CMOS 32-BIT SINGLE CHIP MICROCOMPUTER
S1C33 Family
Startup Manual
## Configuration of product number

### Devices

<table>
<thead>
<tr>
<th>Devices</th>
<th>C</th>
<th>33209</th>
<th>F</th>
<th>00E1</th>
<th>00</th>
</tr>
</thead>
</table>

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

### Specification

### Package
[D: die form; F: QFP, B: BGA]

### Model number

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

### Product classification
[S1: semiconductor]

## Development tools

### S5U1

<table>
<thead>
<tr>
<th>Devices</th>
<th>C</th>
<th>33000</th>
<th>H2</th>
<th>1</th>
<th>00</th>
</tr>
</thead>
</table>

### Packing specifications
[00: standard packing]

### Version
[1: Version 1]

### Tool type
[Hx: ICE
 Dx: Evaluation board
 Ex: ROM emulation board
 Mx: Emulation memory for external ROM
 Tx: A socket for mounting
 Cx: Compiler package
 Sx: Middleware package]

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

### Tool classification
[C: microcomputer use]

### Product classification
[S5U1: development tool for semiconductor products]
The S1C33 Family microprocessors/controllers incorporate a Seiko Epson original 32-bit CMOS RISC Core, ROM, RAM, DMA, Timers, SIO, PLL, A/D converter, and other peripheral modules. They feature high-speed operation, low current consumption, small code size, and an embedded MAC module, and are capable of being used for a wide range of applications from mobile equipment to OA equipment. Furthermore, they can be embedded in ASIC and custom microcomputers.

This manual is written for developers of application systems incorporating the S1C33 Family microprocessors/controllers, and it explains basic programming methods for embedded applications and how to program the peripheral modules using the S1C33 chips, particularly the S1C3301.

This manual assumes that the reader already possesses the following fundamental knowledge:

- Knowledge about C language (based on ANSI C) and C source creation methods
- Knowledge about the gnu C, binutils, gnu make and the linker script for the gnu linker (ld)
- Basic knowledge about assembler language
- Basic knowledge about the general concept of program development by a C compiler and an assembler
- Basic operating methods for Windows 2000 or Windows XP.

The sample code provided in this manual is excerpted from the S1C33 Family C/C++ Compiler Package (S5U1C33001C) Ver. 3 or later. For details of the C compiler, refer to the “S5U1C33001C Manual.”

<Organization of the manual>

This manual contains the following four chapters:
- Chapter 1 will introduce basic knowledge for developing embedded application software.
- Chapter 2 will describe basic programming methods for the S1C33 MCU using sample programs.
- Chapter 3 will describe how to program the peripheral circuits built into the S1C33 MCU.
- Chapter 4 will describe some tips and precautions on programming for the S1C33 MCU.

<Reference manuals>

The following lists the related manuals:
- S1C33000 Core CPU Manual
- S1C33 Family C33 ADV Core CPU Manual
- S5U1C33000C Manual (C Compiler Package for S1C33 Family) (Ver. 4)
- S5U1C33001C Manual (C/C++ Compiler Package for S1C33 Family) (Ver. 3)
- Technical Manual for each S1C33 Family processor

<Information provided on the website>

The information listed below is provided on our website. Enter the webpage from the following address using a user ID and a password:
http://www.epsondevice.com/webapp/MCUToolsDownload/entry.jsp

1. Differences between S5U1C33001C (GNU33 Ver. 2, Ver. 3) and S5U1C33000C (CC33)
   This page summarizes differences between CC33, GNU33 Ver. 2 and GNU33 Ver. 3.

2. Tool correspondence table
   The table lists the correspondence between the S1C33 Family CPU Core, MCU, ICD, reference boards, OS and middlewares. Furthermore, option, library, and assembler differences between the cores are summarized.

3. FAQ
   This page lists the frequently asked questions and their answers.
CONTENTS

– Contents –

1 Basic Knowledge for Embedded Software .................................................................1-1
  1.1 Basic Mechanism to Run Programs .................................................................1-1
  1.2 Startup (Initial Setting) Routine .................................................................1-2

2 Writing Programs for the S1C33 .................................................................................2-1
  2.1 Vector Table and Startup Routine .................................................................2-1
  2.2 Interrupt Handling .........................................................................................2-6
    2.2.1 Prototype Declaration .............................................................................2-6
    2.2.2 NMI .................................................................................................2-6
    2.2.3 Exceptions .........................................................................................2-7
    2.2.4 Software Interrupts .............................................................................2-7
    2.2.5 Cause-of-Interrupt Flag ......................................................................2-7
  2.3 C Compiler and Code Optimization ..................................................................2-9
    2.3.1 Accessing Variables .............................................................................2-9
    2.3.2 volatile Modifier .................................................................................2-9
    2.3.3 Pointer Type Structures and Arrays .................................................2-10
    2.3.4 C Compiler Options for Optimization ..............................................2-10

3 Programming the S1C33 Standard Peripheral Modules ........................................3-1
  3.1 BCU ...............................................................................................................3-1
  3.2 8-bit Programmable Timers ..........................................................................3-3
  3.3 16-bit Programmable Timers .........................................................................3-6
  3.4 Watchdog Timer ............................................................................................3-9
  3.5 Clock Timer .....................................................................................................3-11
  3.6 Serial Interface ...............................................................................................3-14
  3.7 Serial Interface with FIFO ............................................................................3-25
  3.8 Port Interrupts .................................................................................................3-37
  3.9 A/D Conversion ..............................................................................................3-42
  3.10 HSDMA Transfer ..........................................................................................3-45
  3.11 IDMA Transfer ..............................................................................................3-48
  3.12 SLEEP ............................................................................................................3-51

4 Technical Reference ...............................................................................................4-1
  4.1 Boot ................................................................................................................4-1
    4.1.1 Boot from External RAM .......................................................................4-1
    4.1.2 Boot from Flash Memory ........................................................................4-1
  4.2 Linker Script ......................................................................................................4-3
    4.2.1 Usage of .data Section ..........................................................................4-3
    4.2.2 Using the Internal RAM for Program Cache ........................................4-4
  4.3 C Compiler ........................................................................................................4-5
    4.3.1 Arguments .............................................................................................4-5
    4.3.2 Substitutions ..........................................................................................4-6
  4.4 DMA Transfer ..................................................................................................4-7
1 Basic Knowledge for Embedded Software

This chapter was written for programmers who have no experience in software development for embedded applications to introduce important software development concepts that should be understood, such as the basic mechanism to run program and initialization in a startup routine.

1.1 Basic Mechanism to Run Programs

This section explains operations (basic mechanism) when the S1C33 processor (hereafter described as MCU) activates. (See Figure 1.1.1.)

![Diagram of basic mechanism to startup the S1C33 processor]

(1) When an embedded system is powered on, a reset exception occurs and the MCU reads the contents written in the base address of the interrupt vector area (trap table). The interrupt vector area is configured as a table that contains addresses of various exception/interrupt handler routines (functions). When an exception or interrupt occurs, the MCU reads an address from this table to jump to the corresponding handler routine. Write the vector (start address) to the startup routine to be executed when the MCU is reset in the base address.

(2) The MCU reads the reset vector (address) in Step (1) and jumps to the address to execute the startup (initial setting) routine.

(3) First the startup routine should initialize the stack and parameters/resources required for executing the program.

(4) After the initialization has been finished, the startup routine calls the main function.

Embedded applications cannot startup directly from the main function. Be aware that a startup routine must be implemented to run the main program in development of embedded software.
1 BASIC KNOWLEDGE FOR EMBEDDED SOFTWARE

1.2 Startup (Initial Setting) Routine

The startup routine initializes required resources before executing the main routine.
In an application program development for Windows or Linux/UNIX, the C compiler automatically links a startup routine, because the program execution environment is always the same (it is not necessary to change the environment in each application). So the programmer does not need to keep that in mind.
In embedded applications, the resources, such as the devices connected and the memory size/type/location, depend on the system. Therefore, embedded applications need different initial settings according to the system. Also the initialization procedure is different between the MCU models. So the startup routine plays an important role in the embedded system.

A generic startup routine in embedded software should perform the processing listed below.
- Sets the stack pointer.
- Initializes PSR and enables interrupts (IE)
- Initializes the peripheral modules built into the MCU.
  - Initializes the BCU setup parameters.
  - Initializes interrupt settings.
  - Initializes I/O registers.
- Prepares data required for executing the program.
  - Transfers the initial data from a ROM area to a RAM area (copies the .data section).
  - Clears the memory area without initial values to 0 (clears the .bss section to 0).

![Startup Routine Diagram](Figure 1.2.1 Startup Routine)
1 BASIC KNOWLEDGE FOR EMBEDDED SOFTWARE

The stack is a RAM area used to save some data being processed and a return address when a subroutine or function is called. Also interrupts and exceptions use the stack, therefore, a startup routine must reserve a stack area first.

The PSR (Processor Status Register) is a 32-bit register to hold the CPU status and its contents are changed according to the instruction execution results. Since this register affects the program execution, the contents of PSR will be saved when an interrupt/exception occurs and the saved contents will be loaded back to PSR after the interrupt/exception handler has finished. A startup routine sets the IE (Interrupt Enable) bit in PSR to 1 to enable maskable external interrupts and the other PSR bits to 0.

In addition to the settings above, global variables without an initial value must be initialized before the program is able to run. These variables are undefined at reset, they should be initialized with an appropriate value or cleared to 0 (clearing the .bss section).

Global variables with an initial value must be initialized by copying the initial values from the ROM to the RAM (copying the .data section).

On-chip peripheral function and interrupt settings should be initialized according to the system configuration.

The C standard library must be initialized before calling the main function when it is used.

Besides the initialization related to software, the MCU and hardware resources must be initialized before use. Refer to the manuals for the MCU and hardware resources, for initial setup.

In an embedded application, a startup routine as above must be executed before the main function can be called (see Figure 1.2.1).

Bearing the above in mind, develop programs for embedded applications.
1 BASIC KNOWLEDGE FOR EMBEDDED SOFTWARE

THIS PAGE IS BLANK.
2 Writing Programs for the S1C33

This chapter explains how to write programs for the S1C33.
As described in Chapter 1, an embedded application needs to execute a startup routine as a preprocessing before executing the main function. The following explains a processing flow until the main function is called including a startup routine using sample programs.
The programs and peripheral functions shown in the explanation are examples from the S1C33301 unless otherwise specified. Be aware that the functions, control register addresses, and other conditions may be different from those of other models.

2.1 Vector Table and Startup Routine

The S1C33 program must have at least a vector table and a boot routine.
The vector table contains an array of vectors (destination addresses) to trap (interrupt) handler routines that will be executed when interrupts/exceptions occur during program running. So it is also called a trap table.

<table>
<thead>
<tr>
<th>Vector address</th>
<th>base + 0</th>
<th>base + 4–12</th>
<th>base + 16</th>
<th>base + 20</th>
<th>base + 24</th>
<th>base + 28</th>
<th>base + 32–44</th>
<th>base + 48</th>
<th>base + 60</th>
<th>base + 64</th>
<th>base + 924</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero division</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address error</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NMI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Software exception 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Software exception 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>External maskable interrupt 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>External maskable interrupt 215</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

base: Trap table start address
= 0x0080000 (when booting by on-chip ROM)
= 0x0c00000 (when booting by external ROM)

Figure 2.1.1 Configuration of Vector Table

On the other hand, a startup routine is also called a boot routine and it will be executed as a trap handler at initial reset.
In the S1C33 Family MCU, a reset exception occurs by a cold-reset at power on.
The C33 Core reads the reset exception vector (startup routine start address) from the vector table and jumps to the address to execute the initial processing before the main function is able to run.
Figure 2.1.2 shows an S1C33 Family MCU startup flow until the main function is called.
2 WRITING PROGRAMS FOR THE S1C33

The following shows the contents of GNU33\sample\ide\std\dmt3330\8timer\src\vector.c (8-bit timer sample program) as a sample vector table with startup routine in C language.

Vector table and startup routine
/* Prototype */
void boot() __attribute__((interrupt_handler));
void div0() __attribute__((interrupt_handler));
void unalign() __attribute__((interrupt_handler));
void dummy() __attribute__((interrupt_handler));
void nmi() __attribute__((interrupt_handler));
extern void int_8timer0();
extern void int_8timer1();
extern void int_8timer3();
extern void init_bcu(void);
extern void init_int(void);
extern void init_sys(void);
extern void init_ram(void);
extern void exit(void);

/* vector table */
const unsigned long vector[] = {
    (unsigned long)boot,  // 0    0 Reset
    (unsigned long)dummy,  // 4    1 Reserved
    (unsigned long)dummy,  // 8    2 Reserved
    (unsigned long)dummy,  // 12   3 Reserved
    (unsigned long)div0,  // 16   4 Division by zero
    (unsigned long)dummy,  // 20   5 Reserved
    (unsigned long)unalign, // 24   6 Address misaligned exception
    (unsigned long)nmi,  // 28   7 NMI
    (unsigned long)dummy,  // 32   8 Reserved
    (unsigned long)dummy,  // 36   9 Reserved
    (unsigned long)dummy,  // 40  10 Reserved
    (unsigned long)dummy,  // 44  11 Reserved
    (unsigned long)dummy,  // 48  12 Software exception 0
    (unsigned long)dummy,  // 52  13 Software exception 1
    (unsigned long)dummy,  // 56  14 Software exception 2
    (unsigned long)dummy,  // 60  15 Software exception 3
}

Figure 2.1.2 Execution Flow to Call main Function
<table>
<thead>
<tr>
<th>Address</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Port input interrupt 0</td>
</tr>
<tr>
<td>1</td>
<td>Port input interrupt 1</td>
</tr>
<tr>
<td>2</td>
<td>Port input interrupt 2</td>
</tr>
<tr>
<td>3</td>
<td>Port input interrupt 3</td>
</tr>
<tr>
<td>4</td>
<td>Key input interrupt 0</td>
</tr>
<tr>
<td>5</td>
<td>Key input interrupt 1</td>
</tr>
<tr>
<td>6</td>
<td>High-speed DMA Ch.0</td>
</tr>
<tr>
<td>7</td>
<td>High-speed DMA Ch.1</td>
</tr>
<tr>
<td>8</td>
<td>High-speed DMA Ch.2</td>
</tr>
<tr>
<td>9</td>
<td>High-speed DMA Ch.3</td>
</tr>
<tr>
<td>10</td>
<td>Intelligent DMA</td>
</tr>
<tr>
<td>11</td>
<td>16bit timer 0 comp B</td>
</tr>
<tr>
<td>12</td>
<td>16bit timer 0 comp A</td>
</tr>
<tr>
<td>13</td>
<td>16bit timer 1 comp B</td>
</tr>
<tr>
<td>14</td>
<td>16bit timer 1 comp A</td>
</tr>
<tr>
<td>15</td>
<td>16bit timer 2 comp B</td>
</tr>
<tr>
<td>16</td>
<td>16bit timer 2 comp A</td>
</tr>
<tr>
<td>17</td>
<td>16bit timer 3 comp B</td>
</tr>
<tr>
<td>18</td>
<td>16bit timer 3 comp A</td>
</tr>
<tr>
<td>19</td>
<td>16bit timer 4 comp B</td>
</tr>
<tr>
<td>20</td>
<td>16bit timer 4 comp A</td>
</tr>
<tr>
<td>21</td>
<td>16bit timer 5 comp B</td>
</tr>
<tr>
<td>22</td>
<td>16bit timer 5 comp A</td>
</tr>
<tr>
<td>23</td>
<td>Serial interface Ch.0</td>
</tr>
<tr>
<td>24</td>
<td>Serial interface Ch.0</td>
</tr>
<tr>
<td>25</td>
<td>Serial interface Ch.1</td>
</tr>
<tr>
<td>26</td>
<td>Serial interface Ch.1</td>
</tr>
<tr>
<td>27</td>
<td>Serial interface Ch.2</td>
</tr>
<tr>
<td>28</td>
<td>Serial interface Ch.2</td>
</tr>
<tr>
<td>29</td>
<td>Serial interface Ch.3</td>
</tr>
<tr>
<td>30</td>
<td>Serial interface Ch.3</td>
</tr>
<tr>
<td>31</td>
<td>Serial interface Ch.0</td>
</tr>
<tr>
<td>32</td>
<td>Serial interface Ch.0</td>
</tr>
<tr>
<td>33</td>
<td>Serial interface Ch.1</td>
</tr>
<tr>
<td>34</td>
<td>Serial interface Ch.1</td>
</tr>
<tr>
<td>35</td>
<td>Serial interface Ch.2</td>
</tr>
<tr>
<td>36</td>
<td>Serial interface Ch.2</td>
</tr>
<tr>
<td>37</td>
<td>Serial interface Ch.3</td>
</tr>
<tr>
<td>38</td>
<td>Serial interface Ch.3</td>
</tr>
<tr>
<td>39</td>
<td>Serial interface Ch.2</td>
</tr>
<tr>
<td>40</td>
<td>Serial interface Ch.2</td>
</tr>
<tr>
<td>41</td>
<td>Serial interface Ch.3</td>
</tr>
<tr>
<td>42</td>
<td>Serial interface Ch.3</td>
</tr>
<tr>
<td>43</td>
<td>Serial interface Ch.0</td>
</tr>
<tr>
<td>44</td>
<td>Serial interface Ch.0</td>
</tr>
<tr>
<td>45</td>
<td>Serial interface Ch.1</td>
</tr>
<tr>
<td>46</td>
<td>Serial interface Ch.1</td>
</tr>
<tr>
<td>47</td>
<td>Serial interface Ch.2</td>
</tr>
<tr>
<td>48</td>
<td>Serial interface Ch.2</td>
</tr>
<tr>
<td>49</td>
<td>Serial interface Ch.3</td>
</tr>
<tr>
<td>50</td>
<td>Serial interface Ch.3</td>
</tr>
<tr>
<td>51</td>
<td>Serial interface Ch.0</td>
</tr>
<tr>
<td>52</td>
<td>Serial interface Ch.0</td>
</tr>
<tr>
<td>53</td>
<td>Serial interface Ch.1</td>
</tr>
<tr>
<td>54</td>
<td>Serial interface Ch.1</td>
</tr>
<tr>
<td>55</td>
<td>Serial interface Ch.2</td>
</tr>
<tr>
<td>56</td>
<td>Serial interface Ch.2</td>
</tr>
<tr>
<td>57</td>
<td>Serial interface Ch.3</td>
</tr>
<tr>
<td>58</td>
<td>Serial interface Ch.3</td>
</tr>
<tr>
<td>59</td>
<td>Serial interface Ch.0</td>
</tr>
<tr>
<td>60</td>
<td>Serial interface Ch.0</td>
</tr>
<tr>
<td>61</td>
<td>Serial interface Ch.1</td>
</tr>
<tr>
<td>62</td>
<td>Serial interface Ch.1</td>
</tr>
<tr>
<td>63</td>
<td>Serial interface Ch.2</td>
</tr>
<tr>
<td>64</td>
<td>Serial interface Ch.2</td>
</tr>
<tr>
<td>65</td>
<td>Serial interface Ch.3</td>
</tr>
<tr>
<td>66</td>
<td>Serial interface Ch.3</td>
</tr>
<tr>
<td>67</td>
<td>Port input interrupt 4</td>
</tr>
<tr>
<td>68</td>
<td>Port input interrupt 5</td>
</tr>
<tr>
<td>69</td>
<td>Port input interrupt 6</td>
</tr>
<tr>
<td>70</td>
<td>Port input interrupt 7</td>
</tr>
<tr>
<td>71</td>
<td>Port input interrupt 8</td>
</tr>
<tr>
<td>72</td>
<td>Port input interrupt 9</td>
</tr>
<tr>
<td>73</td>
<td>Port input interrupt 10</td>
</tr>
<tr>
<td>74</td>
<td>Port input interrupt 11</td>
</tr>
<tr>
<td>75</td>
<td>Port input interrupt 12</td>
</tr>
<tr>
<td>76</td>
<td>Port input interrupt 13</td>
</tr>
<tr>
<td>77</td>
<td>Port input interrupt 14</td>
</tr>
<tr>
<td>78</td>
<td>Port input interrupt 15</td>
</tr>
<tr>
<td>79</td>
<td>Port input interrupt 16</td>
</tr>
<tr>
<td>80</td>
<td>Port input interrupt 17</td>
</tr>
<tr>
<td>81</td>
<td>Port input interrupt 18</td>
</tr>
<tr>
<td>82</td>
<td>Port input interrupt 19</td>
</tr>
<tr>
<td>83</td>
<td>Port input interrupt 20</td>
</tr>
<tr>
<td>84</td>
<td>Port input interrupt 21</td>
</tr>
<tr>
<td>85</td>
<td>Port input interrupt 22</td>
</tr>
<tr>
<td>86</td>
<td>Port input interrupt 23</td>
</tr>
<tr>
<td>87</td>
<td>Port input interrupt 24</td>
</tr>
</tbody>
</table>

2 WRITING PROGRAMS FOR THE S1C33
2 WRITING PROGRAMS FOR THE S1C33

{(unsigned long)dummy, // 352 88
(unsigned long)dummy, // 356 89
(unsigned long)dummy, // 360 90
(unsigned long)dummy, // 364 91
(unsigned long)dummy, // 368 92
(unsigned long)dummy, // 372 93
(unsigned long)dummy, // 376 94
(unsigned long)dummy, // 380 95
(unsigned long)dummy, // 384 96
(unsigned long)dummy, // 388 97
(unsigned long)dummy, // 392 98
(unsigned long)dummy, // 396 99
(unsigned long)dummy, // 400 100
(unsigned long)dummy, // 404 101
(unsigned long)dummy, // 408 102
(unsigned long)dummy, // 412 103
(unsigned long)dummy, // 416 104
(unsigned long)dummy, // 420 105
(unsigned long)dummy, // 424 106
(unsigned long)dummy, // 428 107
(unsigned long)dummy, // 432 108
(unsigned long)dummy, // 436 109
(unsigned long)dummy, // 440 110
(unsigned long)dummy, // 444 111
(unsigned long)dummy, // 448 112 FIFO Serial interface Ch.0
(unsigned long)dummy, // 452 113 FIFO Serial interface Ch.0
(unsigned long)dummy // 456 114 FIFO Serial interface Ch.0
};

/*******************************************************************************
* boot
* Type :     void
* Ret val :  none
* Argument : void
* Function : Boot program.
*******************************************************************************

void boot(void)
{
asm("xld.w %r15,0x2000"); // Set SP in end of 8KB internal RAM (1)
asm("ld.w %sp,%r15");  // Initialize PSR (2)
asm("ld.w %psr,%r15"); // Set PSR to interrupt enable (3)
asm("ld.w %r15,0b10000");
init_bcu();  // Initialize BCU on boot time (4)
init_int();  // Initialize interrupt controller (5)
init_sys();  // Initialize for sys.c (6)
init_ram();  // Initialize bss section & data section (7)
main();     // Call main (8)
exit();      // In last, go to exit (9)
}

/*******************************************************************************
* dummy
* Type :     void
* Ret val :  none
* Argument : void
* Function : Dummy interrupt program.
*******************************************************************************

void dummy(void)
{
INT_LOOP:
    goto INT_LOOP; (10)
}

/*******************************************************************************
* div0
* Type :     void
* Ret val :  none
*******************************************************************************
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 (/\(x\ y\ z\)) is decimal values indicating the offset address (\(x\)) from the top of the table and the vector number (\(y\)), and the interrupt source (\(z\)). The sample program contains the interrupt handler functions for the 8-bit timers to be used. In addition a dummy routine (dummy) has been written as the interrupt handler when an unused interrupt occurs.

