to adjust the cutoff frequency, making the circuits useful for 22.05 kHz sampling and 16 kHz sampling, respectively. In either case, the ratio of the first-stage synthesizing resistors is 1:64.

### Low-pass filter for 22.05 kHz sampling

Fourth-order op amp method

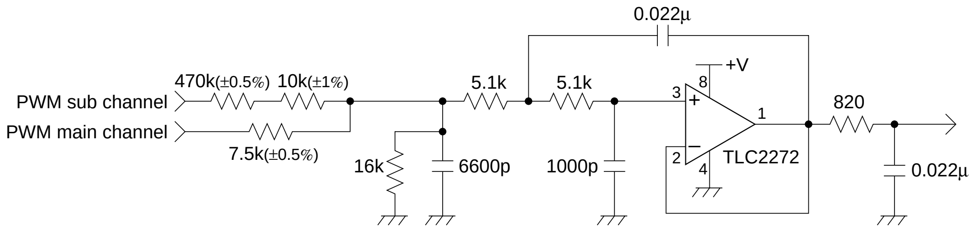


Third-order RC network method

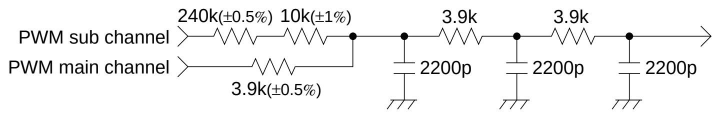


### Low-pass filter for 16 kHz sampling

Fourth-order op amp method

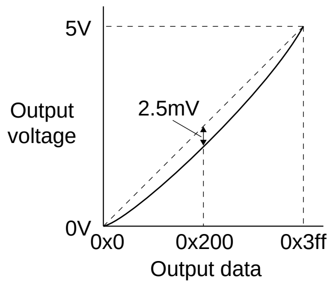


Third-order RC network method



## ● Linearity correction by software

High-resolution PWM technology offers a differentiation accuracy of 1 / 100 LSB or better (actual measured value), which may be said to approach ultimate accuracy. The linearity error is relatively good, with bowl-shaped characteristics. This is because PWM outputs have minute differences in impedance between high and low levels. If the difference between the low-pass filter's first-stage resistance and the S1C chip's internal equivalent resistance is known, the drift can be theoretically calculated. For example, if the first-stage resistance is 3.9 kΩ when the PWM output voltages are 0.0 V and 5.0 V, the middle part of the output curve deflects 2.5 mV downward. The deflection is 1.3 mV for 7.5 kΩ, and 25 mV for 390 Ω.



This deflection is corrected using a table like the one (for 3.9 kΩ) shown below.

