EPSON
EXCEED YOUR VISION

CMOS 16-BIT SINGLE CHIP MICROCONTROLLER

S1C17 Family
Startup Manual
Assembler version

SEIKO EPSON CORPORATION

Rev. 1.1

NOTICE

No part of this material may be reproduced or duplicated in any form or by any means without the written permission of Seiko Epson. Seiko Epson reserves the right to make changes to this material without notice. Seiko Epson does not assume any liability of any kind arising out of any inaccuracies contained in this material or due to its application or use in any product or circuit and, further, there is no representation that this material is applicable to products requiring high level reliability, such as, medical products. Moreover, no license to any intellectual property rights is granted by implication or otherwise, and there is no representation or warranty that anything made in accordance with this material will be free from any patent or copyright infringement of a third party. When exporting the products or technology described in this material, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You are requested not to use, to resell, to export and/or to otherwise dispose of the products (and any technical information furnished, if any) for the development and/or manufacture of weapon of mass destruction or for other military purposes.

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

©SEIKO EPSON CORPORATION 2017, All rights reserved.

Introduction

The S1C17 Family, 16-bit RISC Processors have various peripheral circuits including abundant interfaces that are compatible with different kinds of sensors, and the LCD driver and controller that covers wide display area. The products suitable for mobile devices are provided in this Family with high-speed operation and low power consumption. The Family has a lineup of many products with built-in Flash ROM. Rich development environment and on-chip IC function provided to the products enable customers to minimize any development period.
This document has been released for an application developer who will use the S1C17 Family products, and it explains the basic embedded programming procedure of the S1C17 Family products.

The reader of this document should have the following basic software knowledge.

- The knowledge of C language and how to create assembler source programs.
- General knowledge of C language (ANSI C compatible)
- The knowledge of GNU
- The basic operations of the Windows 2000 or Windows XP OS

Note that sample programs given in this manual have been created by using the S1C17 Family C/C++ Compiler Package (S5U1C17001C) Version 1.2.1.

Manual Configuration:
This manual consists of the following two chapters.
Chapter 1 provides the basic knowledge to create an embedded software.
Chapter 2 explains the basic programming procedure of the S1C17 Family products by using sample programs.
Chapter 3 gives a sample program to explain the mixed method of C and assembler languages.
Chapter 4 describes precautions for assembler programing.

Related Manuals:
The following lists the related manuals that you should reference to

- S1C17 Core Manual
- S5U1C17001C Manual (S1C17 Family C Compiler Package)
- Technical Manual for each S1C17 Family Model

Table of Contents

1. BASIC KNOWLEDGE OF EMBEDDED PROGRAMS 1

1.1 Basic Mechanism for Program Operation 1

1.2 Startup (Initialize) Routine 2

2. PROGRAMMING OF S1C17 FAMILY PROCESSORS 3

2.1 Program Development Procedure using the GNU17 3

2.2 Creating a Vector Table 4

2.2.1 Explanation of Vector Table 5

2.3 Interrupts 6

2.3.1 Reset 6

2.3.2 reti Instruction 6

2.3.3 Address Misalignment 6

2.3.4 NMI 6

2.4 Creating a Startup Routine 7

2.4.1 Explanation of Startup Routine 8

2.4.2 Setting the SP 9

2.4.3 Initializing the .bss/.data Section 10

2.4.4 Enabling an Interrupt (IE) 13

3. C AND ASSEMBLER INTERFACES 14

3.1 How to Mix Assembler and C Sources 14

3.1.1 C → Assembler Function Call 15

3.1.2 Assembler → C Function Call 16

4. PRECAUTIONS ON CREATING ASSEMBLER PROGRAMS 18

4.1 Using "ext" 18

4.2 Extension Instruction 20

4.3 Memory Models 22

REVISION HISTORY 23

1. BASIC KNOWLEDGE OF EMBEDDED PROGRAMS

This chapter is intended to be read by the user who develops an embedded software in the first time, and this chapter explains the basic concept that is very important for the user, including the basic mechanism to operate programs, system initialization by the startup routine and others.

1.1 Basic Mechanism for Program Operation

The operation when the S1C17 processor (called "the MCU" hereafter) starts, the basic operation mechanism is explained first.