The startup routine is executed as the reset exception handler function (boot). The boot routine performs the following sequence:

1. Initializes the stack pointer.
2. Initializes PSR.
3. Enables interrupts.
4. Initializes the BCU.
5. Initializes the interrupt controller.
6. Initializes input buffers.
7. Clears the global variables without an initial value (.bss section) and copies initial data for the global variables with an initial value (.data section) into RAM.
8. Calls the main function.

An embedded application does not necessarily require all the above processes, so execute only the required processes. This sample calls the exit function (9) after the main function has finished to notify the system that the processing has been terminated.

The INT_LOOP local symbols (10) are provided for debugging when an unexpected interrupt occurs. By setting the INT_LOOP location as a breakpoint, occurrence of an unexpected interrupt can be easily trapped.
2 WRITING PROGRAMS FOR THE S1C33

2.2 Interrupt Handling

2.2.1 Prototype Declaration

In the C/C++ compiler (SSU1C33001C), interrupt handler functions can be implemented by declaring a function prototype with

\[
\text{<Type> <Function name> __attribute__((interrupt_handler));}
\]

When an interrupt handler function prototype has been declared, the C/C++ compiler will automatically add the save/restore instructions for general-purpose registers and the \texttt{re\texttt{i}} instruction into the interrupt handler assembler code. Also the save/restore instructions for %ahr (Arithmetic operation High Register) and %alr (Arithmetic operation Low Register) will be added if the interrupt handler contains a multiplication/division or MAC operation instruction. The interrupt handler must contain a \texttt{re\texttt{i}} instruction to exit from the interrupt handling by restoring the PSR and PC values that were saved by the hardware when the interrupt occurred. Therefore, be sure to write the \texttt{re\texttt{i}} instruction at the end of the interrupt handler function if its prototype is not declared.

The following shows a sample C code and the assembler code after being compiled. The sample program below is the 8-bit timer 0 interrupt handler function written in the GNU33\example\std\dm3301\timer\src\drv_8timer.c.

\begin{verbatim}
C code
/* Prototype */
void int_8timer0() __attribute__((interrupt_handler));

/* 8bit timer0 interrupt function */
void int_8timer0(void)
{
    *** 8-bit timer 0 interrupt processing ***
}

Assembler code
/* 8bit timer0 interrupt function */
{
    pushn %r14
    ld.w %0, %alr
    ld.w %r1, %ahr
    pushn %r1

    *** 8-bit timer 0 interrupt processing ***

    popn %1
    ld.w %alr, %0
    ld.w %ahr, %1
    popn %r14
    re\texttt{i}
}
\end{verbatim}

As shown in the above sample code, the C/C++ compiler automatically generates assembler code that includes the register save/restore and \texttt{re\texttt{i}} instructions when an interrupt handler function prototype is declared. To avoid unexpected results, interrupt handler functions should be prototype declared.

2.2.2 NMI

Interrupts are classified under maskable interrupts and non-maskable interrupts (NMI). The CPU will always accept an NMI as it has higher priority than other interrupts.

However, after an initial reset, the hardware masks (disables) all interrupts including NMI until the SP (stack pointer) has been set in order to prevent a malfunction caused by the occurrence of an NMI before the stack pointer is set up.
2.2.3 Exceptions

Exception means the occurrence of an error while the program is being executed. In the S1C33 Family MCU, division by 0 and illegal memory accesses are assigned to causes of exception. When an exception occurs, the C33 Core reads the corresponding vector from the vector table and performs the exception handling.

In exceptions, pay particular attention to address misaligned exceptions. An address misaligned exception occurs when an illegal address is accessed, for example, accessing an odd address during a 16-bit memory read/write operation or accessing an address other than a word boundary address during a 32-bit memory read/write operation. The C33 Core does not allow such accessing.

2.2.4 Software Interrupts

Maskable interrupts include hardware interrupts and software interrupts (software exceptions). Interrupts caused by the timers and peripheral modules are the hardware interrupts. They occur depending on the hardware conditions. On the other hand, a software interrupt can be generated anywhere in the program by executing the int instruction.

When a software interrupt occurs, the C33 Core reads the corresponding vector from the vector table and performs the software interrupt handling.

The following shows a sample program to handle a software interrupt:

```
void softint1 __attribute__((interrupt_handler));

int int_num;

const unsigned long vector[] = {
    (unsigned long)softint1,  // 52 13 software interrupt 1 (1)
};

int main()
{
    asm("int 1");  // software interrupt 1 (2)
}

void softint1()
{
    int_num = 5;    // 3
}
```

The sample program above generates a software interrupt 1 using the "asm("int 1")" code in the main function (2). The sequence jumps to the software interrupt 1 handler function through the vector table (1). The interrupt handler function softint1 executes the interrupt processing (int_num = 5) and returns to the main function (3).

2.2.5 Cause-of-Interrupt Flag

Each maskable hardware interrupt provides a cause-of-interrupt flag and an interrupt enable register. An interrupt occurs if the cause-of-interrupt flag is set when the interrupt has been enabled by the interrupt enable register.

The cause-of-interrupt flag is set to 1 by the hardware when the corresponding cause of interrupt occurs regardless of whether the interrupt is enabled by the interrupt enable register or not. The user program can determine occurrence of the cause of interrupt by reading the cause-of-interrupt flag. The cause-of-interrupt flag that has been set to 1 must be reset in the interrupt handler routine, because the set cause-of-interrupt flag will generate the same interrupt again when the interrupt is enabled by the reti instruction in the interrupt handler. Furthermore, cause-of-interrupt flags are not initialized at initial reset, the software must reset cause-of-interrupt flags before use.

The cause-of-interrupt flag reset method can be selected from either reset-only method or read/write method using the Flag Set/Reset Method Select Register (0x4029F). Reset-only is the default method selected at initial reset. The following describes each reset method.
2 WRITING PROGRAMS FOR THE S1C33

Reset-only method

With this method, the cause-of-interrupt flag is reset by writing 1. The cause-of-interrupt flags for which 0 has been written can be neither set nor reset. Therefore, this method ensures that only a specific cause-of-interrupt flag is reset.

Read/write method

When this method is used, the cause-of-interrupt flag is reset by writing 0 and set by writing 1 same as other registers. All cause-of-interrupt flags for which 0 has been written are reset. Note that the cause-of-interrupt flags for which 1 has been written are set and they may generate interrupts.

Figure 2.2.5.1 shows the basic interrupt handling flow.
2.3 C Compiler and Code Optimization

2.3.1 Accessing Variables

Variables are classified under external variables (variables in RAM, constants in ROM, and static variables, etc., those with absolute addresses) and auto variables (variables placed in the stack). Normally, an external variable is accessed using a register with a 32-bit memory address loaded, while an auto variable is accessed with SP (stack pointer) + offset. The following lists the advantage of auto variables:

- The number of instructions required for access is smaller than external variables, and the processing speed is faster.
- 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.

Although excessive use of auto variables increases stack size making it difficult to predict the upper limit, temporarily used variables are better to define as auto variables for reducing code size.

2.3.2 volatile Modifier

The C compiler provides the volatile modifier that affects the code to access variables. To reduce code size and increase processing speed, generic C compilers perform code optimization to minimize the number of memory accesses and to recycle values placed in the registers. However, this optimization may omit actual memory access even if the C source contains the memory access code. In this case, some memory contents, such as control register values that have been changed by the hardware and variables that have been changed in interrupt handling, may be processed without updating to the current values. The volatile modifier instructs the C compiler to get the variable value every time it is referenced assuming that the value has been changed. In other words, the memory is always accessed when a variable with volatile declaration is referenced, so the variable is processed after updating to the latest value.

The following shows a comparison between assembler codes generated with and without volatile declaration:

**C code**

```c
int i_Normal_Flg;   // Without volatile declaration
volatile int i_Volatile_Flg; // With volatile declaration

while( 1 ){
    /* (1) Without volatile declaration */
    if( i_Normal_Flg == 1 ) {
        break;
    }

    while( 1 ){
        /* (2) With volatile declaration */
        if( i_Volatile_Flg == 2 ) {
            break;
        }
    }
}
```

**Assembler code**

```assembly
/* (1) Without volatile declaration */
cmp %r4,0x1 ← Compared using register value
jrne 0xff

/* (2) With volatile declaration */
ld.w %r4,[%r5] ← Always compared using memory value
cmpeq %r4,0x2
jrne 0xfe
```

In the sample program above, for example, if the variable value has been changed in an interrupt handler, code (1) cannot get the expected results because it does not reference the memory to update. On the other hand, code (2) can get the results in which the current memory value is reflected.
2 WRITING PROGRAMS FOR THE S1C33

2.3.3 Pointer Type Structures and Arrays

As described above, accessing normal external variables increases the number of instructions since it needs to load the 32-bit address of the variable into a register. To reduce code size, structure and array declarations may be used for external variables. An element of a structure or array can be accessed using only the offset value from the pointer to the structure/array, this makes it possible to increase the efficiency of external variable accesses.

2.3.4 C Compiler Options for Optimization

The C compiler optimizes code generation according to the specified switch `-O0`, `-O`, `-O2`, `-O3`, or `-Os`. The `-O2` and `-O3` switches specify optimization to increase the execution speed, and the `-Os` switch specifies optimization to reduce the code size. Unless a switch is specified, code generation is not optimized. The greater the value of `-O`, the higher the functionality of optimization, with the cost that some debugging information may not be output or other problems may arise. If code generation cannot be executed normally, reduce the value of the optimization option. Since `-O2` and `-O3` are provided for speed-priority optimization, the code size may be larger than for `-O`. Normally, `-O` should be specified.

For more information on the optimization options, refer to Section 6.3.2, “Command-line Options,” in the supplied “S5U1C33001C Manual.”
3 Programming the S1C33 Standard Peripheral Modules

This chapter describes how to program the standard peripheral modules of the S1C33 chip, especially for initial settings, using sample programs for reference.

The programs and peripheral functions shown in the explanation are examples from the S1C33301 unless otherwise specified. Be aware that the functions, control register addresses, and other conditions may be different from those of other models.

The constants used in the sample programs have been declared in the header files located in the GNU33\sample_ide\std\dmt\xxx\include directory (xxx represents a peripheral circuit name).

3.1 BCU

The BCU allows the user to configure the memory or I/O device type, size, and other access conditions for each memory area. This section describes how to set the BCU when an SRAM and flash are connected using the sample program located in the GNU33\sample_ide\std\dmt\bcu\src directory for reference.

External memory map and chip enable signals

Areas 4 to 10 in the memory space are open to an external system, each provided with an independent #CE (chip-enable) pin. Although the C33 STD Core is limited to 7 pins for the #CE outputs, it supports the memory space from Area 11 to Area 18 and the area assignment to the #CE pins can be switched using the CEFUNC[1:0] (D[A:9]) bits in the DRAM Timing Set-up Register (0x48130).

<table>
<thead>
<tr>
<th>Pin</th>
<th>CEFUNC = 00</th>
<th>CEFUNC = 01</th>
<th>CEFUNC = 1x</th>
</tr>
</thead>
<tbody>
<tr>
<td>#CE4</td>
<td>#CE11</td>
<td>#CE11</td>
<td>#CE11+#CE12</td>
</tr>
<tr>
<td>#CE5</td>
<td>#CE5</td>
<td>#CE15</td>
<td>#CE15+#CE16</td>
</tr>
<tr>
<td>#CE6</td>
<td>#CE6</td>
<td>#CE6</td>
<td>#CE7+#CE8</td>
</tr>
<tr>
<td>#CE7/#RAS0</td>
<td>#CE7/#RAS0</td>
<td>#CE13/#RAS2</td>
<td>#CE13/#RAS2</td>
</tr>
<tr>
<td>#CE8/#RAS1</td>
<td>#CE8/#RAS1</td>
<td>#CE14/#RAS3</td>
<td>#CE14/#RAS3</td>
</tr>
<tr>
<td>#CE9</td>
<td>#CE9</td>
<td>#CE17</td>
<td>#CE17+#CE18</td>
</tr>
<tr>
<td>#CE10EX</td>
<td>#CE10EX</td>
<td>#CE10EX</td>
<td>#CE9+#CE10EX</td>
</tr>
</tbody>
</table>

(Default: CEFUNC = 00)

Settings for SRAM, ROM, and flash

Settings for SRAM, ROM, and flash can be made for each area below using the BCU registers.

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 writing, 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.
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

Flash and SRAM configuration program

The following shows an S1C33301 sample program to configure the BCU when a flash is connected to Areas 5 and 8 and an SRAM is connected to Area 10:

BCU configuration

```c
void init_bcu(void)
{
    /* Set area 4-5,6 0x4812a <- 0x1111 */
    /* Device size 16 bits, output disable delay 1.5, wait control 1 in area 4-5,6 */
    *(volatile unsigned short *)BCU_A4_A5_A6_ADDR =
    BCU_DFH_15 | BCU_WTH_1 | BCU_SZL_16 | BCU_DFL_15 | BCU_WTL_1;  (1)

    /* Set area 7-8 0x48128 <- 0x0011 */
    /* Device size 16 bits, output disable delay 1.5, wait control 1, */
    /* DRAM is not used in area 7-8 */
    *(volatile unsigned short *)BCU_A7_A8_ADDR =
    BCU_DRAH_NOT | BCU_DRAL_NOT | BCU_SZL_16 | BCU_DFL_15 | BCU_WTL_1;  (2)

    /* Set area 9-10 setting 0x48126 <- 0x0017 */
    /* Device size 16 bits, disable delay 1.5, wait control 1, burst ROM not used */
    *(volatile unsigned short *)BCU_A9_A10_ADDR =
    BCU_BROH_NOT | BCU_BROL_NOT | BCU_SZL_16 | BCU_DFL_15 | BCU_WTL_1;  (3)
}
```

The sample program configures Areas 4–5 and 6 (1), Areas 7–8 (2), and Areas 9–10 (3) as device size = 16 bits, wait = 1 cycle, output disable delay time = 1.5 cycles.

Furthermore, DRAM support in Areas 7–8 (2) and burst ROM support in Areas 9–10 (3) are disabled.

The program above assumes use of two 8-bit SRAMs to connect to the 16-bit bus. When using a 16-bit SRAM, set the external interface method (D3/0x4812E) to #BSL (1). Note that an area pair in which the same configuration is applied does not allow use of 8- and 16-bit devices in combination.
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

3.2 8-bit Programmable Timers

This section shows an example of basic 8-bit timer interrupt control program. The sample program is located in the GNU33\sample_ide\std\dm333018timer\src directory.

The 8-bit timer can output the underflow signal generated by its 8-bit pre-settable down counter to the on-chip interrupt and other peripheral circuits and/or external devices.

8-bit timer interrupt control program

The sample program sets up 8-bit timers 0, 1, and 3 so that the timers generate a periodic interrupt in a different interval for each timer and gets the counter value when each interrupt occurs. The program displays the counter values acquired after all the timers have generated an interrupt. Below is a sample program for timer 0.

Interrupt vector settings for 8-bit timers

| (unsigned long)int_8timer0,       // 208  52 8bit timer 0 |
| (unsigned long)int_8timer1,       // 212  53 8bit timer 1 |
| (unsigned long)dummy,             // 216  54 8bit timer 2 |
| (unsigned long)int_8timer3,       // 220  55 8bit timer 3 |

Setting the vector table

Register the interrupt handler functions in the vector table.

Initialization of 8-bit timer

/* Prototype */
void init_8timer0(void);

/*******************************************************************************
* init_8timer0
*   Type :     void
*   Ret val :  none
*   Argument : void
*   Function : Initialize 8bit timer0.
*******************************************************************************/
void init_8timer0(void)
{
    unsigned char temp;
    /* Set 8bit timer0 interrupt enable on interrupt controller 0x40275 */
    volatile unsigned char *INT_E8TU_ADDR &= ~INT_E8TU0;

    /* Set 8bit timer0 control 0x40160, timer stop */
    volatile unsigned char *T8P_PTRUN0_ADDR &= 0xfe;

    /* Set 8bit timer0 prescaler 0x4014d */
    temp = *(volatile unsigned char *)PRESC_P8TS0_P8TS1_ADDR;
    temp &= 0xf0;
    temp |= PRESC_PTONL_ON | PRESC_CLKDIVL_SEL7;
    *(volatile unsigned char *)PRESC_P8TS0_P8TS1_ADDR = temp;

    /* Set 8bit timer0 reload data 0x40161 */
    *(volatile unsigned char *)T8P_RLD0_ADDR = 0x27;
    // Set reload data (0x27 -> 1ms on OSC3 clock 40MHz)

    /* Set 8bit timer0 clock output off, preset and timer stop 0x40160 */
    *(volatile unsigned char *)T8P_PTRUN0_ADDR =
        T8P_PTOUT_OFF | T8P_PSET_ON | T8P_PTRUN_STOP;

    /* Set 8bit timer0 interrupt CPU request on interrupt controller 0x40292 */
    volatile unsigned char *INT_R16T5_R8TU_RS0_ADDR &= ~INT_R8TU0;
    // IDMA request disable and CPU request enable

    /* Set 8bit timer0 interrupt priority level 3 on interrupt controller 0x40269 */
    temp = *(volatile unsigned char *)INT_P8TM_P5L0_ADDR;
    temp &= 0xf0;
    temp |= INT_PRL_LVL3;
    *(volatile unsigned char *)INT_P8TM_P5L0_ADDR = temp;

S1C33 FAMILY STARTUP MANUAL EPSON 3-3
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

/* Reset 8bit timer0 interrupt factor flag on interrupt controller 0x40285 */
*(volatile unsigned char *)INT_F8TU_ADDR = INT_F8TU0; (8)
// Reset 8bit timer0 underflow interrupt factor flag

/* Set 8bit timer0 interrupt enable on interrupt controller 0x40275 */
*(volatile unsigned char *)INT_E8TU_ADDR |= INT_E8TU0; (9)
// Set 8bit timer0 underflow interrupt enable

/*******************************************************************************
* run_8timer
* Type : void
* Ret val : none
* Argument : unsigned long reg 8bit timer run/stop register address
* Function : Run 8bit timer.
*******************************************************************************/
void run_8timer(unsigned long reg)
{
    /* run timer0 0x40160 */
    *(volatile unsigned char *)reg |= 0x01; (10)
}

(1) Disabling interrupt
   Disable the 8-bit timer 0 interrupt to avoid the occurrence of unexpected interrupts.

(2) Stopping the 8-bit timer counter
   Temporarily stop the timer for setting the timer input clock.

(3) Setting the input clock and starting clock supply to the timer
   Select a divided clock from the prescaler and enable the clock output to start supplying to the timer. The
   sample program selects the prescaler clock generated by dividing the operating clock by 256.

(4) Setting the reload data
   Set the reload data (counter initial data) from which the timer starts counting down. This value determines
   the timer underflow cycle. The underflow cycle is expressed by the equation below.

   \[
   \text{Underflow cycle} = \frac{\text{Reload data} + 1}{\text{Prescaler input clock frequency} \times \text{Prescaler division ratio}}
   \]

   The sample program sets the reload data to 0x27, so an underflow will occur in about 250 µs intervals when
   the operating clock frequency is 40 MHz.