Table example

```
const unsigned char ucAdj18 [] = {     // PWM adjust for 3.9Kohm with 18bit precision
      0x4,     // 0
      0x8,     // 1
      0xc,     // 2
      0x10,    // 3
      0x14,    // 4
      0x17,    // 5
      0x1b,    // 6
      0x1f,    // 7
      0x22,    // 8
      0x26,    // 9
      0x29,    // a
      0x2d,    // b
      0x30,    // c
      0x33,    // d
      0x36,    // e
      0x39,    // f
      0x3c,    // 10
      0x3f,    // 11
      0x42,    // 12
      0x45,    // 13
      0x48,    // 14
      0x4b,    // 15
      0x4d,    // 16
      0x50,    // 17
      0x52,    // 18
      0x55,    // 19
      0x57,    // 1a
      0x5a,    // 1b
      0x5c,    // 1c
      0x5e,    // 1d
      0x60,    // 1e
      0x62,    // 1f
      0x64,    // 20
      0x66,    // 21
      0x68,    // 22
      0x6a,    // 23
      0x6c,    // 24
      0x6d,    // 25
      0x6f,    // 26
      0x71,    // 27
      0x72,    // 28
      0x74,    // 29
      0x75,    // 2a
      0x76,    // 2b
      0x77,    // 2c
      0x79,    // 2d
      0x7a,    // 2e
      0x7b,    // 2f
      0x7c,    // 30
      0x7d,    // 31
      0x7e,    // 32
      0x7e,    // 33
      0x7f,    // 34
      0x80,    // 35
      0x80,    // 36
      0x81,    // 37
      0x81,    // 38
      0x82,    // 39
      0x82,    // 3a
      0x83,    // 3b
      0x83,    // 3c
      0x83,    // 3d
      0x83,    // 3e
      0x83,    // 3f
      0x83,    // 40
      0x83,    // 41
      0x83,    // 42
      0x83,    // 43
      0x82,    // 44
      0x82,    // 45
      0x82,    // 46
      0x81,    // 47
      0x80,    // 48
      0x80,    // 49
      0x7f,    // 4a
      0x7e,    // 4b
      0x7e,    // 4c
      0x7d,    // 4d
      0x7c,    // 4e
      0x7b,    // 4f
      0x7a,    // 50
      0x79,    // 51
      0x78,    // 52
      0x76,    // 53
      0x75,    // 54
      0x74,    // 55
      0x72,    // 56
      0x71,    // 57
      0x6f,    // 58
      0x6d,    // 59
      0x6c,    // 5a
      0x6a,    // 5b
      0x68,    // 5c
      0x66,    // 5d
      0x64,    // 5e
      0x62,    // 5f
      0x60,    // 60
      0x5e,    // 61
      0x5c,    // 62
      0x5a,    // 63
      0x57,    // 64
      0x55,    // 65
      0x52,    // 66
      0x50,    // 67
      0x4d,    // 68
      0x4b,    // 69
      0x48,    // 6a
      0x45,    // 6b
      0x42,    // 6c
      0x3f,    // 6d
      0x3c,    // 6e
      0x39,    // 6f
      0x36,    // 70
      0x33,    // 71
      0x30,    // 72
      0x2d,    // 73
      0x29,    // 74
      0x26,    // 75
      0x22,    // 76
      0x1f,    // 77
      0x1b,    // 78
      0x17,    // 79
      0x14,    // 7a
      0x10,    // 7b
      0xc,     // 7c
      0x8,     // 7d
      0x4,     // 7e
      0x0,     // 7f
};
```

The values in this table have been created as 18-bit precision data by subtracting correction values from 7 high-order bits, so that the values are ultimately added after right-shifting three bits before use for correction. By this correction, the linearity error can be suppressed to about ±0.2 mV on average, or down to about ±1 mV even for large errors. An error of ±1 mV is equivalent to 12 bits ±1 LSB for 5 V.

Unless corrected, the error appears in the waveform as distortion. But errors of up to about 2.5 mV produce no perceptible differences to human ears, and generally does not require correction. In speech middleware, corrective processing is omitted to alleviate software burdens.

## *5.7 Melody Output using a Piezoelectric Buzzer*

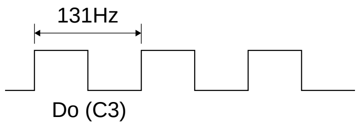
In this section, we discuss producing melody output using PWM and connecting a piezoelectric buzzer.

### ● PWM and melody

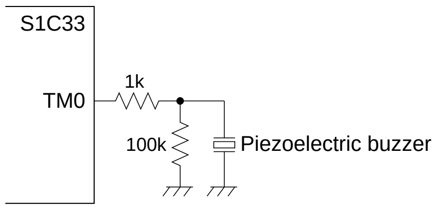
Human ears can discriminate tone on the musical scale by sound frequency. For example, a 131 Hz tone is heard as do (C3). A 262 Hz tone is heard as a do (C4) one-octave higher, while a 65.5 Hz tone is heard as a do (C2) one-octave lower. When one octave (up to 2-fold frequency) is equally divided by 12, with frequency increased by about 6% for each, musical intervals are recognized as being raised by a halftone at a time. The musical scale is expressed in this way.

Seiko Epson's S5U1C331M2S middleware and general melody ICs use PWM (square) waveforms to express these tones. Note that waveforms with perfect 50% duty cycles bear three-fold harmonics, such as 3 times and 9 times the fundamental frequency, providing fairly extensive high-pitched components in addition to the actual musical scale.

### ● 1-channel output

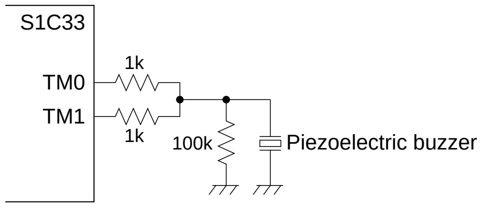


The output waveform of 131 Hz produces a sound corresponding to do (C).



