MF1313-01

EPSON

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

APPLICATION NOTES

ENERGY SAVING EPSON

SEIKO EPSON CORPORATION

PREFACE

Written for developers of application systems incorporating the E0C33 Family of microcomputers, this manual explains how to write a program, design basic circuitry, and produce audio output using the E0C33 chips, particularly the E0C33208. The sample code provided in this manual is excerpted from E0C33 Family C Compiler Package Ver. 2 or later.

CONTENTS

1 ABOUT THE E0C33000 CPU CORE

The E0C33000 is the CPU core shared by all chips in the E0C33 Family of 32-bit CMOS single-chip microcomputers. 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 E0C33 Family processors.

The main features of the E0C33000 are as follows.

- A highly code-efficient instr uction set
- Fast operation and multiplier/accumulator function
- Small CPU core size
- Low curr ent consumption

The E0C33000 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 E0C33XXX)
- Instruction set ..... 16-bit fixed length
  105 discrete instructions
  Main instructions can be executed in one cycle.
- Multiplier/accumulator function .... MAC instruction (16 bits × 16 bits + 64 bits → 64 bits)
  Executed in 2 cycles per MAC operation
- Register set ..... 32-bit general-purpose register × 16
  32-bit special register × 5
- Memory space ..... 28-bit (256 MB) space
  Instruction, data, and I/O mixed type linear space
  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 × 216 sources
  Software exception × 4 sources, 2 types of instruction execution exception
  Vectors are fetched from trap table when branching to the jump address.
- Reset ..... Cold reset (all internal circuits reset)
  Hot reset (buses not reset)
  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)
  SLP instruction (all internal circuits halted)
- Other ..... Little endian (standard)/ big endian
  Harvard architecture

1.2 Memory Map

			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	929
:	
External maskable interrupt 0	64
Software exception 3	60
:	
Software exception 0	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.

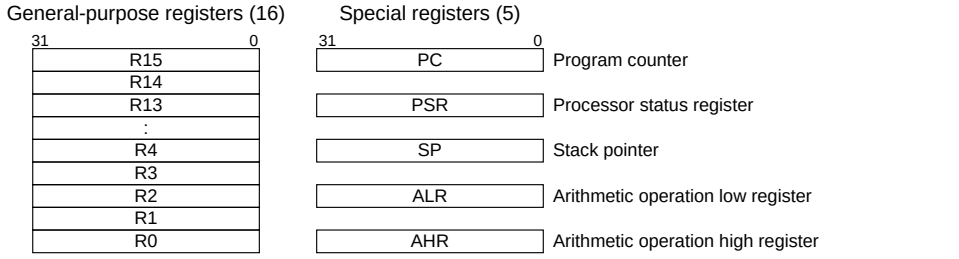
Interrupt sequence

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



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  sign21
EXT  imm13 + EXT  imm13 + CALL  sign8   → CALL  sign31
```

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

- **Extended 3 operand mode**

```
EXT  imm13 + ADD  %R1,%R2               → ADD  %R1,%R2,imm13
EXT  imm13 + EXT  imm13 + ADD  %R1,%R2  → ADD  %R1,%R2,imm26
```

The instruction Reg1 ← Reg1 OP Reg2 is changed to a 3-operand instruction Reg1 ← Reg2 OP imm13/26 by the EXT instruction.

● High code density for C language

Based on the following two concepts, the E0C33 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 sign21` (±1M jump area)
Supports other conditional branching with two instructions (4 bytes).

- **Subroutine call**

`EXT imm13 + CALL sign8 → CALL sign21` (±1M jump area)
Supports almost all subroutine calls with two instructions (4 bytes).
`EXT imm13 + EXT imm13 + JRNE sign9 → JRNE sign31` (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,[%R8] → LD.H %R2,[%R8+imm13]` (Accesses 4K-byte area from R8)
Two instructions (4 bytes)
```
EXT imm13 + EXT imm13 + LD.H %R2,[%R8] → LD.W %R2,[%SP+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)

- **3-operand, register to immediate**

`EXT imm13 + ADD %R2,[%R3] → ADD %R2,%R3,imm13` (R2 = R3 + imm13)
Two instructions (4 bytes)
`EXT imm13 + EXT imm13 + ADD %R2,imm6 → ADD %R2,%R3,imm26` (R2 = R3 + imm26)
Three instructions (6 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

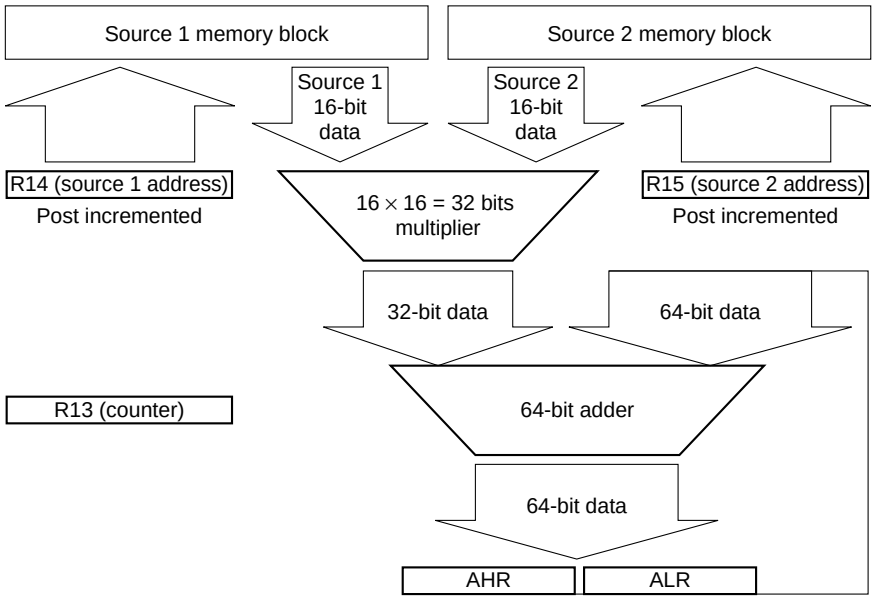
  CALL, JP, RET, RETI, RETD, INT, BRK
  Two to 10 cycles per instruction

- **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 %R13**
R13: Repetition counter (maximum 4 G)
R14: Source 1 address (post incremented)
R15: 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) 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) Execute next instruction upon branching	1 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, retb		Return, interrupt, etc.	3–10
Logic operation	and, or, xor, not	and %rd, %rs and %rd, sign6	%rd = %rd & %rs %rd = %rd & sign6	1

Arithmetic operation	add, sub	add %rd, %rs
		add %rd, imm6
		add %sp, imm12
			%rd = %rd + %rs
			%rd = %rd + imm6
			%sp = %sp + imm12
				1

Compare operation	cmp	cmp %rd, %rs cmp %rd, sign6	%rd - %rs, flag only changes %rd - sign6	1
Carry operation	adc, sbc	adc %rd, %rs	%rd = %rd + %rs + carry flag	1
Multiplication	mlt.h, mlt.uh (16bit) mlt.w, mlt.uw (32bit)	mlt.h %rd, %rs	%alr = %rd × %rs (32 = 16 × 16) %ahr:%alr = %rd × %rs (64 = 32 × 32)	1 5
Division	div0s, div0u, div1, div2s, div3s		Execute division using these in combination	1
Shift	srl, sll (logical shift) sra, sla (arithmetical shift) rr, rl (rotate)	srl %rd, imm4 srl %rd, %rs	%rd = %rd >> imm4 %rd = %rd >> %rs Shift by 0 to 8 bits	1
Memory load	ld.b (signed 8bit load) ld.ub (unsigned 8bit load) ld.h (signed 16bit load) ld.uh (unsigned 16bit load) ld.w (32bit load)	ld.w %rd, [%sp+imm6] ld.w [%sp+imm6], %rs ld.w %rd, [%rb] ld.w %rd, [%rb]+ ld.w [%rb], %rs ld.w [%rb]+, %rs	%rd = [%sp+imm6], stack relative access [%sp+imm6] = %rs %rd = [%rb], register address access %rd = [%rb], %rb = %rb + 4, post inc. [%rb] = %rs [%rb] = %rs, %rb = %rb + 4	1–2
Register load	ld.w	ld.w %rd, %rs ld.w %rd, sign6 ld.w %rd, %ss ld.w %ss, %rs	Copy between registers Store immediate value Copy from special register 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 call sign8 (= call sign21)	ext imm13 ext imm13 call sign8 (= call sign31)

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 sign21	jp sign31
3-operand operation	add, sub, and, or, xor, not, cmp	add %rd, %rs	add %rd, %rs, imm13 3-operand operation	add %rd, %rs, imm26 3-operand operation
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] ld.w [%sp+imm6], %rs	[%sp+imm19] Extend offset value	[%sp+imm32] Extend offset value
Absolute load	ld.b, ld.ub, ld.h, ld.uh, ld.w	ld.w %rd, [%rb] ld.w %rd, [%rb]+ ld.w [%rb], %rs ld.w [%rb]+, %rs	[%rb+imm13] Add offset value	[%rb+imm28] Add offset value
Bit operation	btst, bset, bclr, bnot	btst [%rb], imm3	[%rb+imm13] Add offset value	[%rb+imm26] Add offset value

signX, immX: immediate value, %XX: register

2 WRITING PROGRAMS FOR THE E0C33

This chapter explains how to write programs for the E0C33. The method described here applies to all microcomputers in the E0C33 Family.

2.1 Vector Table and Boot Routine

The E0C33 program must have at least a vector table and a boot routine. When cold reset at power-on, the E0C33 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.

```
        .abs                        ; Directive command located beginning with 0xC00000
        .org     0xc000000
        .code

        .word    BOOT               ; Vector table (consisting of only one boot line)

BOOT:
        xld.w    %r8,0x800          ; Boot program
        ld.w     %sp,%r8            ; Sets SP and calls main
        xcall    main
```

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 code is included in cc33\sample\drv33a104\.

Vector table [drv33a104\16timer\vector.s]

```
        .code

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

.word 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 E0C33, these memory accesses are prohibited.

Redefinition of interrupt vectors [drv33a104\16timer\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 [drv33a104\common\boot.s]

```
/*****************************************************************************/
/*                                                                           */
/*      Copyright (C) SEIKO EPSON CORP. 1997 - 1998                          */
/*                                                                           */
/*      File name: vector.h                                                  */
/*      This is interrupt vector header file.                                */
/*                                                                           */
/*****************************************************************************/

#include       "..\bcu\drv_bcu.h"

;; Stack top
#define SP_TOP          0x07ff   ; sp is in end of 2KB internal RAM

;; Global pointer
#define GP_ADDR         0x0000   ; global pointer is 0x0

      .code
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; BOOT
;;   Type:     void
;;   Ret val:  none
;;   Argument: void
;;   Function: Boot program.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      .global  BOOT
BOOT:
      xld.w    %r8,SP_TOP        ; Set stack pointer
      ld.w     %sp,%r8
      xld.w    %r8,GP_ADDR       ; Set global pointer
      xcall    _init_bcu         ; Initialize BCU on boot time in bcu.s
      xcall    _init_sys         ; Call _init_sys() in sys.c to use simulated I/O
      xcall    main              ; Go to main
      xcall    _exit             ; In last, go to _exit in sys.c to use simulated I/O

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; _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    %r5,BCU_A10_ADDR
      xld.w    %r4,BCU_BW_0|BCU_DRAH_NOT|BCU_DRAL_NOT|BCU_SZL_16|BCU_DFL_15|BCU_WTL_2
      ld.h     [%r5],%r4
      ret
```

This boot routine (BOOT) initializes the stack pointer, the R8 (used in ext33 for global pointer-based optimization), 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 cc33\sample\drv33208\.

Vector table, boot routine [drv33208\16timer\vector.c]

```
/*****************************************************************************
 *                                                                           *
 *      Copyright (C) SEIKO EPSON CORP. 1999                                 *
 *                                                                           *
 *      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 %r8,0x2000");   // Set SP in end of 8KB internal RAM
      asm("ld.w  %sp,%r8");
      asm("ld.w  %r8,0b10000");  // Set PSR to interrupt enable
      asm("ld.w  %psr,%r8");
      asm("ld.w  %r8,0x0");      // Set GPR 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

This section describes interrupt handling routines, in particular methods for saving and restoring the registers. Other routines are written like other ordinary processing routines.

● Routine written in assembler

Example for handling 16-bit timer interrupts [Excerpt from cc33\sample\drv33a104\16timer\demo_16tint.s]

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; int_16timer_u00
;;   Type :     void
;;   Ret val :  none
;;   Argument : void
;;   Function : 16-bit timer 00 underflow interrupt function.
;;                Read 16-bit timer 3 counter data and stop 16-bit timer 00.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      .global   int_16timer_u00
int_16timer_u00:
      pushn     %r15
      xld.w     %r12,T16P_TC30_ADDR ; %r12 <- 16-bit timer 3 counter data reg. addr
      xcall     read_16timer_cnt    ; %r10 <- 16-bit timer 3 counter data
      xld.w     [timer00],%r10

      xld.w     %r12,T16P_PRUN00_ADDR
      xcall     stop_16timer        ; %r12 <- 16-bit timer 00 run/stop register addr

      ld.w      %r4,0x01            ; 16-bit timer 00 interrupt flag on
      xld.w     [t16int00_flg],%r4

      xld.w     %r5,INT_F16T0_F16T1_ADDR ; %r5 <- Interrupt factor register address
      xld.w     %r4,INT_F16TU00     ; Reset 16-bit timer 00 underflow int.factor flag
      ld.b      [%r5],%r4
      popn      %r15
      reti
```

The start label of this routine (int_16timer_u00) is defined as a 16-bit timer interrupt vector (16-bit timer 00 underflow interrupt). When this interrupt occurs, the CPU saves the PC and PSR to the stack before executing this routine. Start by saving all general-purpose registers to the stack using pushn %r15. Then write the required processing code. Finally, restore the contents of the stack to the general-purpose registers using popn %r15, and return to the location where the interrupt occurred using reti. To return from the interrupt handling routine, you must use the reti instruction, which restores the PSR and PC to their states immediately before the interrupt occurrence.

Since multiply/divide operations or MAC operation are unnecessary in this example, the AHR and ALR registers will never be modified within the routine. But if you use the AHR and ALR registers, always save the contents of these registers to the stack, along with those of general-purpose registers, as shown below.

```
      pushn     %r15
      ld.w      %r0,%ahr
      ld.w      %r1,%alr
      pushn     %r1
           |
      popn      %r1
      ld.w      %ahr,%r0
      ld.w      %alr,%r1
      popn      %r15
      reti
```

Conversely, if register use is limited, there is no need to save all general-purpose registers. For example, if you are using only R0 to R3, specify R3 in the pushn and popn instructions. Limiting the registers to be saved helps reduce time and the stack area required for the save.

```
      pushn %r3
           |
      popn %r3
      reti
```

● Routine written in C

Example for handling 16-bit timer interrupts [Excerpt from cc33\sample\drv33208\16timer\demo_16tint.c]

```
/*****************************************************************************
 * int_16timer_c0
 *   Type :     void
 *   Ret val :  none
 *   Argument : void
 *   Function : 16bit timer0 comparison match A interrupt function.
 *****************************************************************************/
void int_16timer_c0(void)
{
      extern volatile int timer0;// Timer counter variable for 16bit timer0

      INT_BEGIN;
      timer0 = read_16timer_cnt(T16P_TC3_ADDR);
      *(volatile unsigned char *)INT_F16T0_F16T1_ADDR = INT_F16TC0;
                        // Reset 16bit timer0 comparison match A interrupt factor flag
      stop_16timer(T16P_PRUN0_ADDR);
      INT_END;
}
```

The respective processing for saving and restoring the registers is defined in INT-BEGIN and INT_END, as follows:
(Excerpt from drv33208\include\common.h)

```
/* Macro */
#define INT_BEGIN   asm("pushn  %r15")
#define INT_END     asm("popn   %r15\n   reti")
```

Here, all general-purpose registers are saved and restored as in the example written in assembler.

Save all general-purpose registers with C, since you do not know which registers will be used. You may also need to save the AHR and ALR registers. In C, multiplication operations are used to calculate addresses for array processing, not just for multiply/divide operations.

Variables in C are sometimes saved to the stack using the pushn instruction. In this case, the preceding example may operate erratically, since the stack pointer loses consistency. This is because the preceding example contains popn and ret after reti, which means popn will not be executed. For this processing, use the sed.exe file provided as a utility. The following example illustrates this, using cc33\sample\int_c as an example.
(int.mak)

```
                |
int.ms : $(SRC_DIR)int.c
      $(GCC33) $(GCC33_FLAG) $(SRC_DIR)int.c
      $(SED) -f int.sed int.ps > int.ps2          ← Filtering by sed
      $(EXT33) $(EXT33_FLAG) int.ps2
int.o : int.ms
      $(AS33) $(AS33_FLAG) int.ms
                |
```

Here, int.c is filtered by sed after being compiled.
(Filter definition in int.sed)

```
s/    pushn.*/;/
s/    popn.*/;/
s/;   \.frame.*/;/
s/;   \.mask.*/;/
s/;   \.fmask.*/        pushn    %r15\
      ld.w     %r0,%ahr\
      ld.w     %r1,%alr\
      ld.w     %r2,%sp\
      pushn    %r2/
s/ret/popn     %r2\
      ld.w     %ahr,%r0\
      ld.w     %alr,%r1\
      ld.w     %sp,%r2\
      popn     %r15\
      reti/
```

For example, the SED processing result of the div0() function in int.c is like the one shown below.

```
void div0()
      {
                int_num = 4;
      }
                ↓
                                                00060  void div0()
 00080068 020F pushn   %r15                     00061       {
 0008006A A430 ld.w    %r0,%ahr
 0008006C A421 ld.w    %r1,%alr
 0008006E A412 ld.w    %r2,%sp
 00080070 0202 pushn   %r2
 00080072 6C4A ld.w    %r10,0x4                 00062             int_num = 4;
 00080074 C000 ext     0x0
 00080076 C000 ext     0x0
 00080078 3C8A ld.w    [%r8],%r10
 0008007A 0242 popn    %r2                      00063       }
 0008007C A003 ld.w    %ahr,%r0
 0008007E A012 ld.w    %alr,%r1
 00080080 A021 ld.w    %sp,%r2
 00080082 024F popn    %r15
 00080084 04C0 reti
```

In this way, the code required to save and restore the general-purpose registers R0–R15 and other registers AHR, ALR, and SP is added before and after function processing, with the reti instruction used for return.

Note: Files processed by sed may not have any function written in them other than exception/interrupt handling.

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

cc33\utility\lib_src\ansilib33\string\src\strcpy.s

```
;*************************************************
; strcpy
;     string copy from src to dest until 0 terminate
;
; arguments : %r12:dest addr, %r13:src addr (0 terminate string)
; return    : %r10:dest addr
;*************************************************

.global strcpy
strcpy:
      ld.w     %r10, %r12              ; return dest add

strcpy_loop:
      ld.ub    %r4, [%r13]+            ; copy src 1 byte to dest
      ld.b     [%r12]+, %r4
      cmp      %r4, 0                  ; continue until 0 terminate
      jrne     strcpy_loop
      ret
```

This routine is called from a C routine as follows.
(Excerpt from cc33\sample\ansilib\sansilib.c)

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

The first and the second arguments are respectively placed in the R12 and the R13 registers when passed, and the return value is stored in the R10 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 R12 to the R14 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 R10 register when returned.

The limitations on register usage within the assembler routine called from a C routine are as follows:
- The R0 to R3 registers must be restored directly without modifying their contents when called. When using these registers, use the pushn/popn instructions to save and restore their contents.
- The contents of the R4 to R15 registers do not need to be saved. However, avoid using registers R9 and R8 whenever possible, since R9 is used to expand the extend instructions of ext33, while R8 is used for global pointer-based optimization by ext33. Be especially careful if you do use these registers.
- The contents of the AHR, ALR, and PSR registers do not need to be saved.

For example, cc33\utility\lib_src\emulib33\fp\src\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 R12 to R14 registers when passed.
- The R10 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:

Tools	Input files	Output files
1. Preprocessor pp33	file.**s**	file.**ps**
2. Instruction extender ext33	file.**ps**	file.**ms**
3. Assembler as33	file.**ms**	file.**o**

∗ The assembly sources (.ps) obtained by compiling C sources cannot be fed into the preprocessor pp33. They must be entered to ext33.

● Types of assembly source files

Shown below are an example and the differences between each assembly source file (.s, .ps, and .ms).

Assembly source (.s) created by user

```
; boot.s
; boot program

#define SP_INI 0x0800   ; sp is in end of 2KB internal RAM          (1)
#define GP_INI 0x0000   ; global pointer %r8 is 0x0                 (1)

      .code                                                         (3)
      .word    BOOT             ; BOOT VECTOR                     (3)
BOOT:
      xld.w    %r8,SP_INI                                         (2)
      ld.w     %sp,%r8          ; set SP
      ld.w     %r8,GP_INI       ; set global pointer
      xcall    main             ; goto main                       (2)
      xjp      BOOT             ; infinity loop                   (2)
```

(1) Quasi directives processed by pp33
(2) Extended instructions processed by ext33
(3) Directive commands processed by as33

Processing this file by pp33 results in the following:

Preprocessor output file (.ps)

```
      .file    "boot.s"                                           (3)
; boot.s
; boot program

;#define SP_INI 0x0800          ; sp is in end of 2KB internal RAM  (1)
;#define GP_INI 0x0000          ; global pointer %r8 is 0x0         (1)