(5) Selecting underflow output and presetting the counter data
   Select whether the underflow signal is to be output outside the IC or not. The sample program sets the timer
   so that it does not output the signal.
   At the same time, preset the reload data that has been set in Step (4) into the down counter.

(6) Setting interrupt/IDMA request
   Select whether the underflow interrupt cause is used to request an interrupt to the CPU or to request an
   IDMA transfer. The sample program selects interrupt request.

(7) Setting the interrupt priority level
   Set the interrupt priority level. When two or more interrupts occur at the same time, the interrupt that has
   the highest priority is accepted first. The interrupt priority level is set to 3 in the sample program.

(8) Resetting the cause-of-interrupt flag
   Cause-of-interrupt flags are undefined after an initial reset, therefore, reset the flag before enabling an
   interrupt.

(9) Enabling the interrupt
   Enable the timer interrupt. The timer will be able to generate a periodical interrupt after it starts counting.

(10) Starting the timer
   Start the timer. The timer keeps outputting an underflow signal periodically until the timer stops.
Interrupt handler
/* Prototype */
void int_8timer0(void) __attribute__((interrupt_handler));

/*******************************************************************************
* int_8timer0
*   Type :     void
*   Ret val :  none
*   Argument : void
*   Function : 8bit timer0 underflow interrupt function.
*                Read 8bit timer3 counter data and stop 8bit timer0.
*******************************************************************************/
void int_8timer0(void)
{
    extern volatile unsigned char timer0;
    extern volatile int t8int0_flg;

    /* interrupt operation */
    timer0 = read_8timer_cnt(T8P_PTD0_ADDR);     (1)
    stop_8timer(T8P_PTRUN0_ADDR);
    t8int0_flg = TRUE;

    /* Reset 8bit timer0 interrupt factor flag register 0x40285 */
    *(volatile unsigned char *)INT_F8TU_ADDR = INT_F8TU0;   (2)
}

(1) Executing the interrupt processing
When an interrupt occurs, the CPU executes the interrupt handler function that has been described in the
vector table.
The sample program saves the counter value at the point when the interrupt occurred and stops the timer.
Furthermore, it sets a software flag used to check whether the timer interrupt has occurred or not.

(2) Clearing the cause-of-interrupt flag
Reset the cause-of-interrupt flag, because it has been set to 1 by the occurrence of the interrupt.
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

3.3 16-bit Programmable Timers

This section shows an example of basic 16-bit timer interrupt control program. The sample program is located in the GNU333/sample_idstd/dmt33301/16timer/src directory.

The 16-bit timer provides the compare data registers A and B. Matching between the counter value and the compare data register value outputs the compare A or B signal to control the interrupt and timer output. The compare data register allows programmable setting of the interrupt cycle and the duty ratio of the output signal.

16-bit timer interrupt control program

The sample program starts the 16-bit timers after setting up them so that interrupts will occur in order of timers 0, 1, 2, and 3. When a timer 0, 1, or 2 interrupt occurs, the program saves the current value of the timer 3 counter and stops the timer that generated the interrupt. When a timer 3 interrupt occurs, the program displays the timer 3 counter values saved by the interrupts of timers 0–2. Below is a sample program for timer 0.

Interrupt vector settings for 16-bit timers

| (unsigned long)dummy, | // 120 30 16bit timer 0 comp B |
| (unsigned long)int_16timer_c0, | // 124 31 16bit timer 0 comp A |
| (unsigned long)dummy, | // 128 32 |
| (unsigned long)dummy, | // 132 33 |
| (unsigned long)int_16timer_u1, | // 136 34 16bit timer 1 comp B |
| (unsigned long)dummy, | // 140 35 16bit timer 1 comp A |
| (unsigned long)dummy, | // 144 36 |
| (unsigned long)dummy, | // 148 37 |
| (unsigned long)dummy, | // 152 38 16bit timer 2 comp B |
| (unsigned long)int_16timer_c2, | // 156 39 16bit timer 2 comp A |
| (unsigned long)dummy, | // 160 40 |
| (unsigned long)dummy, | // 164 41 |
| (unsigned long)int_16timer_u3, | // 168 42 16bit timer 3 comp B |
| (unsigned long)dummy, | // 172 43 16bit timer 3 comp A |

Setting the vector table

Register the interrupt handler functions in the vector table.

Initialization of 16-bit timer

```c
/* Prototype */
void init_16timer0(void);

void init_16timer0(void)
{
    unsigned char temp;

    /* Set 16bit timer0 interrupt enable 0x40272, compare A interrupt disable */
    *(volatile unsigned char *)INT_E16T0_E16T1_ADDR &= ~INT_E16TC0;   (1)

    /* Set 16bit timer0 control 0x48186, timer stop */
    *(volatile unsigned char *)T16P_PRUN0_ADDR &= 0xfe;               (2)

    /* Set 16bit timer0 prescaler 0x40147, prescaler on, CLK/256 */
    *(volatile unsigned char *)PRESC_P16TS0_ADDR = PRESC_PTONL_ON | PRESC_CLKDIVL_SEL5; (3)

    /* Set 16bit timer0 comparison match A data 0x48180, compare A 0x4e */
    *(volatile unsigned short *)T16P_CR0A_ADDR = 0x9c;                (4)

    /* Set 16bit timer1 comparison match B data 0x48182, compare B 0x9c */
    *(volatile unsigned short *)T16P_CR0B_ADDR = 0x138;

    /* Set 16bit timer0 mode 0x48186 */
    /* fine-mode normal, compare buffer disable, output normal, clock internal, */
```
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

/* clock output off, timer reset, timer stop */
*(volatile unsigned char *)T16P_PRUN0_ADDR =
    T16P_SBLFM_NOR | T16P_SELCRB_DIS | T16P_OUTINV_NOR | T16P_CKSL_INT |
    T16P_PTM_OFF | T16P_PSET_ON | T16P_PRUN_STOP;                  (5)

/* Set 16bit timer0 interrupt CPU request 0x40290, compare A,B CPU request */
temp = *(volatile unsigned char *)INT_RP0_RHDM_R16T0_ADDR;
    temp &= 0x3f;
    temp |= INT_RIDMA_DIS;
*(volatile unsigned char *)INT_RP0_RHDM_R16T0_ADDR = temp;

/* Set 16bit timer0 interrupt priority level 3 on interrupt controller 0x40266 */
temp = *(volatile unsigned char *)INT_P16T0_P16T1_ADDR;
    temp &= 0xf0;
    temp |= INT_PRIL_LVL3;
*(volatile unsigned char *)INT_P16T0_P16T1_ADDR = temp;

/* Reset 16bit timer0 interrupt factor flag 0x40282, */
/* Reset compare A interrupt flag */
*(volatile unsigned char *)INT_F16T0_F16T1_ADDR = INT_F16TC0;  (8)

/* Set 16bit timer0 interrupt enable 0x40272, compare A interrupt enable */
*(volatile unsigned char *)INT_E16T0_E16T1_ADDR |= INT_E16TC0;   (9)

} /* run_16timer */
*/

*******************************************************************************
* run_16timer
* Type :     void
* Ret val :  none
* Argument : unsigned long reg       16bit timer run/stop register address
* Function : Run 16bit timer.
*******************************************************************************
void run_16timer(unsigned long reg)
{
    *(volatile unsigned char *)reg |= 0x01;  (10)
}

(1) Disabling interrupt
Disable the 16-bit timer 0 interrupt to avoid the occurrence of unexpected interrupts.

(2) Stopping the 16-bit timer 0 counter
Temporarily stop the timer for setting the timer input clock.

(3) Setting the input clock and starting clock supply to the timer
Select a divided clock of the prescaler and enable the clock output to start supplying to the timer. The sample program selects the prescaler clock generated by dividing the operating clock by 256.

(4) Setting compare data
Set data to be compared with the up counter value.
When the up counter reaches compare data A or B, a compare A interrupt or a compare B interrupt occurs. The sample program sets compare data A to 0x9c and compare data B to 0x138. This makes the compare A interrupt occur in about 1 ms periods, and the compare B interrupt occur in about 2 ms periods when the operating clock = 40 MHz. A compare B interrupt resets the up counter.

(5) Setting 16-bit timer control conditions
Set up the 16-bit timer external output and input clock conditions.
For the external output, select an output mode and a signal active level, and control to start external output.
For the input clock, select either the internal clock or the external clock, and controls to start the timer.
The sample program does not perform external output and selects the internal clock as the input clock.
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

(6) Setting interrupt/IDMA request
Select whether the compare interrupt cause is used to request an interrupt to the CPU or to request an IDMA transfer. The sample program selects interrupt request.

(7) Setting the interrupt priority level
Set the interrupt priority level. When two or more interrupts occur at the same time, the interrupt that has the highest priority is accepted first. The interrupt priority level is set to 3 in the sample program.

(8) Resetting the cause-of-interrupt flag
Cause-of-interrupt flags are undefined after an initial reset, therefore, reset the flag before enabling an interrupt.

(9) Enabling the interrupt
Enable the timer interrupt. The timer will be able to generate a periodical interrupt after it starts counting.

(10) Starting the timer
Start the timer. The timer keeps outputting a compare signal periodically until the timer stops.

Interrupt handler
/* Prototype */
void int_16timer_c0(void) __attribute__((interrupt_handler));

/******************************************************************************
* 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;

    /* interrupt operation */
    timer0 = read_16timer_cnt(T16P_TC3_ADDR);
    stop_16timer(T16P_PRNG0_ADDR);
    /* Reset 16bit timer0 compare A interrupt factor flag */
    *(volatile unsigned char *)INT_F16T0_F16T1_ADDR = INT_F16TC0;
}

(1) Executing the interrupt processing
When an interrupt occurs, the CPU executes the interrupt handler function that has been described in the vector table.
The sample program saves the counter value at the point the interrupt occurred and stops the timer.

(2) Clearing the cause-of-interrupt flag
Reset the cause-of-interrupt flag, because it has been set to 1 by the occurrence of the interrupt.
3.4 Watchdog Timer

This section shows an example of a watchdog timer control program for detecting a CPU runaway. The sample program is located in the GNU33/sample_ide/dmtdm33301/wdtsrc directory.

The watchdog timer is implemented through the use of the 16-bit timer 0. When this function is enabled, an NMI will occur by the compare B signal from the 16-bit timer 0. Resetting the 16-bit timer 0 periodically so as not to generate an NMI, makes it possible to detect a program crash that may not pass through this processing routine.

Watchdog timer control program

This sample program provides a main process loop in which the 16-bit timer 0 is reset. If the timer is not reset, it will output the compare B signal and an NMI will occur. This is regarded as a CPU runaway, and the NMI handler is executed.

Initialization of watchdog timer

```c
/* Prototype */
void init_wdt(void);

/******************************************************************************
* init_wdt
*   Type :     void
*   Ret val :  none
*   Argument : void
*   Function : Initialize WatchDogTimer function.
******************************************************************************/
void init_wdt(void)
{
    /* watchdog timer write protection 0x40170, EWD write enable */
    *(volatile unsigned char *)WDT_WRWD_ADDR = WDT_WRWD_WRT; (1)

    /* watchdog timer enable 0x40171, NMI enable */
    *(volatile unsigned char *)WDT_EWD_ADDR = WDT_EWD_ENABLE; (2)
}
```

(1) Removing write protection of the watchdog timer

To prevent an unexpected NMI from being generated the watchdog timer enable bit (EWD) is normally write-protected. You must set 1 to the write-protect register before this bit can be altered.

(2) Enable the watchdog timer

Enable the watchdog timer facility.

After this, an NMI will occur if the 16-bit timer 0 outputs the compare B signal.
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

Main process

/*******************************************************************************
* main
* Type : void
* Ret val : none
* Argument : void
* Function : 16bit timer demonstration program.
*******************************************************************************/
int main(void)
{
    for (;;)
    {
        /* Main process */
        . . . . .
        (1)
        /* Reset 16bit timer0 */
        *(volatile unsigned char *)T16P_PRUN0_ADDR |= T16P_PSET_ON;
        (2)
    }
}

(1) Main process
Execute the main process.
The sample program repeats executing the main process loop.

(2) Resetting the 16-bit timer 0
The sample program resets the 16-bit timer 0 after the main process has finished. If an NMI occurs by the compare B signal of the 16-bit timer 0 before the timer has been reset, it is determined that there is a CPU runaway.

Refer to Section 3.3, “16-bit Programmable Timers,” for setting up the 16-bit timer.
The compare B signal output interval must be set longer than the main process period, to avoid occurrence of an NMI by the watchdog timer during normal operation.

Interrupt handler

/*******************************************************************************
* nmi
* Type : void
* Ret val : none
* Argument : void
* Function : NMI interrupt program.
*******************************************************************************/
void nmi(void)
{
    /* Reset 16bit timer0 interrupt flag 0x40282 */
    *(volatile unsigned char *)INT_F16T0_F16T1_ADDR = INT_F16TU0;
    /* Stop 16bit timer0 */
    stop_16timer(T16P_PRUN0_ADDR);
    t16i0_flg = TRUE;
    write_str("*** NMI occurred by Watchdog timer ***\n");
    write_str("\n");
}

Interrupt handler
Describe the process to be executed after a CPU runaway.
The sample program resets the cause-of-interrupt flag of the 16-bit timer 0, stops the timer, and then sets a software flag used to check whether an NMI has occurred or not.
3.5 Clock Timer

This section shows an example of a clock timer control program. The sample program is located in the GNU33\sample_ide\std\dm33301\vsrc\dir directory.

The clock timer consists of time/day counters that are clocked by a derived OSC1 clock and each counter data can be read out by software. The clock timer can also generate an interrupt by the output signal of each counter and an alarm interrupt with a time (minute or hour) or day specified.

Clock timer control program

The sample program controls the clock timer to generate an interrupt due to counting up of the 1 Hz counter. The following describes initialization and interrupt handling for the clock timer.

Interrupt vector settings for clock timer

| (unsigned long) int_ct, // 260 65 RTC |

Setting the vector table

Register the interrupt handler functions in the vector table.

Initialization of clock timer

void init_ct(void);

/*******************************************************************************
* init_ct
* Type : void
* Ret val : none
* Argument : void
* Function : Initialize clock timer to use real time clock.
*******************************************************************************/
void init_ct(void)
{
    unsigned char temp;
    /* Set clock timer interrupt enable 0x40277, clock timer interrupt disable */
    *(volatile unsigned char *)INT_EADE_ECTM_EP4_ADDR &= 0xfd; (1)
    /* Stop clock timer 0x40151 */
    *(volatile unsigned char *)CT_TCRUN_ADDR = 0x00; (2)
    /* Reset clock timer 0x40151 */
    *(volatile unsigned char *)CT_TCRUN_ADDR |= CT_TCRST_RST; (3)
    /* Set clock timer interrupt control 0x40152, */
    /* timer interrupt 1hz, timer alarm disable, factor flag reset */
    *(volatile unsigned char *)CT_TCAF_ADDR =
      CT_TCISE_1HZ | CT_TACF_NONE | CT_TCIF_RST | CT_TCAF_RST; (4)
    /* Set clock timer interrupt priority level 3 on interrupt controller 0x4026b */
    temp = *(volatile unsigned char *)INT_PCTM_ADDR;
    temp |= INT_PRIL_LVL3;
    *(volatile unsigned char *)INT_PCTM_ADDR = temp; (5)
    /* Reset clock timer interrupt factor flag on interrupt controller 0x40287 */
    *(volatile unsigned char *)INT_FADE_PCTM_FP4_ADDR = INT_PCTM; (6)
    /* Reset clock timer interrupt factor flag */
    /* Set clock timer interrupt enable on interrupt controller 0x40277 */
    *(volatile unsigned char *)INT_EADE_ECTM_EP4_ADDR |= INT_ECTM; (7)
    /* Set clock timer interrupt enable */
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

 /******************************************************************************
 * main
 * Type : void
 * Ret val : none
 * Argument : void
 * Function: Clock timer demonstration program.
 *******************************************************/

 void main() {
   /* Run clock timer */
   *(volatile unsigned char *)CT_TCRUN_ADDR |= 0x01;  // (8)
 }

(1) Disabling interrupt
Disable the clock timer interrupt to avoid the occurrence of unexpected interrupts.

(2) Stopping the clock timer counter
Stop the clock timer to avoid erroneous operations.

(3) Resetting the clock timer
Reset the clock timer counters. They can be reset only by software. Note that an initial reset does not reset
the clock timer counters. Furthermore, a reset operation cannot be accepted while the clock timer is active,
so resetting must be performed after stopping the clock timer.
Depending on the timer conditions, resetting the timer may generate an interrupt. Therefore, always disable
the clock timer interrupt before resetting.

(4) Selecting a cause of interrupt
Select a signal to be used as a cause of interrupt.
The clock timer can generate a cause of interrupt at the falling edge of the 32 Hz, 8 Hz, 2 Hz, 1 Hz, 1
minute, 1 hour, or 1 day signal.

(5) Setting the interrupt priority level
Set the interrupt priority level. When two or more interrupts occur at the same time, the interrupt that has
the highest priority is accepted first. The interrupt priority level is set to 3 in the sample program.

(6) Resetting the cause-of-interrupt flag
Cause-of-interrupt flags are undefined after an initial reset, therefore, reset the flag before enabling an
interrupt.

(7) Enabling the interrupt
Enable the clock timer interrupt. The clock timer will be able to generate interrupts in the selected cycles
after it starts counting.

(8) Starting the clock timer
Start the clock timer. An interrupt will occur in the selected cycles or at the specified alarm date and time.
Interrupt handler

/* Prototype */
void int_ct(void) __attribute__((interrupt_handler));

/*******************************************************************************
* int_ct
*   Type :     void
*   Ret val :  none
*   Argument : void
*   Function : Clock timer interrupt function.
*******************************************************************************/
void int_ct(void)
{
    extern volatile int ctint_flg;

    /* interrupt operation */
    ctint_flg = TRUE;        (1)

    /* Reset clock timer interrupt factor flag 0x40287 */
    *(volatile unsigned char *)INT_FADE_FCTM_FP4_ADDR = INT_FCTM;  (2)
}

(1) Executing the interrupt processing
    When an interrupt occurs, the CPU executes the interrupt handler function that has been described in the vector table.
    The sample program sets a software flag used to check whether the clock timer interrupt has occurred or not.

(2) Clearing the cause-of-interrupt flag
    Reset the cause-of-interrupt flag, because it has been set to 1 by the occurrence of the interrupt.
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

3.6 Serial Interface

This section shows examples of serial interface control programs. The sample programs are located in the GNU33\sample_idc\std\dmt33301\sif\asy\slv\mst directories.

The serial interface supports clock-synchronized transfer mode and asynchronous transfer mode. In the clock-synchronized transfer mode, transfer data is synchronized to the common clock on both the transmitter and receiver when the data is transferred. Asynchronous transfers are performed by adding a start bit and a stop bit to the start and end points of each serial data.

Serial interface (clock-synchronized slave mode) program

The sample program located in the GNU33\sample_idc\std\dmt33301\sif\asy\slv\src directory configures the serial interface in clock-synchronized slave mode and sends data continuously from the slave device (this chip) to the master device using an interrupt. The following describes how to configure the serial interface Ch.1 in slave mode and other initial settings.

Interrupt vector settings for serial interface (clock-synchronized slave mode)

| (unsigned long) dummy,        | // 240 60 Serial interface Ch.1 |
| (unsigned long) dummy,        | // 244 61 Serial interface Ch.1 |
| (unsigned long) int_sif_empty,| // 248 62 Serial interface Ch.1 |

Setting the vector table

Register the interrupt handler functions in the vector table.

Initialization of serial interface (clock-synchronized slave mode)

/* Prototype */
void init_sync_sif1(void);

/*******************************************************************************/
/* init_sync_sif1 */
/* Type : void */
/* Ret val : none */
/* Argument : void */
/* Function : Initialize synchronous serial channel 1. */
/*******************************************************************************/
void init_sync_sif1(void) {
    unsigned char temp;
    /* Set serial ch.0,1 interrupt enable 0x40276, serial ch.1 interrupt disable */
    *(volatile unsigned char *)INT_ES_ADDR &= ~(INT_ESRX1 | INT_ESERR1);  
    /* Set serial control 0x401e8, send disable, receive disable */
    *(volatile unsigned char *)SIF_SMD1_ADDR = SIF_TXEN_DIS | SIF_RXEN_DIS;
    /* Set serial interface I/O port 0x402d0, #SRDY1, #SCLK1, SOUT1, SIN1 */
    *(volatile unsigned char *)IO_CFP0_ADDR |= IO_CFP07_SRDY1 | IO_CFP06_SCLK1 | IO_CFP05_SOUT1 | IO_CFP04_SIN1;
    /* Set IrDA control 0x401e9, i/f mode normal */
    *(volatile unsigned char *)SIF_IRMD1_ADDR = SIF_IRMD_ORD;
    /* Set serial control 0x401e8, transfer-mode clock-syn slave */
    *(volatile unsigned char *)SIF_SMD1_ADDR = SIF_SMD_SLA;
    /* Set serial control 0x401e8, send enable, receive disable, clock #SCLK */
    *(volatile unsigned char *)SIF_SMD1_ADDR |= SIF_TXEN_ENA | SIF_RXEN_DIS | SIF_SSDCK_SCLK;
    /* Clear serial status */
    *(volatile unsigned char *)SIF_RDBF1_ADDR = SIF_ERR_NON;
    /* Set serial interface ch.1 and A/D converter interrupt priority register */
    /* 0x4026a */
    temp = *(volatile unsigned char *)INT_PSI01_PAD_ADDR;  
    temp &= 0xf0;
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

```
temp |= INT_PRIL_LVL3;
*(volatile unsigned char *)INT_PSIO1_PAD_ADDR = temp;

/* Set serial ch.0,1 interrupt flag 0x40286, clear serial ch.1 interrupt flag */
*(volatile unsigned char *)INT_FS_ADDR = INT_FSTX1 | INT_FSRX1 | INT_FSERR1; (9)

/* Set serial ch.0,1 interrupt enable 0x40276, serial ch.1 interrupt enable */
*(volatile unsigned char *)INT_ES_ADDR |= INT_ESTX1 | INT_ESRX1 | INT_ESERR1; (10)
```

(1) Disabling interrupt
Disable the serial interface interrupt to avoid the occurrence of unexpected interrupts.