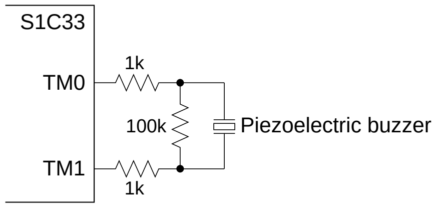
One-channel output drives a piezoelectric buzzer, as shown here.

### ● 2-channel synthesis output



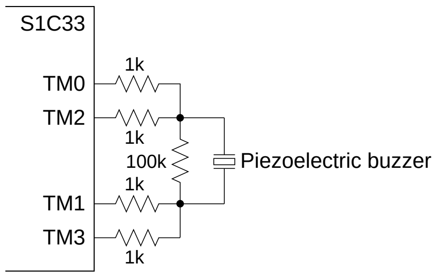
Two or more channels can be synthesized, as shown here.

### ● Differential output



Sound volume can be increased through differential output, using inverted PWM on one channel.

### ● Differential output, 2-channel synthesis output

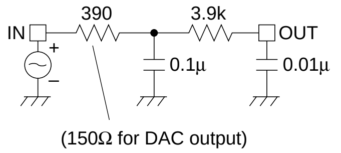


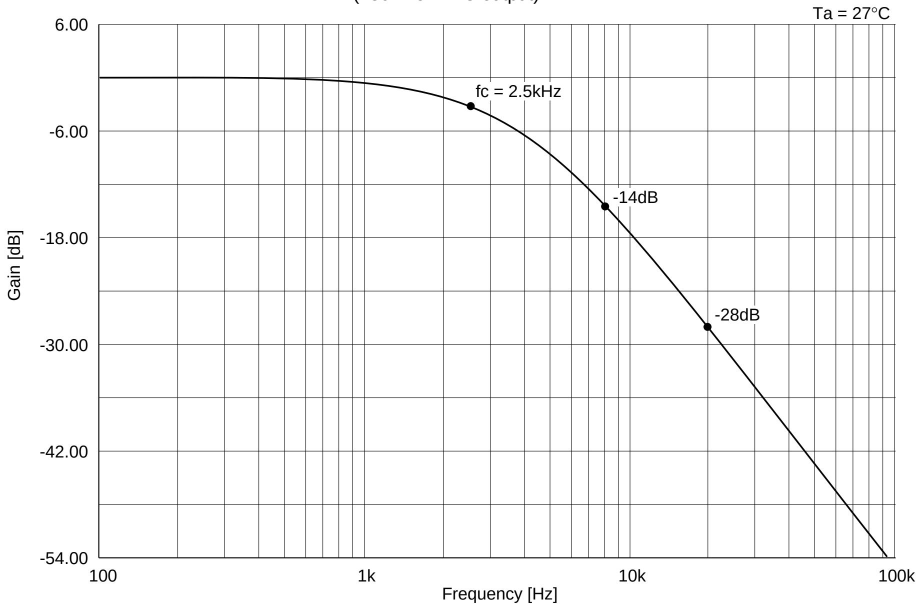
Two-channel synthesis and differential output can be used in combination using two differential outputs.

## *5.8 <Reference Data> Characteristic Graphs*

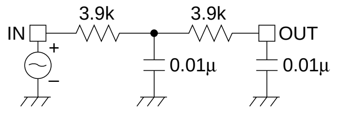
### ● RC second-order low-pass filter frequency response (for 8 kHz sampling)