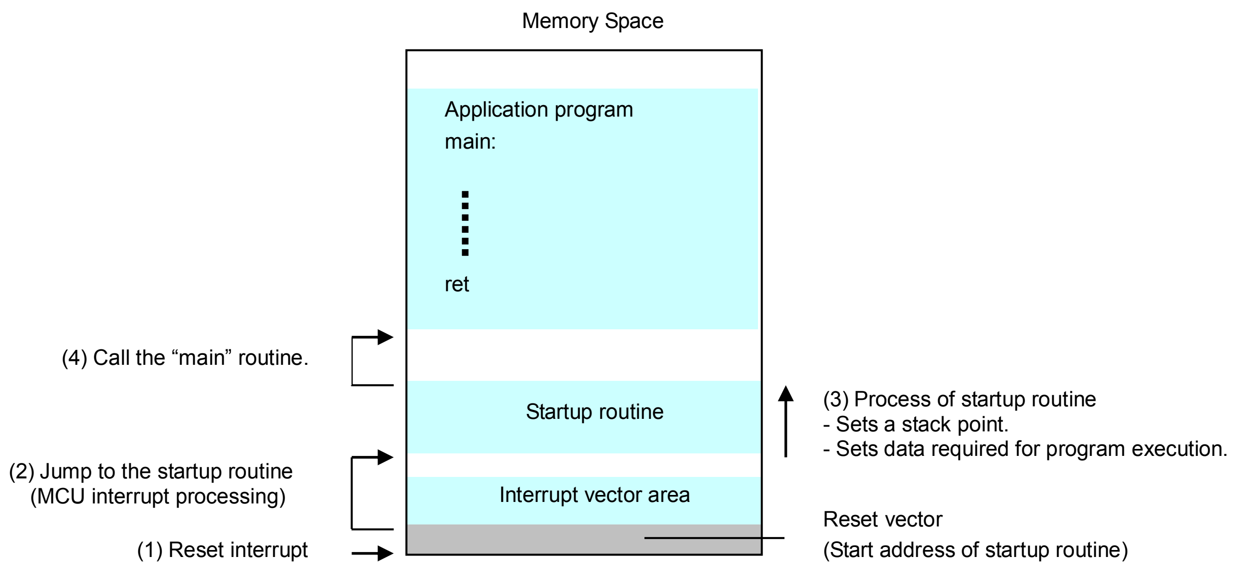


Fig.1.1 Basic Mechanism during Startup of S1C17 Processor

(1) When the MCU is turned ON, a Reset interrupt occurs and the MCU reads the start address of the vector table.
(2) The MCU jumps to the content (address) which was read in Step (1) and calls the Startup (Initialize) routine.
(3) The Startup routine first executes initialization that is required for stack setting and program execution.
(4) When the initialization process is complete, the Startup routine calls the "main" routine.

Note: Addresses of various interrupt process routines are written on the Vector Table. When an interrupt occurs, the appropriate routine address is read from the table and control jumps to the corresponding process routine.

An embedded application program must begin with the Startup routine, but not with the "main" routine. The user needs to understand the Startup routine to operate the program when developing an embedded software.

1.2 Startup (Initialize) Routine

The embedded software uses the Startup routine to execute the required initialization before executing the main routine. Generally, the following processes are executed:
・Setup of stack pointer
・Data setup required for program execution
  - Clear the memory area that has no defaults. (Clearing of .bss section)
  - Transfer the default data from ROM area to RAM area. (Copy of .data section)
・Hardware initialization and interrupt setting

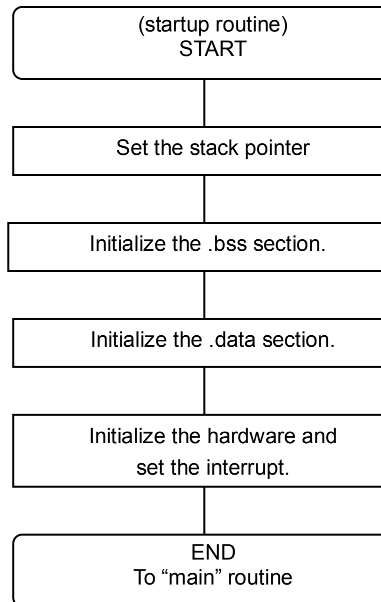


Fig.1.2 Startup Routine

The “stack” is a RAM area where the current processing data, return address and others are saved temporarily when a subroutine or a function is called. As the stack is also used by the interrupt, the stack area needs to be assigned by the startup routine.
When you execute the program, you need to initialize the global variables which have no defaults. Because their setting may be indefinite when reset, you need to initialize them (by clearing the .bss section). Also, if global variables have defaults, you need to copy their defaults from the ROM to RAM (by copying the .data section).
In addition, you need to initialize not only the variables relating to the software execution but also the values required for MCU and other hardware operations. During interrupt setting, you also need to enable an external interrupt that can be masked.