(2) Disabling data transfer
Disable transmission/reception via the serial interface. Setting while the serial interface is active may cause malfunctions.

(3) Configuring the input/output pins
Switch the I/O port pin functions for the serial interface. In clock-synchronized slave mode, SINx, SOUTx, #SCLKx, and #SRDYx pins are used.

(4) Selecting an interface mode
Select either normal interface or IrDA interface. This setting must be initialized because it is undefined at initial reset. The sample program selects normal interface.

(5) Selecting a transfer mode
Select a transfer mode of the serial interface. The sample program selects clock-synchronized slave mode.

(6) Setting the clock and enabling data transfer
The serial interface set in clock-synchronized slave mode uses the clock output from the master device, therefore select an external clock for the operating clock. It is not necessary to setup the prescaler and a timer.
At the same time, data transfer should be enabled using the same register. The sample program performs data transmission in slave mode, so enable transmission only. The clock-synchronized transfer does not allow enabling of transmit/receive operations simultaneously.

(7) Resetting the status register
Reset the error flags in the status register by writing 0.

(8) Setting the interrupt priority level
Set the interrupt priority level. When two or more interrupts occur at the same time, the interrupt that has the highest priority is accepted first. The interrupt priority level is set to 3 in the sample program.

(9) Resetting the cause-of-interrupt flag
Cause-of-interrupt flags are undefined after an initial reset, therefore, reset the flag before enabling an interrupt.

(10) Enabling the interrupt
Enable the serial interface interrupt.
A transmit buffer empty interrupt will occur during transmission.
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

Interrupt handler for serial interface (clock-synchronized slave mode)

/* Prototype */
void int_sif_empty(void) __attribute__((interrupt_handler));

*******************************************************************************
* int_sif_empty
*   Type : void
*   Ret val : none
*   Argument : void
*   Function : Serial I/F ch.1 sending buffer empty interrupt function.
*******************************************************************************
void int_sif_empty(void)
{
    extern volatile int int_empty_flg;
    extern volatile unsigned char str[10];
    extern volatile int i;

    if (i > 9)
    {
        int_empty_flg = TRUE;
    } /* write sending data */
    else
    {
        /* write sending data */
        *(volatile unsigned char *)SIF_TXD1_ADDR = str[i];
        i++;
    }
    /* clear serial sending buffer empty interrupt flag */
    *(volatile unsigned char *)INT_FS_ADDR = INT_FSTX1;
}

(1) Interrupt processing
The serial interface generates a cause of transmit buffer empty interrupt when transmit data is loaded from
the transmit data register to the shift register. An interrupt occurs if the transmit buffer empty interrupt has
been enabled, and the interrupt handler is executed.
The sample program writes transmit data to the transmit data register and sets a software flag to TRUE after
a certain number of data has been transmitted.

(2) Clearing the cause-of-interrupt flag
Reset the cause-of-interrupt flag, because it has been set to 1 by the occurrence of the interrupt.

Serial interface (clock-synchronized master mode) program

The sample program located in the GNU33\sample\ide\std\dmt\sif_mst\src directory configures the serial
interface in clock-synchronized master mode and continuously receives data sent from the slave device to the
master device (this chip) using an interrupt. The following describes how to configure the serial interface Ch.1
in master mode and other initial settings.

Interrupt vector settings for serial interface (clock-synchronized master mode)

(unsigned long) int_sif_error,  // 240  60  Serial interface Ch.1
(unsigned long) int_sif_full,  // 244  61  Serial interface Ch.1
(unsigned long) dummy,       // 248  62  Serial interface Ch.1

Setting the vector table
Register the interrupt handler functions in the vector table.
Initialization of serial interface (clock-synchronized master mode)

/* Prototype */
void init_sync_sif1(void);

/******************************************************************************
* init_sync_sif1
* Type : void
* Ret val : none
* Argument : void
* Function : Initialize synchronous serial channel 1.
*******************************************************************************/
void init_sync_sif1(void)
{
    unsigned char temp;

    /* Set serial ch.0,1 interrupt enable OX40276, serial ch.1 interrupt disable */
    *(volatile unsigned char *)INT_ES_ADDR &= ~(INT_ESTX1 | INT_ESRX1 | INT_ESERR1);

    /* Set serial control OX401e8, send disable, receive disable */
    *(volatile unsigned char *)SIF_SMD1_ADDR = SIF_TXEN_DIS | SIF_RXEN_DIS;

    /* Set serial interface I/O port OX402d0, #SRDY1, #SCLK1, SOUT1, SIN1 */
    *(volatile unsigned char *)IO_CFP0_ADDR = IO_CFP07_SRDY1 | IO_CFP06_SCLK1 | IO_CFP05_SOUT1 | IO_CFP04_SIN1;

    /* Set IrDA control OX401e9, i/f mode normal */
    *(volatile unsigned char *)SIF_IRMD1_ADDR = SIF_IRMD_ORD;

    /* Set serial control OX401e8, transfer-mode clock-syn master */
    *(volatile unsigned char *)SIF_SMD1_ADDR = SIF_SMD_MAS;

    /* Set serial control OX401e8, send disable, receive enable, clock internal */
    *(volatile unsigned char *)SIF_SMD1_ADDR |= SIF_TXEN_DIS | SIF_RXEN_ENA | SIF_SSCK_INT;

    /* Set 8bit timer3 prescaler, prescaler on, CLK/4 */
    *(volatile unsigned char *)PRESC_P8TS2_P8TS3_ADDR = RESC_PTONH_ON | PRESC_CLKDIVH_SEL1;

    /* Set 8bit timer3 reload data OX4016d, reload data 0x40 */
    *(volatile unsigned char *)T8P_RLD3_ADDR = 0x20;

    /* Set 8bit timer3 for serial interface ch.1 OX4016c, */
    /* clock output, reload data pre-set, timer run */
    *(volatile unsigned char *)T8P_PTRUN3_ADDR = T8P_PTOUT_ON | T8P_PSET_ON | T8P_PTRUN_RUN;

    /* Clear serial status */
    *(volatile unsigned char *)SIF_RDBF1_ADDR = SIF_ERR_NON;

    /* Set serial interface ch.1 and A/D converter interrupt priority register */
    /* OX4026a */
    temp = *(volatile unsigned char *)INT_PSI01_PAD_ADDR;
    temp &= 0x0f;
    temp |= INT_PRL_LVL3;
    *(volatile unsigned char *)INT_PSI01_PAD_ADDR = temp;

    /* Set serial ch.0,1 interrupt flag OX4028E, clear serial ch.1 interrupt flag */
    *(volatile unsigned char *)INT_FSTX1 | INT_FSRX1 | INT_FSSERR1;

    /* Set serial ch.0,1 interrupt enable OX40276, serial ch.1 interrupt enable */
    *(volatile unsigned char *)INT_ES_ADDR &= ~(INT_ESTX1 | INT_ESRX1 | INT_ESERR1);
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

1) Disabling interrupt
   Disable the serial interface interrupt to avoid the occurrence of unexpected interrupts.

2) Disabling data transfer
   Disable transmission/reception via the serial interface. Setting while the serial interface is active may cause malfunctions.

3) Configuring the input/output pins
   Switch the I/O port pin functions for the serial interface. In clock-synchronized master mode, SINx, SOUTx, #SCLKx, and #SRDYx pins are used.

4) Selecting an interface mode
   Select either normal interface or IrDA interface. This setting must be initialized because it is undefined at initial reset. The sample program selects normal interface.

5) Selecting a transfer mode
   Select a transfer mode of the serial interface. The sample program selects clock-synchronized master mode.

6) Setting the clock and enabling data transfer
   The serial interface set in clock-synchronized master mode uses the clock internally generated, therefore select the internal clock for the operating clock.
   At the same time, data transfer should be enabled using the same register. The sample program performs data reception in master mode, so enable reception only. The clock-synchronized transfer does not allow enabling of transmit/receive operations simultaneously.

7)–(9) Setting the input clock (8-bit timer)
   The clock-synchronized master mode operates with an internal clock. Each channel uses the source clock listed below.
   Ch.0: 8-bit programmable timer 2 output clock
   Ch.1: 8-bit programmable timer 3 output clock
   Ch.2: 8-bit programmable timer 4 output clock
   Ch.3: 8-bit programmable timer 5 output clock
   The sample program uses the serial interface Ch.1, so set up the 8-bit timer 3.
   In Step (9), the sample program turns the clock output on to start supplying the clock to the serial interface.
   Refer to Section 3.2, “8-bit Programmable Timers,” for setting up the 8-bit timer.

10) Resetting the status register
    Reset the error flags in the status register by writing 0.

11) Setting the interrupt priority level
    Set the interrupt priority level. When two or more interrupts occur at the same time, the interrupt that has the highest priority is accepted first. The interrupt priority level is set to 3 in the sample program.

12) Resetting the cause-of-interrupt flag
    Cause-of-interrupt flags are undefined after an initial reset, therefore, reset the flag before enabling an interrupt.

13) Enabling the interrupt
    Enable the serial interface interrupt.
    A receive buffer full interrupt will occur during reception and a receive error interrupt may occur.
Interrupt handler for serial interface (clock-synchronized master mode)

/* Prototype */
void int_sif_error(void) __attribute__((interrupt_handler));
void int_sif_full(void) __attribute__((interrupt_handler));

/******************************************************************************
* int_sif_error
* Type : void
* Ret val: none
* Argument: void
* Function: Serial I/F ch.0 receiving error interrupt function.
******************************************************************************/
void int_sif_error(void)
{
    extern volatile int int_error_flg;
    int_error_flg = TRUE;  // Serial error interrupt processing

    /* clear serial error interrupt flag */
    *(volatile unsigned char *)INT_FS_ADDR = INT_FSERR0;  // Clearing the cause-of-interrupt flag
}

/******************************************************************************
* int_sif_full
* Type : void
* Ret val: none
* Argument: void
* Function: Serial I/F ch.0 receiving buffer full interrupt function.
******************************************************************************/
void int_sif_full(void)
{
    extern volatile unsigned char str[10];
    extern volatile int i;

    /* read receiving data */
    str[i] = *(volatile unsigned char *)SIF_RXD1_ADDR; // Receive buffer full interrupt processing
    //write_hex((unsigned long)str[i]);

    /* clear serial receiving buffer full interrupt cause flag */
    *(volatile unsigned char *)INT_FS_ADDR = INT_FSRX1;  // Clearing the cause-of-interrupt flag

    i++;
}

(1) Serial error interrupt processing
When the serial interface detects a parity error, framing error, or overrun error during data receiving, it sets
the receive error interrupt cause flag. An interrupt occurs if the serial error interrupt has been enabled, and
the interrupt handler is executed. In clock-synchronized mode, a parity error and a framing error will not
occur.
The sample program sets a software flag to TRUE to check whether the serial interface interrupt has
occurred or not.

(2) Receive buffer full interrupt processing
When the serial interface has finished a data reception and the received data is loaded from the shift register
to the receive data register, it sets the receive buffer full interrupt cause flag. An interrupt occurs if the serial
error interrupt has been enabled, and the interrupt handler is executed.
The sample program reads the received data from the receive data register.

(3) Clearing the cause-of-interrupt flag
Reset the cause-of-interrupt flag, because it has been set to 1 by the occurrence of the interrupt.
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

After the above settings have been made, clock-synchronized transfer can be started. The following shows the operations of the serial interface after data transfer begins.

Data reception (master mode)
1. Waits until the #SRDY signal is set to low by the slave device.
2. Starts inputting the synchronous clock to the serial interface after the #SRDY signal goes low.
3. Takes data sent from the slave device into the shift register. When the MSB has been received, the received data is transferred to the receive data register.
4. A receive data buffer full interrupt occurs when the received data is loaded to the receive data register and it starts an interrupt processing.

Data transmission (slave mode)
1. When transmit data is set to the transmit data register, the serial interface sets the #SRDY signal to low and waits for the clock input from the master device.
2. The transmit data in the transmit data register is loaded to the shift register when the synchronous clock is input to the #SCLK pin.
3. A transmit buffer empty interrupt occurs when transmit data is loaded to the shift register, and it starts an interrupt processing. At the same time, the #SRDY signal goes high.
4. The data in the shift register is output to the master device in synchronization with the clock. The #SRDY signal goes low after the MSB has been output to the master device.

The sample program writes data to the transmit data register in the interrupt handler executed in transmission Step 3 to transmit data continuously.

Serial interface (asynchronous mode) program
The sample program located in the GNU33\sample_idstd\dmt33301\sif_asyn\src directory configures the serial interface in asynchronous mode and performs continuous data transfer with the external serial device. The following describes how to initialize serial interface Ch.0.

Interrupt vector settings for serial interface (asynchronous mode)

<table>
<thead>
<tr>
<th>Vector Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x224</td>
<td>Serial interface Ch.0</td>
</tr>
<tr>
<td>0x228</td>
<td>Serial interface Ch.0</td>
</tr>
<tr>
<td>0x232</td>
<td>Serial interface Ch.0</td>
</tr>
</tbody>
</table>

Setting the vector table
Register the interrupt handler functions in the vector table.

Initialization of serial interface (asynchronous reception)

```c
/* Prototype */
void init_async_sif0(void);

/*******************************************************************************
* init_async_sif0
*   Type :     void
*   Ret val :  none
*   Argument : void
*   Function : Initialize asynchronous serial channel 0.
*******************************************************************************/
void init_async_sif0(void)
{
    unsigned char temp;
    /* Set serial ch.0,1 interrupt enable 0x40276, serial ch0 interrupt disable */
    *(volatile unsigned char *)INT_ES_ADDR &= ~(INT_ESTX0 | INT_ESRX0 | INT_ESERR0); (1)
    /* Set serial control 0x401e3, send disable, receive disable */
    *(volatile unsigned char *)SIF_SMD0_ADDR = SIF_TXEN_DIS | SIF_RXEN_DIS; (2)
    /* Set serial interface I/O port 0x402d0, #SRDY0, #SCLK0, SOUT0, SINO */
    *(volatile unsigned char *)IO_CFP0_ADDR | IO_CFP01_SOUT0 | IO_CFP00_SIN0; (3)
    /* Set IrDA control 0x401e4, i/f mode normal */
```
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

*(volatile unsigned char *)SIF_IRMD0_ADDR = SIF_IRMD_ORD;
/* Set serial control 0x401e3, transfer-mode clock-syn master */
*(volatile unsigned char *)SIF_SMD0_ADDR = SIF_SMD_8BIT;
/* Set IrDA control 0x401e4, dividing frequency 1/16 */
*(volatile unsigned char *)SIF_IRMD0_ADDR |= SIF_DIVMD_16;
/* Set serial control 0x401e3, send disable, receive enable, */
/* parity off, parity-mode even, stop-bit 1, clock internal */
*(volatile unsigned char *)SIF_SMD0_ADDR |=
SIF_TXEN_DIS | SIF_RXEN_ENA | SIF_EPR_OFF | SIF_PMD_EVEN |
SIF_STPB_1 | SIF_SSCK_INT;
/* Set 8bit timer2 prescaler, prescaler on, CLK/4 */
*(volatile unsigned char *)PRESC_PTONL_ON |PRESC_CLKDIVL_SEL1;
/* Set 8bit timer2 reload data 0x40169, reload data 0x40 */
*(volatile unsigned char *)T8P_RLD2_ADDR = 0x20;
/* Set 8bit timer2 for serial interface ch.0 0x40168, */
/* clock output, reload data pre-set, timer run */
*(volatile unsigned char *)T8P_PTRUN2_ADDR =
T8P_PTOUT_ON | T8P_PSET_ON | T8P_PTRUN_RUN;
/* Clear serial status */
*(volatile unsigned char *)SIF_RDBF1_ADDR = SIF_ERR_NON;
/* Set serial interface ch.0 and 8bit timer interrupt priority */
/* register 0x40269 */
temp = *(volatile unsigned char *)INT_P8TM_P8IO0_ADDR;
temp &= 0x0f;
temp |= INT_PRIH_LVL3;
*(volatile unsigned char *)INT_P8TM_P8IO0_ADDR = temp;
/* Set serial ch.0,1 interrupt flag 0x40286, clear serial ch.0 interrupt flag */
*(volatile unsigned char *)INT_FADDR =
INT_FSTX0 | INT_FSRX0 | INT_FSSERR0;
/* Set serial ch.0,1 interrupt enable 0x40276, serial ch.0 interrupt enable */
*(volatile unsigned char *)INT_ES_ADDR |=
INT_ESTX0 | INT_ESRX0 | INT_ESERR0;

(1) Disabling interrupt
Disable the serial interface interrupt to avoid the occurrence of unexpected interrupts.

(2) Disabling data transfer
Disable transmission/reception via the serial interface. Setting while the serial interface is active may cause
malfunctions.

(3) Configuring the input/output pins
Switch the I/O port pin functions for the serial interface. In asynchronous mode, SINx and SOUTx pins are
used.

(4) Selecting an interface mode
Select either normal interface or IrDA interface. This setting must be initialized because it is undefined at
initial reset. The sample program selects normal interface.

(5) Selecting a transfer mode
Select a transfer mode of the serial interface. The sample program selects 8-bit asynchronous mode.

(6) Setting the clock for asynchronous mode
Set the register to configure the input/output specifications of the IrDA interface and to select a division
ratio to generate the clock for the asynchronous mode.
The sample program does not use IrDA and sets the clock division ratio to 1/16.
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

(7) Setting the input clock, transfer enable bit, and data format

Either an internal clock or external clock can be selected in asynchronous mode. At the same time, data transfer should be enabled and a data format should be selected. The asynchronous mode allows configuration of the data bit length, stop bit, and parity bit.

The sample program configures the serial interface in 8-bit asynchronous mode, internal clock used, data bit length = 8 bits, stop bit = 1 bit, with no parity bit, and enables transmission.

(8)–(10) Setting the 8-bit timer

The serial interface operates with the clock generated by the 8-bit timer when the internal clock is selected as the input clock, so turn the clock output from the 8-bit timer on. Refer to “Serial interface (clock-synchronized master mode) program” for the source clock used in each channel. Refer to Section 3.2, “8-bit Programmable Timers,” for setting up the 8-bit timer.

(11) Resetting the status register

Reset the error flags in the status register by writing 0.

(12) Setting the interrupt priority level

Set the interrupt priority level. When two or more interrupts occur at the same time, the interrupt that has the highest priority is accepted first. The interrupt priority level is set to 3 in the sample program.

(13) Resetting the cause-of-interrupt flag

Cause-of-interrupt flags are undefined after an initial reset, therefore, reset the flag before enabling an interrupt.

(14) Enabling the interrupt

Enable the serial interface interrupt.

A receive buffer full interrupt will occur during reception and a receive error interrupt may occur.

Interrupt handler for serial interface (asynchronous reception)

Refer to “Interrupt handler for serial interface (clock-synchronized master mode)” for the receive interrupts (receive buffer full interrupt and serial error interrupt) that occur in asynchronous transfer.

Initialization of serial interface (asynchronous transmission)

```
/* Prototype */
void set_sif_mode(unsigned char);

/*******************************************************************************/
/* main */
/* Type : void */
/* Ret val : none */
/* Argument : void */
/* Function : Serial interface demonstration program. */
*******************************************************************************/
int main(void)
{
    unsigned char sif_mode;
    /* Set serial interface ch.0 transmit mode */
    sif_mode = SIF_TXEN_ENA | SIF_RXEN_DIS | SIF_EPR_OFF |
               SIF_PMD_EVEN | SIF_STPB_1 | SIF_SSCK_INT | SIF_SMD_8BIT;
    (1)
    set_sif_mode(sif_mode);

    /*******************************************************************************/
    /* set_sif_mode */
    /* Type : void */
    /* Ret val : none */
    /* Argument : unsigned char mode Serial mode */
    /* Function : Set serial interface mode. */
    ******************************************************************************/
    void set_sif_mode(unsigned char mode)
    {
        /* Set serial ch.0,1 interrupt enable 0x40276, serial ch0 interrupt disable */
        *(volatile unsigned char *)INT_ES_ADDR &=
        -(INT_ESTX0 | INT_ESRX0 | INT_ESERR0);
    (2)
```
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

/* Set serial control 0x401e3, send disable, receive disable */
*(volatile unsigned char *)SIF_SMD0_ADDR = SIF_TXEN_DIS | SIF_RXEN_DIS;  \(3\)

/* Clear serial status */
*(volatile unsigned char *)(SIF_RDBF0_ADDR) = SIF_ERR_NON;  \(4\)

/* Set serial control register */
*(volatile unsigned char *)(SIF_SMD0_ADDR) = mode;
// Set serial control register

/* Set serial ch.0,1 interrupt flag 0x40286, clear serial ch.0 interrupt flag */
*(volatile unsigned char *)INT_FS_ADDR =
  INT_FSTX0 | INT_FSRX0 | INT_FSERR0;  \(6\)

/* Set serial ch.0,1 interrupt enable 0x40276, serial ch.0 interrupt enable */
*(volatile unsigned char *)INT_ES_ADDR |=
  INT_ESTX0 | INT_ESRX0 | INT_ESERR0;  \(7\)
}

(1) Setting the transfer mode, input clock, transfer enable bit, and data format
Select a transfer mode of the serial interface. For the operating clock, either an internal clock or external
clock can be selected in asynchronous mode. At the same time, data transfer should be enabled and a data
format should be selected. The asynchronous mode allows configuration of the data bit length, stop bit, and
parity bit.
The sample program configures the serial interface in 8-bit asynchronous mode, internal clock used, data
bit length = 8 bits, stop bit = 1 bit, with no parity bit, and enables transmission.