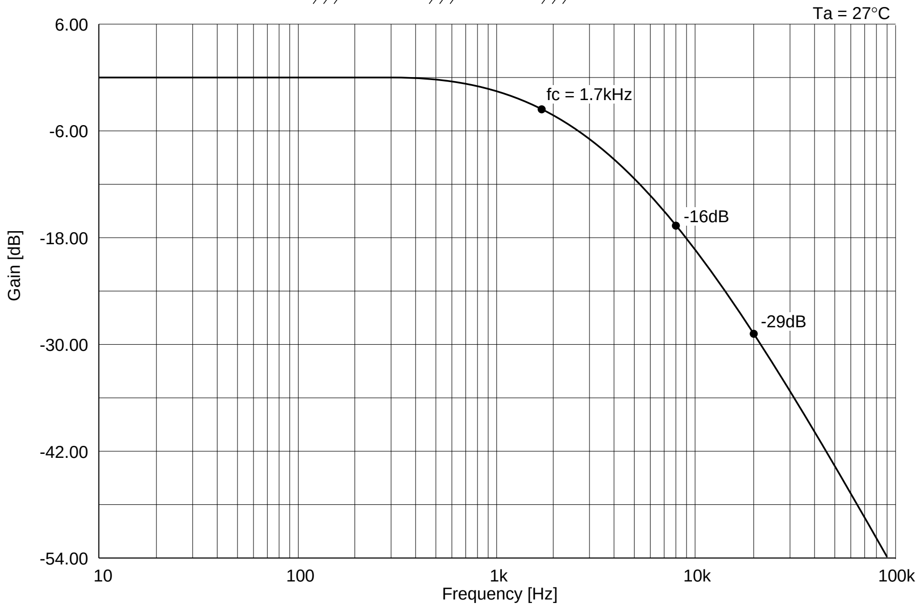
#### (1) fc = 2.5 kHz





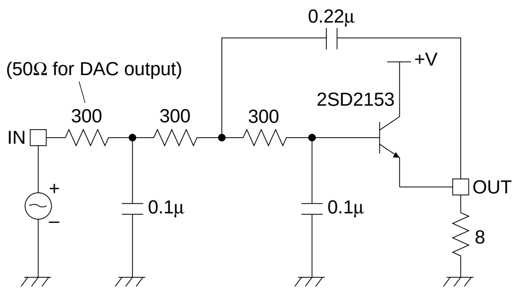
#### (2) fc = 1.7 kHz

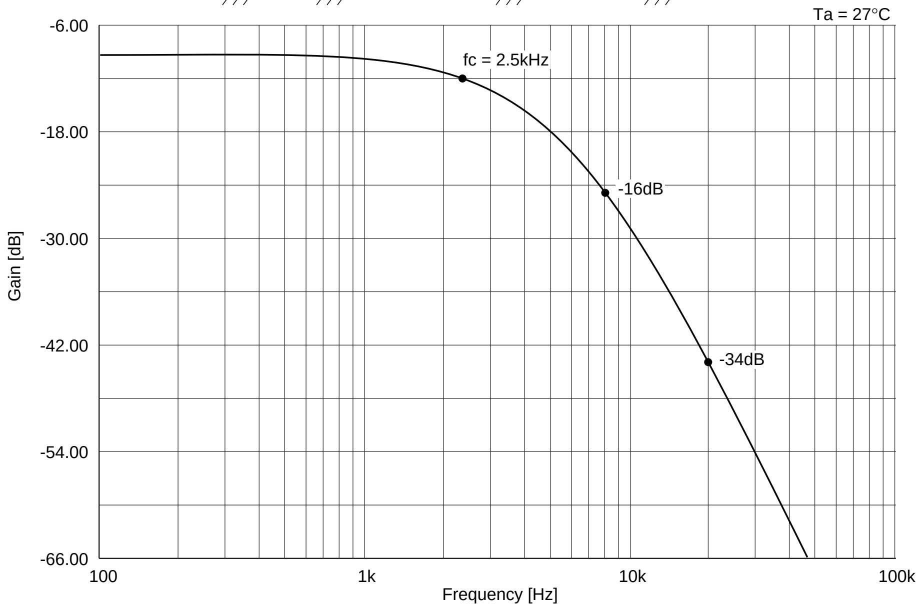




## ● Transistor third-order low-pass filter frequency response (for 8 kHz sampling)

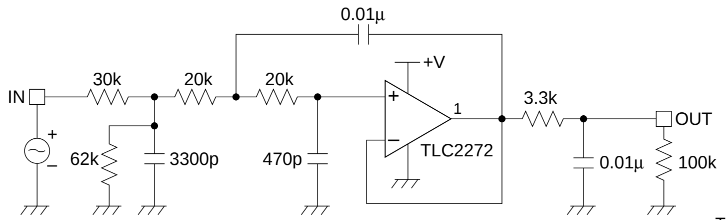