For the embedded applications, the startup routine is first executed and then the “main” routine is called.

You should consider these basics and develop an embedded device program.

2. PROGRAMMING OF S1C17 FAMILY PROCESSORS

This chapter explains how to create a program that is common to the S1C17 Family processors.
As explained in Chapter 1, the startup routine needs to be executed for an embedded application as the preprocessing of “main” routine execution. The following gives a sample program and explains the standard process flow until you call the startup routine and the “main” routine.

2.1 Program Development Procedure using the GNU17

GNU17 is the integrated program development environment that contains a series of software tools and utilities for compiling of C source programs and for assembling and debugging of assembler source programs.
To install the GNU17, access to the “EPSON Microcontroller User Site” and to the “S1C17 Family” and download the “S1C17 Software Integrated Development Environment GNU17. ”
The following shows the standard program development flow using the GNU17.

(1) Developing a project
    Develop a new project using the GNU17.
(2) Creating a source program
    Create a source file using the GNU17 editor or a general-purpose editor, and add this file to the project.
(3) Building the program
    Using GNU17, set up the startup options and linker scripts from C compiler to the linker. When you execute the build from GNU17, an elf-format object file that can be debugged and a ROM data file (psa file) which is the object file converted into S-record format.
(4) Debugging
    Check the program operation and debug it using the elf-format object file (created by the linker) and the S-record format ROM data file. You can set and start debugging from the GNU17.

For the detailed information, refer to the “Software Development Procedure” of the “S5U1C17001C MANUAL.” You can find out the “S5U1C17001C MANUAL” under the “EPSON¥GNU17¥doc” directory when you have installed the GNU17.

2.2 Creating a Vector Table

The vector table and the startup routine are minimum required to execute an S1C17 program. This section explains the vector table. The startup routine is explained in Section 2.4 of this manual.

List 2.1 An Example Vector Table

```
;--------BOOT.s---------
.section .rodata                          } Declare rodata section

.global BOOT                              }
.global DUMMY                             } Declare Global
.global NMI                               }

;--- vector table ---
                    ; NO      BASE+
.long BOOT          ; 0        00         }
.long DUMMY         ; 1        04         }
.long NMI           ; 2        08         }
.long DUMMY         ; 3        0C         }
.long DUMMY         ; 4        10         }
     .                                    } (1)
     .                                    }
     .                                    }
.long DUMMY         ; 29       74         }
.long DUMMY         ; 30       78         }
.long DUMMY         ; 31       7C         }


;--- Non Maskable Interrupt function ---
NMI:
jpr -1


;--- Dummy Interrupt function ---
DUMMY:
jpr -1
```

The vector table is created using assembler by placing constant (1) after declaring rodata section.

For details, refer to the section and link descriptions in the S5U1C17001C Manual,
the description of constant pseudo-instruction (.rodata, data) in the S5U1C17001C Manual,
and the vector table description in the S1C17701 Technical Manual.

2.2.1 Explanation of Vector Table

The “vector table” stores an array of vectors (destination addresses) to each interrupt process routine that is executed if an interrupt occurs during program execution.

Table 2.1 Configuration of Vector Table

Vector No. or software interrupt No.	interrupt	Vector address
0 (0x00)	Reset	TTBR+0x00
1 (0x01)	Address misaligned interrupt	TTBR+0x04
2 (0x02)	NMI	TTBR+0x08
3 (0x03)	Maskable external interrupt 3	TTBR+0x0c
:	:	:
31 (0x1f)	Maskable external interrupt 31	TTBR+0x7c

The “TTBR” (Trap Table Base Register) shown on Table 2.1 identifies the start address of the vector table.
Note: As the TTBR value depends on the processor model used, refer to the corresponding Technical Manual for the actual TTBR.

2.3 Interrupts

The S1C17 Core can accept up to 32 types of interrupts. (The first 3 interrupts are used for reset, address misalignment, and NMI.)
The interrupt process routine is called when the corresponding interrupt cause is accepted. You must code the appropriate routine process. As the interrupt cause and setting vary depending on the model used, refer to the corresponding Technical Manual for details.

2.3.1 Reset

A reset interrupt occurs at system power-on. During the reset process, the reset vector is called from the beginning of vector table and it is set on the PC. This allows control to jump to the startup routine of reset vector and to execute the program.

2.3.2 reti Instruction

“reti” is a return instruction used for the interrupt process routine. The interrupt process saves the PSR as well as the return address in the stack. Therefore the PSR content must be restored by the reti instruction. The reti instruction reads from the stack in the order of from the PC to PSR.
Be sure to execute the reti instruction at the end of the interrupt process routine. Doing so allows the PC to return to the position where the interrupt has occurred, and enables the PSR value to be returned from the stack to the instruction sequence to resume the interrupted processing.