(2) Disabling interrupt
Disable the serial interface interrupt to avoid the occurrence of unexpected interrupts.

(3) Disabling data transfer
Disable transmission/reception via the serial interface. Setting while the serial interface is active may cause
malfunctions.

(4) Resetting the status register
Reset the error flags in the status register by writing 0.

(5) Setting the transmit parameters
Set the transmit parameters prepared in Step (1) into the register.

(6) Resetting the cause-of-interrupt flag
Cause-of-interrupt flags are undefined after an initial reset, therefore, reset the flag before enabling an
interrupt.

(7) Enabling the interrupt
Enable the serial interface interrupt.
A transmit buffer empty interrupt will occur during transmission.

Interrupt handler for serial interface (asynchronous transmission)
Refer to “Interrupt handler for serial interface (clock-synchronized slave mode)” for the transmission interrupt
(transmit buffer empty interrupt) that occur in asynchronous transfer.
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

After the above settings have been made, asynchronous transfer can be started. The following shows the operations of the serial interface after data transfer begins.

Data reception
1. The serial interface starts data sampling when a start bit is input.
2. After the start bit is sampled, the data bits that follow are taken into the shift register from the LSB to the MSB.
3. The parity and stop bits that follow the MSB are sampled.
4. After the stop bit has been sampled, data in the shift register is loaded to the receive data register. The parity check is performed when the received data is loaded to the receive data register and a receive buffer full interrupt occurs and it starts executing an interrupt handler.

Data transmission
1. The serial interface transfers data in the transmit data register to the shift register in synchronization with the sampling clock. At the same time, it outputs a start bit from the SOUTx pin.
2. A transmit buffer empty interrupt occurs when transmit data is loaded to the shift register, and it starts an interrupt processing.
3. After the start bit has been output, the data bits are output in order from the LSB to the MSB in synchronization with the rising edge of the clock.
4. The parity and stop bits are output following the MSB.

The sample program writes data to the transmit data register in the interrupt handler executed in transmission Step 2 to transmit data continuously.
3.7 Serial Interface with FIFO

This section shows control program examples of the serial interface with FIFO. The sample programs are located in the GNU33\sample_ide\std\dmt33301\fsif\async\slv\msr directories.
The serial interface with FIFO incorporates a 4-byte receive data buffer and a 2-byte transmit data buffer in addition to the serial interface functions allowing continuous transmission/reception.

Serial interface with FIFO (clock-synchronized slave mode) program

The sample program located in the GNU33\sample_ide\std\dmt33301\fsif\async\slv\src directory configures the serial interface with FIFO in clock-synchronized slave mode and sends data continuously from the slave device (this chip) to the master device using an interrupt. The following describes how to configure the serial interface with FIFO Ch.0 in slave mode and other initial settings.

Interrupt vector settings for serial interface with FIFO (clock-synchronized slave mode)

| (unsigned long) dummy, | 448  112 FIFO Serial interface Ch.0 |
| (unsigned long) dummy, | 452  113 FIFO Serial interface Ch.0 |
| (unsigned long) int_fsif_empty | 456  114 FIFO Serial interface Ch.0 |

Serial interface with FIFO (clock-synchronized slave mode) program

The following describes how to configure the serial interface with FIFO Ch.0 in slave mode and other initial settings.

Interrupt vector settings for serial interface with FIFO (clock-synchronized slave mode)

| (unsigned long) dummy, | 448  112 FIFO Serial interface Ch.0 |
| (unsigned long) dummy, | 452  113 FIFO Serial interface Ch.0 |
| (unsigned long) int_fsif_empty | 456  114 FIFO Serial interface Ch.0 |

Setting the vector table

Register the interrupt handler functions in the vector table.

Initialization of serial interface with FIFO (clock-synchronized slave mode)

```c
/* Prototype */
void init_fifo_bcu(void);
void init_sync_sif0(void);

/******************************************
* init_fifo_bcu
* Type : void
* Ret val : none
* Argument : void
* Function : Initialize BCU for FIFO serial I/F.
*******************************************/
void init_fifo_bcu(void)
{
    unsigned short temp;
    /* Set area 5-18 endian and external/internal access control 0x48132 */
    temp = *(volatile unsigned short *)BCU_EC_IO_ADDR;
    temp &= 0xfdfd;
    temp |= BCU_A6IO_INT;
    *(volatile unsigned short *)BCU_EC_IO_ADDR = temp;
    /* Set area 4-6 BCU register, fifo i/f selection 0x4812a */
    temp = *(volatile unsigned short *)BCU_A4_A5_A6_ADDR;
    temp &= 0xf0ff;
    temp |= BCU_WTH_1;
    *(volatile unsigned short *)BCU_A4_A5_A6_ADDR = temp;
}

/******************************************
* init_sync_sif0
* Type : void
* Ret val : none
* Argument : void
* Function : Initialize synchronous FIFO serial channel 0.
*******************************************/
void init_sync_sif0(void)
{
    /* Set FIFO serial ch.0 interrupt enable 0x402b1, */
    /* FIFO serial ch.0 interrupt disable */
    *(volatile unsigned char *)INT_EFS_ADDR &=
        ~(INT_EFS_TX0 | INT_EFS_RX0 | INT_EFS_ERR0);
    /* Set FIFO serial interface control, each mode disable 0x300203 */
    *(volatile unsigned char *)FSIF_SMD0_ADDR =
        FSIF_TXEN_DIS | FSIF_RXEN_DIS;
}
```
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

/* Set P00-P03 extended function, #FSRDY0, #FSCLK0, FSOUT0, FSIN0 0x300040 */
*(volatile unsigned char *)FSIF_EFP_ADDR =
    FSIF_EFP01_FSRDY | FSIF_EFP02_FSCLK | FSIF_EFP01_FSOUT |
    FSIF_EFP00_FSIN;  (5)

/* Set FIFO serial interface IrDA register, i/f mode normal 0x300204 */
*(volatile unsigned char *)FSIF_IRMD0_ADDR = FSIF_IRMD_ORD;  (6)

/* Set FIFO serial interface control register, */
/* transfer mode clock-sync slave 0x300203 */
*(volatile unsigned char *)FSIF_SMD0_ADDR = FSIF_SMD_SLA;  (7)

/* Set FIFO serial interface IrDA register, #FSRDY0 normal 0x300204 */
*(volatile unsigned char *)FSIF_IRMD0_ADDR |= FSIF_SRDYL_NML;  (8)

/* Set FIFO serial interface control register, transfer enable, */
/* #FSCLK0 0x300203 */
*(volatile unsigned char *)FSIF_SMD0_ADDR |=
    FSIF_TXEN_ENA | FSIF_RXEN_DIS | FSIF_SSCK_FSCLK;  (9)

/* Set FIFO serial interface status register, error flag clear 0x300202 */
*(volatile unsigned char *)FSIF_RDBF0_ADDR &= 0xe3;  (10)

/* Set FIFO serial interface interrupt priority level 3 on interrupt */
/* controller 0x402b0 */
*(volatile unsigned char *)INT_PFSIO_ADDR = INT_PRIH_LVL3;  (11)

/* Set FIFO serial ch.0 interrupt flag 0x402b2, */
/* clear serial ch.0 interrupt flag */
*(volatile unsigned char *)INT_FFS_ADDR =
    INT_FFSSTX0 | INT_FFSRX0 | INT_FFSERR0;  (12)

/* Set FIFO serial ch.0 interrupt enable 0x402b1, */
/* serial ch.0 interrupt enable */
*(volatile unsigned char *)INT_EFS_ADDR =
    INT_EFSSTX0 | INT_EFSSRX0 | INT_EFSSERR0;  (13)

(1), (2) Setting the BCU
Addresses 0x300200 to 0x300209 in Area 6 are allocated for the control registers of the serial interface with FIFO. Therefore, set the BCU registers to enable the internal device to be accessed in Area 6 before the control registers can be accessed.

(3) Disabling interrupt
Disable interrupts of the serial interface with FIFO to avoid the occurrence of unexpected interrupts.

(4) Disabling data transfer
Disable transmission/reception via the serial interface with FIFO. Setting while the serial interface is active may cause malfunctions.

(5) Configuring the input/output pins
Switch the I/O port pin functions for the serial interface with FIFO. In clock-synchronized slave mode, FSIN0, FSOUT0, #FSCLK0, and #FSRDY0 pins are used.

(6) Selecting an interface mode
Select either normal interface or IrDA interface. This setting must be initialized because it is undefined at initial reset. The sample program selects normal interface.

(7) Selecting a transfer mode
Select a transfer mode of the serial interface with FIFO. The sample program selects clock-synchronized slave mode.

(8) Setting the #FSRDY0 control method
Set the #FSRDY0 signal control method at a receive buffer full status. By selecting mask for the #FSRDY0 signal control, the #FSRDY0 signal is fixed at high in receive buffer full status to avoid occurrence of an overrun error.
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

(9) Setting the clock and enabling data transfer
The serial interface with FIFO set in clock-synchronized slave mode uses the clock output from the master device, therefore select an external clock for the operating clock. It is not necessary to setup the prescaler and a timer.

At the same time, data transfer should be enabled using the same register. The sample program performs data transmission in slave mode, so enable transmission only. The clock-synchronized transfer does not allow enabling of transmit/receive operations simultaneously.

(10) Resetting the status register
Reset the error flags in the status register by writing 0.

(11) Setting the interrupt priority level
Set the interrupt priority level. When two or more interrupts occur at the same time, the interrupt that has the highest priority is accepted first. The interrupt priority level is set to 3 in the sample program.

(12) Resetting the cause-of-interrupt flag
Cause-of-interrupt flags are undefined after an initial reset, therefore, reset the flag before enabling an interrupt.

(13) Enabling the interrupt
Enable interrupts of the serial interface with FIFO.
A transmit buffer empty interrupt will occur during transmission.

Interrupt handler for serial interface with FIFO (clock-synchronized slave mode)

```c
/* Prototype */
void int_fsif_empty(void) __attribute__((interrupt_handler));

/*******************************************************************************/
/* int_fsif_empty */
/* Type : void */
/* Ret val : none */
/* Argument : void */
/* Function : FIFO serial I/F ch.0 sending buffer empty interrupt function. */
*******************************************************************************/
void int_fsif_empty(void)
{
    extern volatile int int_empty_flg;
    extern volatile unsigned char str[10];
    extern volatile int i;

    if (i > 9)
    {
        int_empty_flg = TRUE;  // (1)
    }
    else
    {
        /* write sending data */
        *(volatile unsigned char *)FSIF_TXD0_ADDR = str[i];  // (1)
        i++;
    }

    /* clear serial sending buffer empty interrupt flag */
    *(volatile unsigned char *)INT_FFS_ADDR = INT_FFSTX0;  // (2)
}
```

(1) Interrupt processing
The serial interface with FIFO generates a cause of transmit buffer empty interrupt when transmit data is loaded from the transmit data register to the shift register. An interrupt occurs if the transmit buffer empty interrupt has been enabled, and the interrupt handler is executed.

The sample program writes transmit data to the transmit data register and sets a software flag to TRUE after a certain number of data has been transmitted.

(2) Clearing the cause-of-interrupt flag
Reset the cause-of-interrupt flag, because it has been set to 1 by the occurrence of the interrupt.
## 3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

### Serial interface with FIFO (clock-synchronized master mode) program

The sample program located in the GNU33\sample_ide\std\dmt33301\vsif_mst\src directory configures the serial interface with FIFO in clock-synchronized master mode and continuously receives data sent from the slave device to the master device (this chip) using an interrupt. The following describes how to configure the serial interface with FIFO Ch.0 in master mode and other initial settings.

#### Interrupt vector settings for serial interface with FIFO (clock-synchronized master mode)

<table>
<thead>
<tr>
<th>Vector Name</th>
<th>Address</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>(unsigned long)int_fsif_error</td>
<td>448</td>
<td>112 FIFO Serial interface Ch.0</td>
</tr>
<tr>
<td>(unsigned long)int_fsif_full</td>
<td>452</td>
<td>113 FIFO Serial interface Ch.0</td>
</tr>
<tr>
<td>(unsigned long)dummy</td>
<td>456</td>
<td>114 FIFO Serial interface Ch.0</td>
</tr>
</tbody>
</table>

#### Setting the vector table

Register the interrupt handler functions in the vector table.

#### Initialization of serial interface with FIFO (clock-synchronized master mode)

```c
/* Prototype */
void init_fifo_bcu(void);
void init_sync_sif0(void);

/* Function : Initialize BCU for FIFO fifo-serial I/F. */
void init_fifo_bcu(void)
{
    unsigned short temp;
    
    /* Set area 5-18 endian control and external/internal access control 0x48132 */
    temp = *(volatile unsigned short *)BCU_EC_IO_ADDR;
    temp &= 0xfdfd;
    temp |= BCU_A6IO_INT;
    *(volatile unsigned short *)BCU_EC_IO_ADDR = temp;

    /* Set area 4-6 BCU register, fifo i/f selection 0x4812a */
    temp = *(volatile unsigned short *)BCU_A4_A5_A6_ADDR;
    temp &= 0xf0ff;
    temp |= BCU_WTH_1;
    *(volatile unsigned short *)BCU_A4_A5_A6_ADDR = temp;
}

/* Function : Initialize synchronous FIFO serial channel 0. */
void init_sync_sif0(void)
{
    /* Set FIFO serial ch.0 interrupt enable 0x402b1, */
    /* FIFO serial ch.0 interrupt disable */
    *(volatile unsigned char *)INT_EFS_ADDR &= ~ (INT_EFSRX0 | INT_EFSERR0);

    /* Set FIFO serial interface control, each mode disable 0x300203 */
    *(volatile unsigned char *)FSIF_SMD0_ADDR = FSIF_TXEN_DIS | FSIF_RXEN_DIS;

    /* Set FIFO serial interface control, each mode enable 0x300204 */
    *(volatile unsigned char *)FSIF_SMD0_ADDR = FSIF_TXEN_EN | FSIF_RXEN_EN;

    /* Set P00-P03 extended function, fifo i/f port selection 0x300040 */
    *(volatile unsigned char *)FSIF_P00P03_ADDR =
        FSIF_EFP00_FSIN | FSIF_EFP01_FSOUT | FSIF_EFP02_FSClk | FSIF_EFP03_FSRdy;

    /* Set FIFO serial interface IrDA register, i/f mode selection 0x300204 */
    *(volatile unsigned char *)FSIF_IRMD0_ADDR = FSIF_IRMD_ORD;
```
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

/* Set FIFO serial interface control register, */
*(volatile unsigned char *)FSIF_SMD0_ADDR = FSIF_SMD_MAS;  (7)
/* Set FIFO serial interface IrDA register, i/f mode setting 0x300204 */
*(volatile unsigned char *)FSIF_IRMD0_ADDR |=
FSIF_SRDYL_NML | FSIF_FINT0_LV2 | FSIF_DIVMD_16;  (8)
/* Set FIFO serial interface control register, transfer mode setting 0x300203 */
*(volatile unsigned char *)FSIF_SMD0_ADDR |=
FSIF_TXEN_DIS | FSIF_RXEN_ENA | FSIF_SSCK_INT;  (9)
/* Set FIFO serial interface baudrate reload data 0x300206,0x300207, */
/* 9600bps(40M) */
*(volatile unsigned char *)FSIF_BRTRD_LSB = 0x80;  (10)
*(volatile unsigned char *)FSIF_BRTRD_MSB = 0x00;
/* Set FIFO serial interface status register, error flag clear 0x300202 */
*(volatile unsigned char *)FSIF_RDBF0_ADDR &= 0xe3;  (11)
/* Set FIFO serial interface interrupt priority level 3 on interrupt */
/* controller 0x402b0 */
*(volatile unsigned char *)INT_PFSIO_ADDR = INT_PRIH_LVL3;  (12)
/* Set FIFO serial ch.0 interrupt flag 0x402b2, */
/* clear serial ch.0 interrupt flag */
*(volatile unsigned char *)INT_FFS_ADDR =
INT_FFSTX0 | INT_FFSRX0 | INT_FFSERR0;  (13)
/* Set FIFO serial ch.0 interrupt enable 0x402b1, */
/* serial ch.0 interrupt enable */
*(volatile unsigned char *)INT_EFS_ADDR |=
INT_EFSTX0 | INT_EFSRX0 | INT_EFSERR0;  (14)
/* Set FIFO serial interface baudrate control register 0x300205 */
*(volatile unsigned char *)FSIF_BTRUN_ADDR = FSIF_BTRUN_ON;  (15)
}

(1), (2) Setting the BCU
Addresses 0x300200 to 0x300209 in Area 6 are allocated for the control registers of the serial interface with FIFO. Therefore, set the BCU registers to enable the internal device to be accessed in Area 6 before the control registers can be accessed.

(3) Disabling interrupt
Disable interrupts of the serial interface with FIFO to avoid the occurrence of unexpected interrupts.

(4) Disabling data transfer
Disable transmission/reception via the serial interface with FIFO. Setting while the serial interface is active may cause malfunctions.

(5) Configuring the input/output pins
Switch the I/O port pin functions for the serial interface with FIFO. In clock-synchronized master mode, FSIN0, FSOUT0, #FSCLK0, and #FSRDY0 pins are used.

(6) Selecting an interface mode
Select either normal interface or IrDA interface. This setting must be initialized because it is undefined at initial reset. The sample program selects normal interface.

(7) Selecting a transfer mode
Select a transfer mode of the serial interface with FIFO. The sample program selects clock-synchronized master mode.
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

(8) Setting the #FSRDY0 control method and receive FIFO level
Set the #FSRDY0 signal control method at a receive buffer full status and the receive FIFO level. By selecting mask for the #FSRDY0 signal control, the #FSRDY0 signal is fixed at high in receive buffer full status to avoid occurrence of an overrun error. The receive FIFO level is the number of data in the receive data buffer to generate a cause of receive buffer full.
The sample program sets the #FSRDY0 to be output normally and receive FIFO level = 2.

(9) Setting the clock and enabling data transfer
The serial interface with FIFO set in clock-synchronized master mode uses the clock internally generated, therefore select the internal clock for the operating clock.
At the same time, data transfer should be enabled using the same register. The sample program performs data reception in master mode, so enable reception only. The clock-synchronized transfer does not allow enabling of transmit/receive operations simultaneously.

(10) Setting the input clock (baud rate)
The clock-synchronized master mode operates with the clock internally generated. The serial interface with FIFO uses the dedicated baud-rate timer (10-bit programmable timer) to generate the input clock. The sample program sets an initial value to the reload data register in the baud-rate timer so that the baud-rate timer will generate a clock in 9600 bps baud rate when the operating clock = 40 MHz.

(11) Resetting the status register
Reset the error flags in the status register by writing 0.

(12) Setting the interrupt priority level
Set the interrupt priority level. When two or more interrupts occur at the same time, the interrupt that has the highest priority is accepted first. The interrupt priority level is set to 3 in the sample program.

(13) Resetting the cause-of-interrupt flag
Cause-of-interrupt flags are undefined after an initial reset, therefore, reset the flag before enabling an interrupt.

(14) Enabling the interrupt
Enable interrupts of the serial interface with FIFO.
A receive buffer full interrupt will occur during reception and a receive error interrupt may occur.

(15) Starting the baud-rate timer
Set the baud-rate control registers to start the baud-rate timer.

Interrupt handler for serial interface with FIFO (clock-synchronized master mode)
/*@Prototype*/
void int_fsif_error(void) __attribute__((interrupt_handler));
void int_fsif_full(void) __attribute__((interrupt_handler));
/******************************************************************************
* int_fsif_error
* Type : void
* Ret val : none
* Argument : void
* Function : FIFO serial I/F ch.0 receiving error interrupt function.
*******************************************************************************/
void int_fsif_error(void)
{
    extern volatile int int_error_flg;
    int_error_flg = TRUE; (1)
    /* clear serial error interrupt flag */
    *((volatile unsigned char *)INT_FFS_ADDR = INT_FFSERR0); (3)
}
/******************************************************************************
* int_fsif_full
* Type : void
* Ret val : none
* Argument : void
* Function : FIFO serial I/F ch.0 receiving buffer full interrupt function.
*******************************************************************************/
void int_fsif_full(void)
{
    extern volatile unsigned char str[10];
    extern volatile int i;

    /* read receiving data */
    while (*((volatile unsigned char *)FSIF_RDBF0_ADDR & 0x01)!=0)
    {
        str[i] = *((volatile unsigned char *)FSIF_RXD0_ADDR);  //serial error interrupt processing (1)
        //write_hex((unsigned long)str[i]);
        i++;
    }

    /* clear serial receiving buffer full interrupt flag */
    *((volatile unsigned char *)INT_FFS_ADDR) = INT_FFSRX0;   //receive buffer full interrupt processing (2)
}

(1) Serial error interrupt processing
When the serial interface with FIFO detects a parity error, framing error, or overrun error during data receiving, it sets the receive error interrupt cause flag. An interrupt occurs if the serial error interrupt has been enabled, and the interrupt handler is executed. In clock-synchronized mode, a parity error and a framing error will not occur.
The sample program sets a software flag to TRUE to check whether the serial interface interrupt has occurred or not.