### fc = 2.5 kHz

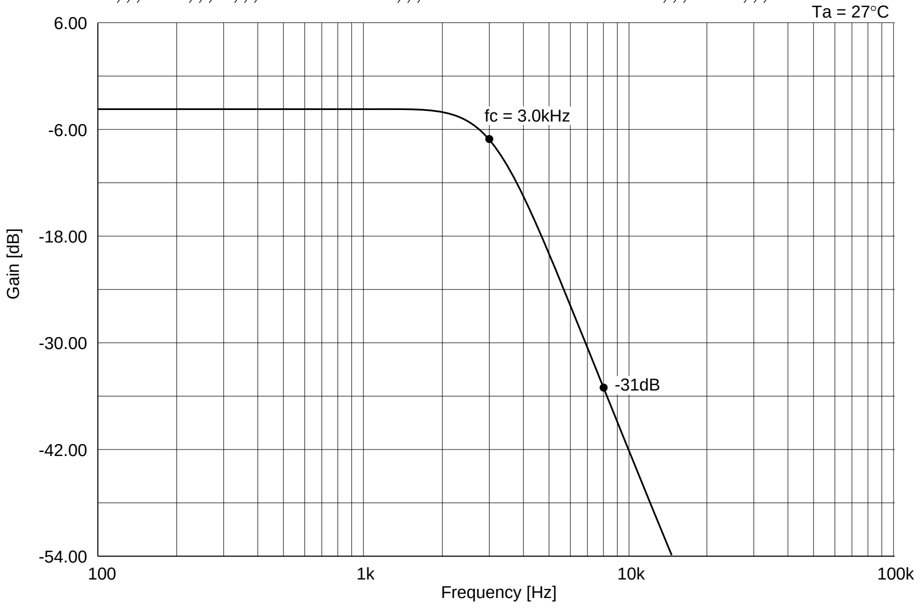




## ● Op amp fourth-order low-pass filter frequency response (for 8 kHz sampling)

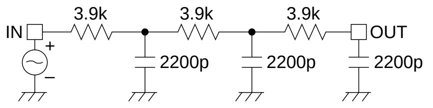
**fc = 3 kHz**

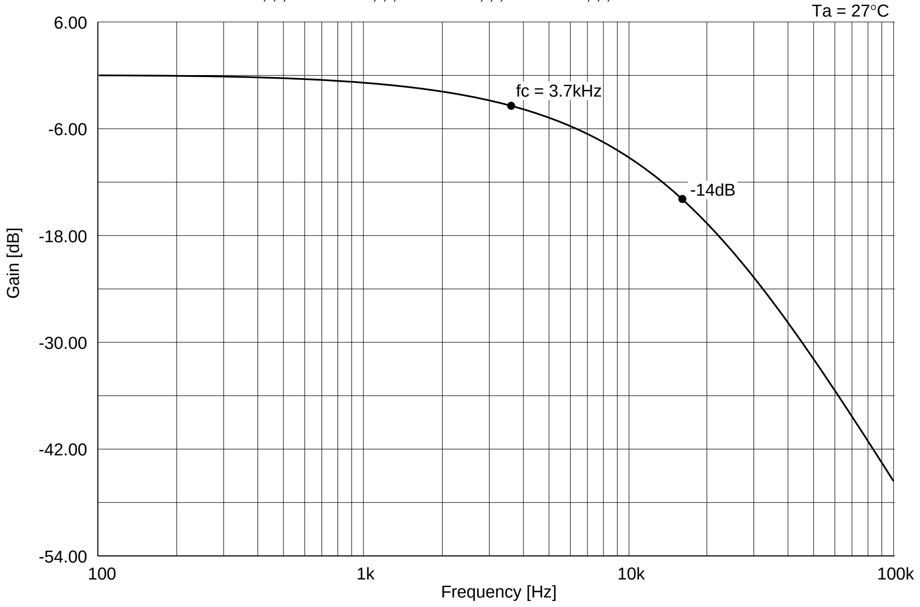




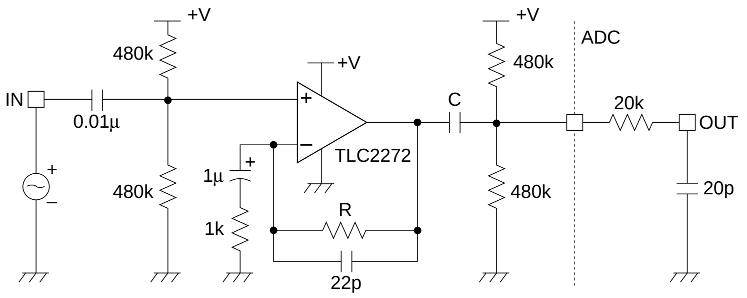
## ● RC third-order low-pass filter frequency response (for 16 kHz sampling)