2.3.3 Address Misalignment

A load instruction to access to memory or I/O area has the fixed size of data to be transferred by the instruction. Its address must be on a boundary of each data size.

Table 2.2 Load Instructions and Address Boundaries

Instruction	Transfer data size	Address
ld.b / ld.ub	Bytes (8 bits)	Byte boundaries (for all addresses)
ld	16 bits	16-bit boundaries (The least significant bit of address is 0.)
ld.a	32 bits	32-bit boundaries (The low-order 2 bits of address are 00.)

If the specified address of the load instruction does not meet these conditions, the processor considers an address misaligned interrupt and transitions to the interrupt process.
On List 2.1, if an address misaligned interrupt occurs, control jumps to the “dummy” function and starts the indefinite loop process. Correct it appropriately.

2.3.4 NMI

There are two types of interrupts: maskable interrupts and non-maskable interrupts.
A non-maskable interrupt is shorted as NMI. The CPU accepts this NMI unconditionally in preference to other interrupts.
On List 2.1, if an NMI occurs, control jumps to “NMI” and the unlimited loop occurs. Correct it appropriately.

2.4 Creating a Startup Routine

List 2.2 An Example of Startup Routine

```
;--- Boot function ---
.text
.align 1
BOOT:
xld.a    %sp,0x0fc0                              ;Set the SP.

xcall    clearBss                                ;Call “clearBss”
xcall    copyLmaToVma                            ;Call “copyLmaToVma”

ei                                               ;Interrupt enabled
xcall    main                                    ;Call the "main" routine.
xcall    end                                     ;Call the "end" routine.

ret

;--- ClearBss function ---
clearBss:
  * For details on the "clearBss" program, refer to Section 2.4.3, ".bss/.data."
         ret

;--- copyLmaToVma function ---
copyLmaToVma:
  * For details on the "copyLmaToVma" program, refer to Section 2.4.3, ".bss/.data."
         ret
```

2.4.1 Explanation of Startup Routine

The startup routine specifies that a reset interrupt occurs during system power-on (at initial reset) and that the function is called from the vector table that corresponds to the interrupt.