(2) Receive buffer full interrupt processing
When the serial interface with FIFO has finished a data reception and received data more than the specified number are loaded to the receive data buffer, it sets the receive buffer full interrupt cause flag. An interrupt occurs if the serial error interrupt has been enabled, and the interrupt handler is executed.
The sample program reads the received data from the receive data register until the receive data buffer becomes empty.

(3) Clearing the cause-of-interrupt flag
Reset the cause-of-interrupt flag, because it has been set to 1 by the occurrence of the interrupt.

After the above settings have been made, clock-synchronized transfer can be started. The following shows the operations of the serial interface with FIFO after data transfer begins.

Data reception (master mode)
1. Waits until the #FSRDY0 signal is set to low by the slave device.
2. Starts inputting the synchronous clock to the serial interface with FIFO after the #FSRDY0 signal goes low.
3. Takes data sent from the slave device into the shift register. When the MSB has been received, the received data is transferred to the receive data buffer. After that, the user program can read the received data.
4. When the number of data in the receive data buffer exceeds the predefined value, a receive buffer full interrupt occurs and it starts executing an interrupt handler.

Data transmission (slave mode)
1. When transmit data is set to the transmit data buffer, the serial interface with FIFO sets the #FSRDY0 signal to low and waits for the clock input from the master device. Up to two bytes of data can be written to the transmit data buffer.
2. The transmit data in the transmit data buffer is loaded to the shift register when the synchronous clock is input to the #FSCCLK0 pin. At this time the #FSRDY0 goes high.
3. The data in the shift register is output to the master device in synchronization with the clock. The #FSRDY0 signal goes low after the MSB has been output to the master device. The serial interface with FIFO repeats Steps 2 and 3 until all data in the transmit data buffer have been output.
4. When the last data in the transmit data buffer has been loaded to the shift register, a transmit buffer empty interrupt occurs and it starts executing an interrupt handler.
The sample program writes data to the transmit data buffer in the interrupt handler executed in transmission Step 4 to transmit data continuously.
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

Serial interface with FIFO (asynchronous mode) program

The sample program located in the GNU\33\sample_idc\std\dmt\33\fifo_async\src directory configures the serial interface with FIFO in asynchronous mode and performs continuous data transfer with the external serial device. The following describes how to initialize serial interface with FIFO Ch.0.

### Interrupt vector settings for serial interface with FIFO (asynchronous mode)

| (unsigned long)int_fsif_error,  // 448  112 FIFO Serial interface Ch.0 |
| (unsigned long)int_fsif_full,  // 452  113 FIFO Serial interface Ch.0 |
| (unsigned long)int_fsif_empty  // 456  114 FIFO Serial interface Ch.0 |

#### Setting the vector table

Register the interrupt handler functions in the vector table.

#### Initialization of serial interface with FIFO (asynchronous reception)

```c
#include <bitset> // for bit set operations

#define BCU_A6IO_INT   0x48132
#define BCU_WTH_1      0x300203
#define FSIF_SMD_8BIT  0x300203
#define FSIF_TXEN_DIS  0x01
#define FSIF_RXEN_DIS  0x01
#define FSIF_IRMD_ORD  0x01
#define FSIF_EFP01_FSOUT 0x01
#define FSIF_EFP00_FSIN 0x01

void init_fifo_bcu(void)
{
    unsigned short temp;
    /* Set area 5-18 endian control and external/internal access control 0x48132 */
    temp = *(volatile unsigned short *)BCU_EC_IO_ADDR;
    temp &= 0xfdfd;
    temp |= BCU_A6IO_INT;
    *(volatile unsigned short *)BCU_EC_IO_ADDR = temp;

    /* Set area 4-6 BCU register, fifo i/f selection 0x4812a */
    temp = *(volatile unsigned short *)BCU_A4_A5_A6_ADDR;
    temp &= 0xf0ff;
    temp |= BCU_WTH_1;
    *(volatile unsigned short *)BCU_A4_A5_A6_ADDR = temp;
}

void init_async_fsif0(void)
{
    /* Set FIFO serial ch.0 interrupt enable 0x402b1, */
    /* FIFO serial ch.0 interrupt disable */
    *(volatile unsigned short *)INT_EFS_ADDR &=
    ~(INT_EFSTX0 | INT_EFSRX0 | INT_EFSERR0);

    /* Set FIFO serial interface control, each mode disable 0x300203 */
    *(volatile unsigned short *)FSIF_SMD0_ADDR =
    FSIF_TXEN_DIS | FSIF_RXEN_DIS;

    /* Set FIFO serial interface IrDA register, i/f mode selection 0x300204 */
    *(volatile unsigned short *)FSIF_IRMD0_ADDR =
    FSIF_IRMD_ORD;

    /* Set FIFO serial interface control register, transfer mode 8bit 0x300203 */
    *(volatile unsigned short *)FSIF_SMD0_ADDR = FSIF_SMD_8BIT;
}
```
(1), (2) Setting the BCU
Addresses 0x300200 to 0x300209 in Area 6 are allocated for the control registers of the serial interface with FIFO. Therefore, set the BCU registers to enable the internal device to be accessed in Area 6 before the control registers can be accessed.

(3) Disabling interrupt
Disable interrupts of the serial interface with FIFO to avoid the occurrence of unexpected interrupts.

(4) Disabling data transfer
Disable transmission/reception via the serial interface with FIFO. Setting while the serial interface is active may cause malfunctions.

(5) Configuring the input/output pins
Switch the I/O port pin functions for the serial interface with FIFO. In asynchronous mode, FSIN0 and FSOUT0 pins are used.

(6) Selecting an interface mode
Select either normal interface or IrDA interface. This setting must be initialized because it is undefined at initial reset. The sample program selects normal interface.

(7) Selecting a transfer mode
Select a transfer mode of the serial interface with FIFO. The sample program selects asynchronous mode.
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

(8) Setting the #FSRDY0 control method, receive FIFO level, and clock division ratio
Set the #FSRDY0 signal control method at a receive buffer full status, the receive FIFO level, and the
division ratio for the clock in asynchronous mode. By selecting mask for the #FSRDY0 signal control, the
#FSRDY0 signal is fixed at high in receive buffer full status to avoid occurrence of an overrun error. The
receive FIFO level is the number of data in the receive data buffer to generate a cause of receive buffer full.
Also the same register is used to set the division ratio for the clock in asynchronous mode.
The sample program sets the #FSRDY0 to be output normally, receive FIFO level = 2, and clock division
ratio = 1/16.

(9) Setting the input clock, transfer enable bit, and data format
Either an internal clock or external clock can be selected in asynchronous mode. At the same time,
data transfer should be enabled and a data format should be selected. The asynchronous mode allows
configuration of the data bit length, stop bit, and parity bit.
The sample program selects internal clock, data bit length = 8 bits, stop bit = 1 bit, with no parity bit, and
enables transmission.

(10) Resetting the status register
Reset the error flags in the status register by writing 0.

(11) Setting the input clock (baud rate)
When internal clock is selected, the serial interface with FIFO uses the dedicated baud-rate timer (10-bit
programmable timer) to generate the input clock. The sample program sets an initial value to the reload data
register in the baud-rate timer so that the baud-rate timer will generate a clock in 9600 bps baud rate when
the operating clock = 40 MHz.

(12) Setting the interrupt priority level
Set the interrupt priority level. When two or more interrupts occur at the same time, the interrupt that has
the highest priority is accepted first. The interrupt priority level is set to 3 in the sample program.

(13) Resetting the cause-of-interrupt flag
Cause-of-interrupt flags are undefined after an initial reset, therefore, reset the flag before enabling an
interrupt.

(14) Enabling the interrupt
Enable interrupts of the serial interface with FIFO.
A receive buffer full interrupt will occur during reception and a receive error interrupt may occur.

(15) Starting the baud-rate timer
Set the baud-rate control registers to start the baud-rate timer.

Interrupt handler for serial interface with FIFO (asynchronous reception)
Refer to “Interrupt handler for serial interface with FIFO (clock-synchronized master mode)” for the receive
interrupts (receive buffer full interrupt and serial error interrupt) that occur in asynchronous transfer.

Initialization of serial interface with FIFO (asynchronous transmission)

```c
/* Prototype */
void set_fsif_mode(unsigned char);

/******************* set_fsif_mode(unsigned char) ********************/

/* main */
* Type : void
* Ret val : none
* Argument : void
* Function : FIFO serial interface demonstration program.
***************************************************************************/
int main(void)
{
    unsigned char sif_mode;

    /* Set FIFO serial interface ch.0 transmit mode */
    sif_mode = FSIF_TXEN_ENA | FSIF_RXEN_DIS | FSIF_EPR_OFF |
                   FSIF_PMD_EVEN | FSIF_STPB_1 | FSIF_SCKINT | FSIF_SMD_8BIT; // (1)
```
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

```c
set_fsif_mode(sif_mode);
}

/******************************************************************************
* set_fsif_mode
* Type :     void
* Ret val :  none
* Argument : unsigned char mode      Serial mode
* Function : Set FIFO serial interface mode.
*******************************************************************************/
void set_fsif_mode(unsigned char mode)
{
  /* Set FIFO serial ch.0 interrupt enable 0x402b1, */
  /* FIFO serial ch.0 interrupt disable */
  *(volatile unsigned char *)INT_EFS_ADDR &= ~INT_EFSTX0 | INT_EFSRX0 | INT_EFSERR0;

  /* Set FIFO serial interface control, each mode disable 0x300203 */
  *(volatile unsigned char *)FSIF_SMD0_ADDR = FSIF_TXEN_DIS | FSIF_RXEN_DIS;

  /* Set FIFO serial control register 0x300203 */
  *(volatile unsigned char *)FSIF_SMD0_ADDR = mode;

  /* Clear FIFO serial status 0x300202 */
  *(volatile unsigned char *)FSIF_RDBF0_ADDR = FSIF_ERR_NON;

  /* Set FIFO serial ch.0 interrupt flag 0x402b2, */
  /* clear serial ch.0 interrupt flag */
  *(volatile unsigned char *)INT_FFS_ADDR = INT_FFSTX0 | INT_FFSRX0 | INT_FFSERR0;

  /* Set FIFO serial ch.0 interrupt enable 0x402b1, serial ch.0 interrupt enable */
  *(volatile unsigned char *)INT_EFS_ADDR &= INT_EFSTX0 | INT_EFSRX0 | INT_EFSERR0;
}
```

(1) Setting the transfer mode, input clock, transfer enable bit, and data format
Select a transfer mode of the serial interface with FIFO. For the operating clock, either an internal clock or
external clock can be selected in asynchronous mode. At the same time, data transfer should be enabled and
a data format should be selected. The asynchronous mode allows configuration of the data bit length, stop
bit, and parity bit.
The sample program configures the serial interface with FIFO in
8-bit asynchronous mode, internal clock
used, data bit length = 8 bits, stop bit = 1 bit, with no parity bit, and enables transmission.

(2) Disabling interrupt
Disable interrupts of the serial interface with FIFO to avoid the occurrence of unexpected interrupts.

(3) Disabling data transfer
Disable transmission/reception via the serial interface with FIFO. Setting while the serial interface is active
may cause malfunctions.

(4) Setting the transmit parameters
Set the transmit parameters prepared in Step (1) into the register.

(5) Resetting the status register
Reset the error flags in the status register by writing 0.

(6) Resetting the cause-of-interrupt flag
Cause-of-interrupt flags are undefined after an initial reset, therefore, reset the flag before enabling an
interrupt.

(7) Enabling the interrupt
Enable interrupts of the serial interface with FIFO.
A transmit buffer empty interrupt will occur during transmission.
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

Interrupt handler for serial interface with FIFO (asynchronous transmission)
Refer to “Interrupt handler for serial interface with FIFO (clock-synchronized slave mode)” for the transmission interrupt (transmit buffer empty interrupt) that occur in asynchronous transfer.

After the above settings have been made, asynchronous transfer can be started. The following shows the operations of the serial interface with FIFO after data transfer begins.

Data reception
1. The serial interface with FIFO starts data sampling when a start bit is input.
2. After the start bit is sampled, the data bits that follow are taken into the shift register from the LSB to the MSB.
3. The parity and stop bits that follow the MSB are sampled.
4. After the stop bit has been sampled, data in the shift register is loaded to the receive data buffer. The data loaded to the receive data buffer can be read out. The parity check is performed when the received data is loaded to the receive data buffer.
5. When the number of data in the receive data buffer exceeds the predefined value, a receive buffer full interrupt occurs and it starts executing an interrupt handler.

Data transmission
1. The serial interface with FIFO transfers data in the transmit data buffer to the shift register in synchronization with the sampling clock. At the same time, it outputs a start bit from the FSOUT0 pin.
2. After the start bit has been output, the data bits are output in order from the LSB to the MSB in synchronization with the rising edge of the clock.
3. The parity and stop bits are output following the MSB.
   The serial interface with FIFO repeats Steps 1 to 3 until all data in the transmit data buffer have been output.
4. When the last data in the transmit data buffer has been loaded to the shift register, a transmit buffer empty interrupt occurs and it starts executing an interrupt handler.

The sample program writes data to the transmit data buffer in the interrupt handler executed in transmission Step 4 to transmit data continuously.
3.8 Port Interrupts

This section shows an example of an input interrupt control program. The sample program is located in the GNU33\sample_id\std\dtm33301\gpio\gpio_key\src directory. There are two types of input interrupts are available, port input interrupt and key input interrupt. A port input interrupt occurs at the signal edge or signal level input to the port selected. A key input interrupt occurs when the pin statuses of the port group selected and the input comparison register value become unmatched.

Port input interrupt control program

This sample program sets the port input interrupt (FPT0) to be generated at the rising edge of the K50 port input. Below describes initialization of the port input interrupt condition and handling the interrupt.

Interrupt vector settings for port input interrupts

| (unsigned long) int_io0, // 64 | 16 Port input interrupt 0 |
| (unsigned long) dummy, // 68 | 17 Port input interrupt 1 |
| (unsigned long) dummy, // 72 | 18 Port input interrupt 2 |
| (unsigned long) dummy, // 76 | 19 Port input interrupt 3 |
| (unsigned long) dummy, // 272 | 68 Port input interrupt 4 |
| (unsigned long) dummy, // 276 | 69 Port input interrupt 5 |
| (unsigned long) dummy, // 280 | 70 Port input interrupt 6 |
| (unsigned long) dummy, // 284 | 71 Port input interrupt 7 |

Initialization of port input interrupts

/* Prototype */
void init_port(void);

/*******************************************************************************
* init_port
* Type :     void
* Ret val :  none
* Argument : void
* Function : Initialize I/O port function.
*******************************************************************************/

void init_port(void)
{
    unsigned char temp;

    /* input port0-3 and key input0,1 interrupt enable 0x40270, */
    /* input port0 interrupt disable */
    *(volatile unsigned char *)INT_EP0_EK_ADDR &= ~INT_EP0;   // (1)

    /* Set input/output port function 0x402c0, port K50 */
    *(volatile unsigned char *)IN_CFK5_ADDR &= 0xfe;          // (2)

    /* FPT0-FPT3 interrupt port selection 0x402c6, FPT0 K50 */
    temp = *(volatile unsigned char *)IN_SPT0_SPT3_ADDR;
    temp &= 0xfc;
    temp |= 0x1;
    *(volatile unsigned char *)IN_SPT0_SPT3_ADDR |= temp;    // (3)

    /* FPT0-FPT7 interrupt input polarity selection 0x402c8, */
    /* FPT0 high level or edge rising */
    *(volatile unsigned char *)IN_SPP_ADDR |= 0x01;           // (4)

    /* FPT0-FPT7 interrupt edge/level selection 0x402c9, FPT0 edge */
    *(volatile unsigned char *)IN_SEP_ADDR |= 0x01;          // (5)

    /* input port0-3, HSDMA and 16bit timer0 IDMA request 0x40290, */
    /* input port0 CPU request */
    *(volatile unsigned char *)INT_RP0_RHDM_R16T0_ADDR &= 0xfe; // (6)

    /* port input0,1 interrupt priority register 0x40260, */
    /* input port0 priority level 3 */
    temp = *(volatile unsigned char *)INT_PP0_PP1_ADDR;
    temp &= 0xf0;
    temp |= INT_PRIL_LVL3;                                   // (7)
    *(volatile unsigned char *)INT_PP0_PP1_ADDR = temp;
}
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

/* input port0-3 and key input0,1 interrupt factor flag 0x40280. */
/* input port0 reset interrupt flag */
*(volatile unsigned char *)INT_FP0_PK_ADDR = INT_FP0;
/* input port0-3 and key input0,1 interrupt enable 0x40270. */
/* input port0 interrupt enable */
*(volatile unsigned char *)INT_EP0_EK_ADDR |= INT_EP0;
}

(1) Disabling interrupt
Disable the port input interrupt to avoid the occurrence of unexpected interrupts.

(2) Setting the input port
Select the port function to be used.
The sample program configures K50 as an input port.

(3) Selecting the input pin
Select the pin to be used for the port input interrupt.
The table below lists the ports assigned to each cause of interrupt.

Table 3.8.1 Selecting Pins for Port Input Interrupts

<table>
<thead>
<tr>
<th>Cause of interrupt</th>
<th>Control bit</th>
<th>SPT settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPT7</td>
<td>SPT7[1:0]</td>
<td>P27 P07 P33 K67</td>
</tr>
<tr>
<td>FPT6</td>
<td>SPT6[1:0]</td>
<td>P26 P06 P32 K66</td>
</tr>
<tr>
<td>FPT5</td>
<td>SPT5[1:0]</td>
<td>P25 P05 P31 K65</td>
</tr>
<tr>
<td>FPT4</td>
<td>SPT4[1:0]</td>
<td>P24 P04 K64 K64</td>
</tr>
<tr>
<td>FPT3</td>
<td>SPT3[1:0]</td>
<td>P23 P03 K63 K63</td>
</tr>
<tr>
<td>FPT2</td>
<td>SPT2[1:0]</td>
<td>P22 P02 K62 K62</td>
</tr>
<tr>
<td>FPT1</td>
<td>SPT1[1:0]</td>
<td>P21 P01 K61 K61</td>
</tr>
<tr>
<td>FPT0</td>
<td>SPT0[1:0]</td>
<td>P20 P00 K50 K60</td>
</tr>
</tbody>
</table>

The sample program selects K50 (FPT0) as the cause of interrupt.

(4), (5) Setting the interrupt condition
Select a condition to generate port input interrupts. A port input interrupt can be generated due to the input signal level or at the signal edge. The table below lists the port input interrupt conditions.

Table 3.8.2 Port Input Interrupt Condition

<table>
<thead>
<tr>
<th>SEPTx (0x402C9•Dx)</th>
<th>SPPTx (0x402C8•Dx)</th>
<th>FPTx interrupt condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Rising edge</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Falling edge</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>High level</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Low level</td>
</tr>
</tbody>
</table>

The sample program selects the rising edge for the FPT0 interrupt condition.

(6) Setting interrupt[IDMA] request
Select whether the cause of interrupt is used to request an interrupt to the CPU or to request an IDMA transfer. The sample program selects interrupt request.

(7) Setting the interrupt priority level
Set the interrupt priority level. When two or more interrupts occur at the same time, the interrupt that has the highest priority is accepted first. The interrupt priority level is set to 3 in the sample program.

(8) Resetting the cause-of-interrupt flag
Cause-of-interrupt flags are undefined after an initial reset, therefore, reset the flag before enabling an interrupt.

(9) Enabling the interrupt
Enable the port input interrupt.
After this, an interrupt will occur at the rising edge of the K50 port input signal.
Port input interrupt handler

/* Prototype */
void int_io0(void) __attribute__((interrupt_handler));

/*******************************************************************************
* int_io0
*   Type :     void
*   Ret val :  none
*   Argument : void
*   Function : I/O port FPT0 interrupt function.
*******************************************************************************/
void int_io0(void)
{
    unsigned int j;
    volatile extern unsigned char int_io0_flg;

    int_io0_flg = TRUE;

    /* input port0-3 and key input0-1 interrupt factor flag 0x40280, */
    /* reset interrupt factor flag */
    *(volatile unsigned char *)INT_FP0_FK_ADDR = INT_FP0;
}

(1) Interrupt processing
An interrupt occurs when the selected port has met the set interrupt conditions. When an interrupt occurs,
the CPU executes the interrupt handler function that has been described in the vector table.
The sample program sets a software flag used to check whether the port input interrupt has occurred or not.

(2) Clearing the cause-of-interrupt flag
Reset the cause-of-interrupt flag, because it has been set to 1 by the occurrence of the interrupt.