      .code
      .word BOOT                ; BOOT VECTOR
BOOT:
      .loc     10                                                  (3)
      xld.w    %r8,0x0800                                         (2)
      .loc     11                                                  (3)
      ld.w     %sp,%r8          ; set SP
      .loc     12                                                  (3)
      ld.w     %r8,0x0000       ; set global pointer               (2)
      .loc     13                                                  (3)
      xcall    main             ; goto main
      .loc     14                                                  (3)
      xjp      BOOT             ; infinity loop
      .endfile                                                     (3)
```

Shown below are sections of the .s file processed by pp33.

(1) The directive commands of pp33 beginning with # are processed, with the statements themselves changed to comments.
(2) When the symbol defined by #define in (1) appears, it is replaced with the value or string defined for that symbol.
(3) When pp33 is executed after specifying the -g option, the information necessary for source-level debugging is inserted.

Processing this file by ext33 results in the following:

Instruction extender output file (.ms)

```
      .file    "boot.s"
; boot.s
; boot program

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

      .code
      .word BOOT                 ; BOOT VECTOR
BOOT:
      .loc     10
      ext      0x20              ;     xld.w %r8,0x0800                    (1)
      ld.w     %r8,0x0
      .loc     11
      ld.w     %sp,%r8           ; set SP
      .loc     12
      ld.w     %r8,0x0000        ; set global pointer
      .loc     13
      ext      main@rh           ;     xcall main        ; goto main       (2)
      ext      main@rm
      call     main@rl
      .loc     14
      jp       BOOT              ;     xjp   BOOT        ; infinity loop   (3)
      .endfile
```

Shown below are sections of the .ps file processed by ext33.
The extended instructions of ext33 beginning with x in (1) to (3) are expanded into the basic as33 instructions according to operand values. At this time, the number of instructions expanded is limited to the smallest possible.
(1) is expanded into two instructions required for the immediate data of 0x800.
(2) is a label in another file and its address unknown; it is expanded into three instructions that can always be called.
(3) is a label within the same file, so that its relative address is calculated; as a result, it is expanded into one instruction.

● 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 pp33 and expanded into basic instructions that can be assembled by as33. The operators used to specify numeric values with an expression are also processed by pp33.

Preprocessor quasi directives [cc33\sample\asm\pp.s]

```
; pp.s         1998.1.5
; sample source for pp33

#include "pp.def"                    ; include file
#define SP_IRAM                      ; definition for #ifdef
#ifdef SP_IRAM                       ; condition assemble
      #define SP_INIT_ADDR 0x400     ; set number to defnum symbol
#else
      #define SP_INIT_ADDR 0x880000
#endif
#define BLK_ADDR  0x0+0x10           ; You can use arithmetic operators.
                                     ; operators : +,-,*,/,%%,>>,<<,&,!,^,~,
                                     ; ^H,^M,^L,^AH,^AL,(,)
#define gpr %r8                      ;
#define GP_INIT_ADDR 0x0             ;
```

```
#macro  FILL_AREA $1 $2 $3                  ; macro argument is $1, $2, --- $32
      xld.w       %r1,$1                    ; $1 is start address
      xld.w       %r2,$2                    ; $2 is fill pattern (8bit)
      xld.w       %r3,$3                    ; $3 is fill size (byte address)
$$1:                                        ; $$1 -- $$64 is local jump label
      cmp         %r3,0
      jreq        $$2
      ld.b        [%r1]+,%r2
      sub         %r3,1
      jp          $$1
$$2:
#endm

      .word BOOT
BOOT:
      ext         SP_INIT_ADDR^H
      ext         SP_INIT_ADDR^M
      ld.w        %r0,SP_INIT_ADDR^L
      ld.w        %sp,%r0
      ld.w        gpr,GP_INIT_ADDR
      FILL_AREA   BLK_ADDR 0b01010101 10    ; fill 0x10-0x1f with 0x55
      FILL_AREA   BLK_ADDR+0x10 0 10        ; fill 0x20-0x2f with 0x00
      jp          BOOT
```

● 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 as33.

Assembler directive commands [cc33\sample\asm\as_directive.ms]

```
.abs                                        ; absolute file
; as_directive.ms       1997.2.15
; sample source for as33 directives

      .set RAM1 0x0                         ; set absolute data

      .code                                 ; start code section
      .global  BOOT                         ; BOOT become global symbol
      .org     0x80000                      ; set absolute address
      .word    BOOT                         ; 32bit data
BOOT:                                       ; label in code section
      ld.w     %r8,0
      xld.w    %r1,[DATA1]
      xld.w    [RAM1],%r1
      xld.ub   %r2,[DATA1+8]
      xld.b    [COMM1],%r2
      jp       BOOT
      .half    0x0000                       ; same with nop

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

      .org     0x0
      .comm    COMM1 4                      ; 4 byte global bss data area
      .lcomm   LCOMM1 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.

- Instructions supported by as33 and owned by the CPU core itself (basic instructions)
- Macro instructions expanded by ext33 (extended instructions)

Pooling all instructions of these two types produces a large number of available instructions, particularly an extensive list of instructions for ext33.
Until you are familiar with programming the E0C33, 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 E0C33000 instruction set, which are assembled into machine codes by as33. 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 or symbol mask.

Example:
```
jr    LABEL      ; Specify label
ext   LABEL+4@h  ; Specify label + displacement + symbol mask
ext   LABEL+4@m
ld.w  %r9,LABEL+4@l
ld.w  %r1,[%r9]
```

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 ext33 instructions.

Basic instruction list [cc33\sample\asm\as_inst.ms]

```
; as_inst.ms   1997.2.23
; sample source for as33 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
```

● Extended instructions

The extended instructions beginning with "x" are provided to facilitate the use of the instruction extension function supported by the ext instruction. These extended instructions are expanded into basic instructions with or without the ext instruction according to the operand value.

Extended instructions [cc33\sample\asm\ext.ms]

```
; ext.ms       1997.4.30
; sample source for ext33 extended instructions
; not for execution, just for ext33 extension only

      .word BOOT
BOOT:

; summary of major patterns

; xld.w
      xld.w    %r8,0                   ; immediate load
      xld.w    %r1,DATA1               ; symbol immediate load
      xld.w    %r2,DATA1+4             ; symbol+offset

      xadd     %r1,%r2,0x12345678      ; 3 operand for xadd, xsub

      xand     %r14,%r15,0xff000000    ; 3 operand for xand, xoor, xxor
      xnot     %r8,0b1111100000

      xsrl     %r3,8                   ; immediate shift
      xrr      %r7,%r8                 ; register shift
                                       ; for xsrl, xsll, xsra, xsla, xrr, xrl

      xld.w    %r1,[0x1234568]         ; immediate address
      xld.ub   %r5,[DATA1]             ; symbol address
      xbtst    [COMM1+0x400],2         ; symbol address + offset
      xld.uh   %r10,[%sp+0x222]        ; sp relative
      xld.b    [%sp],%r7               ; sp relative
      xld.uh   %r1,[%r15+0x1234568]    ; resister + immediate address
      xld.h    %r5,[%r11+DATA1]        ; register + symbol address
      xbset    [%r9+COMM1+0x400],2     ; register + symbol address + offset
                                       ; for xld.w, xld.uh, xld.h, xld.ub, xld.b,
                                       ;     xbset, xbclr, xbtst, xbnot

      xjp      -2                      ; immediate relative
      xjrgt.d  BOOT                    ; symbol relative
                                       ; for xjp, xjreq, xjrne, xjrgt, xjrge, xjrlt,
                                       ; xjrle, xjrugt, xjruge, xjrult, xjrule, xcall
                                       ; And with ".d"

; more detail samples

; xld.w                load immediate to register operation

      xld.w    %r8,0                   ; decimal
      xld.w    %r0,0x12345678          ; hex
      xld.w    %r0,0b10101             ; binary
      xld.w    %r1,DATA1               ; symbol
      xld.w    %r2,DATA1+4             ; symbol+offset(hex,dec,bin)
      xld.w    %r2,DATA1+0x5
      xld.w    %r2,DATA1+0b110

; xadd, xsub   add and sub, arithmetic operations

      xadd     %r1,%r2,0x12345678      ; 3 operand No.1
      xsub     %r2,%r1,0x12345         ; 3 operand No.2
      xadd     %r0,%r1,1               ; 3 operand No.3
      xsub     %r2,%r2,5               ; 3 operand No.4
      xadd     %r1,%r2,%sp             ; for C compiler
      xsub     %sp,%sp,%r1             ; for C compiler

; xand, xoor, xxor, xnot
;              and, or, xor, and not, logical operations

      xand     %r14,%r15,0xff000000    ; 3 operand No.1
      xoor     %r12,%r11,0xfedc        ; 3 operand No.2
      xxor     %r9,%r9,-1              ; 3 operand No.3
      xnot     %r8,0b1111100000
```

```
; xsrl, xsll, xsra, xsla, xrr, xrl     shift operations

      xsrl     %r3,8                   ; immediate shift No.1
      xsll     %r4,15                  ; immediate shift No.2
      xsra     %r5,17                  ; immediate shift No.3
      xsla     %r6,31                  ; immediate shift No.4
      xrr      %r7,%r8                 ; register shift

; xld.w, xld.uh, xld.h, xld.ub, xld.b, xbset, xbclr, xbtst, xbnot
;              load, bit operation from/to absolute address

      xld.w    %r1,[0x1234568]         ; immediate address No.1
      xld.uh   %r2,[0xABC]             ; immediate address No.2
      xld.h    [10], %r3               ; immediate address No.3
      xld.ub   %r4,[0]                 ; immediate address No.4
      xld.b    %r5,[DATA1]             ; symbol address No.1
      xbnot    [COMM1],1               ; symbol address No.2
      xbtst    [COMM1+0x400],2         ; symbol address + offset No.1
      xbset    [COMM1+0x10],3          ; symbol address + offset No.2
      xbclr    [COMM1+1],4             ; symbol address + offset No.3

; xld.w, xld.uh, xld.h, xld.ub, xld.b, xbset, xbclr, xbtst, xbnot
;              load, bit operation from/to SP relative address

      xld.w    %r15,[%sp+0x4444444]    ; sp relative No.1
      xld.uh   %r10,[%sp+0x222]        ; sp relative No.2
      xld.b    [%sp],%r7               ; sp relative No.3
      xbset    [%sp+0x14],5            ; sp relative No.4

; xld.w, xld.uh, xld.h, xld.ub, xld.b, xbset, xbclr, xbtst, xbnot
;              load, bit operation from/to register relative address

      xld.w    %r1,[%r15+0x1234568]    ; + immediate address No.1
      xld.uh   %r2,[%r14+0xABC]        ; + immediate address No.2
      xld.h    [%r13+10], %r3          ; + immediate address No.3
      xld.ub   %r4,[%r12]              ; + immediate address No.4
      xld.b    %r5,[%r11+DATA1]        ; + symbol address No.1
      xbnot    [%r10+COMM1],1          ; + symbol address No.2
      xbtst    [%r9+COMM1+0x400],2     ; + symbol address + offset No.1
      xbset    [%r8+COMM1+0x10],3      ; + symbol address + offset No.2
      xbclr    [%r7+COMM1+1],4         ; + symbol address + offset No.3

; xld.w                load word operation from sp register for C support

      xld.w    [%sp],%sp
      xld.w    [%sp+0x2468],%sp
      xld.w    [0x12340],%sp
      xld.w    [COMM1],%sp
      xld.w    [COMM1+4],%sp
      xld.w    [%r5],%sp
      xld.w    [%r6+0b1100],%sp
      xld.w    [%r7+DATA1],%sp
      xld.w    [%r7+DATA1+200],%sp

; xjp, xjreq, xjrne, xjrgt, xjrge, xjrlt, xjrle, xjrugt, xjruge, xjrult
; xjrule, xcall and with .d            relative branchs

NEAR:
      xjp.d    -2                      ; immediate relative No.1
      xjreq    800                     ; immediate relative No.2
      xjrne    0x1000000               ; immediate relative No.3
      xjrgt.d  BOOT                    ; symbol relative No.1
      xjrge    COMM1                   ; symbol relative No.2
      xjruge   NEAR                    ; symbol relative No.3

      .data
DATA1:
      .word    0x12345678

      .comm COMM1 4
```

● make file

Execution of make is indispensable in obtaining the final object file by efficient execution of the necessary tools after correcting source files. Shown below are examples of make files: one with suffixes defined, and one with suffixes undefined. In most cases, you can use either make file, since they are easily created with wb33. But in cases involving manual correction for additional processed files, the make file with suffixes defined may prove preferable.

make file using suffix definition

```
# make file made by wb33

# macro definitions for tools & dir

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

# macro definitions for tool flags

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

# suffix & rule definitions

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

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

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

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

# dependency list start

suf.srf : suf.cm \
    boot.o \
    main.o \

   $(LK33) $(LK33_FLAG) suf.cm

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

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

# dependency list end
```

make file not using suffix definition

```
# make file made by wb33

# macro definitions for tools & dir

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

# macro definitions for tool flags

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

# suffix & rule definitions

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

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

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

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

# dependency list start

nosuf.srf : nosuf.cm \
    boot.o \
    main.o \

   $(LK33) $(LK33_FLAG) nosuf.cm

## boot.s
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.c
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

# dependency list end
```

```
# optimaization by 2 pass make

opt:
   $(MAKE) -f suf.mak
   $(TOOL_DIR)\cwait 2
   $(EXT33) $(EXT33_CMX_FLAG) suf.cmx
   $(MAKE) -f suf.mak

# clean files except source

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

```
# optimaization by 2 pass make

opt:
   $(MAKE) -f nosuf.mak
   $(TOOL_DIR)\cwait 2
   $(EXT33) $(EXT33_CMX_FLAG) nosuf.cmx
   $(MAKE) -f nosuf.mak

# clean files except source

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

2.5 C and Code Optimization

This section explains how to optimize the instruction code generated by the C compiler, using main.c in cc33\sample\ccode as an example. Note that ccode is found in the CC33 Ver 3.0 or later package.
The original source is shown below.

Assembly source boot routine [ccode\boot.s]

```
; boot.s  1997.2.13
; boot program

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

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

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;
    }
```

When this program is compiled in the default state, the following code results.

Code derived by compiling [ccode\default.dis]

```
**** Disassemble code and source code ****
  Addr   Code  Unassemble               Line        Source
00080000 0004 ***
00080002 0008 ***
                                            --- boot.s ---
                                       00001  ; boot.s  1997.2.13
                                       00002  ; boot program
                                       00003
                                       00004  #define SP_INI 0x0800; sp is in ..
                                       00005  #define GP_INI 0x0000; global ..
                                       00006
                                       00007     .code
                                       00008     .word BOOT        ; BOOT VECTOR
                                       00009  BOOT:
00080004 C020 ext     0x20             00010     xld.w %r8,SP_INI
00080006 6C08 ld.w    %r8,0x0
00080008 A081 ld.w    %sp,%r8          00011     ld.w  %sp,%r8     ; set SP
0008000A 6C08 ld.w    %r8,0x0          00012     ld.w  %r8,GP_INI  ; set ..
0008000C C000 ext     0x0              00013     xcall main        ; goto main
0008000E C000 ext     0x0
00080010 1C02 call    0x2
00080012 1EF9 jp      0xf9             00014     xjp   BOOT        ; infinity ..
                                            --- main.c ---
                                       00001  /* main.c 1999.7.28 */
                                       00002  /* sample program for optimize*/
                                       00003
                                       00004  struct ST gst;
                                       00005  int a;
                                       00006  struct ST {
                                       00007     int s1;
                                       00008     int s2;
                                       00009  };
                                       00010
                                       00011  main()
00080014 840D sub     %sp,0xd          00012      {
                                       00013     int b;
                                       00014     struct ST st;
                                       00015     int ar[10];
                                       00016
00080016 6C1C ld.w    %r12,0x1         00017     a = 1;
00080018 C000 ext     0x0
0008001A C000 ext     0x0
0008001C 6C89 ld.w    %r9,0x8
0008001E 3C9C ld.w    [%r9],%r12
00080020 6C2A ld.w    %r10,0x2         00018     b = 2;
00080022 5CAA ld.w    [%sp+0xa],%r10
                                       00019
00080024 6C3A ld.w    %r10,0x3         00020     st.s1 = 3;
00080026 5CBA ld.w    [%sp+0xb],%r10
00080028 6C4A ld.w    %r10,0x4         00021     ar[3] = 4;
0008002A 5C3A ld.w    [%sp+0x3],%r10
                                       00022
0008002C A41D ld.w    %r13,%sp         00023     sub1(a, &b);
0008002E 628D add     %r13,0x28
00080030 1C0D call    0xd
00080032 1C0E call    0xe              00024     sub2();
                                       00025
00080034 6C5A ld.w    %r10,0x5         00026     gst.s2 = 5;
00080036 C000 ext     0x0
00080038 C000 ext     0x0
0008003A 6C49 ld.w    %r9,0x4
0008003C 3C9A ld.w    [%r9],%r10
0008003E A41C ld.w    %r12,%sp         00027     sub3(&st, ar);
00080040 62CC add     %r12,0x2c
00080042 A41D ld.w    %r13,%sp
00080044 1C0F call    0xf
00080046 800D add     %sp,0xd          00028      }
00080048 0640 ret
```

```
                                        00029
                                        00030  sub1(a,b)
                                        00031      int a;
                                        00032      int *b;
                                        00033      {
0008004A 3CDC ld.w    [%r13],%r12       00034     *b = a;
0008004C 0640 ret                       00035      }
                                        00036
                                        00037  sub2()
                                        00038      {
                                        00039     volatile char *vp;
                                        00040
0008004E C000 ext     0x0               00041     vp = (volatile char *)0x40000;
00080050 D000 ext     0x1000
00080052 6C0B ld.w    %r11,0x0
00080054 6C2A ld.w    %r10,0x2          00042     *vp = 2;
00080056 34BA ld.b    [%r11],%r10
                                        00043
00080058 C000 ext     0x0               00044     *(volatile char *)(0x48000) |= 0x1;
0008005A D200 ext     0x1200
0008005C 6C0B ld.w    %r11,0x0
0008005E B0B0 bset    [%r11],0x0
00080060 0640 ret                       00045      }
                                        00046
                                        00047  sub3(st, ar)
                                        00048      struct ST *st;
                                        00049      int ar[];
                                        00050      {
00080062 6C4A ld.w    %r10,0x4          00051     st->s2 = 4;
00080064 C004 ext     0x4
00080066 3CCA ld.w    [%r12],%r10
00080068 6C5A ld.w    %r10,0x5          00052     ar[5]=5;
0008006A C014 ext     0x14
0008006C 3CDA ld.w    [%r13],%r10
0008006E 0640 ret                       00053      }
```

● About external variables and auto variables

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

```
                                        00005  int a;          ← Defines external variable
                                        00006  struct ST {
                                        00007     int s1;
                                        00008     int s2;
                                        00009  };
                                        00010
                                        00011  main()
00080014 840D sub     %sp,0xd           00012      {
                                        00013     int b;       ← Defines auto variable
                                        00014     struct ST st;
                                        00015     int ar[10];
                                        00016
00080016 6C1C ld.w    %r12,0x1          00017     a = 1;
00080018 C000 ext     0x0
0008001A C000 ext     0x0
0008001C 6C89 ld.w    %r9,0x8           ← External variable has its address first placed in R9,
0008001E 3C9C ld.w    [%r9],%r12        ← then accessed based on R9
00080020 6C2A ld.w    %r10,0x2            00018      b = 2;
00080022 5CAA ld.w    [%sp+0xa],%r10    ← auto variable is accessed as offset relative to the stack
```

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

Normally, an external variable is accessed following the procedure
1) Place 32-bit value (variable's address) in R9
2) Access memory based on R9

Thus, four instructions are required.

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

an auto variable in the stack area of 63 bytes or less when offset is byte accessed, 126 bytes or less when half-word accessed, or 252 bytes or less when word accessed, may be accessed with one instruction, or beyond that, with two instructions. 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 5555s 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.

```
                                        00037  sub2()
                                        00038      {
                                        00039     volatile char *vp;
                                        00040
0008004E C000 ext     0x0               00041     vp = (volatile char *)0x40000;
00080050 D000 ext     0x1000
00080052 6C0B ld.w    %r11,0x0
00080054 6C2A ld.w    %r10,0x2          00042     *vp = 2;
00080056 34BA ld.b    [%r11],%r10                                  ← Access
                                        00043
00080058 C000 ext     0x0               00044     *(volatile char *)(0x48000) |= 0x1;
0008005A D200 ext     0x1200
0008005C 6C0B ld.w    %r11,0x0
0008005E B0B0 bset    [%r11],0x0                                   ← bset access
00080060 0640 ret                       00045      }
```

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.

```
                                    00030  sub1(a,b)
                                    00031       int a;
                                    00032       int *b;
                                    00033       {
0008004A 3CDC ld.w    [%r13],%r12   00034      *b = a;      ← Access by one instruction
0008004C 0640 ret                   00035       }
```

● 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.

```
                                    00011  main()
00080014 840D sub     %sp,0xd       00012       {
                                    00013      int b;
                                    00014      struct ST st;
                                    00015      int ar[10];
                                    00016
                                      |
                                    00019
00080024 6C3A ld.w    %r10,0x3      00020      st.s1 = 3;
00080026 5CBA ld.w    [%sp+0xb],%r10                      ← Accesses auto variable
00080028 6C4A ld.w    %r10,0x4      00021      ar[3] = 4;
0008002A 5C3A ld.w    [%sp+0x3],%r10                      ← Accesses auto variable
                                    00022
0008002C A41D ld.w    %r13,%sp      00023      sub1(a, &b);
0008002E 628D add     %r13,0x28
00080030 1C0D call    0xd
00080032 1C0E call    0xe           00024      sub2();
                                    00025
00080034 6C5A ld.w    %r10,0x5      00026      gst.s2 = 5;
00080036 C000 ext     0x0                                 ← Accesses external variable
00080038 C000 ext     0x0
0008003A 6C49 ld.w    %r9,0x4
0008003C 3C9A ld.w    [%r9],%r10
0008003E A41C ld.w    %r12,%sp      00027      sub3(&st, ar);
00080040 62CC add     %r12,0x2c
00080042 A41D ld.w    %r13,%sp
00080044 1C0F call    0xf
00080046 800D add     %sp,0xd       00028       }
00080048 0640 ret
```

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

```
                                    00047  sub3(st, ar)
                                    00048       struct ST *st;
                                    00049       int ar[];
                                    00050       {
00080062 6C4A ld.w    %r10,0x4      00051      st->s2 = 4;
00080064 C004 ext     0x4                                 ← Access as offset
00080066 3CCA ld.w    [%r12],%r10                         ← Two instructions
00080068 6C5A ld.w    %r10,0x5      00052      ar[5]=5;
0008006A C014 ext     0x14                                ← Access as offset
0008006C 3CDA ld.w    [%r13],%r10                         ← Two instructions
0008006E 0640 ret                   00053       }
```

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.