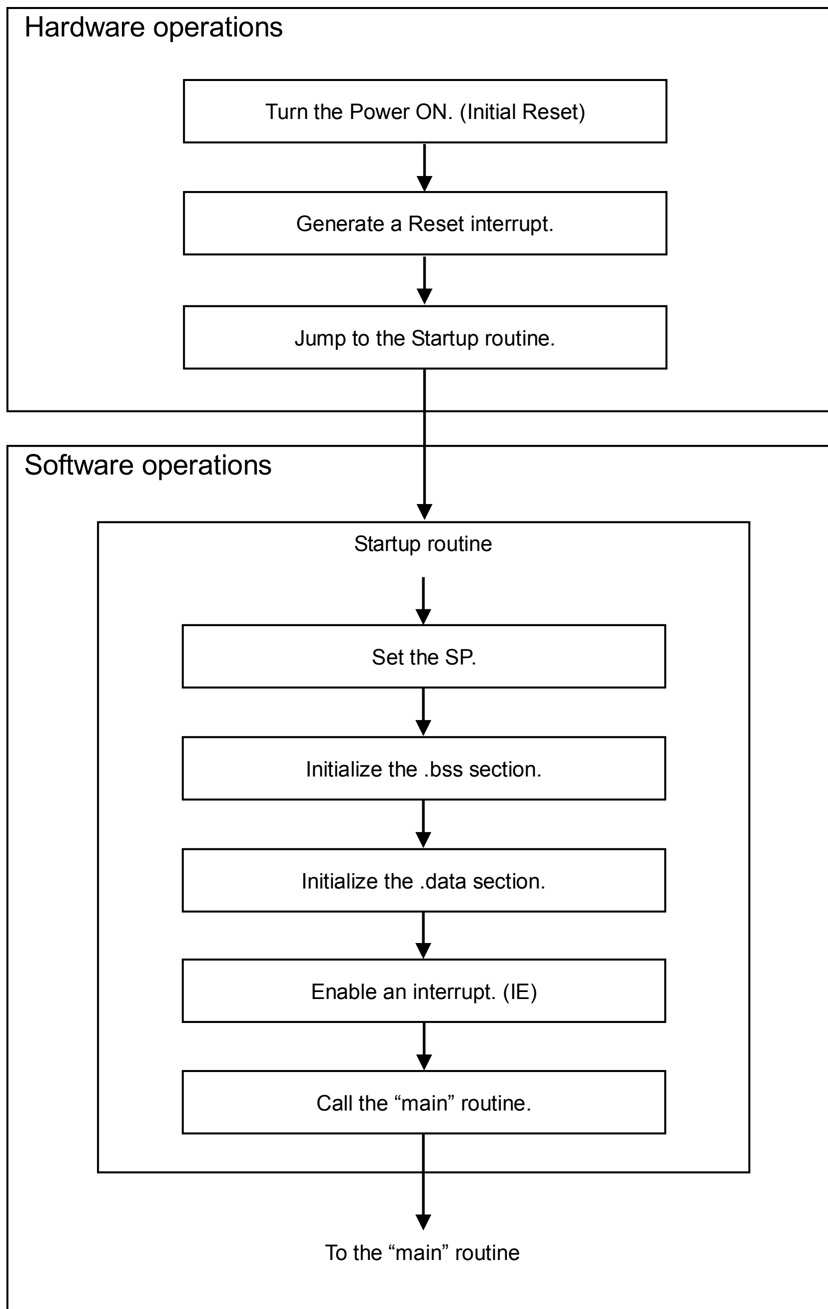


Fig.2.1 Operation Flow of Startup Routine

2.4.2 Setting the SP

First, set the start address of the stack in the SP (Stack Pointer) by issuing the "ld" instruction of the assembler.

List 2.3 SP Setup Example

```
xld.a    %sp, 0x0fc0          ; Set SP in RAM
```

Although address "0x0fc0" is set on List 2.3, you can set an address of any RAM area. Take care not to overlap the stack area by the RAM data storage area.
The following explains the SP setting by considering that address 0x0fc0 is set on List 2.3.
The S1C17 Series CPU places the stacks in the smaller address direction. As the S1C17701 has the RAM area of 0x0000 to 0x1000, the maximum value you can set for the SP is 0x1000. However, the 0x0fc0 to 0x0fff area is reserved for on-chip debugging. Therefore, address 0x0fc0 is set to avoid the area overlapping.
Note: For the address of RAM area and on-chip debugger area, refer to the corresponding Technical Manual.

The following gives a reference chart to save registers in the stack and to transition to the SP setting.
Example: ld.a -[%SP],%r0
Explanation of instructions: The stack pointer value is decremented by 4 bytes, and the 24-bit data of "r0" register is transferred to its address. The 32-bit data, having the high-order 8 bits of all 0s, is written in the memory.
(1) SP=SP-4
(2) R0→ [SP]

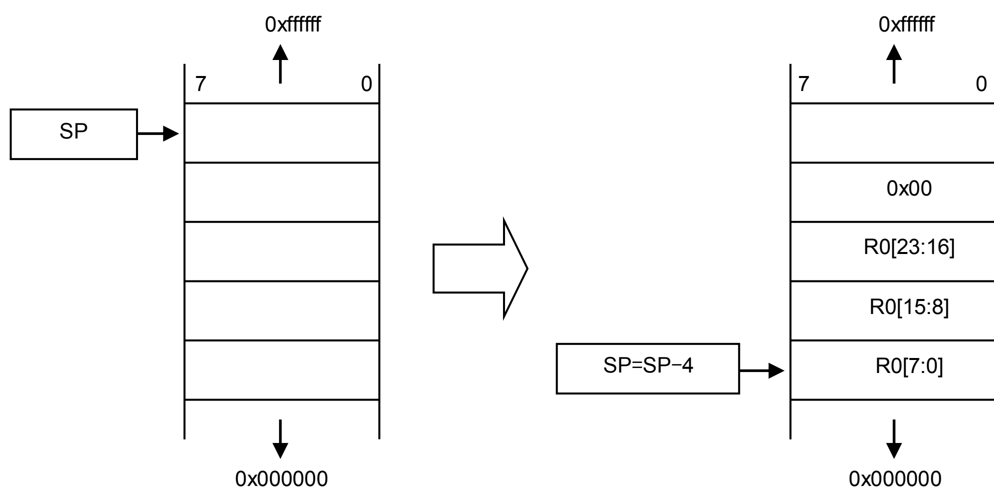


Fig.2.2 SP and Stack

2.4.3 Initializing the .bss/.data Section

Before explaining the initialization of ".bss/.data sections," the following explains the memory configuration of the project you have created with GNU17. Fig.2.3 shows the memory configuration of the S1C17701.