Key input interrupt control program
This sample program sets the K50 port to generate a key input interrupt (FPK0) when the input status and the
input comparison register value become unmatched. Below describes initialization of the key input interrupt
condition and handling the interrupt.

Interrupt vector settings for key input interrupts
(unsigned long)int_key0, // 80   20 Key input interrupt 0
(unsigned long)dummy,  // 84   21 Key input interrupt 1

Initialization of key input interrupts
/* Prototype */
void init_port(void);

/*******************************************************************************
* init_port
*   Type :     void
*   Ret val :  none
*   Argument : void
*   Function : Initialize I/O port function.
*******************************************************************************/
void init_port(void)
{
    unsigned char temp;

    /* Address for input port0-3 and key input0,1 */
    /* interrupt enable register 0x40270 */
    *(volatile unsigned char *)INT_EP0_EK_ADDR &= ~INT_EK0;
    /* set input/output port function 0x402c0, port K50 */
    *(volatile unsigned char *)IN_CFK5_ADDR &= 0xfe;
    temp = *(volatile unsigned char *)IN_SPPK_ADDR;
    temp &= 0xfc;
    temp |= 0x0;

    /* Address for FPK0-FPK1 interrupt port selection register 0x402ca, port K50 */
    temp = *(volatile unsigned char *)IN_SPPK_ADDR;
    temp &= 0xfc;
    temp |= 0x0;
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

*(volatile unsigned char *)IN_SPPK_ADDR |= temp;
/* FPK0 input mask register 0x402ce, FPK00 enable */
*(volatile unsigned char *)IN_SMPK0_ADDR |= 0x1;
/* FPK0 input comparison register 0x402cc, FPK00 high */
*(volatile unsigned char *)IN_SCPK0_ADDR |= 0x1;
/* Address for port input0,1 interrupt priority register 0x40260 */
temp = *(volatile unsigned char *)INT_PK0_PK1_ADDR;
temp &= 0xf0;
temp |= INT_PRIL_LVL3;
*(volatile unsigned char *)INT_PK0_PK1_ADDR = temp;
/* Address for input port0-3 and key input0,1 */
/* interrupt factor flag register 0x40280 */
*(volatile unsigned char *)INT_FP0_FK_ADDR = INT_FK0;
/* Address for input port0-3 and key input0,1 */
/* interrupt enable register 0x40270 */
*(volatile unsigned char *)INT_EP0_EK_ADDR |= INT_EK0;
}

(1) Disabling interrupt
Disable the key input interrupt to avoid the occurrence of unexpected interrupts.

(2) Setting the input port
Select the port function to be used.
The sample program configures K50 as an input port.

(3) Selecting the input pin
Select the pin to be used for key input interrupt.
The table below lists the ports assigned to each cause of interrupt.

<p>| Table 3.8.3 Selecting Pins for Key Input Interrupts |
|-------------------------------|----------------|</p>
<table>
<thead>
<tr>
<th>Cause of interrupt</th>
<th>Control bit</th>
<th>SPPK settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPK1</td>
<td>SPPK1[1:0]</td>
<td>P2[7:4]</td>
</tr>
<tr>
<td></td>
<td>(0x402CA+D[3:2])</td>
<td>K5[7:4]</td>
</tr>
<tr>
<td>FPK0</td>
<td>SPPK0[1:0]</td>
<td>P2[4:0]</td>
</tr>
<tr>
<td></td>
<td>(0x402CA+D[1:0])</td>
<td>K5[4:0]</td>
</tr>
</tbody>
</table>

The sample program selects K5[4:0] (FPK0) as the cause of interrupt.

(4) Setting the input mask register
Select the ports to be used for key input interrupt.
An interrupt occurs when the ports selected here have met the interrupt condition.
The sample program enables the K50 interrupt (SMPK00) only.

(5) Setting the input comparison register
Set the interrupt condition for the ports that have been enabled to generate interrupts in Step (4).
The sample program selects the rising edge.

(6) Setting the interrupt priority level
Set the interrupt priority level. When two or more interrupts occur at the same time, the interrupt that has the highest priority is accepted first. The interrupt priority level is set to 3 in the sample program.

(7) Resetting the cause-of-interrupt flag
Cause-of-interrupt flags are undefined after an initial reset, therefore, reset the flag before enabling an interrupt.

(8) Enabling the interrupt
Enable the key input interrupt.
After this, an interrupt will occur at the rising edge of the K50 port input signal.
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

Key input interrupt handler

```c
/* Prototype */
void int_key0(void) __attribute__((interrupt_handler));

/*******************************************************************************
* int_key0
*   Type :     void
*   Ret val :  none
*   Argument : void
*   Function : I/O port FPK0 interrupt function.
*******************************************************************************/
void int_key0(void)
{
    unsigned int j;
    volatile extern unsigned char int_key0_flg;

    int_key0_flg = TRUE;  \(1\)

    /* Reset key input0 interrupt factor flag 0x40280 */
    *(volatile unsigned char *)INT_FP0_FK_ADDR = INT_FK0;  \(2\)
}
```

(1) Interrupt processing

An interrupt occurs when the selected input port status and the input comparison register value become unmatched. When an interrupt occurs, the CPU executes the interrupt handler function that has been described in the vector table.

The sample program sets a software flag used to check whether the key input interrupt has occurred or not.

(2) Clearing the cause-of-interrupt flag

Reset the cause-of-interrupt flag, because it has been set to 1 by the occurrence of the interrupt.
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

3.9 A/D Conversion

This section shows an example of the A/D conversion program that uses a software trigger. The sample program is located in the GNU33\sample_id\dmt33301\ad\src directory. In the S1C33 chip, analog inputs to the specified pins can be A/D converted using the on-chip A/D converter.

A/D conversion control program

The sample program executes A/D conversion for the voltage applied between the AD0 and Vss pins 128 times in continuous mode. Below describes initialization of the A/D converter and handling the interrupt.

Interrupt vector settings for A/D converter

(unsigned long)int_ad, // 256  64 A/D converter

Setting the vector table

Register the interrupt handler functions in the vector table.

Initialization of A/D converter

/* Prototype */
void init_ad(void);

void init_ad(void)
{
    unsigned char temp;
    /* Set A/D converter interrupt enable on interrupt controller 0x40277 */ (1)
    *(volatile unsigned char *)INT_EADE_ECTM_EP4_ADDR &= ~INT_EADE;
    /* Set A/D converter interrupt disable */
    /* Set A/D converter enable 0x40244 */ (2)
    *(volatile unsigned char *)AD_OWE_ADDR &= ~AD_ADE_ENA;
    /* A/D disable */
    /* Set A/D converter port select 0x402c3 */ (3)
    *(volatile unsigned char *)IN_CFK6_ADDR = IN_CFK60_AD0;
    /* A/D AD0 port */
    /* Set A/D operation mode select 0x4025f */ (4)
    *(volatile unsigned char *)AD_CADV_ADDR = AD_CADV_CMP;
    /* 209 compatible mode */
    /* Set A/D converter prescaler set 0x4014f */ (5)
    *(volatile unsigned char *)PRESC_PSAD_ADDR =
        PRESC_PTONL_ON | PRESC_CLKDIVL_SEL4;
    /* Set A/D converter prescaler(CLK/32) */
    /* Set A/D converter status register 0x40242, 0x40243, 0x40244, 0x40245 */ (6)
    *(volatile unsigned char *)AD_CH_ADDR = AD_MS_CON | AD_TS_SOFT;
    /* A/D converter software trigger and continuous mode */
    *(volatile unsigned char *)AD_CS_ADDR = AD_CS_0 | AD_CE_0; (7)
    /* A/D converter start channel AD0 and A/D end channel AD0 */
    *(volatile unsigned char *)AD_OWE_ADDR =
        AD_ADE_ENA | AD_ADST_STOP | AD_OWE_NOERR; (8)
    /* A/D converter enable, A/D converter stop, */
    /* A/D converter over write error clear */
    *(volatile unsigned char *)AD_ST_ADDR = AD_ST_9; (9)
    /* A/D converter sampling 9 clocks */
    /* Set A/D converter interrupt CPU request on interrupt controller 0x40293 */ (10)
    *(volatile unsigned char *)INT_RS1_RADE_RP4_ADDR = ~INT_RADE;
    /* IDMA request disable and CPU request enable */
    /* Set A/D converter interrupt priority level 3 on interrupt controller */ (11)
    /* 0x4026a */
    temp = *(volatile unsigned char *)INT_PRI1_pad_ADDR;
    temp &= 0x0f;
    temp |= INT_PRIH_LVL3;
    *(volatile unsigned char *)INT_PRI1_PAD_ADDR = temp;
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

/* Reset A/D converter interrupt factor flag on interrupt controller 0x40287 */
*(volatile unsigned char *)INT_FADE_FCTM_FP4_ADDR = INT_FADE; (12)

// Reset A/D converter interrupt factor flag

/* Set A/D converter interrupt enable on interrupt controller 0x40277 */
*(volatile unsigned char *)INT_EADE_ECTM_EP4_ADDR |= INT_EADE; (13)

// Set A/D converter interrupt enable

(1) Disabling interrupt
Disable the A/D converter interrupt to avoid the occurrence of unexpected interrupts.

(2) Disabling the A/D converter
Disable the A/D converter to avoid erroneous operations during setup.

(3) Selecting the input pins
Select the input pins to be used for the A/D converter.
The sample program uses K60 (AD0) only.

(4) Setting the operation mode
Select either 33209 compatible mode of which functions are compatible with the C33 analog block for the existing models or the advanced mode allowing use of the extended functions. Table below shows differences between 33209 compatible mode and advanced mode.

<table>
<thead>
<tr>
<th>Function</th>
<th>33209 compatible mode</th>
<th>Advanced mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reading conversion results</td>
<td>The conversion results are read from the A/D conversion result register common to all channels. When converting for multiple channels, the A/D conversion result register must be read before conversion for the next channel has completed.</td>
<td>The conversion results can be read from the conversion result buffer provided for each channel. Thus the conversion result for the current channel will not be lost even when the conversion for the next channel is completed during a multiple channel conversion.</td>
</tr>
<tr>
<td>Conversion-complete flag, overwrite error flag</td>
<td>One bit is assigned for the flag and is commonly used in all channels.</td>
<td>Different flags are provided for each channel.</td>
</tr>
<tr>
<td>Comparison with upper/lower-limit values</td>
<td>Not supported.</td>
<td>An upper-limit value and a lower-limit value can be set and conversion results of the specified channel can be checked whether they are within the specified range or not.</td>
</tr>
<tr>
<td>Intermits</td>
<td>Conversion-complete interrupt only can be generated. The interrupts cannot be masked in channel units.</td>
<td>Conversion-complete interrupts and out-of-range interrupts can be generated. Conversion complete interrupts for the specified channels can be masked.</td>
</tr>
</tbody>
</table>

The sample program selects 33209 compatible mode.

(5) Selecting the input clock and starting clock supply
Select a prescaler divided clock and start supplying it to the A/D converter.
The sample program selects the 1/32 clock divided by the prescaler as the input clock.

(6) Setting an A/D conversion mode and a trigger source
Select an A/D conversion mode and a trigger source to start conversion.
The A/D converter has two conversion modes selectable, normal mode and continuous mode. In normal mode, A/D conversion is performed successively within the specified range of A/D channels and is completed after these conversions are executed in one operation. In the continuous mode, A/D conversion within the specified range of A/D channels is executed repeatedly until the software stops.
The trigger source to start A/D conversion can be selected from the external trigger, 8-bit timer 0, 16-bit timer 0 and software trigger. The sample program selects continuous mode and a software trigger.

(7) Selecting A/D conversion start/end channels
Select the A/D conversion start and end channels within the channels that have been configured for analog inputs. The A/D converter performs A/D conversion successively from the start channel to the end channel in one process. The sample program selects channel 0 (AD0) only.
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

(8) Enabling A/D conversion and resetting the error flag
Set A/D enable bit to enable A/D conversion.
At the same time, reset the overwrite error flag.

(9) Setting the sampling time
Set the time for sampling the analog input signal.
The control register allows selection from four conditions. However, use the A/D converter with the default
sampling time (nine clocks).

(10) Setting interrupt/IDMA request
Select whether the A/D conversion completed interrupt cause is used to request an interrupt to the CPU or
to request an IDMA transfer. The sample program selects interrupt request.

(11) Setting the interrupt priority level
Set the interrupt priority level. When two or more interrupts occur at the same time, the interrupt that has
the highest priority is accepted first. The interrupt priority level is set to 3 in the sample program.

(12) Resetting the cause-of-interrupt flag
Cause-of-interrupt flags are undefined after an initial reset, therefore, reset the flag before enabling an
interrupt.

(13) Enabling the interrupt
Enable the A/D converter interrupt. The A/D converter will be able to generate an interrupt after it finishes
an A/D conversion.

Interrupt handler
/* Prototype */
void int_ad(void) __attribute__((interrupt_handler));

*******************************************************************************
* 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_adr; // A/D data address
    extern volatile int ad_int_flg;           // A/D converter interrupt flag
    *ad_adr = read_ad_data();                // Read A/D converter data
    write_hex(*ad_adr);                     // Output A/D converter data
    ad_adr++;
    ad_int_flg = TRUE;                      // A/D converter interrupt flag on
    /* Reset A/D converter interrupt factor flag 0x40287 */
    *(volatile unsigned char *)INT_FADE_FCTM_FP4_ADDR = INT_FADE;
}

(1) Executing the interrupt processing
When an interrupt occurs, the CPU executes the interrupt handler function that has been described in the
vector table.
The sample program reads the conversion results and displays them in the [Simulated I/O] window.
Furthermore, it sets a software flag used to check whether the A/D converter interrupt has occurred or not.

(2) Clearing the cause-of-interrupt flag
Reset the cause-of-interrupt flag, because it has been set to 1 by the occurrence of the interrupt.
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

3.10 HSDMA Transfer

This section shows an example of HSDMA transfer program. The sample program is located in the GNU33\sample_ide\std\dmt\hsdma\src directory. The HSDMA instantaneously responds to a DMA request for data transfer.

HSDMA transfer program

The sample program invokes HSDMA Ch.1 by issuing a software trigger to transfer data from the source address to the destination address. Below describes initialization of the HSDMA and handling the interrupt.

Interrupt vector settings for HSDMA

<table>
<thead>
<tr>
<th>Function</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt vector</td>
<td>dummy</td>
<td>88</td>
</tr>
<tr>
<td></td>
<td>int_hsdma1</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>dummy</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>dummy</td>
<td>100</td>
</tr>
</tbody>
</table>

Setting the vector table

Register the interrupt handler functions in the vector table.

Initialization of HSDMA

```c
/* Prototype */
void init_hsdma(unsigned char, unsigned long, unsigned long);

unsigned long dummy, // 88 22 High-speed DMA Ch.0
(unsigned long)int_hsdma1, // 92 23 High-speed DMA Ch.1
(unsigned long)dummy, // 96 24 High-speed DMA Ch.2
(unsigned long)dummy, // 100 25 High-speed DMA Ch.3
```

```c
void init_hsdma(unsigned char ch, unsigned long src, unsigned long dst)
{
    /* Set HSDMA interrupt enable on interrupt controller 0x40271 */
    *(volatile unsigned char *)INT_EHDM_EIDM_ADDR &= ~INT_EHSDMA1;
    /* Disable HSDMA transfer 0x4823c, disable HSDMA transfer */
    *(volatile unsigned char *)HSDMA_HS1EN_ADDR &= ~HSDMA_HSEN_ENA;
    /* HSDMA trigger mode 0x40298, software trigger */
    *(volatile unsigned char *)HSDMA_HSD0S_HSD1S_ADDR = HSDMA_HSD1_SOFT;
    /* HSDMA mode and control 0x48232, dual mode */
    *(volatile unsigned short *)(HSDMA_TC0H_D0DIR_DUALM0_ADDR + ch * 0x10) = HSDMA_DUAL_DUAL;
    /* HSDMA destination address 0x48238, destination address increment, */
    /* destination address 0xc0000 */
    *(volatile unsigned long *)(HSDMA_D0ADRL_ADDR + ch * 0x10) =
        HSDMA_DATSIZE_HALF | HSDMA_INC_INIT | dst;
    /* HSDMA source address 0x48234, HWsize, increment, source address 0x80000 */
    *(volatile unsigned long *)(HSDMA_S0ADRL_ADDR + ch * 0x10) =
        HSDMA_DATSIZE_HALF | HSDMA_INC_INIT | src;
    /* HSDMA transfer count and block size 0x48230, count 1, size 0x80 */
    *(volatile unsigned short *)(HSDMA_BLKLEN0_TC0L_ADDR + ch * 0x10) =
        0x0100 | 0x0080;
    /* Set HSDMA interrupt priority level 3 on interrupt controller 0x40263 */
    *(volatile unsigned char *)INT_PHSD0_PHSD1_ADDR = INT_PRIH_LVL3;
    /* Reset HSDMA interrupt flag on interrupt controller 0x40281 */
    *(volatile unsigned char *)INT_FHDM_FIDM_ADDR = INT_FHSDMA1;
    /* Set HSDMA interrupt enable on interrupt controller 0x40271 */
}
```
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

*(volatile unsigned char *)INT_EHDM_EIDM_ADDR |= INT_EHSDMA1;  
/* Enable IDMA transfer 0x4823c, Enable HSDMA transfer */
*(volatile unsigned char *)HSDMA_HS1EN_ADDR |= HSDMA_HSEN_ENA;
}

(1) Disabling interrupt
Disable the HSDMA interrupt to avoid the occurrence of unexpected interrupts.

(2) Disabling the HSDMA channel
Disable the HSDMA channel before setting the control information.

(3) Selecting a trigger source
Select a trigger source to invoke HSDMA. When selecting a cause of interrupt as the trigger source, the HSDMA activates due to occurrence of the cause of interrupt. Note that the HSDMA does not reset the cause-of-interrupt flag that has been set to generate the interrupt. If the interrupt used as the trigger source has been enabled, the interrupt request is output to the CPU after a DMA transfer has finished. The sample program selects a software trigger.

(4) Setting the address mode, transfer counter (high-order bits), and transfer direction (only in single address mode)
Select either single address mode or dual address mode. Also select a transfer direction when using single address mode.
The transfer counter setup bits in this register correspond to D[23:16] of the transfer counter in single/continuous transfer mode, or D[15:8] in block transfer mode.
The sample program selects dual address mode and sets the transfer counter for block transfer.

(5) Setting the destination address and transfer mode
Set the data transfer destination address and a destination address control method. The destination address control method can be selected from increment, decrement, and fixed. The HSDMA transfer mode can be selected from single transfer, continuous transfer, and block transfer modes.
The sample program sets the destination address to 0xcd0000 and selects block transfer mode and destination address increment control.

(6) Setting the transfer data size and source address
Set the transfer data bit size, source address and a source address control method.
The sample program sets the transfer data size to 16 bits and the source address to 0xcc0000, and select source address increment control.

(7) Setting the block size and transfer counter (low-order bits)
Set the number of data transfer (transfer counter low-order bits). The transfer counter setup bits in this register correspond to D[15:0] of the transfer counter in single/continuous transfer mode, or D[7:0] in block transfer mode. When using block transfer mode, set the block size for one transfer. Be sure to set anything other than 0 for the block size.
The sample program sets the number of transfer to one and the block size to 0x80.

(8) Setting the interrupt priority level
Set the interrupt priority level. When two or more interrupts occur at the same time, the interrupt that has the highest priority is accepted first. The interrupt priority level is set to 3 in the sample program.

(9) Resetting the cause-of-interrupt flag
Reset the cause-of-interrupt flag before enabling the interrupt to avoid the occurrence of unexpected interrupts.

(10) Enabling the interrupt
Enable the HSDMA interrupt. The HSDMA will be able to generate an interrupt after it finishes a data transfer.

(11) Enabling HSDMA transfer
Enable HSDMA transfer.
After this, the HSDMA will perform data transfer when the trigger source has occurred.
**HSDMA interrupt handler**

```c
/* Prototype */
void int_hsdma1(void) __attribute__((interrupt_handler));

/******************************************************************************
* int_hsdma1
*   Type :     void
*   Ret val :  none
*   Argument : void
*   Function : HSDMA ch.1 interrupt function.
*******************************************************************************/
void int_hsdma1(void)
{
    extern volatile int hdmaint_flg;
    hdmaint_flg = TRUE; (1)
    /* Reset HSDMA ch.1 interrupt factor flag 0x40281 */
    *(volatile unsigned char *)INT_FHDM_FIDM_ADDR = INT_FHSDMA1; (2)
}
```

(1) Executing the interrupt processing

When an HSDMA transfer has completed, an interrupt occurs and the CPU executes the interrupt handler function that has been described in the vector table.

The sample program sets a software flag used to check whether the HSDMA interrupt has occurred or not.

(2) Clearing the cause-of-interrupt flag

Reset the cause-of-interrupt flag, because it has been set to 1 by the occurrence of the interrupt.
3  PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

3.11  IDMA Transfer

This section shows an example of IDMA transfer program. The sample program is located in the GNU33\sample_ide\std\dmt3301\idma\src directory.
The IDMA operates with the control information stored in the RAM.

IDMA transfer program

The sample program invokes IDMA by issuing a software trigger to transfer data from the source address to the
destination address. Below describes initialization of the IDMA Ch.0 and handling the interrupt.