● Call optimization

```
                                          --- boot.s ---
                                    00001  ; boot.s  1997.2.13
                                    00002  ; boot program
                                    00003
                                    00004  #define SP_INI 0x0800; sp ..
                                    00005  #define GP_INI 0x0000; global ..
                                    00006
                                    00007     .code
                                    00008     .word BOOT        ; BOOT VECTOR
                                    00009  BOOT:
00080004 C020 ext     0x20          00010     xld.w %r8,SP_INI
00080006 6C08 ld.w    %r8,0x0
00080008 A081 ld.w    %sp,%r8       00011     ld.w  %sp,%r8     ; set SP
0008000A 6C08 ld.w    %r8,0x0       00012     ld.w  %r8,GP_INI  ; set
0008000C C000 ext     0x0           00013     xcall main        ; ..  ← call
0008000E C000 ext     0x0                                           ←
00080010 1C02 call    0x2                                           ← 3 instructions
00080012 1EF9 jp      0xf9          00014     xjp   BOOT        ; infinity ..
```

Used to call a routine in another file, call is normally expanded as a precautionary measure into three instructions. This ensures that the program always branches to a routine, no matter where in the E0C33 address space it may be located. Two instructions (ext + call) may also be used to make the program branch to a location 2M bytes forward or backward from that point. For example, when the entire program is stored in 2MB of flash memory, all call instances can be turned into two instructions without problems. In such cases, use the ext33 -near flag.

In ccode\tes.mak

```
#EXT33_FLAG =            ← Comment out this default using #
EXT33_FLAG = -near       ← Use this one
```

After making this change, execute make clean once, then reexecute make. All call instances are turned into two instructions.

(From ccode\near.dis)

```
0008000C C000 ext     0x0           00013     xcall main        ; goto main
0008000E 1C03 call    0x3
```

In addition to call, other branch instructions that cause the program jump to a label within the file are also optimized into one or two instructions by ext33. Instructions that jump to a label outside the file (as shown above) are normally expanded into three instructions, or into two instructions when accompanied by the -near flag. (However, the 2-pass make described further below generates more intelligent processing.)

● Global pointer optimization of external variables

For access to external variables, ext33 provides several methods of optimization.
In ccode\test.mak, specify

```
EXT33_FLAG = -gp 0x0
```

and global pointer optimization is implemented.

Before use of this optimize function, the global pointer address must be set in R8 at boot time. Here, because the variable area starts from 0, the value set is 0.
(From ccode\gp.dis)

```
0008000A 6C08 ld.w    %r8,0x0       00012     ld.w %r8,GP_INI  ; set global pointer
```

The external variable 'a' is accessed as offset relative to the base indicated by R8 (global pointer).

```
00080016 6C1C ld.w    %r12,0x1      00017     a = 1;
00080018 C000 ext     0x0                     ← Base in R8
0008001A C008 ext     0x8                     ←
0008001C 3C8C ld.w    [%r8],%r12              ← Accessed using three instructions
```

Note that the offset from the base address in R8 is a maximum of 26 bits and that all external variable accesses must occur on the positive side of the base address. For this reason, we recommend using R8 as 0. (However, the 2-pass make described further below generates more intelligent processing.)

● Two-pass optimization of call and external variables

(From ccode\test.mak)

```
# optimaization by 2 pass make

opt:
      $(MAKE) -f test.mak
      $(TOOL_DIR)\cwait 2
      $(EXT33) $(EXT33_CMX_FLAG) test.cmx
      $(MAKE) -f test.mak
```

As shown above, run make once, then reexecute the sections below ext33 based on this information. This process is referred to as a 2-pass make. In this case, the ext33 flag in the second pass is set by default as follows:

```
EXT33_CMX_FLAG = -lk test -c
```

This is the specification required for ext33 to optimize code generation, using the map and symbol information created by lk33 in the first pass.

One target to be optimized in this way is branch instructions, such as call.

(From ccode\2pass.dis)

```
 0008000C C000 ext     0x0               00013     xcall main        ; goto main
 0008000E 1C03 call    0x3
```

The call to an external file is turned into two instructions. Although similar to the -near flag, this optimization causes ext33 to calculate the distance from call to the label to determine whether it should consist of two or three instructions. For large distances, call is expanded into three instructions.

The second target of optimization is an access to external variables.

(From ccode\default.dis)

```
 00080016 6C1C ld.w    %r12,0x1          00017     a = 1;
 00080018 C000 ext     0x0
 0008001A C000 ext     0x0
 0008001C 6C89 ld.w    %r9,0x8
 0008001E 3C9C ld.w    [%r9],%r12
```

Four instructions are normally required.

(From ccode\2pass.dis)

```
 00080016 6C1C ld.w    %r12,0x1          00017     a = 1;
 00080018 6C89 ld.w    %r9,0x8
 0008001A 3C9C ld.w    [%r9],%r12
```

In this example, access is turned into two instructions. Since ext33 can obtain address information for variable 'a' in the second pass, the address is stored in R9 using the fewest number of instructions. Here, two instructions are used to perform an access, but this is a special case occurring only at the beginning of internal RAM. Access generally requires three instructions and is limited to an address range of up to 0x3ffff.

The 2-pass make can be used in combination with global pointer optimization.

(From ccode\gp.dis)

```
 00080016 6C1C ld.w    %r12,0x1          00017     a = 1;
 00080018 C000 ext     0x0
 0008001A C008 ext     0x8
 0008001C 3C8C ld.w    [%r8],%r12
```

With global pointer optimization alone, access is performed with three instructions. When combined with a 2-pass make (flag settings: EXT33_CMX_FLAG = -lk test -gp 0x0 -c),

(From ccode\gp2pass.dis)

```
 00080016 6C1C ld.w    %r12,0x1          00017     a = 1;
 00080018 C008 ext     0x8
 0008001A 3C8C ld.w    [%r8],%r12
```

the number of instructions is reduced to two. However, this two-instruction case occurs only for a 4KB range from R8, beyond which access expands to three instructions. For a 2-pass make, access in the negative direction from R8 is expanded into the ordinary format not using R8.

● 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.
3) Do not use the R8 register in user applications. Global pointer optimization is effective for all external variables only if the address area consists of a 26-bit space.
4) Execute a 2-pass make. This is effective for variables in the internal RAM area and optimizes the call instruction.

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

2.6 Mapping by Linker

● Absolute files

There are two types of source files: absolute files, for which absolute addresses are specified at the source level, and relocatable files for which the source itself is relocatable and addresses are specified by a linker. C source files are available only as relocatable files. The assembler recognizes both relocatable and absolute files.

Example of an absolute file:

```
      .abs                      ; Directive command starting with 0xc00000
      .org 0xc000000
      .code

      .word BOOT                ; Vector table (consisting of a single boot line)

BOOT:
      xld.w    %r8,0x800        ; Boot program
      ld.w     %sp,%r8          ; Sets SP and calls main
      xcall    main
```

The file is declared to be an absolute file by the .abs directive command, and its address is determined by the .org directive command. This programming technique is useful for performing a simple test with one file. Due to various limitations, this technique is not recommended for full-scale development using multiple files.

● CODE, DATA, and BSS sections

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

CODE section This section stores program code and ROM data.
DATA section This section stores R/W'able data with initial values.
BASS 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 CODE section

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

Compiling the above results in the following.

```
gcc2_compiled.:
__gnu_compiled_c:
      .global  b
      .data
      .align   2
b:
      .word    1                ← b is data in the DATA section
      .global  c
      .code
      .align   2
c:
      .word    2                ← c is data in the CODE section
      .code
      .align   1
      .global  main
main:
;     .frame   %sp,4,$31
;     .mask    0x80000000,-4
;     .fmask   0x00000000,0
      xld.w    %r10,[c]         ← All instructions are placed in the CODE section
      xld.w    [b],%r10
      xld.w    [a],%r10
      ret

      .comm    a 4              ← a is placed in the BSS section
```

Note that the classification of and directive commands for CODE, DATA, and BSS incorporate UNIX concepts. (In UNIX, CODE is referred to as a TEXT section.)
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.

● Ordinary maps

In general, create all files as relocatable files. Specify addresses in a linker command file.
Example: cc33\sample\ansilib.cm

```
;Map set
-code 0x0c00000                  ; set relative code section start address
-bss  0x0800000                  ; set relative bss  section start address

-code 0x0080000 {boot.o}         ; set code sections to absolute address
                          |
```

In this example, the location of the code in boot.o begins at address 0x80000. The code in all other files is located at contiguous addresses in the order of the files specified, starting with 0xc00000. All variables are located starting with address 0x800000 in the order in which they are linked. Absolute addresses are specified only for one block starting with address 0x80000, where one file of boot.o is located. You can place multiple files in this block, or specify multiple blocks.
(Reference)

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

-code 0x0080100 {test2.o test3.o} ; set code sections to absolute address
-data 0x0081100 {test2.o test3.o} ; set data sections to absolute address
-bss  0x0000200 {test2.o test3.o} ; set bss  sections to absolute address
```

● Method for using the DATA section

The DATA section is a R/W'able variable area with initial values. (For more information on each section, see "● CODE, 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.

For 2), the data must be transferred with the boot program. For 1) and 3), linking must be performed by the virtual section function of a linker. This procedure is illustrated using cc33\sample\usection.

Method for specifying a linker command file

Example: Excerpt from usection\usection.cm

```
-objsym                          ← Start by entering this specification
                                   Create a label (section symbol) for transfer by the boot program.
-section BOOT = 0x80000
-section CODE = 0x80100
-section DCOPY                   ← Label indicating RAM area for data expansion

-code BOOT {boot.o}      ; set boot.o to absolute address
-code CODE               ; code section start address
-bss  0x0000000          ; bss  section start address

-udata DCOPY                     ← UDATA sections are successively mapped after BSS.
```

UDATA is a temporary section for symbol resolution, with its actual body located after the code section and stored in ROM. Linking produces the following map.
(Excerpt from usection\usection.map)

```
Data Section mapping
Address     Vaddress    Size        File            ID  Attr
00080128    00000004    00000000    boot.o          1   REL     ← The actual body is at 0x80128
00080128    00000004    00000004    main.o          1   REL     ← Mapped to virtual address 0x4
```

Write the above specification in the linker command file. For more information on the linker command -udata, see the linker section, "Virtual and Union (U) Section" in the E0C33 Family C Compiler Package Manual.

Transfer when booting

As described below, transfer data to RAM using section symbols before executing the program.
Example: Excerpt from usection\boot.s

```
      .code
      .word BOOT                ; BOOT VECTOR
BOOT:
      xld.w    %r8,SP_INI
      ld.w     %sp,%r8          ; set SP
      ld.w     %r8,GP_INI       ; set global pointer

; copy all data section to DCOPY area

      xld.w    %r12, __START_DEFAULT.DATA    ; data start addr
      xld.w    %r13, __START_DCOPY           ; RAM area addr
      xld.w    %r14, __SIZEOF_DEFAULT.DATA   ; copy size (byte)
      xcall    HCOPY_LOOP
               |
               |
HCOPY_LOOP:
      ld.uh    %r4,[%r12]+      ; read half from src addr
      ld.h     [%r13]+,%r4      ; write half to dest addr
      sub      %r14,2           ; decrement 2 byte
      jrgt     HCOPY_LOOP
      ret
```

The -objsym specification in the linker command file creates symbols corresponding to the source address for transfer, size, and destination address. Use these symbols as you copy.

● Caching the program to internal RAM

When the program resides in external ROM or flash memory, program access requires one to two wait states. To eliminate wait states and speed up processing, copy the program to internal RAM. This technique is illustrated below, using cc33\sample\usection as an example.

Method for specifying the linker command file

Example: Excerpt from usection\usection.cm

```
;Map set

-objsym                          ← Start by entering this specification.
                                   Create a label (section symbol) for transfer by the boot program.
-section BOOT = 0x80000
-section CODE = 0x80100
-section DCOPY
-section CCACHE                  ← Label indicating RAM cache area

-code BOOT {boot.o}      ; set boot.o to absolute address
-code CODE               ; code section start address
-bss  0x0000000          ; bss  section start address

-udata DCOPY
-ucode CCACHE {main.o}           ← CCACHE is located after DCOPY in the BSS section.
```

main.o is placed in CCACHE. Specifying multiple files here results in sharing of the CCACHE area. (The same RAM area may be used as a time-multiplexed program cache.)

Linking produces the following map.
(From usection\usection.map)

```
Code Section mapping
Address     Vaddress    Size        File                        ID  Attr
00080000    --------    00000048    boot.o                      0   REL
00080100    00000008    00000028    main.o                      0   REL

Data Section mapping
Address     Vaddress    Size        File                        ID  Attr
00080128    00000004    00000000    boot.o                      1   REL
00080128    00000004    00000004    main.o                      1   REL

Bss Section mapping
Address     Vaddress    Size        File                        ID  Attr
00000000    --------    00000000    boot.o                      2   REL
00000000    --------    00000004    main.o                      2   REL
```

Although the actual body of main.o is located at 0x80100, it is linked for execution at 0x8. Write the following specification in the linker command file. For more on linker command -udata, see the linker section, "Virtual and Union (U) Section" in the E0C33 Family C Compiler Package Manual.

Transfer when booting

Transfer data to RAM using the following section symbols before executing the program. In cases involving sharing by multiple files, transfer the object in the same way just before execution.
Example: usection\boot.s

```
; boot.s  1997.3.29
; boot program for usection function

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

; boot up program set SP and %r8(global pointer)

      .code
      .word BOOT                ; BOOT VECTOR
BOOT:
      xld.w    %r8,SP_INI
      ld.w     %sp,%r8          ; set SP
      ld.w     %r8,GP_INI       ; set global pointer

; copy all data section to DCOPY area

      xld.w    %r12, __START_DEFAULT.DATA    ; data start addr
      xld.w    %r13, __START_DCOPY           ; RAM area addr
      xld.w    %r14, __SIZEOF_DEFAULT.DATA   ; copy size (byte)
      xcall    HCOPY_LOOP

; copy main.o code to CCACHE area

      xld.w    %r12, __START_main_code       ; code start addr
      xld.w    %r13, __START_CCACHE          ; RAM area addr
      xld.w    %r14, __SIZEOF_main_code      ; copy size (byte)
      xcall    HCOPY_LOOP                          ← Transfer using section symbol

; start main

      xcall    main              ; goto main       ← Execute main.o (after jumping to internal RAM)
      xjp      BOOT              ; infinity loop

; copy %r12 addr to %r13 addr with %r14 size

HCOPY_LOOP:
      ld.uh    %r4,[%r12]+       ; read half from src addr
      ld.h     [%r13]+,%r4       ; write half to dest addr
      sub      %r14,2            ; decrement 2 byte
      jrgt     HCOPY_LOOP
      ret
```

3 PROGRAMMING THE E0C33 PERIPHERAL FUNCTIONS

This chapter describes some basic methods for programming the peripheral functions of the E0C33 chip.

Note: Unless otherwise noted, the peripheral functions and following example code apply to the E0C33208. 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 E0C33208, 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 E0C33 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

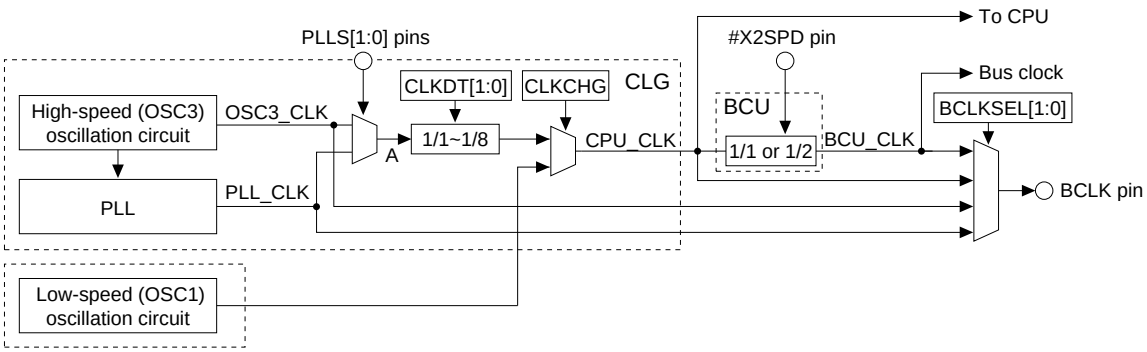
The control bits for setting up BCU for special purposes are available at addresses 0x4812E and 0x48130. Two frequently-used control bits are described below.

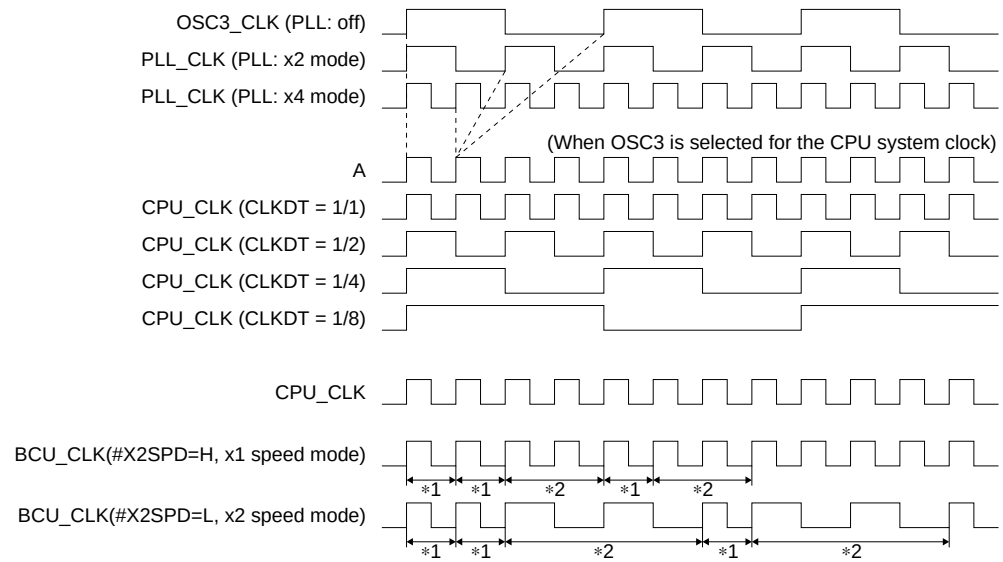
BCLK (0x4812E•DF): BCLK output enable

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)





∗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).)

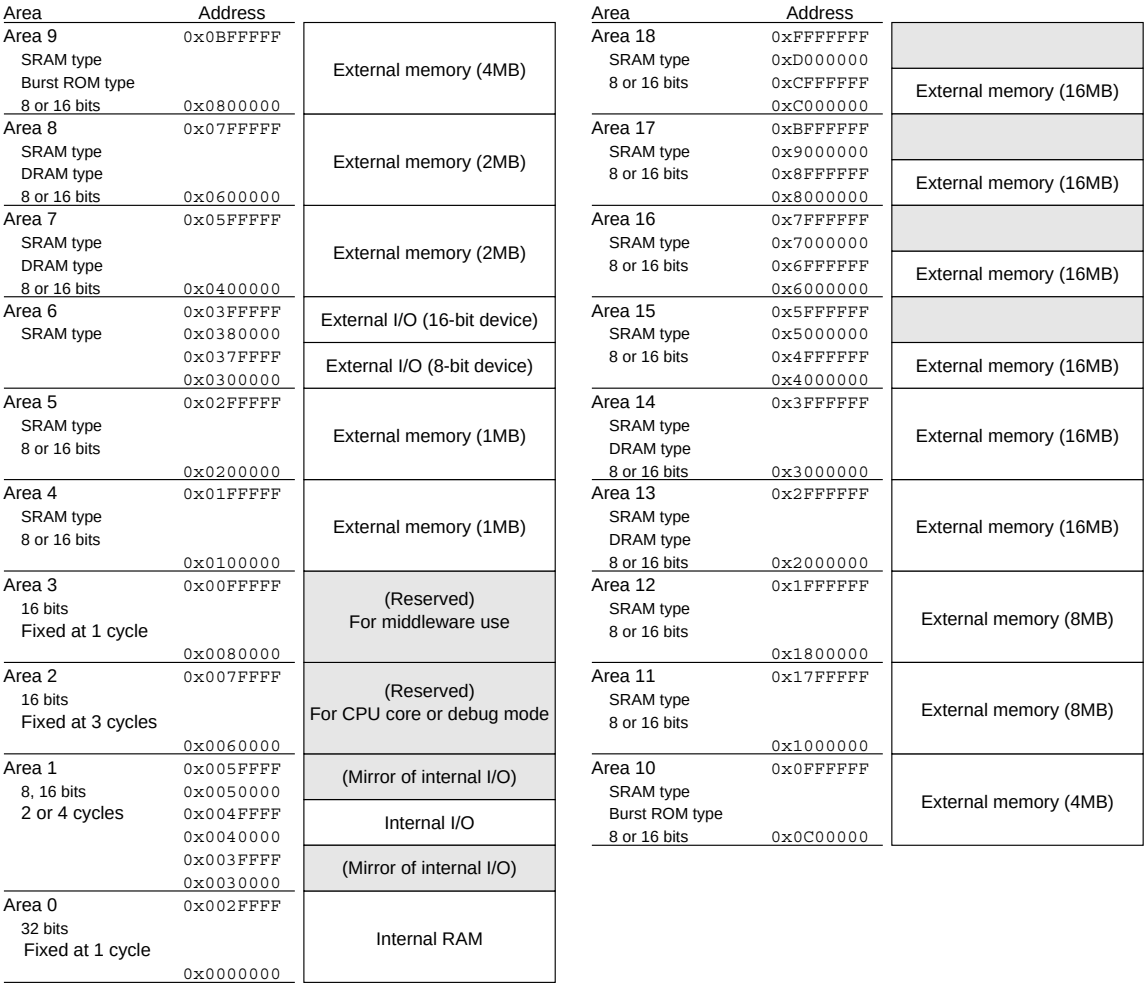
Clock system (E0C33208)

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

Because the E0C33208 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")



E0C33 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 0x037FFFF 0x0300000	External I/O (16-bit device) 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 0x8FFFFFF 0x8000000	(Mirror of External memory 6) External memory 6 (16MB)
Area 15 (#CE15) SRAM type 8 or 16 bits	0x5FFFFFF 0x5000000 0x4FFFFFF 0x4000000	(Mirror of External memory 5) 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 0x037FFFF 0x0300000	External I/O (16-bit device) External I/O (8-bit device)

CEFUNC = "01"

Area	Address	
Area 17+18 (#CE17+18) SRAM type 8 or 16 bits	0xFFFFFFF 0xD000000 0xCFFFFFF 0xC000000 0xBFFFFFF 0x9000000 0x8FFFFFF 0x8000000	(Mirror of External memory 7') External memory 7' (16MB) (Mirror of External memory 7) External memory 7 (16MB)
Areas 15–16 (#CE15+16) SRAM type 8 or 16 bits	0x7FFFFFF 0x7000000 0x6FFFFFF 0x6000000 0x5FFFFFF 0x5000000 0x4FFFFFF 0x4000000	(Mirror of External memory 6') External memory 6' (16MB) (Mirror of External memory 6) 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 the 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 cc33 ver.2.

● ICD33 trace auxiliary interrupt program

This sample is an code example for reinforcing the ICD33 trace function. The ICD33 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 ICD33 using that PC value as a starting point.

Vector section

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

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

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

(1) Vector table setting

Register the interrupt routine in the vector table.

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

Initialization and interrupt service routine