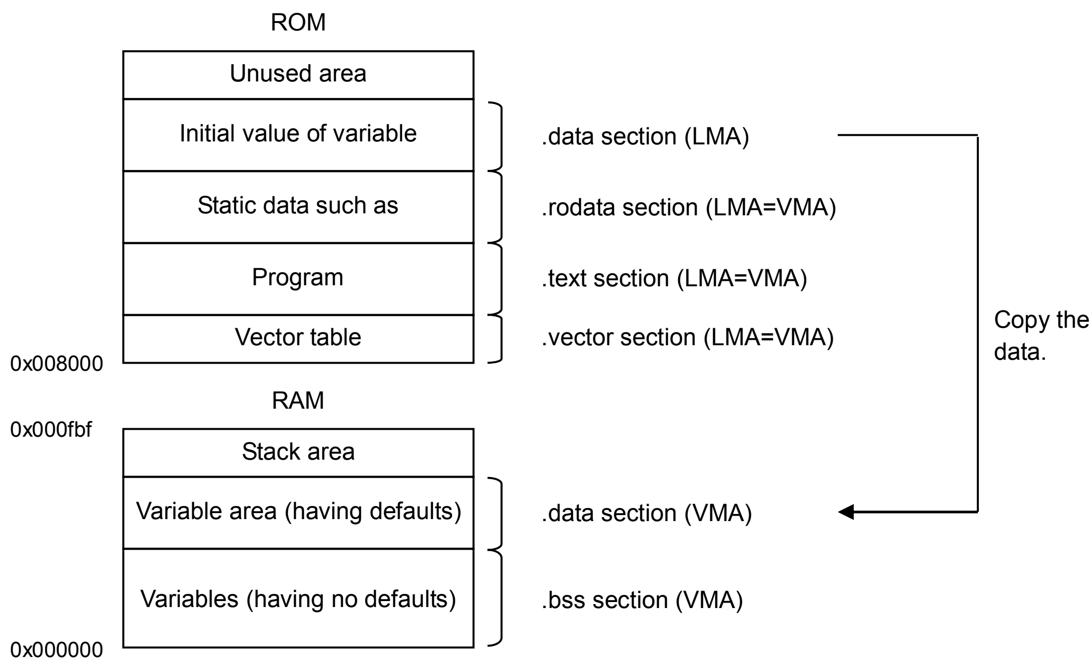
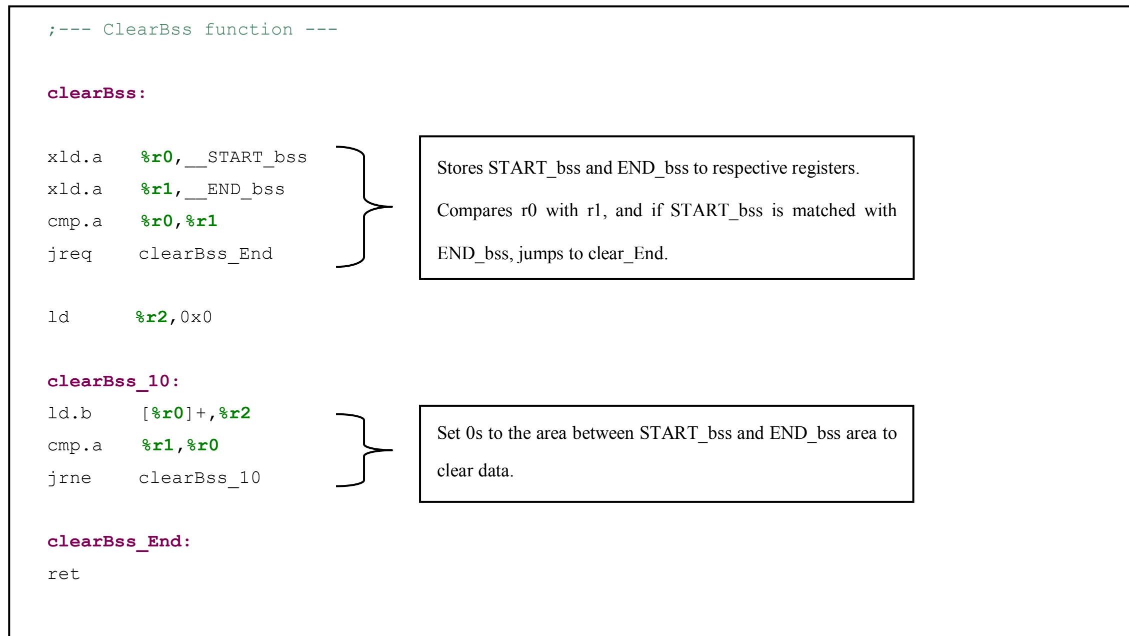