**fc = 3.7 kHz**



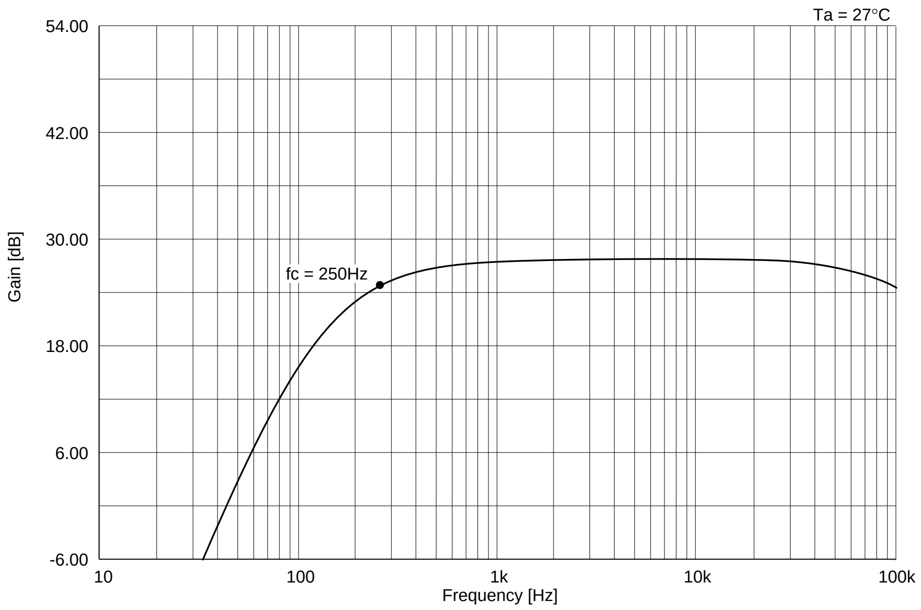


## ● AC amp high-pass filter frequency response (for 8 kHz sampling)

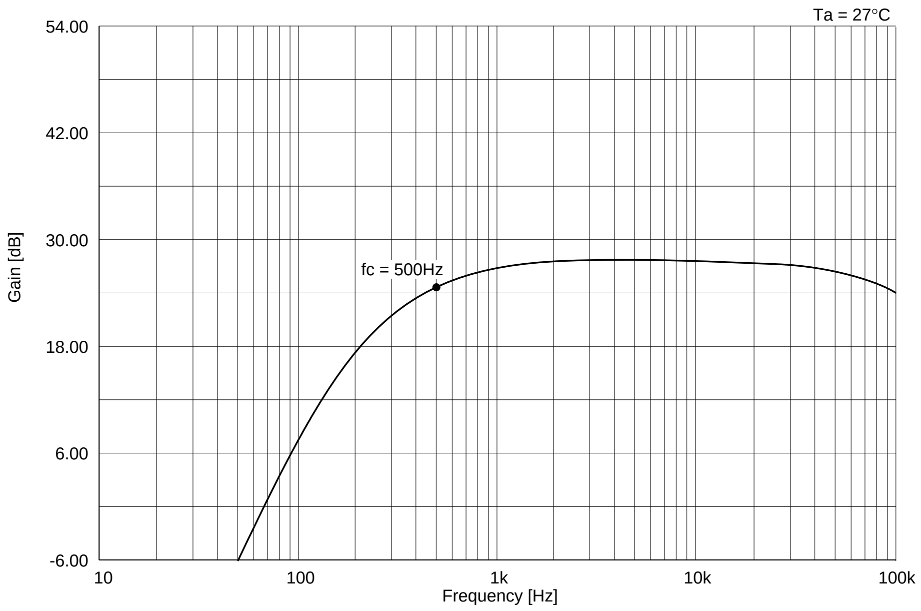


(1) R = 24kΩ, C = 0.0048µF (fc = 250Hz)
(2) R = 24kΩ, C = 0.0015µF (fc = 500Hz)