```
.global INIT_8TIMER

INIT_8TIMER:
      pushn    %r1
;psr set
      ld.w     %r0,0x10                                                    (1)
      ld.w     %psr,%r0 ;IE enable

;8bit timer 0 set
      xld.w    %r0,0x40146                                                 (2)
      xld.w    %r1,0x00
      ld.b     [%r0],%r1          ;8timer0 clock division enable

      xld.w    %r0,0x40269                                                 (4)
      xld.w    %r1,0x03
      ld.b     [%r0],%r1          ;8timer0 interrupt priority level 3

      xld.w    %r0,0x4014d                                                 (2)
      xld.w    %r1,0x0f
      ld.b     [%r0],%r1          ;8timer0 clock division ratio is 1/256

      xld.w    %r0,0x40275                                                 (4)
      xld.w    %r1,0x01
      ld.b     [%r0],%r1          ;8timer0 interrupt enable

      xld.w    %r0,0x40285                                                 (4)
      xld.w    %r1,0x01
      ld.b     [%r0],%r1          ;8timer0 interrupt flag reset

      xld.w    %r0,0x40160                                                 (3)
      xld.w    %r1,0x0
      ld.b     [%r0]+,%r1         ;clock out off,stop
      xld.w    %r1,0x75           ;interrupt every 30000 clocks
      ld.b     [%r0]+,%r1         ;set reload data

      xld.w    %r0,0x40160                                                 (5)
      xld.w    %r1,0x1
      ld.b     [%r0],%r1          ;clock out off,preset,start

      popn     %r1
      ret

.global TIME_INT

TIME_INT:                                                                  (6)
      pushn    %r1
      xld.w    %r1,0x40285
      xld.w    %r0,0x01
      ld.b     [%r1],%r0          ;8timer0 interrupt flag reset
      popn     %r1
      reti
```

For the sake of explanation, the sequence in which the above routine is processed differs from the order in which explanations are provided below.

(1) Enabling interrupts

Enable the IE flag (to enable interrupts). (This processing should be performed after (5).)

```
      ld.w     %r0,0x10
      ld.w     %psr,%r0               ; IE enable
```

(2) Setting the prescaler

Set the prescaler's divided clock for the 8-bit timer operating clock.

```
      xld.w    %r0,0x40146
      xld.w    %r1,0x00
      ld.b     [%r0],%r1              ;8timer0 clock division enable
```

Set the prescaler's division ratio to 1/256.

```
      xld.w    %r0,0x4014d
      xld.w    %r1,0x0f
      ld.b     [%r0],%r1              ;8timer0 clock division ratio is 1/256
```

(3) Setting the 8-bit timer

The reload data 0x75 is used to generate an interrupt every 30,208 clock periods of the prescaler's input clock (by default, OSC3 or PLL output).
(0x75 +1) × 256 = 30,208 clock periods

With the CPU core operating at 20 MHz, an interrupt is generated every 1.5 ms.

```
xld.w    %r0,0x40160
xld.w    %r1,0x0
ld.b     [%r0]+,%r1              ;clock out off,stop
xld.w    %r1,0x75                ;interrupt every 30000 clocks
ld.b     [%r0]+,%r1              ;set reload data
```

(4) Setting the interrupt controller

Set the priority level of the 8-bit timer interrupt to 3.

```
xld.w    %r0,0x40269
xld.w    %r1,0x03
ld.b     [%r0],%r1               ;8timer0 interrupt priority level 3
```

Reset the interrupt factor flag.

```
xld.w    %r0,0x40285
xld.w    %r1,0x01
ld.b     [%r0],%r1               ;8timer0 interrupt flag reset
```

Enable the 8-bit timer interrupt.

```
xld.w    %r0,0x40275
xld.w    %r1,0x01
ld.b     [%r0],%r1               ;8timer0 interrupt enable
```

(5) Start the 8-bit timer.

```
xld.w    %r0,0x40160
xld.w    %r1,0x1
ld.b     [%r0],%r1               ;clock out off,preset,start
```

(6) Processing when interrupt is generated

Shown below is the simplest interrupt routine, which saves R1 only and clears the interrupt factor flag.

```
TIME_INT:
   pushn    %r1
   xld.w    %r1,0x40285
   xld.w    %r0,0x01
   ld.b     [%r1],%r0               ;8timer0 interrupt flag reset
   popn     %r1
   reti
```

The interrupt factor flag is not automatically cleared by an interrupt. It must be cleared with an interrupt routine to avoid generating the same interrupt again.

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 melody33 middleware as an example. Note that the E0C33A104's 16-bit timer significantly differs in functionality from that of the E0C33208.

● Interrupt settings

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

Vector section

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

      .code

      .word    RESET
      .word    RESERVED
      .word    RESERVED
      .word    RESERVED
      .word    ZERODIV
      .word    RESERVED
      .word    ADDRERR
      .word    NMI
      .word    RESERVED
      .word    RESERVED
      .word    RESERVED
      .word    RESERVED
      .word    SOFTINT0
      .word    SOFTINT1
      .word    SOFTINT2
      .word    SOFTINT3
      .word    INT0
      .word    INT1
      .word    INT2
               |
      .word    INT28
      .word    INT29
      .word    INT30                                                    (1)
      .word    INT31
      .word    INT32
      .word    INT33
```

(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

```
      .lcomm   MDY_PSR          0x4

      .global mdyIntOff
mdyIntOff:                                                             (1)
      xld.w    %r4,MDY_PSR
      ld.w     %r5,%psr         // save %psr and IE disable
      ld.w     [%r4],%r5
      ld.w     %r4,0
      ld.w     %psr,%r4
      ret
      .global mdyIntOn
mdyIntOn:                                                              (2)
      xld.w    %r4,MDY_PSR
      ld.w     %r5,[%r4]
      ld.w     %psr,%r5         // 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 melody33 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 MON33 middleware as an example.

● Asynchronous communications using an external clock

The following example is an assembly source excerpted from mon33\src\m3s_sci.s. In this example, communications are controlled by polling rather than by using interrupts.

Initialize routine

```
#ifdef SIO0
      #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
#else
      #define  STDR     0x000401e5     ;transmit data register(ch1)
      #define  SRDR     0x000401e6     ;receive data register(ch1)
      #define  SSR      0x000401e7     ;serial status register(ch1)
      #define  SCR      0x000401e8     ;serial control register(ch1)
      #define  SIR      0x000401e9     ;IrDA control register(ch1)
      #define  PIO_SET  0x70           ;port function register
#endif

#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

      .code
;*******************************************************************************
;
;     void m_io_init()
;              serial port initial function
;
;*******************************************************************************
      .global  m_io_init
m_io_init:
      ld.w     %r0,SIR_SET             ;1/16 mode                          (1)
      xld.b    [SIR],%r0               ;SIR set
      ld.w     %r0,SCR_SET                                                 (2)
      xld.b    [SCR],%r0               ;SCR set(#SCLK input 1.843MHz)
      xld.w    %r0,PIO_SET                                                 (3)
      xld.b    [PIO],%r0               ;IO port set
      xld.w    %r0,SCR_EN|SCR_SET                                          (4)
      xld.b    [SCR],%r0               ;SCR set
      ret
```

(1) Selecting the division ratio
Set the division ratio of the sampling clock to 1/16.

```
      ld.w     %r0,SIR_SET       ;1/16 mode
      xld.b    [SIR],%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 MON33 communications, a 1.843 MHz external clock is fed from DMT33MON (115,200 bps).

```
      ld.w     %r0,SCR_SET
      xld.b    [SCR],%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    %r0,PIO_SET
      xld.b    [PIO],%r0         ;IO port set
```

(4) Enabling transmit/receive
Enable transmit/receive operations.

```
      xld.w    %r0,SCR_EN|SCR_SET
      xld.b    [SCR],%r0         ;SCR set
```

Transmit routine

```
;*****************************************************************************
;
;     void m_snd_1byte( sdata )
;               1 byte send function
;                 IN : uchar sdata (R12)  send data
;
;*****************************************************************************
      .global  m_snd_1byte
m_snd_1byte:
      pushn    %r3               ;save r3-r0
snd000:
      xbtst    [SSR],0x1         ;TDBE1(bit1) == 0(full) ?                (1)
      jreq     snd000            ;if full, jp snd000
      xld.b    [STDR],%r12       ;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    [SSR],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 R12.

```
   xld.b    [STDR],%r12       ;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
rcv000:
      xbtst    [SSR],0x0         ;RDBF1(bit0) == 0(empty) ?               (1)
      jreq     rcv000            ;if empty, jp rcv000

      ld.w     %r10,0x0                                                   (2)
      xbtst    [SSR],0x4         ;FER1(bit4) == 0 ?
      jreq     rcv010
      xbclr    [SSR],0x4         ;FER1(bit4) 0 clear
      ld.w     %r10,0x1          ;return 1
rcv010:
      xbtst    [SSR],0x3         ;PER1(bit3) == 0 ?
      jreq     rcv020
      xbclr    [SSR],0x3         ;PER1(bit3) 0 clear
      ld.w     %r10,0x2          ;return 2
rcv020:
      xbtst    [SSR],0x2         ;OER1(bit2) == 0 ?
      jreq     rcv030
      xbclr    [SSR],0x2         ;OER1(bit2) 0 clear
      ld.w     %r10,0x3          ;return 3
rcv030:
      xld.b    %r0,[SRDR]        ;read data                               (3)
      xld.b    [m_rcv_data],%r0  ;read data set
      popn     %r3               ;restore r3-r0
      ret
```

(1) Wait for receive
Wait until receive data is placed in the buffer.

```
rcv000:
   xbtst    [SSR],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     %r10,0x0
   xbtst    [SSR],0x4          ;FER1(bit4) == 0 ?
   jreq     rcv010
   xbclr    [SSR],0x4          ;FER1(bit4) 0 clear
   ld.w     %r10,0x1           ;return 1
rcv010:
   xbtst    [SSR],0x3          ;PER1(bit3) == 0 ?
   jreq     rcv020
   xbclr    [SSR],0x3          ;PER1(bit3) 0 clear
   ld.w     %r10,0x2           ;return 2
rcv020:
   xbtst    [SSR],0x2          ;OER1(bit2) == 0 ?
   jreq     rcv030
   xbclr    [SSR],0x2          ;OER1(bit2) 0 clear
   ld.w     %r10,0x3           ;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,[SRDR]         ;read data
   xld.b    [m_rcv_data],%r0   ;read data set
```

● Asynchronous communications using an internal clock

The following example is an assembly source file excerpted from mon33\dmt33001\m3s_sci.s. 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 E0C33A104, it may be used in the same way as for the E0C33208, except that no pull-up processing is required.

Initialize routine

```
#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

      .code
;***************************************************************************
;
;     void m_io_init()
;              serial port initial function
;
;***************************************************************************
      .global  m_io_init
m_io_init:
      ld.w     %r0,0x00                                                  (1)
      xld.b    [SIR1],%r0             ;SIR1 set
      ld.w     %r0,0x03                                                  (2)
      xld.b    [SCR1],%r0             ;SCR1 set
      xld.w    %r0,0x30                                                  (3)
      xld.b    [PIO],%r0              ;IO port set
      xld.w    %r0,0xff                                                  (4)
      xld.b    [IOU],%r0              ;pull up set
      xld.w    %r0,0x80                                                  (5)
      xld.b    [P8TS3],%r0            ;P8TS3 set
```

```
xld.w    %r0,0x7                ;38400bps                    (6)
xld.b    [RLD3],%r0             ;RLD3 set
ld.w     %r0,0x07                                            (7)
xld.b    [PT3],%r0              ;PT3 set
xld.w    %r0,0xc3                                            (8)
xld.b    [SCR1],%r0             ;SCR1 set
ret
```

(1) Selecting the division ratio
Set the division ratio of the sampling clock to 1/16.

```
ld.w     %r0,0x00
xld.b    [SIR1],%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).

```
ld.w     %r0,0x03
xld.b    [SCR1],%r0         ;SCR1 set
```

(3) Selecting input/output pin functions
Set the pins shared with I/O ports for serial interface mode.

```
xld.w    %r0,0x30
xld.b    [PIO],%r0          ;IO port set
```

(4) Setting pull-ups (E0C33A104)
Enable pull-ups for the serial interface input pins. This processing is used for the E0C33A104 but is not required for the E0C33208. For real-world applications, we recommend connecting pull-up resistors external to the chip regardless of microcomputer type.

```
xld.w    %r0,0xff
xld.b    [IOU],%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 E0C33208, you can also select 1/1 (0x40146•D3 = 1).

```
xld.w    %r0,0x80
xld.b    [P8TS3],%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 MDT33001 (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    %r0,0x7            ;38400bps
xld.b    [RLD3],%r0         ;RLD3 set
```

(7) Starting the 8-bit timer
Start the 8-bit timer.

```
ld.w     %r0,0x07
xld.b    [PT3],%r0          ;PT3 set
```

(8) Enabling transmit/receive
Enable transmit/receive operations.

```
xld.w    %r0,0xc3
xld.b    [SCR1],%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 \cc33\sample\drv33208\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 cc33\sample\drv33208\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 About IDMA Settings

E0C33208 provides a function that, during the boot process, allows the contents of memory to be transferred by IDMA before vector fetch and program execution (OTP/internal ROM emulation mode, with EA10MD pins set to "01"). The following section gives an example of using this function to transfer the contents of external ROM to high-speed SRAM (used for internal ROM emulation), with booting performed from there. The source code can be found in cc33\sample\dmt33005pd.

● Using OTP DMA

IDMA table definition [m3s_otp.s]

```
;1st word for IDMA ch0
#define LNKEN0  0x80000000    ;IDMA link enable
#define LKCHN0  0x01000000    ;IDMA link field is ch1
#define TC0     0x00000001    ;256KB is 0x20000 times with half word
#define BLKLEN0 0x00000000
;2nd word for IDMA ch0
#define DINTEN0 0x00000000    ;DMA end interrupt is disable
#define DATSIZ0 0x40000000    ;half word
#define SRCINC0 0x30000000    ;address increment
#define SRADR0  0x00c003b8    ;source address is 0xc003b8(0x48126)
;3rd word for IDMA ch0
#define DMOD0   0x40000000    ;successive transfer mode
#define DSINC0  0x30000000    ;address increment
#define DSADR0  0x00048126    ;destination address
;1st word for IDMA ch1
#define LNKEN1  0x00000000    ;IDMA link disable
#define LKCHN1  0x00000000    ;IDMA link field is noting
#define TC1     0x00020000    ;256KB is 0x20000 times with half word
#define BLKLEN1 0x00000000
;2nd word for IDMA ch1
#define DINTEN1 0x00000000    ;DMA end interrupt is disable
#define DATSIZ1 0x40000000    ;half word
#define SRCINC1 0x30000000    ;address increment
#define SRADR1  0x00200000    ;source address is external flash 0x200000
;3rd word for IDMA ch1
#define DMOD1   0x40000000    ;successive transfer mode
#define DSINC1  0x30000000    ;address increment
#define DSADR1  0x00c00000    ;destination address is internal RAM 0x80000
;emulation ROM size setting
#define A10IR   0x4037        ;internal ROM size 256KB 16 bit wait 7 output disable 3.5
.code
      .word LNKEN0|LKCHN0|TC0|BLKLEN0
      .word DINTEN0|DATSIZ0|SRCINC0|SRADR0
      .word DMOD0|DSINC0|DSADR0
      .word LNKEN1|LKCHN1|TC1|BLKLEN1
      .word DINTEN1|DATSIZ1|SRCINC1|SRADR1
      .word DMOD1|DSINC1|DSADR1
      .half A10IR
```

IDMA uses channels 0 and 1. Channel 0 is used to set the size of the internal ROM, while channel 1 is used to perform the actual transfer of memory contents. Create an IDMA table and set its position using a linker command file as described below.
(dmt33005pd.cm)

```
-code 0x0c003a0{ m3s_otp.o }; set absolute code  section m3s_otp.o for DMA ch0
```

When reset, the IDMA table begins with 0xc003a0.

(1) Setting IDMA channel 0

Immediately after a reset, channel 0 transfers 16 bits of data once. The source address of the transfer is 0xc003b8, and the data transferred is ".half A10IR = 0x4037". The transfer destination is the address 0x48126 (area 10–9 setup register) in the BCU. The size of the internal ROM to be emulated is thereby set to 256 KB, corresponding to maximum RAM for the EPOD33001.

```
;1st word for IDMA ch0
#define LNKEN0  0x80000000    ;IDMA link enable
#define LKCHN0  0x01000000    ;IDMA link field is ch1
#define TC0     0x00000001    ;256KB is 0x20000 times with half word
#define BLKLEN0 0x00000000
;2nd word for IDMA ch0
#define DINTEN0 0x00000000    ;DMA end interrupt is disable
#define DATSIZ0 0x40000000    ;half word
#define SRCINC0 0x30000000    ;address increment
#define SRADR0  0x00c003b8    ;source address is 0xc003b8(0x48126)
;3rd word for IDMA ch0
#define DMOD0   0x40000000    ;successive transfer mode
#define DSINC0  0x30000000    ;address increment
#define DSADR0  0x00048126    ;destination address
   |
#define A10IR   0x4037        ;internal ROM .. 256KB 16 bit wait 7 out. disable 3.5
.code
   .word LNKEN0|LKCHN0|TC0|BLKLEN0
   .word DINTEN0|DATSIZ0|SRCINC0|SRADR0
   .word DMOD0|DSINC0|DSADR0
            |
   .half A10IR
```

When transfer across channel 0 is completed, IDMA channel 1 is prompted to start transfer by a link function.

(2) Setting IDMA channel 1

Channel 1 is set to transfer 16-bit data from 0x200000 (flash memory area in the DMT33005PD) to 0xc00000 (internal ROM area of the EPOD33001) 128K times.

```
;1st word for IDMA ch1
#define LNKEN1  0x00000000    ;IDMA link disable
#define LKCHN1  0x00000000    ;IDMA link field is noting
#define TC1     0x00020000    ;256KB is 0x20000 times with half word
#define BLKLEN1 0x00000000
;2nd word for IDMA ch1
#define DINTEN1 0x00000000    ;DMA end interrupt is disable
#define DATSIZ1 0x40000000    ;half word
#define SRCINC1 0x30000000    ;address increment
#define SRADR1  0x00200000    ;source address is external flash 0x200000
;3rd word for IDMA ch1
#define DMOD1   0x40000000    ;successive transfer mode
#define DSINC1  0x30000000    ;address increment
#define DSADR1  0x00c00000    ;destination address is internal RAM 0xc00000
                 |
   .word LNKEN1|LKCHN1|TC1|BLKLEN1
   .word DINTEN1|DATSIZ1|SRCINC1|SRADR1
   .word DMOD1|DSINC1|DSADR1
```

For this reset-time IDMA only, the 0xc00000 area is accessed on #CE10EX (external) as the source and #CE10IN as the destination. Other addresses are accessed the same way as for ordinary IDMA.

(3) Execution

After IDMA transfer is completed, the CPU boots from 0xc00000 and executes the transferred program.

3.7 Setting Up HSDMA

This section explains how to set up HSDMA, using an example program that transfers data to PWM (16-bit timer) for audio output.

● DMA transfer to 16-bit timer

The example program shown below uses HSDMA channel 3 and 16-bit timer 5 as a trigger to transfer data from RAM to 16-bit timer 1.