Fig.2.3 Memory Configuration Example (S1C17701)

Place the program and data in the ROM being assigned at address 0x8000 and later as shown in Fig.2.3. The program is assumed to be executed as it is in the storage address (LMA) of the ROM. Also, the static data is assumed to be read directly from the ROM and used.
Place the variable area (having no defaults) in the RAM at address 0x0 and later, and use it as the variable area (having defaults) later. Store the defaults of variables in the ROM, and the application program copies them to the RAM.
For more information about these sections, refer to the "S5U1C17001C MANUAL."

The following explains how to initialize the .bss section.
The ".bss section" stores variables having no defaults. The area from "__START_bss" to "__END_bss" is set to all 0s, and its data is cleared.

List 2.4 Initializing the .bss Section



```
;--- ClearBss function ---

clearBss:

xld.a   %r0,__START_bss
xld.a   %r1,__END_bss
cmp.a   %r0,%r1
jreq    clearBss_End

ld      %r2,0x0

clearBss_10:
ld.b    [%r0]+,%r2
cmp.a   %r1,%r0
jrne    clearBss_10

clearBss_End:
ret
```


"__START_bss" and "__END_bss" are defined in the "Linker script file (file.lds)."

__START_bss .... The start address of .bss section
__END_bss .... The end address of .bss section

The following explains how to initialize the data section.
The ".data section" stores variables having defaults. Data of the RAM (LMA) at "__START_data_lma" and later is copied to the ROM (VMA) area between "__START_data" and "__END_data".

List 2.5 Initializing the .data Section

```
;--- copyLmaToVma function ---
copyLmaToVma:

xld.a     %r0,__START_data
xld.a     %r1,__START_data_lma
xld.a     %r2,__END_data
sub.a     %r2,%r0
jreq      copyLmaToVma_End

copyLmaToVma_10:
ld.b      %r3,[%r1]+
ld.b      [%r0]+,%r3
sub       %r2,0x01
jrne      copyLmaToVma_10

copyLmaToVma_End:
ret
```

Stores START_data START_data_lma END_data in the register, and subtracts START_data END_data to allow for jumping to copyLmaToVma_End if no variables have defaults.

Copies the START_data_lma data to the area between START_data and END_data.
Exits from the loop when the flag bit in sub %r2, 0x01 turns to 0.

"__START_data" and "__START_data_lma" and "__END_data" are defined in the "Linker script file (file.lds)."

__START_data	.... The start address of .data section
__START_data_lma	.... The start address of .data section LMA part
__END_data	.... The end address of .data section

2.4.4 Enabling an Interrupt (IE)

The IE (Interrupt Enable) bit of the PSR (Processor Status Register) is set to 1 by the “ei” instruction, and an external maskable interrupt is enabled.

List 2.6 Enabling an Interrupt (IE)

```
ei      ; interrupt enable
```

PSR is an 8-bit register that holds the CPU status data, and its content varies depending on the executed instruction result. Except for the IE bit state, you cannot directly change the content of this PSR using the program.

	7	6	5	4	3	2	1	0
PSR	IL [2:0]			IE	C	V	Z	N
Default	0	0	0	0	0	0	0	0

IL:	Interrupt level	(0 to 7: Interrupt)
IE:	Interrupt enabled	(1: Enabled; 0: Disabled)
C:	Carry flag	(1: With carry/borrow; 0: None)
V:	Overflow flag	(1: Overflowed; 0: None)
Z:	Zero flag	(1: Zero, 0: None-zero)
N:	Negative flag	(1: Negative; 0: Positive)

Fig.2.4 PSR

This is a reference information only. To disable an external maskable interrupt, use the “di” instruction of the assembler as shown on List 2.7.

List 2.7 Disabling an Interrupt

```
di      ; interrupt disable
```

3. C AND ASSEMBLER INTERFACES

This chapter explains how to write a program by mixing assembler and C language.

3.1 How to Mix Assembler and C Sources

Following rules of arguments, return values, and register content protection enables you to move freely between C and assembler routines.
When the GNU17 compiles C sources, respective registers are used for the following purpose. The following explains which area stores each argument and return value.