### (1) fc = 250 Hz



### (2) fc = 500 Hz



EPSON **International Sales Operations**

## AMERICA

**EPSON ELECTRONICS AMERICA, INC.**

**- HEADQUARTERS -**
150 River Oaks Parkway
San Jose, CA 95134, U.S.A.
Phone: +1-408-922-0200 Fax: +1-408-922-0238

**- SALES OFFICES -**

**West**
1960 E. Grand Avenue
El Segundo, CA 90245, U.S.A.
Phone: +1-310-955-5300 Fax: +1-310-955-5400

**Central**
101 Virginia Street, Suite 290
Crystal Lake, IL 60014, U.S.A.
Phone: +1-815-455-7630 Fax: +1-815-455-7633

**Northeast**
301 Edgewater Place, Suite 120
Wakefield, MA 01880, U.S.A.
Phone: +1-781-246-3600 Fax: +1-781-246-5443

**Southeast**
3010 Royal Blvd. South, Suite 170
Alpharetta, GA 30005, U.S.A.
Phone: +1-877-EEA-0020 Fax: +1-770-777-2637

## EUROPE

**EPSON EUROPE ELECTRONICS GmbH**

**- HEADQUARTERS -**
Riesstrasse 15
80992 Munich, GERMANY
Phone: +49-(0)89-14005-0 Fax: +49-(0)89-14005-110

**DÜSSELDORF BRANCH OFFICE**
Altstadtstrasse 176
51379 Leverkusen, GERMANY
Phone: +49-(0)2171-5045-0 Fax: +49-(0)2171-5045-10

**UK & IRELAND BRANCH OFFICE**
Unit 2.4, Doncastle House, Doncastle Road
Bracknell, Berkshire RG12 8PE, ENGLAND
Phone: +44-(0)1344-381700 Fax: +44-(0)1344-381701

**FRENCH BRANCH OFFICE**
1 Avenue de l' Atlantique, LP 915 Les Conquerants
Z.A. de Courtaboeuf 2, F-91976 Les Ulis Cedex, FRANCE
Phone: +33-(0)1-64862350 Fax: +33-(0)1-64862355

**BARCELONA BRANCH OFFICE**
**Barcelona Design Center**
Edificio Testa, Avda. Alcalde Barrils num. 64-68
E-08190 Sant Cugat del Vallès, SPAIN
Phone: +34-93-544-2490 Fax: +34-93-544-2491

**Scotland Design Center**
Integration House, The Alba Campus
Livingston West Lothian, EH54 7EG, SCOTLAND
Phone: +44-1506-605040 Fax: +44-1506-605041

## ASIA

**EPSON (CHINA) CO., LTD.**
23F, Beijing Silver Tower 2# North RD DongSanHuan
ChaoYang District, Beijing, CHINA
Phone: 64106655 Fax: 64107319

**SHANGHAI BRANCH**
7F, High-Tech Bldg., 900, Yishan Road
Shanghai 200233, CHINA
Phone: 86-21-5423-5577 Fax: 86-21-5423-4677

**EPSON HONG KONG LTD.**
20/F., Harbour Centre, 25 Harbour Road
Wanchai, Hong Kong
Phone: +852-2585-4600 Fax: +852-2827-4346
Telex: 65542 EPSCO HX

**EPSON TAIWAN TECHNOLOGY & TRADING LTD.**
14F, No. 7, Song Ren Road, Taipei 110
Phone: 02-8786-6688 Fax: 02-8786-6660

**HSINCHU OFFICE**
No. 99, Jiangong Rd., Hsinchu City 300
Phone: +886-3-573-9900 Fax: +886-3-573-9169

**EPSON SINGAPORE PTE., LTD.**
No. 1 Temasek Avenue, #36-00
Millenia Tower, SINGAPORE 039192
Phone: +65-6337-7911 Fax: +65-6334-2716

**SEIKO EPSON CORPORATION KOREA OFFICE**
50F, KLI 63 Bldg., 60 Yoido-dong
Youngdeungpo-Ku, Seoul, 150-763, KOREA
Phone: 02-784-6027 Fax: 02-767-3677

**GUMI OFFICE**
6F, Good Morning Securities Bldg.
56 Songjeong-Dong, Gumi-City, 730-090, KOREA
Phone: 054-454-6027 Fax: 054-454-6093

**SEIKO EPSON CORPORATION**
**ELECTRONIC DEVICES MARKETING DIVISION**

**ED International Marketing Department**
421-8, Hino, Hino-shi, Tokyo 191-8501, JAPAN
Phone: +81-(0)42-587-5814 Fax: +81-(0)42-587-5117

**S1C33 Family**
Application Note for Standard Core
(S5U1C33001C)

**SEIKO EPSON CORPORATION**
ELECTRONIC DEVICES MARKETING DIVISION

■ **EPSON Electronic Devices Website**

http://www.epsondevice.com

**Document code: 404862100**
**Issue February, 2004**
**Printed in Japan Ⓛ A**