Initial settings

```
                                |
;; PWM output timer
#define PRESC16_1           0x40148  ; 16bit Timer 1, Prescaler
#define TMCTRL16_1          0x4818e  ; 16bit Timer 1
#define COMPARE_A16_1       0x48188
#define COMPARE_B16_1       0x4818a
#define TIMER16_1_IMASK     0x40272
#define P2_FUNCTION_SELECT  0x402d8  ; Function select output port

;; trigger timer
#define SPK_INTR_LEVEL_0    0x04
#define PRESC16_5           0x4014c  ; 16bit Timer 5, Prescaler
#define TMCTRL16_5          0x481ae  ; 16bit Timer 5
#define COMPARE_A16_5       0x481a8
#define COMPARE_B16_5       0x481aa
#define TIMER16_5_IMASK     0x40274
#define TIMER16_5_IFLAG     0x40284
#define TIMER16_5_ILEVEL    0x40268  ; Upper 4-bits

;; HSDMA
#define HSDMA_IMASK         0x40271
#define HSDMA_IFLAG         0x40281
#define HSDMA3_IFLAG_CLR    0x08
#define HSDMA_01_ILEVEL     0x40263
#define HSDMA_23_ILEVEL     0x40264
#define HSDMA_23_TRIGGER    0x40299
#define HSDMA3_ENABLE       0x4825c
#define HSDMA3_TFLAG        0x4825e
#define HSDMA3_TRANSFER     0x48250

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
#define PRESC16_3           0x0004014a  ; 16bit Timer 3, Prescaler
#define TMCTRL16_3          0x0004819e  ; 16bit Timer 3
#define COMPARE_A16_3       0x00048198
#define COMPARE_B16_3       0x0004819a
#define TIMER16_3_IMASK     0x00040273
#define TIMER16_3_IFLAG     0x00040283
#define TIMER16_3_ILEVEL    0x00040267  ; Upper 4-bits

                            |
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; void SpkOpen_0(BYTE *SpkParams, int ReloadValue)
;;;  dose not destroy r12 - r15
;;;
SpkOpen_0:

    ; init HSDMA triger timer
        ld.w    %r5,0x00                                              (1)
        xld.b   [TMCTRL16_5],%r5      ; Timer     Stop
        xld.b   [PRESC16_5],%r5       ; Prescaler Stop

        xld.w   %r4,TIMER16_5_IMASK
        bclr    [%r4],7               ; disable comparison A intr.
        bclr    [%r4],6               ; disable comparison B intr.

        xld.w   %r5,0xf0
        xld.b   [TIMER16_5_IFLAG],%r5 ; clear comparison A B factor flags

        xld.h   [COMPARE_B16_5],%r13  ; set compare B value (ReloadValue)
```

```
    ; init HSDMA
        ld.w    %r5,0                                                    (2)
        xld.h   [HSDMA3_ENABLE],%r5    ; HSDMA Ch3 disable (stop)

        ld.w    %r5,HSDMA3_IFLAG_CLR
        xld.b   [HSDMA_IFLAG],%r5      ; clear DMA Ch3 interrupt flag clear
        xbset   [HSDMA_IMASK],3        ; enable DMA Ch3 end interrupt

        xld.w   %r4,HSDMA_23_ILEVEL
        ld.b    %r5,[%r4]
        and     %r5,0x0f               ; mask lower 4bit
        ld.w    %r6,0x4                ; HSDMA Ch3 interrupt level = 4
        sll     %r6,0x04               ; level is upper 4bit
        or      %r5,%r6
        ld.b    [%r4],%r5              ; set interrupt level

        ld.w    %r5,8                                                    (3)
        sll     %r5,4
        xld.b   [HSDMA_23_TRIGGER],%r5 ; HSDMA trigger is 16bit timer
                                       ; Ch.5 compare B
                                  |

                                  |
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; int SpkNext(BYTE *SpkParams)
;;;  interrupt routine
;;;  dose not destroy r9,r10 r12 - r15
;;;  In DMA version, change to control register
;;;     r9  : (*Length)
;;;     r10 : Length
;;;     r12 : SpkParams
;;;
SpkNext:
        xcall   QueueRead           ; QueueRead dose not destroy r12 - r15
        cmp     %r10,0x00           ; r9:*Length, r10:Length, r11:*Buffer
        jreq    RetSpkNext          ; if (Failed) QueueEmpty

                                  |

    ; Write HSDMA register
        xld.w   %r4,HSDMA3_TRANSFER ; start of dma transfer register            (4)

        ld.w    %r6,%r10
        xoor    %r6,%r6,0x80000000
        ld.w    [%r4]+,%r6          ; Length and address mode dual

        ld.w    %r7,%r11            ; source address (buffer pointer)
        xoor    %r7,%r7,0x70000000  ; addr increment, harf word transfer
        ld.w    [%r4]+,%r7          ; for 10bit PWM

        xld.w   %r8,COMPARE_A16_1   ; 16bit timer compareA register
        ld.w    [%r4]+,%r8          ; dest address, single

    ; Enable HSDMA
        ld.w    %r5,1
        ld.h    [%r4],%r5           ; HSDMA enable (start)  HSDMA3_ENABLE

RetSpkNext:
        ret
```

(1) Initializing the HSDMA triggering timer

In this example, 16-bit timer 5 is used as a trigger for HSDMA. Temporarily suspend timer 5 operations and turn off the prescaler clock output for timer 5.

```
; init HSDMA triger timer
   ld.w    %r5,0x00
   xld.b   [TMCTRL16_5],%r5      ; Timer     Stop
   xld.b   [PRESC16_5],%r5       ; Prescaler Stop
```

With the interrupt controller, disable the timer 5 interrupt and clear the interrupt factor flag.

```
xld.w   %r4,TIMER16_5_IMASK
bclr    [%r4],7               ; disable comparison A intr.
bclr    [%r4],6               ; disable comparison B intr.
xld.w   %r5,0xf0
xld.b   [TIMER16_5_IFLAG],%r5 ; clear comparison A B factor flags
```

Set the HSDMA cycle data (e.g. 8k times per second) passed via R13 from the calling routine in the compare B register.

```
xld.h   [COMPARE_B16_5],%r13  ; set compare B value (ReloadValue)
```

(2) Initializing HSDMA interrupt

Always confirm that HSDMA is disabled before setting HSDMA. If set while operating, the register may be read or written to incorrectly.

```
; init HSDMA
   ld.w    %r5,0
   xld.h   [HSDMA3_ENABLE],%r5   ; HSDMA Ch3 disable (stop)
```

With the interrupt controller, clear the the HSDMA channel 3 interrupt factor flag and enable the interrupt.

```
   ld.w    %r5,HSDMA3_IFLAG_CLR
   xld.b   [HSDMA_IFLAG],%r5     ; clear DMA Ch3 interrupt flag clear
   xbset   [HSDMA_IMASK],3       ; enable DMA Ch3 end interrupt
```

Set the interrupt level to 4.

```
   xld.w   %r4,HSDMA_23_ILEVEL
   ld.b    %r5,[%r4]
   and     %r5,0x0f              ; mask lower 4bit
   ld.w    %r6,0x4               ; HSDMA Ch3 interrupt level = 4
   sll     %r6,0x04              ; level is upper 4bit
   or      %r5,%r6
   ld.b    [%r4],%r5             ; set interrupt level
```

(3) Setting HSDMA trigger

Set 16-bit timer 5 for the HSDMA channel 3 trigger.

```
   ld.w    %r5,8
   sll     %r5,4
   xld.b   [HSDMA_23_TRIGGER],%r5 ; HSDMA trigger is 16bit timer
                                  ; Ch.5 compare B
```

(4) Setting HSDMA transfer conditions

Set the DMA transfer count (data count) passed via R10 from the calling routine. Set transfer mode to dual-address transfer.

```
; Write HSDMA register
   xld.w   %r4,HSDMA3_TRANSFER   ; start of dma transfer register

   ld.w    %r6,%r10
   xoor    %r6,%r6,0x80000000
   ld.w    [%r4]+,%r6            ; Length and address mode dual
```

Set the transfer source address, increment mode, and 16-bit data size. The transfer source address is the start address of the buffer (RAM) in which PWM data has been prepared.

```
   ld.w    %r7,%r11              ; source address (buffer pointer)
   xoor    %r7,%r7,0x70000000    ; addr increment, harf word transfer
   ld.w    [%r4]+,%r7            ; for 10bit PWM
```

Set the compare A register for PWM 16-bit timer as the transfer destination. Select single transfer mode, in which one data unit is transferred by one DMA operation.

```
   xld.w   %r8,COMPARE_A16_1     ; 16bit timer compareA register
   ld.w    [%r4]+,%r8            ; dest address, single
```

Enable HSDMA.

```
; Enable HSDMA
   ld.w    %r5,1
   ld.h    [%r4],%r5             ; HSDMA enable (start)  HSDMA3_ENABLE
```

DMA transfer is now executed as many times as set at the frequency of the compare B cycle set in 16-bit timer 5.

Interrupt processing at the end of DMA transfer

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; SpkIntr0
;;;  HSDMA transfer end interrupt
;;;
SpkIntr0:
        pushn   %r15                                                (1)
        ld.w    %r0,%ahr
        ld.w    %r1,%alr
        pushn   %r1
        xld.w   %r12,SPK_PARAMS_0       ; SpkParams
        xld.w   %r13,HSDMA_IFLAG        ; IFlagReg                  (2)
        xld.w   %r14,HSDMA3_IFLAG_CLR   ; IFlag clear data
        ld.b    [%r13],%r14             ; clear DMA Ch3 interrupt flag
        ld.w    %r15,[%sp+0x12]         ; OldPSR                    (3)
        xcall   QueueNext
        popn    %r1                                                 (1)
        ld.w    %alr,%r1
        ld.w    %ahr,%r0
        popn    %r15
        reti
```

Register the start address (SpkIntr0) of this handling routine as the vector for HSDMA channel 3. Process the interrupt generated each time a number of HSDMA transfers set is completed.

(1) Saving and restoring registers
At the beginning and end of interrupt processing, save and restore all of R0–R15, AHR and ALR.

```
pushn   %r15
ld.w    %r0,%ahr
ld.w    %r1,%alr
pushn   %r1
    |
popn    %r1
ld.w    %alr,%r1
ld.w    %ahr,%r0
popn    %r15
reti
```

(2) Resetting the interrupt factor flag
Clear the interrupt factor flag.

```
xld.w   %r13,HSDMA_IFLAG        ; IFlagReg
xld.w   %r14,HSDMA3_IFLAG_CLR   ; IFlag clear data
ld.b    [%r13],%r14             ; clear DMA Ch3 interrupt flag clear
```

(3) Processing for the subsequent transfer
Disable HSDMA and restore transfer conditions before starting the next data transfer.

```
ld.w    %r15,[%sp+0x12]         ; OldPSR
xcall   QueueNext
```

3.8 Clock Settings

The E0C33208 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 cc33\sample\drv33208\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 cc33\sample\drv33208\osc of cc33 ver.3 or later (not included in ver.2).

Main routine [demo_osc.c]

```
void 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 DMT33007, 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.

25µs (=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 Other Sample Programs

In addition to previous examples discussed in this chapter, sample programs involving peripheral functions can be found in cc33\sample\drv33208 (drv33a104 for the 33A104). Refer to these examples, if necessary. An electronic version of this manual in Japanese text format can be found in cc33\sample\apnote (ver.3 or later). Although the print version has been significantly altered by the inclusion of descriptions, the sample programs referenced in this manual can be copied via a simple Copy and Paste.

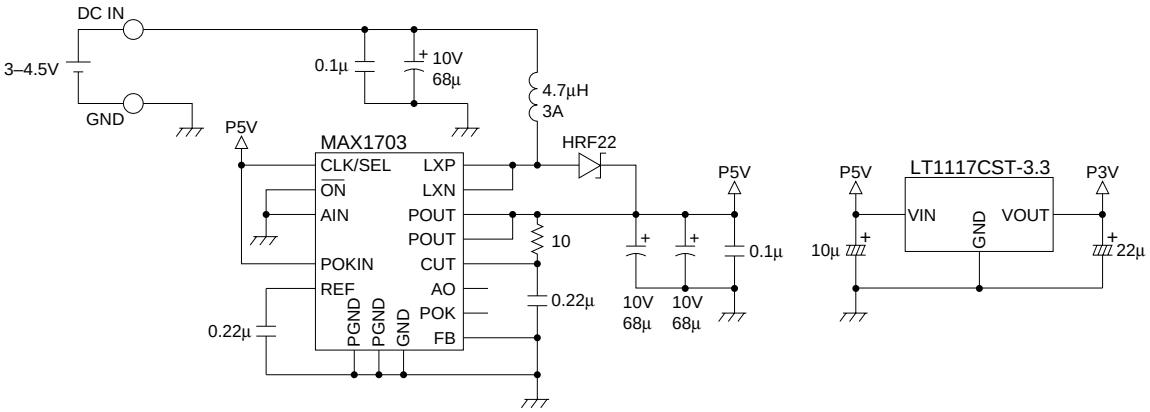
4 THE BASIC E0C33 CHIP BOARD CIRCUIT

This chapter explains the basic circuit design of the E0C33208.

4.1 Power Supply

Here, we'll explain the power supply based on a DC-DC converter, using the DMT33005 circuit as an example.

● DC-DC converter



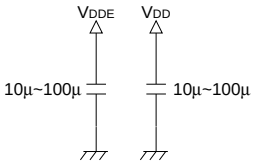
This power supply circuit steps up the 3 to 4.5 V input voltage with a switching regulator to generate a 5 V power supply for the external I/O and memory block, as well as for the analog block. This 5 V power supply is stepped down with a linear regulator to generate a 3.3 V power supply for the CPU core. Because the E0C33208'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 DMT33005. 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 DMT33005 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 DMT33005, 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.

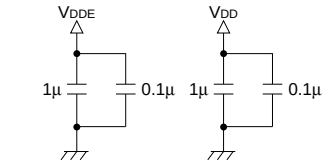
● Decoupling capacitors

Use the following four methods for noise abatement between power supply and ground lines.

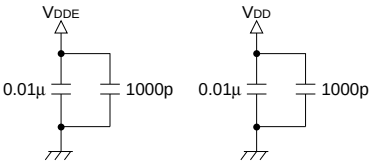
1) Use a circuit board comprised of four or more layers, and provide full-surface GND and full-surface VDD layers.
2) Attach a 100 µF electrolytic capacitor per circuit board. For small circuit boards, attach a 10 µF tantalum capacitor.
3) Attach a 1 µF + 0.1 µF laminated ceramic capacitor to the CPU 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.



Mount for each board



Mount for every one or several IC blocks



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. E0C33208 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.

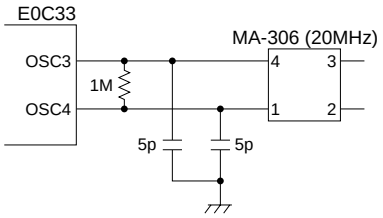
When using double-sided circuit boards, reinforce GND as much as possible to ensure equivalent GND potentials at each location. To prevent voltage fluctuations, use a decoupling capacitor to reinforce power supply lines on each block.

4.2 Oscillation Circuit

The following section discusses oscillation circuits, referring to the DMT33005 and EPOD33001 as examples.

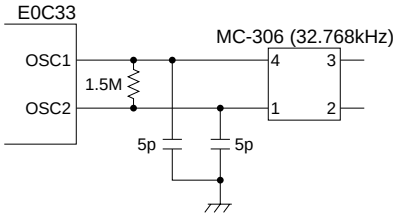
● 20 MHz resonator

This example applies to the DMT33005 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.



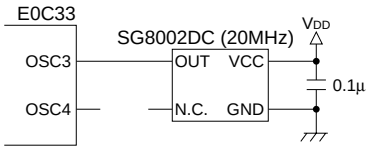
● 32 kHz resonator

This example applies to the DMT33005 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.



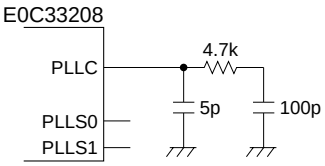
● 20 MHz oscillator

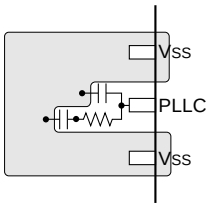
This example applies to the EPOD33001 when the oscillator's output clock is fed to the OSC3 pin. Make sure the voltage level of the input clock is the same as that of the operating clock (V_{DD}) of the CPU core (e.g. 3.3 V). The same applies when an external clock is fed to the OSC1 pin.



● 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 E0C33208 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:

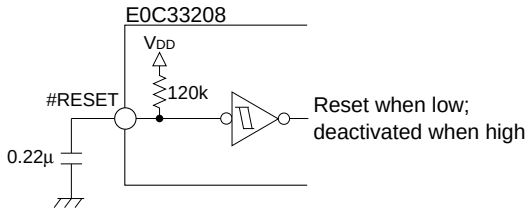
OSC3	PLLS1	PLLS0	X2SPD	Core clock	Bus clock
20 MHz	0	0	1	20 MHz	20 MHz
20 MHz	1	1	0	40 MHz	20 MHz
15 MHz	0	1	0	60 MHz	30 MHz

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.

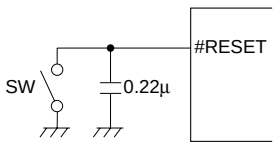
● Reset by an RC network

The E0C33208'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.

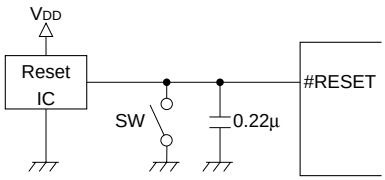
When using the ICD33 for debugging, we recommend attaching a reset switch to your system. When you encounter difficulty connecting the ICD33 and target, this lets you hold down the reset switch while turning on the ICD33, then release the reset switch, ensuring that the ICD33 and target are connected. 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 DMT33008LV (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.

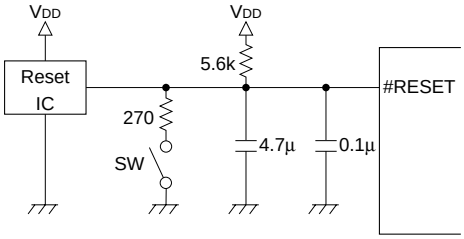
Adding this IC to the above reset circuit results in the following (excerpt from the DMT33008LV circuit):



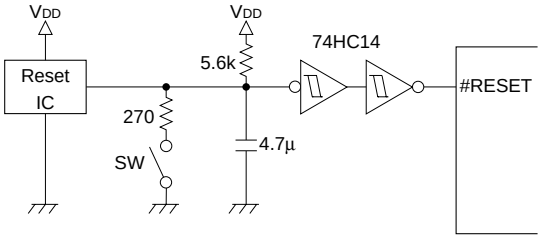
● Protecting reset against noise

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.



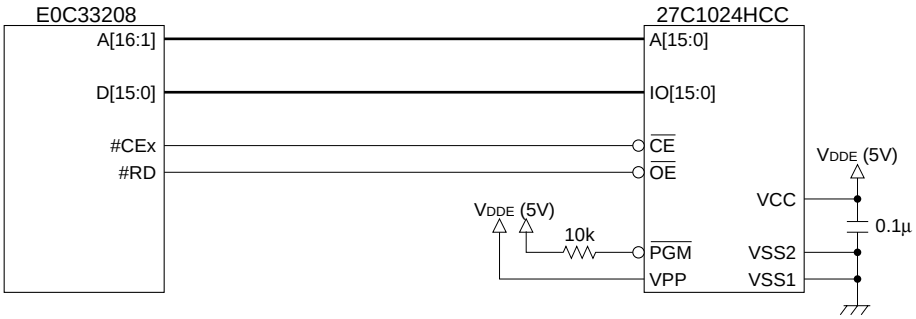
In this example of noise protection, a 74HC14 (Schmitt type inverter) is inserted to drive reset with logic. This renders the reset circuit significantly resistant to noise.

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 DMT33005 and ICD33 as examples, a ROM connection diagram is shown below.

● Connecting x16 ROM



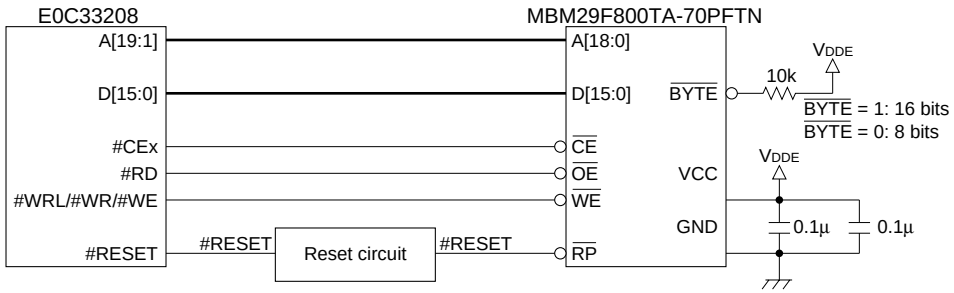
The DMT33005 has a 1M-bit EPROM packaged in a 44-pin PLCC. The E0C33208 I/O and this ROM both operate at 5 V.
When the bus clock speed is 20 MHz, the ROM access time requirements are as follows:
For 2-cycle read with one wait state (bus cycle = 100 ns), ROM access time of 80 ns (or 75 ns for 3.3 V) or 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 DMT33005 as an example, the following is a diagram of a x16-type flash memory connection.

● Connecting x16 flash memory

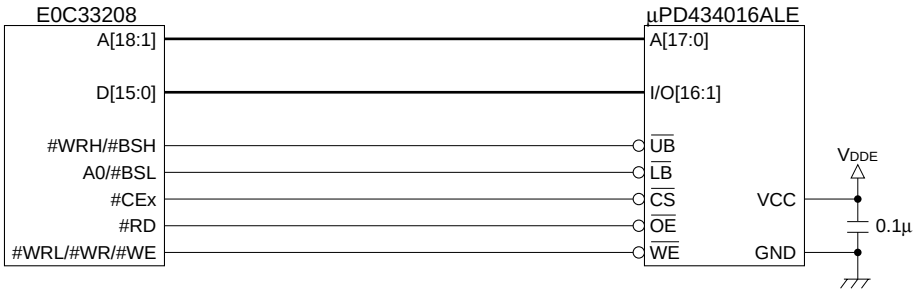
An 8M-bit flash memory in a 48-pin TSOP package is connected directly to the chip. A x8/x16 dual-type flash memory labeled "29F800" is used in a x16 configuration.