Interrupt vector settings for IDMA

(unsigned long)int_idma, // 104  26 Intelligent DMA

Setting the vector table

Register the interrupt handler functions in the vector table.

Control information settings of IDMA

/*******************************************************************************
* main
* Type :     void
* Ret val :  none
* Argument : void
* Function : IDMA demonstration program.
*******************************************************************************/
int main(void)
{
    /* Set IDMA ch.0 */
    first_wd = IDMA_LNKEN_ENA | IDMA_LINK | IDMA_CNT | IDMA_BSIZE; (1)
    second_wd = IDMA_DINTEN_ENA | IDMA_DATSIZ_HW |
                IDMA_SRINC_INC | IDMA_SRC_START0; (2)
    third_wd = IDMA_DMOD_BLOCK | IDMA_DSINC_INC | IDMA_DST_START0; (3)
    write_idma_info((unsigned long *)
                   &dma_control[0]), first_wd, second_wd, third_wd); (4)

    (1) Setting control information (1st word)
    Configure the IDMA control information. In the 1st word, set the IDMA link to be enabled/disabled, IDMA
    link field, transfer counter, and block size (for block transfer mode only).
    When the IDMA link in this channel is enabled, the IDMA will activate data transfer of the channel
    specified in the IDMA link field after a data transfer by this channel has finished. When using block transfer
    mode, be sure to set anything other than 0 for the block size.

    (2) Setting control information (2nd word)
    Configure the IDMA control information. In the 2nd word, set the end-of-transfer interrupt to be enabled/ 
    disabled, data size, source address control, and source address. For the data size, set one data bit size. For
    the source address control, select a source address update operation.

    (3) Setting control information (3rd word)
    Configure the IDMA control information. In the 3rd word, set the transfer mode, destination address control
    and destination address. For the destination address control, select a destination address update operation.

    (4) Writing the control information
    Write the IDMA control information configured in Steps (1) to (3) to the base address in the RAM.
Initialization of IDMA

/* Prototype */
void init_idma(unsigned long, unsigned char);

/*******************************************************************************
* init_idma
*   Type :     void
*   Ret val :  none
*   Argument : unsigned long addr      IDMA control information start address
*              unsigned char ch        IDMA start channel
*   Function : Initialize IDMA control information start address and channel.
*******************************************************************************/
void init_idma(unsigned long addr, unsigned char ch)
{
    /* Set IDMA interrupt enable on interrupt controller 0x40271 */
    *(volatile unsigned char *)INT_EHDM_EIDM_ADDR &= ~INT_EIDMA; (1)

    /* Set IDMA control information address 0x48200 */
    *(volatile unsigned long *)IDMA_DBASEL_ADDR = addr; (2)

    /* Set IDMA start channel 0x48204 */
    *(volatile unsigned char *)IDMA_DCHN_ADDR = ch; (3)

    /* Set IDMA interrupt priority level 3 on interrupt controller 0x40265 */
    *(volatile unsigned char *)INT_PDM_ADDR = INT_PRIL_LVL3; (4)

    /* Reset IDMA interrupt flag on interrupt controller 0x40281 */
    *(volatile unsigned char *)INT_FHDM_FIDM_ADDR = INT_FIDMA; (5)

    /* Set IDMA interrupt enable on interrupt controller 0x40271 */
    *(volatile unsigned char *)INT_EHDM_EIDM_ADDR |= INT_EIDMA; (6)
}

(1) Disabling interrupt
Disabling the IDMA interrupt to avoid the occurrence of unexpected interrupts.

(2) Setting the base address
Set the start address of the control information to the IDMA base address register. The IDMA base address
must be aligned to a word (32-bit) boundary.

(3) Setting the IDMA start channel
Set the IDMA channel number. This determines the RAM address to which the IDMA control information
is written.

(4) Setting the interrupt priority level
Set the interrupt priority level. When two or more interrupts occur at the same time, the interrupt that has
the highest priority is accepted first. The interrupt priority level is set to 3 in the sample program.

(5) Resetting the cause-of-interrupt flag
Reset the cause-of-interrupt flag before enabling the interrupt to avoid the occurrence of unexpected
interrupts.

(6) Enabling the interrupt
Enable the IDMA interrupt. The IDMA will be able to generate an interrupt after it finishes a data transfer.
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

IDMA transfer main routine

/*******************************************************************************
* main
* Type :     void
* Ret val : none
* Argument : void
* Function : IDMA demonstration program.
*******************************************************************************/
int main(void)
{
    /* Disable IDMA transfer 0x48205 */
    *(volatile unsigned char*)IDMA_DMAEN_ADDR &= 0xfe;  // (1)

    /* Set IDMA ch.0 */
    first_wd = IDMA_LNKEN_ENA | IDMA_LINK | IDMA_CNT | IDMA_BSIZE;
    second_wd = IDMA_DINTEN_ENA | IDMA_DATSIZ_HW | IDMA_SRINC_INC | IDMA_SRC_START0;
    third_wd = IDMA_DMOD_BLOCK | IDMA_DSINC_INC | IDMA_DST_START0;
    write_idma_info((unsigned long *)
    (&dma_control[0]), first_wd, second_wd, third_wd);

    /* Enable IDMA transfer */
    *(volatile unsigned char*)IDMA_DMAEN_ADDR |= 0x01;  // (3)

    /* Start IDMA transfer */
    *(volatile unsigned char *)IDMA_DCHN_ADDR |= 0x80;  // (4)
}

(1) Disabling IDMA transfer
Disable IDMA transfer while the control information is written to avoid erroneous operations.

(2) Setting the control information
Write the control information in the RAM in the procedure described above.

(3) Enabling IDMA transfer
Enable IDMA transfer.

(4) Starting IDMA transfer
Set the software trigger bit to start IDMA transfer.

IDMA interrupt handler

/*******************************************************************************
* int_idma
* Type :     void
* Ret val : none
* Argument : void
* Function : IDMA interrupt function.
*******************************************************************************/
void int_idma(void)
{
    extern volatile int idmaint_flg;
    idmaint_flg = TRUE;  // (1)

    /* HSDMA ch.0,1 and IDMA interrupt factor flag 0x40281, */
    /* Reset interrupt factor flag */
    *(volatile unsigned char *)INT_FHDM_FIDM_ADDR = INT_FIDMA;  // (2)
}

(1) Executing the interrupt processing
When an IDMA transfer has completed, an interrupt occurs and the CPU executes the interrupt handler function that has been described in the vector table.
The sample program sets a software flag used to check whether the IDMA interrupt has occurred or not.

(2) Clearing the cause-of-interrupt flag
Reset the cause-of-interrupt flag, because it has been set to 1 by the occurrence of the interrupt.
3 PROGRAMMING THE S1C33 STANDARD PERIPHERAL MODULES

3.12 SLEEP

This section describes how to enter SLEEP mode using the sample program located in the GNU33\sample_ide\std\dm3301\slp\src directory. In SLEEP mode, the high-speed (OSC3) oscillation circuit stops operating and the CPU stops. It stops all the peripheral circuits except for the low-speed (OSC1) oscillation circuit and the clock timer, making it possible to save power.

SLEEP program

The sample program executes the s lp instruction after setting conditions required at wakeup from SLEEP mode. SLEEP mode is canceled by an NMI. After SLEEP mode is cancelled, the 8-bit timer is used to disable clock supply until the high-speed (OSC3) oscillation circuit is stabilized.

slp execute routine

/* Prototype */
void slp_exe(void);
void set_8timer1(void);
void int_8timer1() __attribute__((interrupt_handler));

/*******************************************************************************
* slp_exe
* Type : void
* Ret val : none
* Argument : void
* Function : slp execution
*******************************************************************************/
void slp_exe(void)
{
    /* Set for stabilizing OSC3 by 8bit timer1 */
    set_8timer1();
    /* 8bit timer1 interrupt disable */
    *(volatile unsigned char *)INT_E8TU_ADDR &= ~INT_E8TU1;
    /* 8bit timer1 on for slp */
    *(volatile unsigned char *)OSC_PF1ON_ADDR &= ~OSC_8T1ON_OFF;
    /* 8bit timer1 run */
    *(volatile unsigned char *)T8P_PTRUN1_ADDR |= T8P_PTRUN_RUN;
    /* slp */
    asm("slp");
    /* Reset 8bit timer1 interrupt factor flag on interrupt controller 0x40285 */
    *(volatile unsigned char *)INT_F8TU_ADDR = INT_F8TU1;
}

(1) Setting the 8-bit timer
The high-speed (OSC3) oscillation circuit needs a certain time to stabilize its oscillation after it starts oscillating by canceling SLEEP mode. To disable clock supply while the high-speed (OSC3) oscillation is unstable, the 8-bit timer can be used to wait for oscillation stabilization after SLEEP mode is cancelled. This step sets the oscillation stabilization wait time to the 8-bit timer.

(2) Disabling interrupt
Disable the 8-bit timer interrupt to use the 8-bit timer for waiting for oscillation stabilization.

(3) Enabling high-speed (OSC3) oscillation waiting function
Enable the high-speed (OSC3) oscillation waiting function after canceling SLEEP mode. When this function is enabled, the operating clock will not be supplied to the CPU until the 8-bit timer set in Step (1) underflows after SLEEP mode is cancelled.

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

(5) Executing the s lp instruction
Execute the s lp instruction to set the CPU to enter SLEEP mode. In SLEEP mode, the CPU and high-speed (OSC3) oscillation circuit stop operating.

(6) Resetting the cause-of-interrupt flag
After wakeup from SLEEP mode, reset the cause-of-interrupt flag of the 8-bit timer that has been set to 1 due to a timer underflow.
4 Technical Reference

This chapter describes technical contents and precautions on development for applications using the S1C33 Family. The programs and peripheral functions shown in the explanation are example in the S1C33301 unless otherwise specified.

4.1 Boot

4.1.1 Boot from External RAM

The S1C33 Family processor reads the reset vector from the vector table after an initial reset. The vector table is normally located at the boot address (0xc00000 in case of the S1C33301). To boot the system from an external RAM, change the vector table base address using TTBR (Trap Table Base Register, 0x48134–0x48137) to locate the vector table in the external RAM. Note that the base address must be set to TTBR in 1KB units. Use the commands shown below to change the base address from the debugger.

Sample debug commands to change TTBR

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set {char}0x4812d=0x59</td>
<td>(1) Enable TTBR write protection</td>
</tr>
<tr>
<td>set {long}0x48134=0x0600000</td>
<td>(2) Set address</td>
</tr>
</tbody>
</table>

(1) Remove TTBR write protection

Normally, writing to TTBR is disabled. Before data can be written to TTBR, the write protection must be removed by writing 0b01011001 (0x59) to the TTBR write-protect register.

(2) Setting the base address

Set the base address to TTBR. The vector table is located beginning with the address set here. The example above sets the base address to 0x600000.

4.1.2 Boot from Flash Memory

Before the system can boot from a Flash memory, the boot program must be written to the Flash memory from the debugger in the procedure shown below. The Flash memory write/erase program "am29f800.elf" is located in the GNU33tool0xs33g directory.

Sample debug commands to write program to Flash memory

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td># load symbol information file am29f800.elf</td>
<td>(1)</td>
</tr>
<tr>
<td># decide debugger mode and its port target icd usb</td>
<td>(2)</td>
</tr>
<tr>
<td># load to memory load am29f800.elf</td>
<td>(3)</td>
</tr>
<tr>
<td>// flash memory setting c33 fls 0xC00000 0xC0fffff FLASH_ERASE FLASH_LOAD</td>
<td>(4)</td>
</tr>
<tr>
<td>c33 file 0xC00000 0 0</td>
<td>(5)</td>
</tr>
<tr>
<td># load symbol information file 33301serial_syn_slv.elf</td>
<td>(6)</td>
</tr>
<tr>
<td># decide debugger mode and its port target icd usb</td>
<td>(7)</td>
</tr>
<tr>
<td># load to memory load</td>
<td>(8)</td>
</tr>
</tbody>
</table>
4 TECHNICAL REFERENCE

(1) Reading the debug information
   Read the debug information from the elf format object file.
   The sample command reads the debug information from the write/erase program that supports the Flash
   memory mounted on the target system.

(2) Connecting the target system
   Set the debugger mode to establish the connection between the target system and the debugger. The sample
   command sets the debugger mode to establish the connection with the SSU1C33001H (ICD Ver. 3) via the
   USB interface.

(3) Loading the program
   Read a program from an elf format object file and load it into the target system memory.
   The sample command loads the write/erase program that supports the Flash memory mounted on the target
   system.

(4) Setting Flash memory parameters
   Set the parameters used to write data to the Flash memory. The parameters to be specified are Flash
   memory start and end addresses and the address from which the write/erase program is stored. At this time,
   data is still not written to the Flash memory.

(5) Erasing the Flash memory
   Erase the Flash memory contents. The command parameters specified are the flash memory control address
   and the erase start and end block numbers. This example specifies both the erase start and end block as 0 to
   erase all blocks in the Flash memory.

(6) Reading the debug information
   Read the debug information from the elf format object file.
   The command in this step reads the debug information from the program to be written to the Flash memory.

(7) Connecting the target system
   Set the debugger mode to establish the connection between the target system and the debugger. The sample
   command sets the debugger mode to establish the connection with the SSU1C33001H (ICD Ver. 3) via the
   USB interface.

(8) Loading the program
   Read a program from an elf format object file and load it into the target system.
   The loaded program is written to the Flash memory by the flash memory write program in the target
   system.

In the procedure above, the target system is able to start up by the boot program written in the Flash memory.
4 TECHNICAL REFERENCE

4.2 Linker Script

4.2.1 Usage of .data Section

Data with an initial value that can be read/written are located in the .data section. The .data section is used by copying the initial values written in the ROM into the RAM with the user program as in the figure shown below. Refer to “3.8.2 Sections” in the “S5U1C33001C Manual” supplied with this package for details of sections.

Satisfy the following three conditions to make practical use of the .data section:

1. The initial values of the variables are provided in the ROM.
2. The initial values in the ROM are copied to the RAM before use.
3. The program operates with the data expanded in the RAM.

However, the GNU33 IDE automatically generates a linker script in which the link conditions are described and the linker automatically determines the ROM addresses to which the initial values of the .data section are located. Therefore, the user program must copy the ROM data to the RAM using the section symbols generated by the linker. Perform this processing only in the user’s initialize routine. Use the sample boot program written in the GNU33\sample_ide\stdm333301\xxx\common\init.c file (xxx represents a peripheral circuit name) to simply implement the initial data transfer function.

The following shows sample boot programs in C and assembler codes.

Data transfer program

Before the program can be run, the initial data must be transferred into the RAM using the section symbols as in the sample below. The GNU33 IDE generates the symbols that represent the source address and destination area start/end addresses when it generates a linker script file. Use these symbols to copy data.

Sample assembler code

```asm
/* DATA section copy */
asm("xld.w %r4,__START_data"); ← Data area start address in the RAM
asm("xld.w %r5,__END_data"); ← Data area end address in the RAM
asm("xld.w %r6,__START_data_lma"); ← Data storage area start address in the ROM

/* data copy */
asm("dat_cpy:");
asm("cmp %r4,%r5");
asm("jreq ram_exit");
asm("ld.b [%r6]+,%r7");
asm("ld.b [v4]+,%r7");
asm("jp dat_cpy");
asm("ram_exit:");
```

Sample C code

```c
extern char __START_data;
extern char __END_data;
extern char __START_data_lma;
char *src = &__START_data_lma;
char *dst = &__START_data;
while (dst < &__END_data)
{
    *dst++ = *src++;
}
```
4 TECHNICAL REFERENCE

4.2.2 Using the Internal RAM for Program Cache

The program code located in an external ROM or Flash memory is accessed with one or two wait cycles. By copying the program into the internal RAM as in the figure below, it can be executed with 0 wait cycles increasing the processing speed.

![Diagram of program storage](image)

To use the internal RAM for a program cache, the linker script file must be edited similar to the .data section described above. Refer to "5.7.6 Editing a Linker Script" and "9 Linker" in the "S1C33001C Manual" supplied with this package for setting a linker script file.

**Data transfer program**

Before the program can be run, the program code must be transferred into the RAM using the section symbols as the sample below. Furthermore, a RAM area can be shared with multiple objects by copying each object code before executing. The GNU33 IDE generates the symbols that represent the source address and destination area start/end addresses when it generates a linker script file. Use these symbols to copy data.

**Sample data transfer program**

```c
/* TEXTU1 section copy */
asm("xld.w \$r4,__START_textul"); ← Program area start address in the RAM
asm("xld.w \$r5,__END_textul"); ← Program area end address in the RAM
asm("xld.w \$r6,__START_textul_lma"); ← Program storage area start address in the ROM

/* data copy */
asm("dat_cpy: ");
asm("cmp \$r4,\$r5");
asm("jreq ram_exit");
asm("ld.b [%r6]+,\$r7");
asm("ld.b [%r4]+,\$r7");
asm("jp dat_cpy");
asm("ram_exit:");
```

---

4-4 EPSON S1C33 FAMILY STARTUP MANUAL
4.3 C Compiler

4.3.1 Arguments

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Method of use</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r0</td>
<td>Register used as a frame pointer</td>
</tr>
<tr>
<td>%r1</td>
<td>Register that need have their values saved when calling a function</td>
</tr>
<tr>
<td>%r2</td>
<td>Registers that need have their values saved when calling a function</td>
</tr>
<tr>
<td>%r3</td>
<td></td>
</tr>
<tr>
<td>%r4</td>
<td>Register for storing returned values (8/16/32-bit data, 32 low-order bits of double-type data)</td>
</tr>
<tr>
<td>%r5</td>
<td>Register for storing returned values (32 high-order bits of double-type data)</td>
</tr>
<tr>
<td>%r6</td>
<td>Register for passing argument (1st word)</td>
</tr>
<tr>
<td>%r7</td>
<td>Register for passing argument (2nd word)</td>
</tr>
<tr>
<td>%r8</td>
<td>Register for passing argument (3rd word)</td>
</tr>
<tr>
<td>%r9</td>
<td>Register for passing argument (4th word)</td>
</tr>
<tr>
<td>%r10</td>
<td>Scratch register/unused</td>
</tr>
<tr>
<td>%r11</td>
<td></td>
</tr>
<tr>
<td>%r12</td>
<td></td>
</tr>
<tr>
<td>%r13</td>
<td></td>
</tr>
<tr>
<td>%r14</td>
<td></td>
</tr>
<tr>
<td>%r15</td>
<td>Default data area pointer register*</td>
</tr>
</tbody>
</table>

Table 4.3.1.1 Method of Using General-Purpose Registers by C Compiler

* When the -medda32 option (default data area is not used) is not specified

As the above table shows, four registers %r6 to %r9 are reserved for passing arguments to the function to be called. If the number of arguments exceeds these registers (double type and long long type arguments occupies two registers), the overflowed arguments are passed to the function via the stack. In this case, a register for passing an argument points to the stack address. A memory access occurs when the stack is used and it increases the overhead as compared to a function call with four or less arguments. To enhance code efficiency, it is better to reduce arguments used for calling a function to four or less.
4 TECHNICAL REFERENCE

4.3.2 Substitutions

When substituting a setting value to an I/O register in which multiple functions are assigned, in some cases the bits to be unaltered must be masked. Note that the efficiency of the assembler code after compiled in this case depends on masking and substitution procedure.

The following describes the differences by procedures, using a sample routine to set an 8-bit timer interrupt priority level.

(1) When the I/O register value is copied to a temporary variable and is loaded back to the register after substituting a new value with masking

C code

```c
temp = *(volatile unsigned char *)INT_P8TM_PSIO0_ADDR;
temp &= 0xF0;
temp |= INT_PRIL_LVL3;
*(volatile unsigned char *)INT_P8TM_PSIO0_ADDR = temp;
```

Assembler code

```
sub  %r5,0x29
ld.ub %r4,[%r5]
ext 0x3
xand %r4,0xf0
or  %r4,0x3
ld.b [(%r5),%r4]
```

(2) When the I/O register memory is directly masked and substituted with a new value

C code

```
*(volatile unsigned char *)INT_P8TM_PSIO0_ADDR &= 0x0F;
*(volatile unsigned char *)INT_P8TM_PSIO0_ADDR |= INT_PRIL_LVL3;
```

Assembler code

```
ld.ub %r4,[%r5]
xand %r4,0xffffffff (sign expansion)
ld.b [(%r5),%r4]
ld.ub %r4,[%r5]
or  %r4,0x3
ld.b [(%r5),%r4]
```

As the samples show, memory operations are the major difference. The assembler code in sample (1) appears complicated, but sample (2) includes more memory operation codes than (1). Memory operations cause overhead and dependency to be increased, so sample (1) is better for code efficiency.
### 4.4 DMA Transfer

The HSDMA supports two data transfer modes, dual-address transfer and single-address transfer.

#### Dual-Address Transfer

The dual-address transfer mode first reads data from the source address into the on-chip temporary register, and then writes data in the temporary register to the destination address. This allows data transfer between memories. On the other hand, the single-address transfer mode performs reading from the source and writing to the destination at the same time using the external bus. Although data transfer between memories cannot be performed, it is capable of high-speed data transfer without a temporary register used.

However, a DMA transfer stops memory access by the C33 Core, therefore the C33 Core enters idle status when a DMA occurs while the C33 Core is executing the program. Depending on the condition, data transfer without the DMA used may be processed fast.