4.6 Connecting SRAM

● Connecting x16 SRAM

In the example shown below, one 4M-bit, x16 SRAM is connected to the chip.

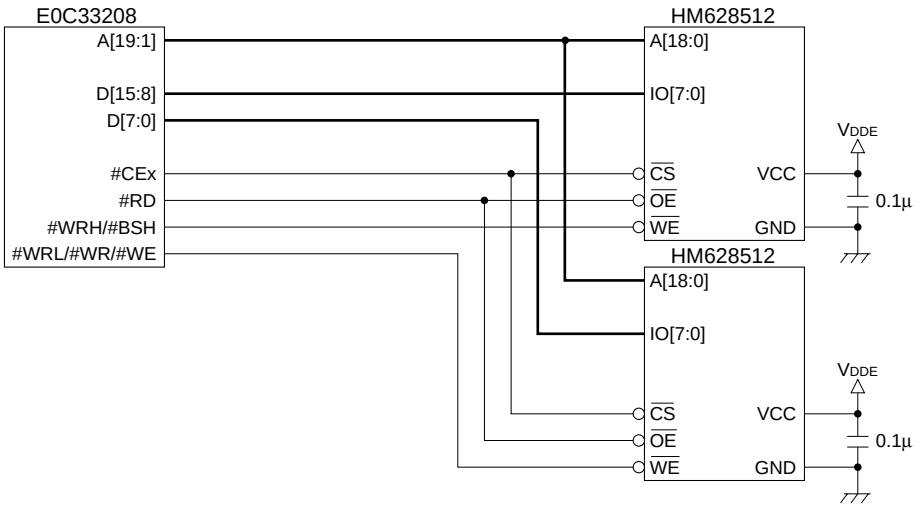


This type of RAM cannot be accessed with a default BCU setting. If BCU is changed to BSL mode, the RAM becomes operational with the wiring shown above. BSL mode is selected by setting D3 at 0x4812E to 1. Setting D3 = 0 selects regular A0 mode.

Note: In the E0C33208 and E0C332L01, BSL mode cannot be used in combination with ICD33 debugging. Use the MON33 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 DMT33005.



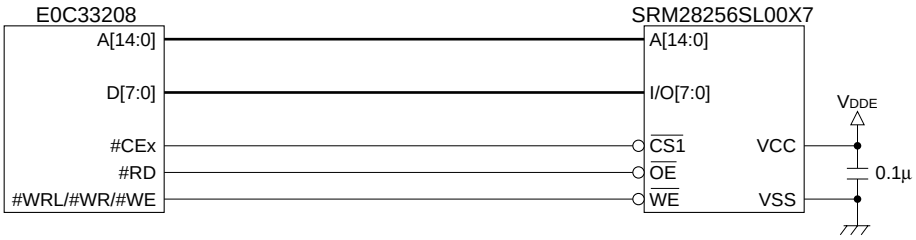
The address, #CE, and #RD outputs may be connected directly to the chip, although the 2-device connection may increase their load capacitance.

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 DMT33005 (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.

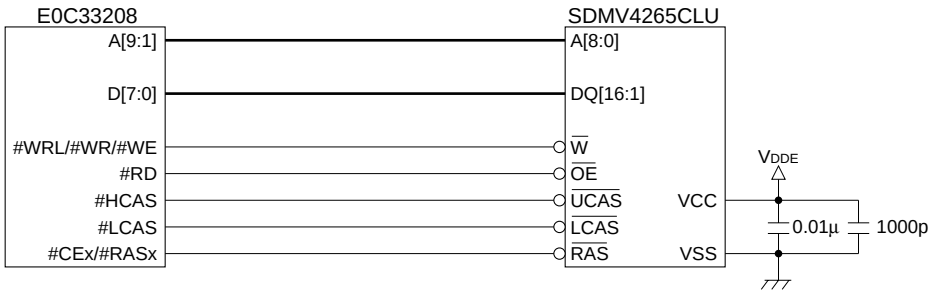


By default, the BCU is set to 16-bit size. Change its setting to 8-bit and set each area's setup register D6 or DE bit to 1.

4.7 Connecting DRAM

Using the DMT33006LV as an example, the following shows a DRAM connection diagram. Note that a DRAM pattern is prepared on the DMT33006LV, but that no DRAMs are yet mounted.

● Connecting 4M-bit, x16 DRAM

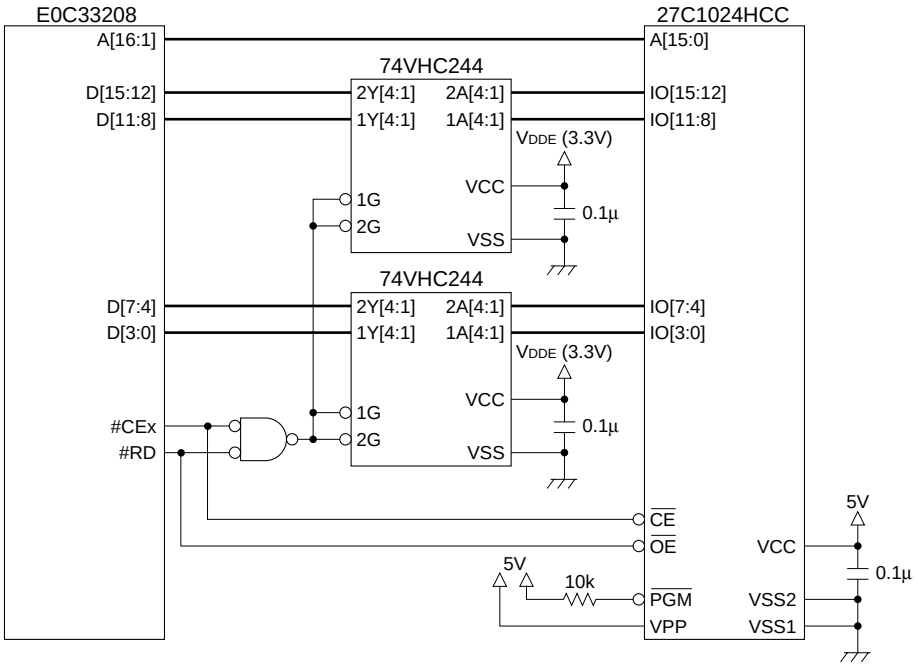


For more information on BCU settings, see Section 3.1, "Setting Up BCU".

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 E0C33208 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 E0C33 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 ICD33 (the CPU, however, is the E0C33A104).

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.

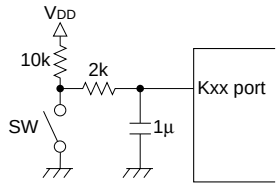
4.9 Ports

● Processing unused I/O (P) ports

By default, the unused I/O ports are set for input. Connect unused ports to VDD or VSS, or switch them for output immediately after booting. Take care that ports connected to VDD or VSS are never set for output.

● Eliminating chattering on input (K, P) ports

Except for K60–K67, the K and P ports are Schmitt inputs with a pull-up resistor of about 120 kΩ, 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 VDD constitutes a rise time of about 10 ms, eliminating several ms of chattering. Turn-on from VDD 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.

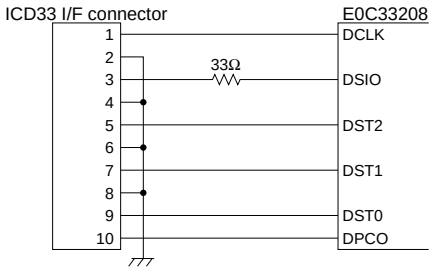
For pins which are not Schmitt inputs, use a 74HC14 or equivalent to eliminate chattering. To determine if a particular pin is a Schmitt input, see the user's manual supplied with each IC. (For the E0C33208/204/202 Technical Manual, see Appendix B, "Pin Characteristics".)

4.10 Connections for Debugging

Using the DMT33005 as an example, this section explains how to connect the ICD33 and the DMT33MON for MON33.

● Connecting the ICD33

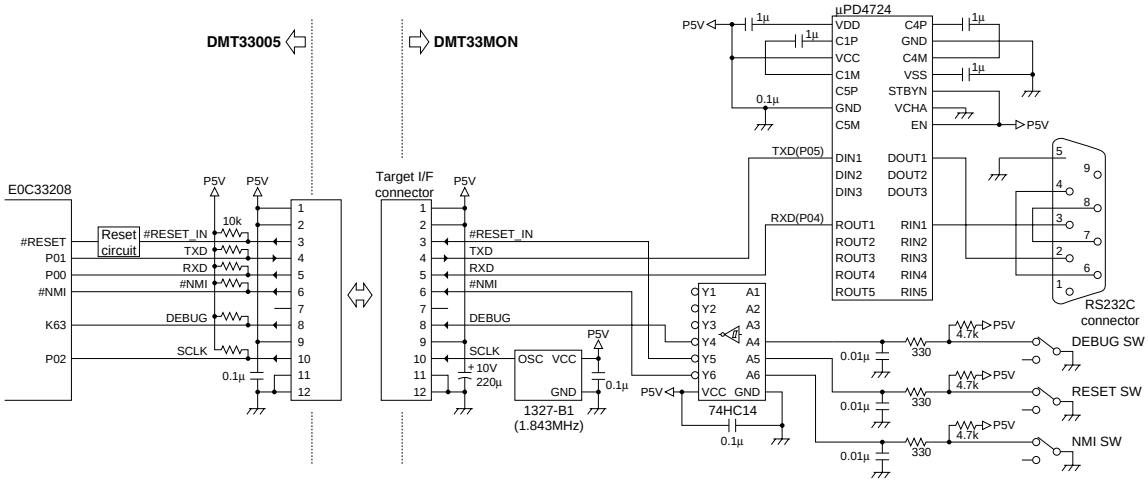
The E0C33208 has six dedicated pins to which a debugger can be connected, including DCLK, DSIO, DST2, DST1, DST0, and DPCO.
Add a 33 Ω resistor in series to DSIO. Use a total of 10 lines to connect to the ICD33, including four additional GND lines.



If the above pattern cannot be laid out on the circuit board, use aerial wiring to connect, without inserting the 33 Ω resistor. The ICD33 will function with this connection.
You can also disable the PC trace function with the ICD33 (by pushing the rightmost DIP switch down) and connect to the ICD33 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.

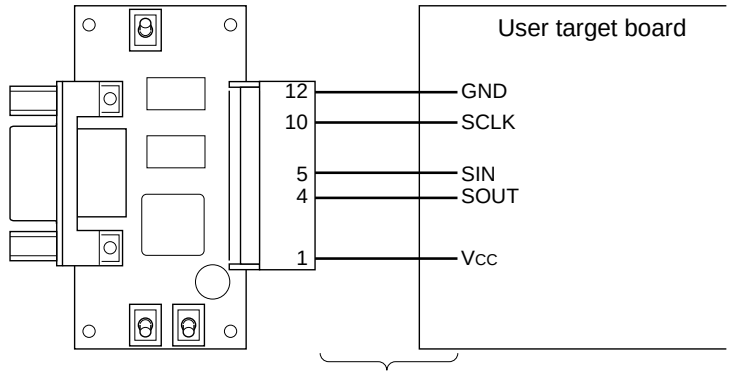
● Connection with the DMT33MON

MON33 uses the following resources: 10K bytes of ROM, 4K bytes of RAM, and one serial interface channel.
The diagram shown below depicts a DMT33MON circuit diagram and an interface component on the target board.



The MDT33005 is connected to the E0C33 to allow use of all DMT33MON 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.



There are five essential pins: SCLK, SIN, SOUT, GND, and VDD. VDD is 5 V for the DMT33MON, and 3.3 V for the DMT33MONLV.

5 SPEAKER OUTPUT AND EXTERNAL ANALOG CIRCUIT USING FINE PWM

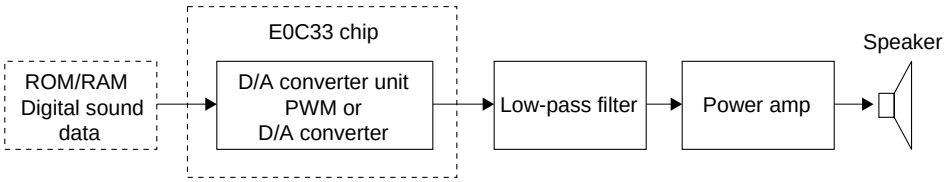
5.1 General Sound Output Circuits Based on Microcomputer

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.



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.

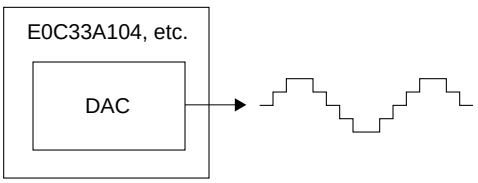
5.1.1 D/A Converter Unit

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

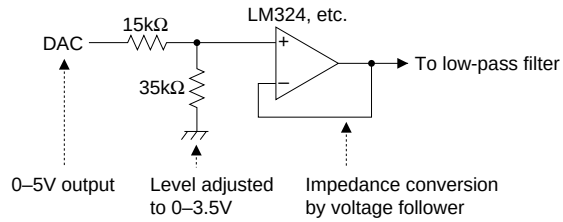
This method uses the DAC incorporated in a microcomputer to output sound.



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.

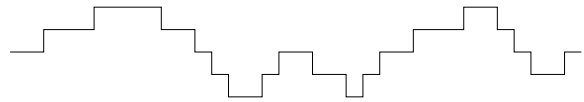


The input voltage is limited by the op amp used. Because an inexpensive CMOS-type op amp (e.g. LM324) is used in this example, the input voltage is divided by resistors to adjust it into the range 0 to 3.5 V. In this case, since a current flows into GND through 15 kΩ + 35 kΩ resistors, care must be taken 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.

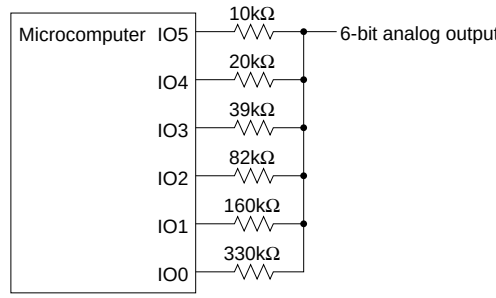


For output with 8 kHz of sampling frequency, for example, write digital data every 1/8000 seconds into the DAC in software. Because the output does not change states until the next data write, the output is in noncontiguous staircase form. This is quantization noise, which degrades audio quality centering 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 E0C33A104 chip's internal 8-bit DAC may be used for audio output, 8-bit resolution is generally considered inadequate for audio quality. The E0C33208 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.

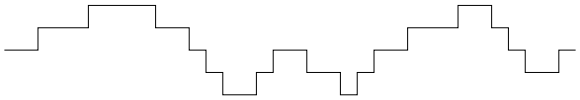
● Conversion by resistor ladder

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.



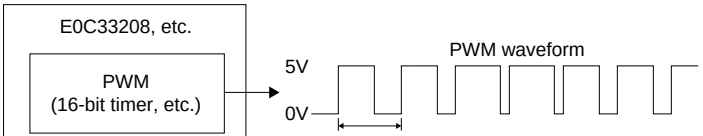
Resistor values selected from the E24 series
10 kΩ and 20 kΩ have a 1% error; others are 5% accurate.

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.

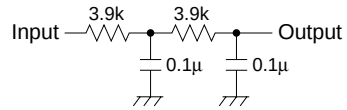
The accuracy of the PWM method is determined by the resolution of the pulse width. To realize 8-bit accuracy, one cycle must be 256 × 80 kHz, requiring a 20 MHz reference clock. The audio output library for the E0C33208 realizes 10-bit accuracy with PWM, providing high audio quality comparable to that of a 10-bit DAC. Normally, 10-bit accuracy requires 1024 × 80 kHz = 80 MHz clock, but the E0C33208 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 E0C33 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

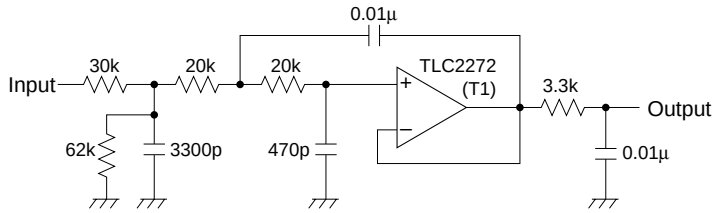
The quantization noise generated during D/A conversion degrades audio quality. To the human ear, this noise appears as roughness in the sound, with shrill high tones. Resolving this problem requires careful design of the low-pass filter unit to eliminate quantization noise. Costs must also be considered.
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

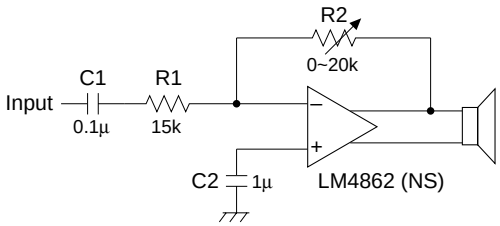
In Seiko Epson's speech and music middleware (e.g. VOX33, TS33, and SOUND33), x2 oversampling technology is used in audio output to significantly reduce software quantization noise, reducing the load on the low-pass filter unit. The result is such that even when using filters above the fourth-order, 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.

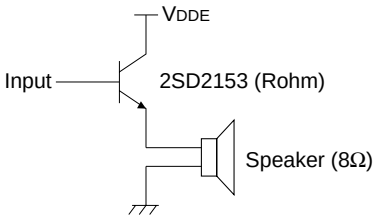
● Example of a power amp



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.

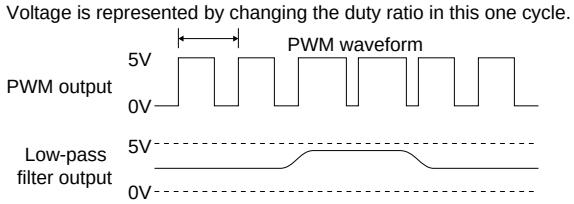
The E0C33A104 by itself is capable only of 8-bit output, using its internal 8-bit DAC. The E0C33208 can produce 8 to 32 kHz, 10 to 15-bit output thanks to its PWM, providing ample capabilities for most applications.

5.3 10-bit D/A Conversion by PWM

The E0C33208 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".)

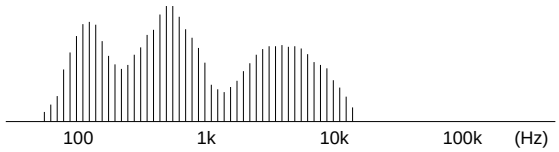
● Differences between PWM and DAC

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.

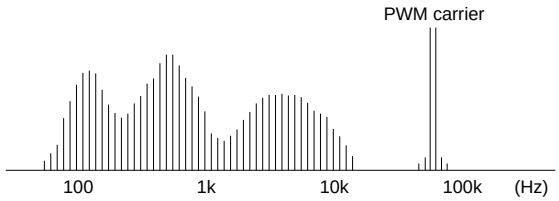


The human ear perceives frequency spectrum as sound rather than waveforms.

Spectrum of DAC output



Spectrum of PWM output

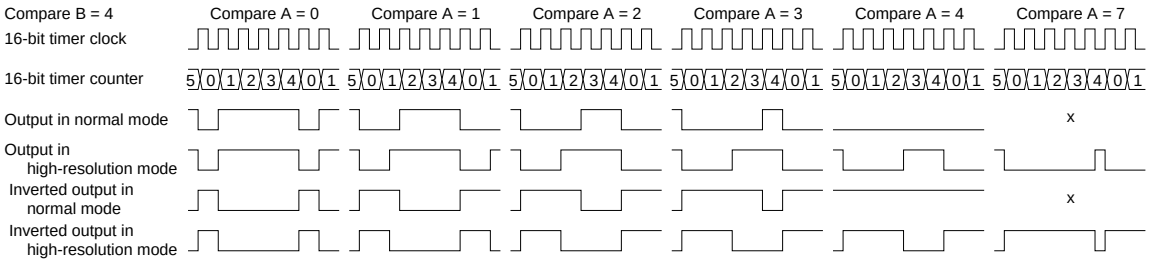


PWM output has significant power near the carrier frequency, but in the same spectrum as that of DAC output in the audible frequency range. Thus, although the output waveforms of PWM and DAC are quite different, both PWM and DAC outputs are perceived as identical by human ears. Because the PWM carrier 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 E0C33208 is 10-bit accurate, so that control of one clock width requires 80 kHz × 1024 = 80 MHz clock. The E0C33208 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 cc33\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

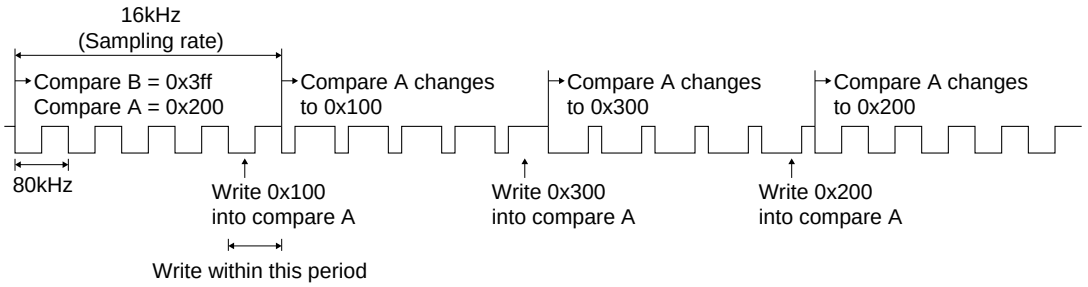
Because the compare data buffer is enabled in (4), compare A data can be written to asynchronously with a count operation.

```
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

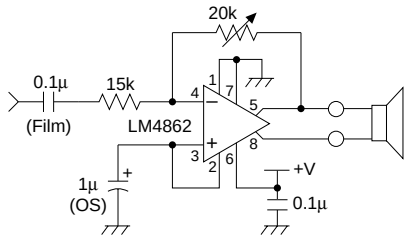
For audio output, set compare B to 80 kHz or higher in terms of cycle and write the data to be D/A converted directly into the compare A data buffer asynchronously every sampling period (8 to 32 kHz).



5.4 Examples of Audio Output Analog Circuits

● Power amp

Shown below is the power amp circuit mounted in the DMT33AMP.



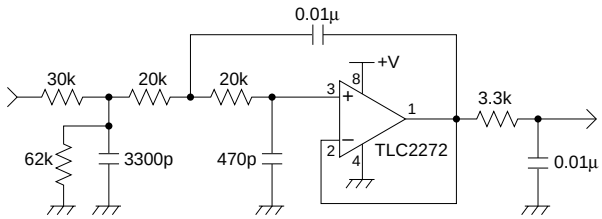
Film capacitors are better than ceramic capacitors as capacitors for signal reception. Inexpensive polyethylene film capacitors may be used without problems. Because ceramic capacitors exhibit minute hysteresis, use of this capacitor type in a circuit in which signal passes directly may result in signal 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 DMT33AMP and DMT33AMP2.

Fourth-order low-pass filter for 8 kHz sampling (DMT33AMP)



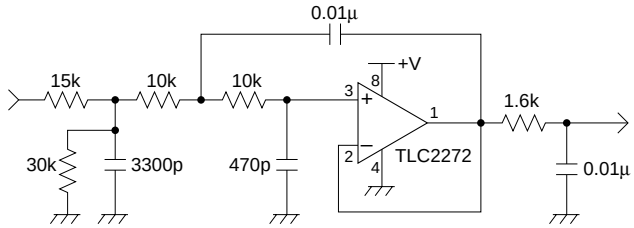
To eliminate 8 kHz sampling quantization noise, choose a cutoff frequency in the range 3.5 kHz to 2.7 kHz. In this filter, the cutoff frequency is set to 3.0 kHz. As the cutoff frequency rises, quantization noise becomes audible at around 3.5 kHz (when using x2 oversampling). The first dividing resistor 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 (DMT33AMP2)

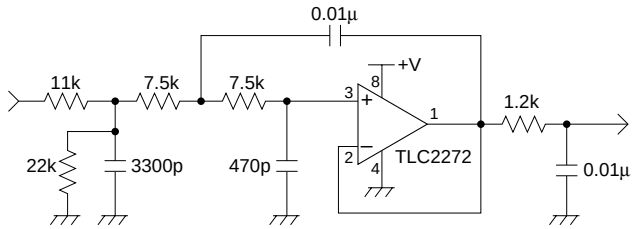


Configured in the same way as the 8 kHz sampling circuit, this filter has a cutoff frequency set to 6.1 kHz. If all resistor values are halved without changing capacitor values, the cutoff frequency doubles while the characteristic curve remains unchanged. The same is true when all capacitor values are 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.

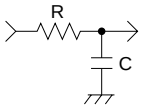
Fourth-order low-pass filter for 22.05 kHz sampling (DMT33AMP2)



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.

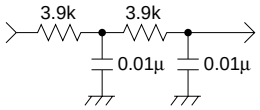
● Low-pass filter comprised of an RC network

The first-order low-pass filter consisting of R and C is configured as shown below. Its cutoff frequency is obtained by calculating $1/(2\pi \times R \times C)$.



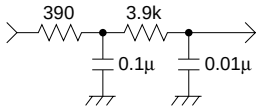
The attenuation factor is 6 dB/oct. When frequency doubles, the waveform is halved. For this reason, audible quantization noise cannot be entirely eliminated. Thus, two such filters are used, with one placed above the other. The configuration creates an effective low-pass filter for cost-priority systems. 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 DMT33AMP).



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 E0C33208 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.

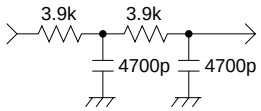
When connecting to the DAC of the E0C33A104, change the 390 Ω resistor in the preceding stage to 150 Ω, since the DAC's output section contains an internal resistor of approximately 250 Ω in series.

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.

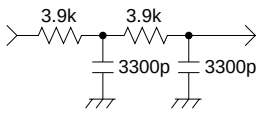
When using a two-high stack of RC networks, take care that the impedance of the following stage is never lower than that of the preceding stage. Characteristics may otherwise become degraded to the point of unusability.

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

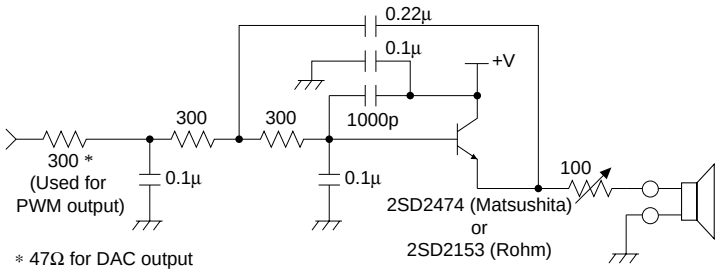


With this circuit, the 0.01 µF capacitor for 8 kHz sampling is nearly divided by 3 to 3300 pF. The cutoff frequency is approximately 6 kHz. If a greater emphasis on high tones is desired, change 3300 pF to 2200 pF. Note that quantization noise will increase.

● 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.

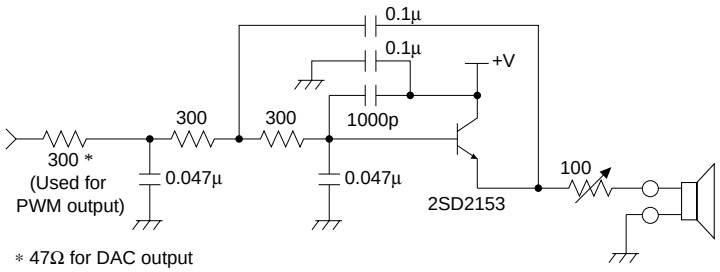
Transistor amp circuit for 8 kHz sampling



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.

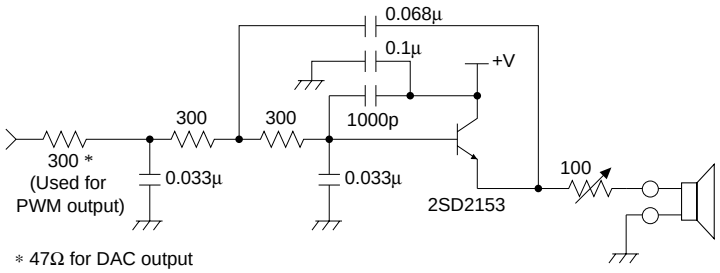
When entering from the DAC of the E0C33A104, change the 300 Ω resistor in the preceding stage to 47 Ω (300 Ω minus the DAC's internal resistor of about 250 Ω). Note that the 0.1 µF capacitor connected to +V is used to decouple the power supply, and that the 1000 pF is used to prevent oscillation. Without 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

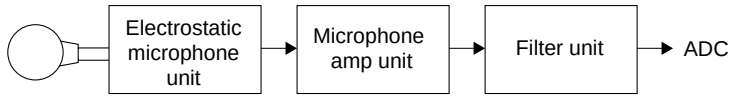


The low-pass filter cutoff frequency is about 8 kHz.

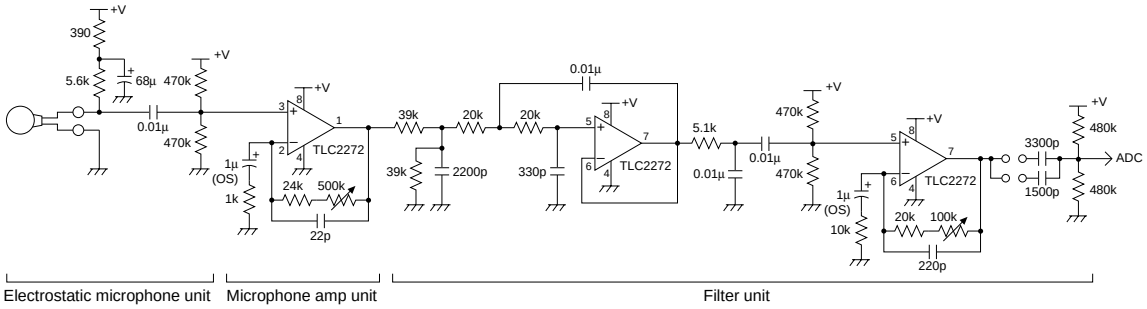
The low-pass filters used here can may be used in combination with the E0C33208 PWM or E0C33A104 DAC.

5.5 Example of a Sound Input Analog Circuit

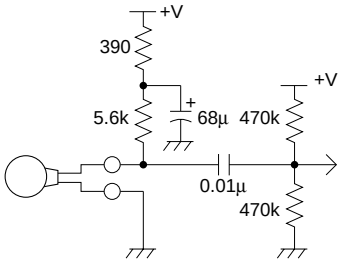
This section explains how to enter sound using an A/D converter.



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 DMT33AMP circuit).

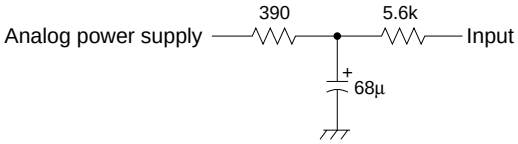


● Electrostatic microphone unit



Electrostatic microphone and AC coupling

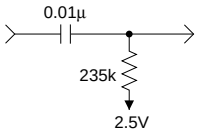
The manufacturer's original recommendation for the 5.6 kΩ resistor inserted in the line-feed power to the electrostatic microphone was originally 1.5 kΩ. This is because the potential difference here constitutes the input signal level; we therefore increased the resistor value to reduce the burden on 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.

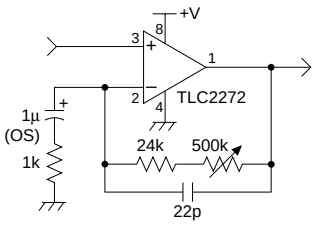
The 0.01 µF capacitor and 470 kΩ resistors, one to GND and one to the power supply, are used for AC coupling to 1/2 power supply voltage, and to cut the DC component as a first-order high-pass filter. The cutoff frequency is approximately 70 Hz, below which frequencies are attenuated.

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



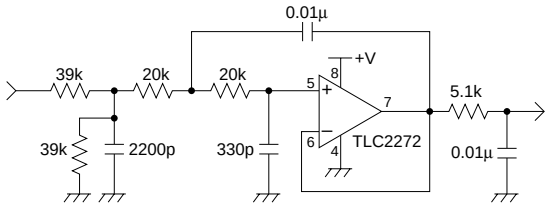
The gain for this AC amplifier may be adjusted in the range of 24-fold to 524-fold using a variable resistor. Combined with the 3.7-fold gain in the electrostatic microphone unit, this amounts to a gain of 90-fold to 2,000-fold. However, because 524-fold is used for experimental purposes, the amp as 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.

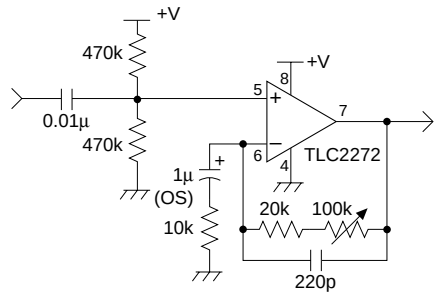
● Filter unit

Fourth-order low-pass filter



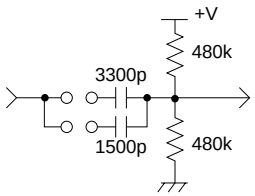
Mounted on the DMT33AMP board is a microphone low-pass filter with 3.5 kHz cutoff, as shown above. This filter cuts unwanted high-frequency components, improving perceived sound quality. The effect is not dramatic, and the filter may be omitted. Here, the amplitude is halved with a dividing resistor, 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

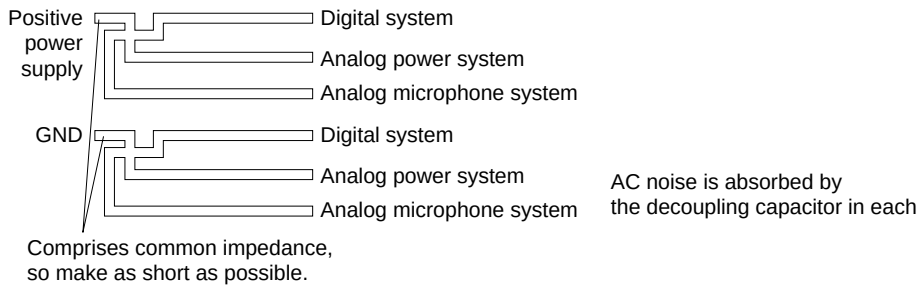


Here, a high-pass filter is used for AC coupling to 1/2 power supply voltage and to cut low tones that adversely affect sound compression. The relationship between capacitor capacitances and cutoff frequencies is shown below.
4800 pF: 250 Hz cutoff
3300 pF: 300 Hz cutoff
1500 pF: 500 Hz cutoff

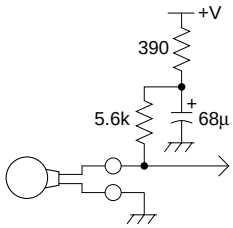
Although the default capacitance for the DMT33AMP is 4800 pF, other capacitances may be tried, depending on the usage environment. For example, the VSX sound compression included in the VOX33 sound compression/expansion middleware may yield better results at 500 Hz, since it is susceptible to DC noise.

● 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 MDT33AMP.



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 DMT33005, 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 DMT33005 + DMT33AMP 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 E0C33208 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.

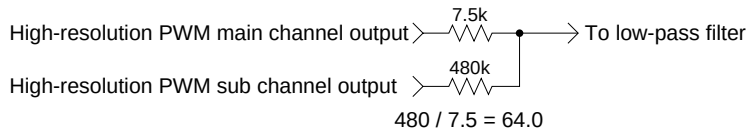
(3) Soft adjust PWM
During PWM output, this technique deploys corrective software processing to produce high-accuracy output, with a linearity error as small as 0.01%.

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.

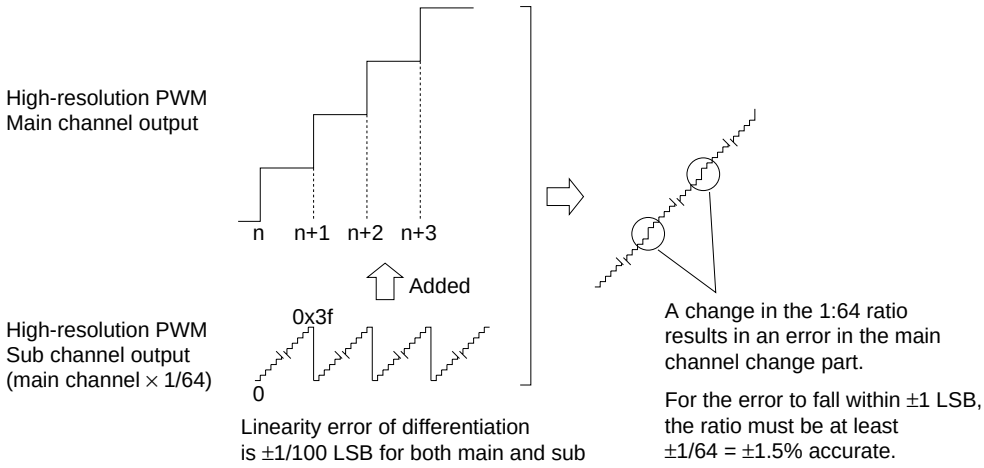


High-resolution PWM provides extremely high differentiation accuracy, with an error of 1/100 LSB or less when actually measured. (Use PLL at x2 or better. Using x1 OSC3 directly as is destroys the duty ratio, making it impossible to obtain this level of differentiation accuracy. For 1-channel high-resolution 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

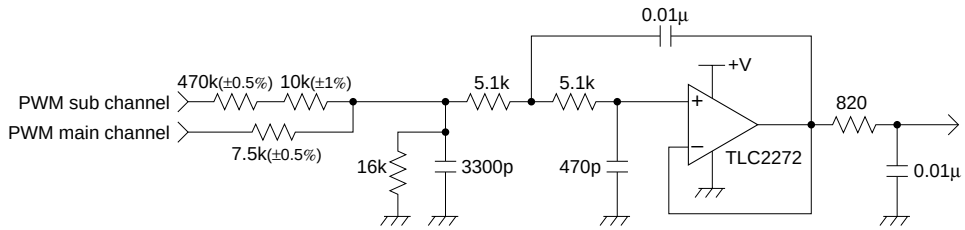


Since a differentiation accuracy of 15 bits ±0.5 LSB more or less applies to 63/64 patterns in which the sub channel changes to other values, audio quality is not degraded as much by the error. Nevertheless, we recommend using resistors with small error values, about 0.5% accuracy, if possible. At worst, 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.

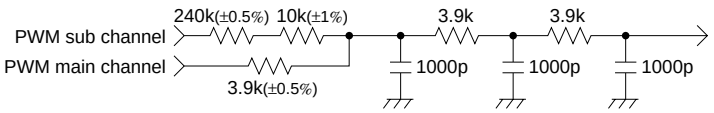
● Circuit example (DMT33AMP3)

Low-pass filter for 32 kHz or higher sampling

Fourth-order op amp block



Third-order RC network



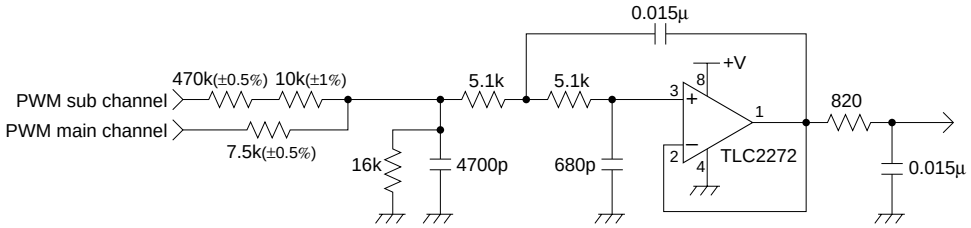
Before the ordinary low-pass filter, add the first-stage synthesizing resistors and connect two-channel PWM outputs. Make sure the ratio of the synthesizing resistors is as close to 64.0-fold as possible (by calculation, within ±0.2% error, from 63.87-fold to 64.13-fold). Use resistance values in the 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.

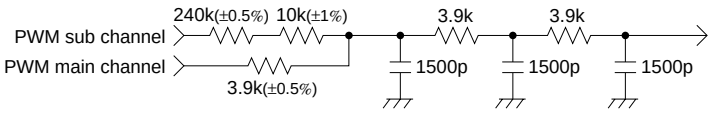
For the circuits shown below, capacitor values have been changed to adjust the cutoff frequency, making the circuits useful for 22.05 kHz sampling and 16 kHz sampling, respectively. In either case, the ratio of the first-stage synthesizing resistors is 1:64.

Low-pass filter for 22.05 kHz sampling

Fourth-order op amp method

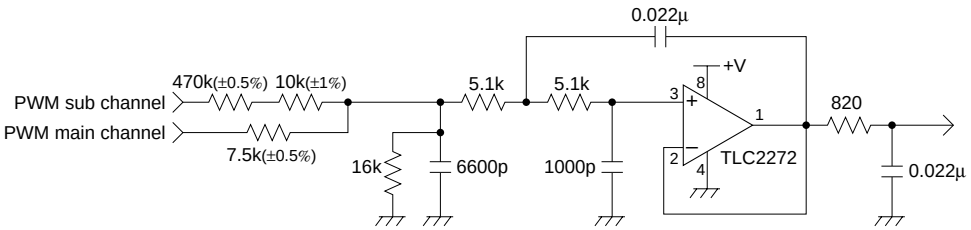


Third-order RC network method

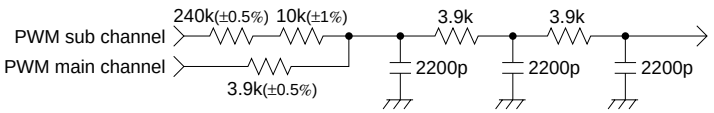


Low-pass filter for 16 kHz sampling

Fourth-order op amp method

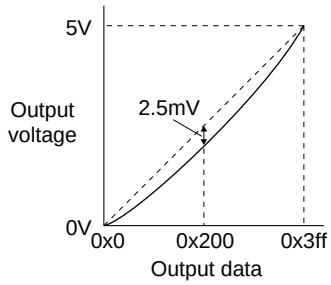


Third-order RC network method



● 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 E0C 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

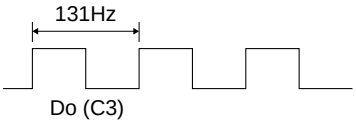
In this section, we discuss producing melody output using PWM and connecting a piezoelectric buzzer.

● PWM and melody

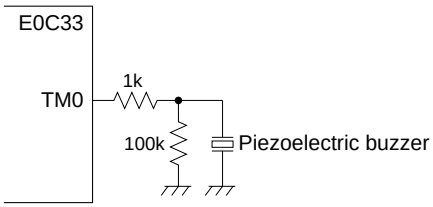
Human ears can discriminate tone on the musical scale by sound frequency. For example, a 131 Hz tone is heard as do (C3). A 262 Hz tone is heard as a do (C4) one-octave higher, while a 65.5 Hz tone is heard as a do (C2) one-octave lower. When one octave (up to 2-fold frequency) is equally divided by 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 melody33 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

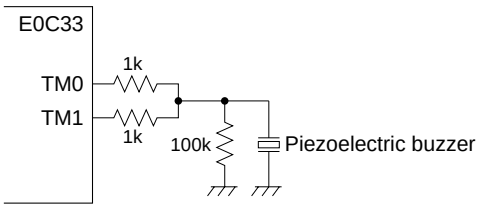


The output waveform of 131 Hz produces a sound corresponding to do (C).



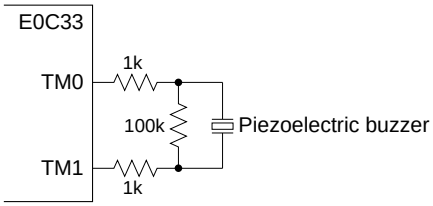
One-channel output drives a piezoelectric buzzer, as shown here.

● 2-channel synthesis output



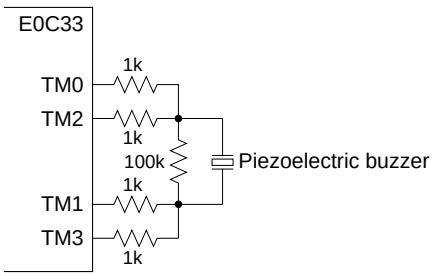
Two or more channels can be synthesized, as shown here.

● Differential output



Sound volume can be increased through differential output, using inverted PWM on one channel.

● Differential output, 2-channel synthesis output

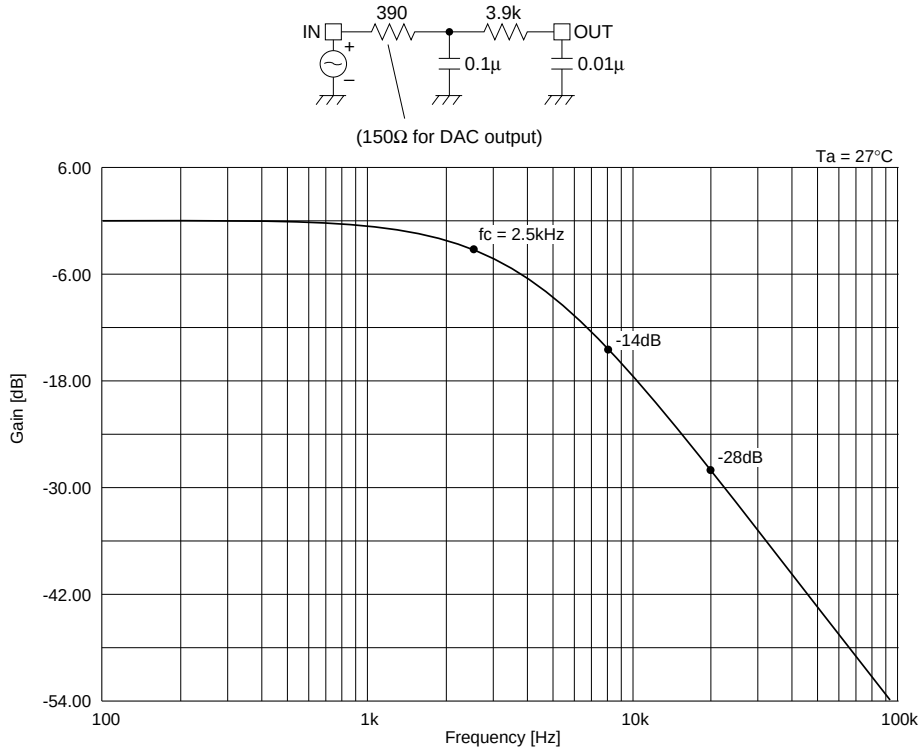


Two-channel synthesis and differential output can be used in combination using two differential outputs.

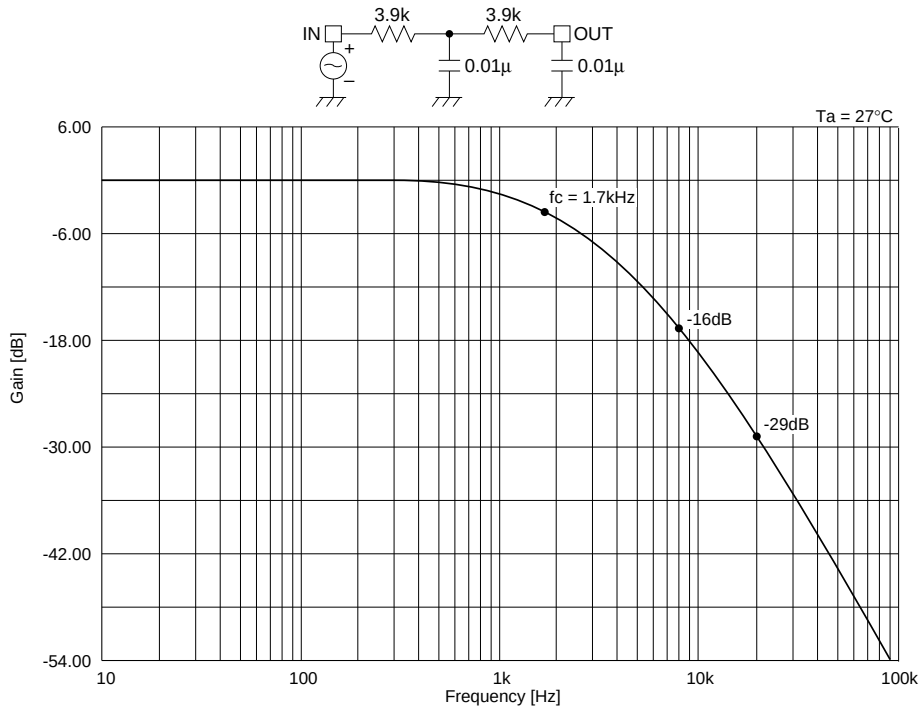
5.8 <Reference Data> Characteristic Graphs

● RC second-order low-pass filter frequency response (for 8 kHz sampling)

(1) fc = 2.5 kHz

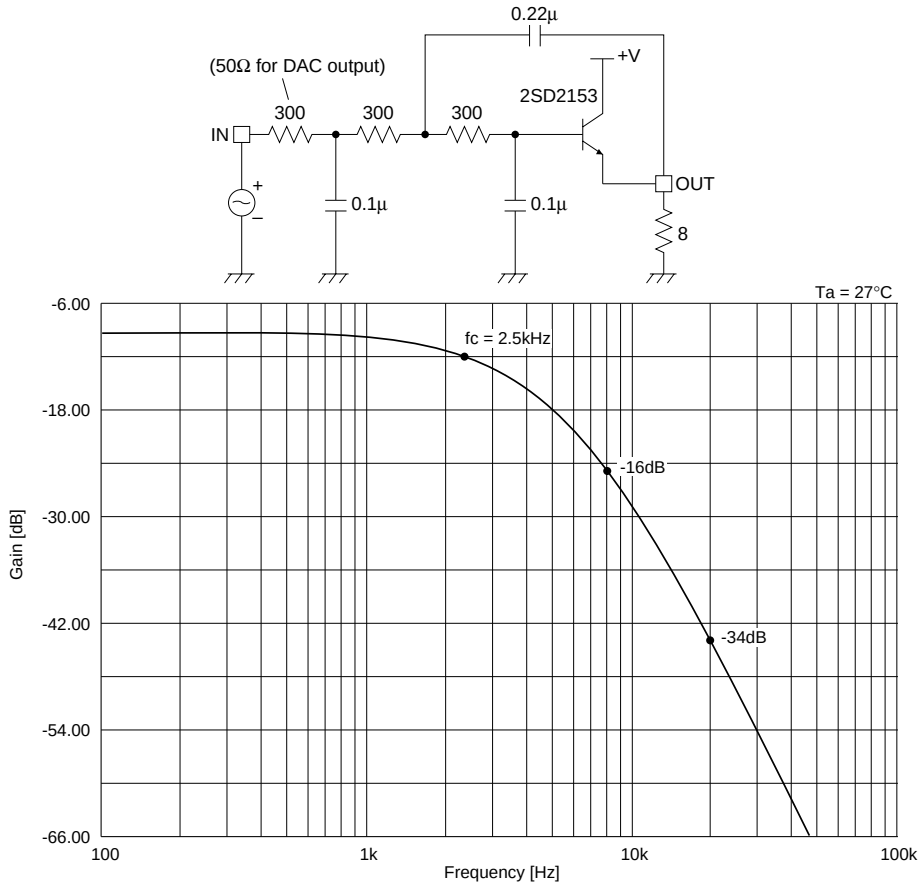


(2) fc = 1.7 kHz



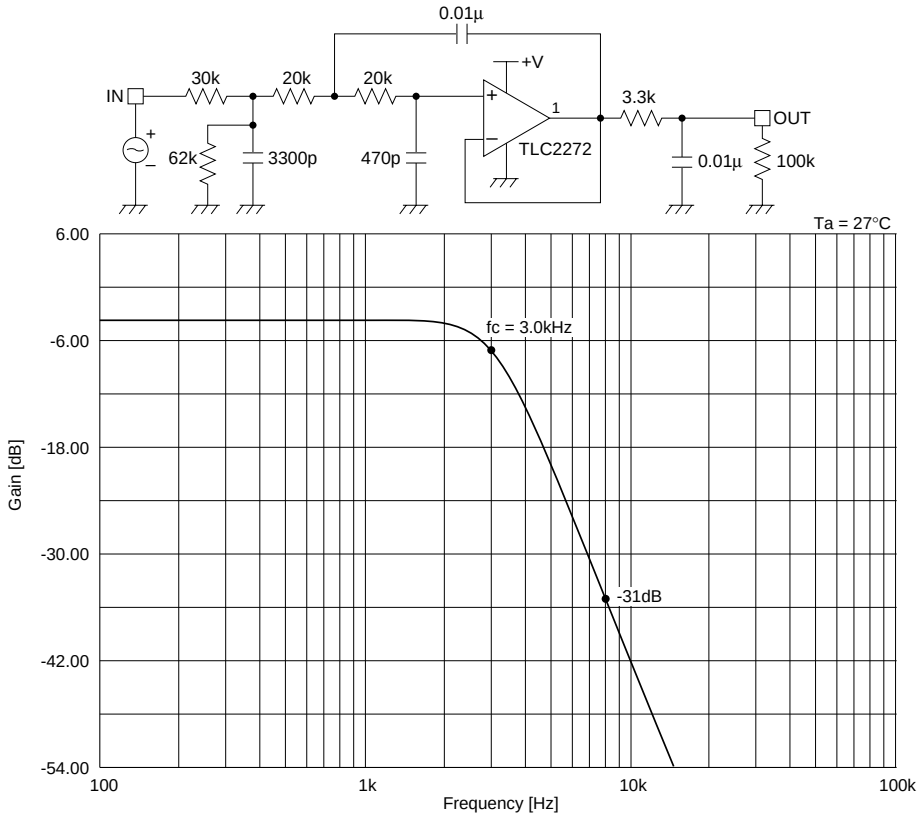
● Transistor third-order low-pass filter frequency response (for 8 kHz sampling)

fc = 2.5 kHz



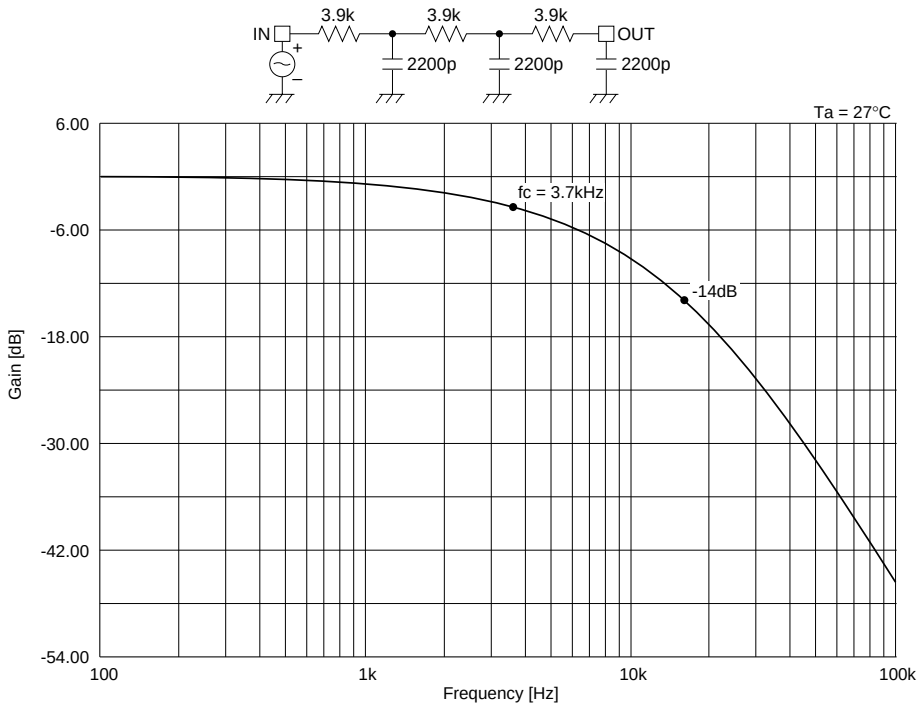
● Op amp fourth-order low-pass filter frequency response (for 8 kHz sampling)

fc = 3 kHz

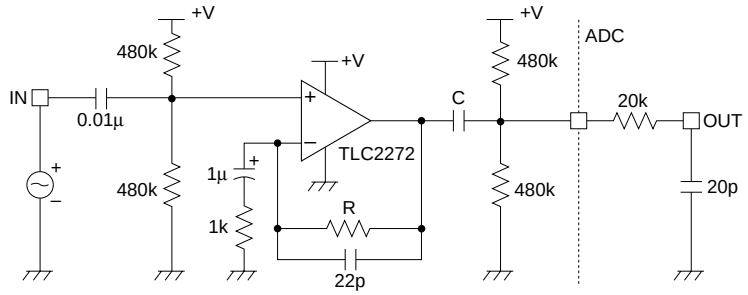


● RC third-order low-pass filter frequency response (for 16 kHz sampling)

fc = 3.7 kHz

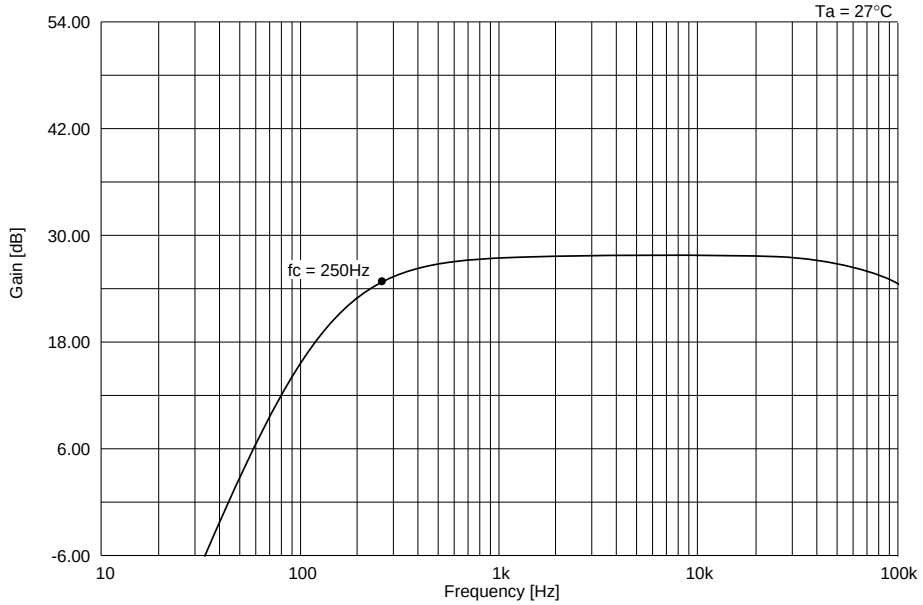


● AC amp high-pass filter frequency response (for 8 kHz sampling)

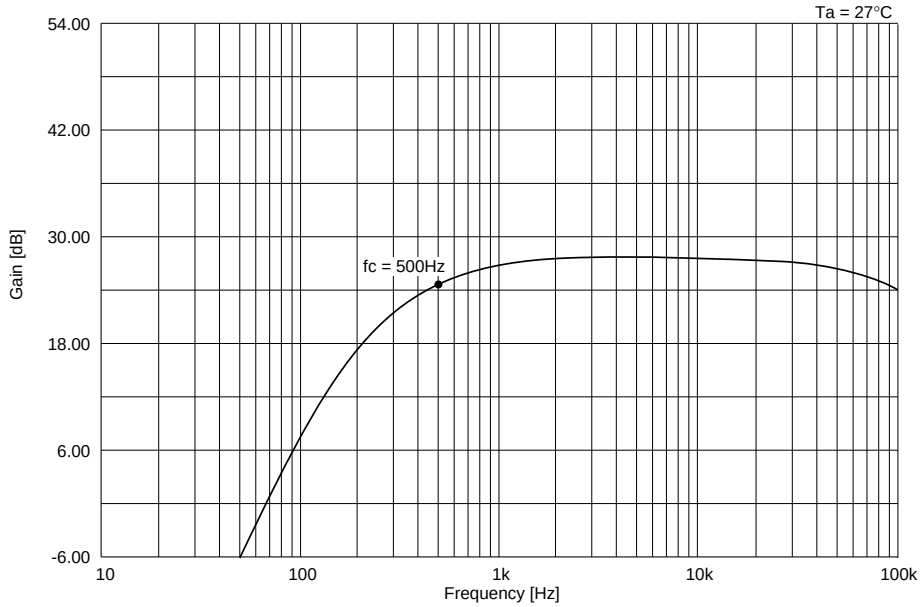


(1) R = 24kΩ, C = 0.0048µF (fc = 250Hz)
(2) R = 24kΩ, C = 0.0015µF (fc = 500Hz)

(1) fc = 250 Hz



(2) fc = 500 Hz



EPSON International Sales Operations

AMERICA

EPSON ELECTRONICS AMERICA, INC.

- HEADQUARTERS -
1960 E. Grand Avenue
El Segundo, CA 90245, U.S.A.
Phone: +1-310-955-5300 Fax: +1-310-955-5400

- SALES OFFICES -

West
150 River Oaks Parkway
San Jose, CA 95134, U.S.A.
Phone: +1-408-922-0200 Fax: +1-408-922-0238

Central
101 Virginia Street, Suite 290
Crystal Lake, IL 60014, U.S.A.
Phone: +1-815-455-7630 Fax: +1-815-455-7633

Northeast
301 Edgewater Place, Suite 120
Wakefield, MA 01880, U.S.A.
Phone: +1-781-246-3600 Fax: +1-781-246-5443

Southeast
3010 Royal Blvd. South, Suite 170
Alpharetta, GA 30005, U.S.A.
Phone: +1-877-EEA-0020 Fax: +1-770-777-2637

EUROPE

EPSON EUROPE ELECTRONICS GmbH

- HEADQUARTERS -
Riesstrasse 15
80992 Munich, GERMANY
Phone: +49-(0)89-14005-0 Fax: +49-(0)89-14005-110

- GERMANY -
SALES OFFICE
Altstadtstrasse 176
51379 Leverkusen, GERMANY
Phone: +49-(0)2171-5045-0 Fax: +49-(0)2171-5045-10

- UNITED KINGDOM -
UK BRANCH OFFICE
Unit 2.4, Doncastle House, Doncastle Road
Bracknell, Berkshire RG12 8PE, ENGLAND
Phone: +44-(0)1344-381700 Fax: +44-(0)1344-381701

- FRANCE -
FRENCH BRANCH OFFICE
1 Avenue de l' Atlantique, LP 915 Les Conquerants
Z.A. de Courtaboeuf 2, F-91976 Les Ulis Cedex, FRANCE
Phone: +33-(0)1-64862350 Fax: +33-(0)1-64862355

ASIA

- CHINA -
EPSON (CHINA) CO., LTD.
28F, Beijing Silver Tower 2# North RD DongSanHuan
ChaoYang District, Beijing, CHINA
Phone: 64106655 Fax: 64107319

SHANGHAI BRANCH
4F, Bldg., 27, No. 69, Gui Jing Road
Caohejing, Shanghai, CHINA
Phone: 21-6485-5552 Fax: 21-6485-0775

- HONG KONG, CHINA -
EPSON HONG KONG LTD.
20/F., Harbour Centre, 25 Harbour Road
Wanchai, HONG KONG
Phone: +852-2585-4600 Fax: +852-2827-4346
Telex: 65542 EPSCO HX

- TAIWAN -
EPSON TAIWAN TECHNOLOGY & TRADING LTD.
10F, No. 287, Nanking East Road, Sec. 3
Taipei, TAIWAN
Phone: 02-2717-7360 Fax: 02-2712-9164
Telex: 24444 EPSONTB

HSINCHU OFFICE
13F-3, No. 295, Kuang-Fu Road, Sec. 2
HsinChu 300, TAIWAN
Phone: 03-573-9900 Fax: 03-573-9169

- SINGAPORE -
EPSON SINGAPORE PTE., LTD.
No. 1 Temasek Avenue, #36-00
Millenia Tower, SINGAPORE 039192
Phone: +65-337-7911 Fax: +65-334-2716

- KOREA -
SEIKO EPSON CORPORATION KOREA OFFICE
50F, KLI 63 Bldg., 60 Yoido-dong
Youngdeungpo-Ku, Seoul, 150-763, KOREA
Phone: 02-784-6027 Fax: 02-767-3677

- JAPAN -
SEIKO EPSON CORPORATION
ELECTRONIC DEVICES MARKETING DIVISION

Electronic Device Marketing Department
IC Marketing & Engineering Group
421-8, Hino, Hino-shi, Tokyo 191-8501, JAPAN
Phone: +81-(0)42-587-5816 Fax: +81-(0)42-587-5624

ED International Marketing Department Europe & U.S.A.
421-8, Hino, Hino-shi, Tokyo 191-8501, JAPAN
Phone: +81-(0)42-587-5812 Fax: +81-(0)42-587-5564

ED International Marketing Department Asia
421-8, Hino, Hino-shi, Tokyo 191-8501, JAPAN
Phone: +81-(0)42-587-5814 Fax: +81-(0)42-587-5110

In pursuit of **"Saving" Technology**, Epson electronic devices.
Our lineup of semiconductors, liquid crystal displays and quartz devices
assists in creating the products of our customers' dreams.
Epson IS energy savings.

EPSON

SEIKO EPSON CORPORATION
ELECTRONIC DEVICES MARKETING DIVISION

■ EPSON Electronic Devices Website

http://www.epson.co.jp/device/

Issue MAY 2000, Printed in Japan Ⓜ A