Table 3.1 Usage of general-purpose registers

Registers	Usage
R0	Register for passing argument (First word) Scratch register Register for storing return values (8/16-bit data, pointer, low 16 bits of 32-bit data)
R1	Register for passing argument (Second word) Scratch register Register storing return values (High 16 bits of 32-bit data)
R2	Register for passing argument (Third word) Scratch register
R3	Register for passing argument (Fourth word) Scratch register
R4	A register of which value is ensured before/after calling functions.
R5	
R6	
R7	

Follow the instructions of Table 3.1 to use general-purpose registers

*Scratch register
A register of which value is not ensured before/after calling functions.

For details, refer to the register usage description in the S5U1C17001C Manual.

3.1.1 C → Assembler Function Call

This section describes how to call a routine created with assembler to a program written in C language.

List 3.1 C Program Calling the strcpy Routine

```
/* #include */
#include <string.h>
int main(void){
    char pchMem[15];
    strcpy(pchMem,"strcpy test");     //Call the strcpy routine
    return 0;
}
```

List 3.1 C program calls the strcpy routine. The strcpy routine copies strings to a character type array. The called strcpy routine stores the first argument "pchMem" pointer and the second argument "strcpy test" pointer in general-purpose registers, R0 and R1 respectively. The return values are stored in the R0 register at the start address of pchMem. The list 3.1 program calls the list 3.2 program.

List 3.2 Assembler Program strcpy Routine

```
;--- strcpy program ---
.section .text
.align 1
.global strcpy

strcpy:
ld.a     %r3,%r0
strcpy_loop:
ld.b     %r2,[%r1]+
ld.b     [%r3]+,%r2
cmp      %r2,0
jrne     strcpy_loop

ret
```

Stores the first argument in %r0 and the second argument in %r1, and copies strings to the character type array until the second argument is incremented to the NULL character.

This type of program enables a C source to call assembler routines.
If you need to use the R4 to R7 registers, be sure to save them to the stack in advance.

For details, refer to the register usage description in the S5U1C17001C Manual.

3.1.2 Assembler → C Function Call

This section describes how to call a function created with C language to a program written in assembler. Create a program based on the description in Section 3.1, rules for mixing assembler and C.

● **Assembler → C language function call (when arguments and return values are not passed in the stack)**
To call a C function to an assembler program, use "xcall."
The called function starts processing using data stored in R0, R1, R2, and R3 registers for arguments. Store necessary arguments in respective registers before calling the C function. Note that if there are not sufficient registers to store the arguments, the arguments are stored in the stack. When the C function call process has been completed, the return values are stored in the R0 and R1 registers. Each register has 16 bits of data storage capacity. Note that data exceeding the capacity will be stored in the stack.
The following gives an example of a simple program to explain a flow until a function call is completed.

List 3.3 C Program addi Function

```
;--- ter program---
short
addi (unsigned short a, unsigned short b){
    return short (a + b) ;
    }
```

To call list 3.3, arrange a program as shown in list 3.4.

List 3.4 Assembler Program addi Function Call

```
;---text section---
.section .text
.align 1
.global main
main:
ld      %r0, 0x1                        ; Stores 0x1 to %r0 (first argument)
ld      %r1, 0x2                        ; Stores 0x2 to %r1 (second argument)
xcall   addi                            ; Calls C function, the addi function
main_loop:
cmp     %r0, 0x3                        ; Compares a return value (%r0) with 0x3
jrne    main_loop                       ; To main_loop if any difference
ret
```

● **Assembler → C language function call (when arguments and return values are passed in the stack)**
The following explains an example of passing arguments and return values in the stack when an assembler routine calls a C function.
The function of list 3.5 has 64 bits of the first argument, 16 bits of the second argument, and 64 bits of the return value.

List 3.5 C Program addi Function

```
;--- addi program---
long long
addi (unsigned long long a, unsigned short b){
    return (long long) (a + b) ;
}
```

List 3.6 shows an assembler routine that calls the function.

List 3.6 Assembler Program addi Function Call with 64 bits of argument and return value

```
;---text section---
.section .text
.align 1
.global main
main:
ld.a     [%sp]-,%r4
xld      %r0,0xffff
xld      %r1,0xffff
xld      %r2,0xffff
xld      %r3,0xffff
ld       %r4,0x1
sub.a    %sp,0xc
ld       [%sp+0x8],%r4
ld       [%sp+0x6],%r3
ld       [%sp+0x4],%r2
ld       [%sp+0x2],%r1
ld       [%sp+0x0],%r0
ld.a     %r0,%sp
xcall    addi
ld       %r0,[%sp+0x0]
ld       %r1,[%sp+0x2]
ld       %r2,[%sp+0x4]
ld       %r3,[%sp+0x6]
add.a    %sp,0xc
ld.a     %r4,[%sp]+
ret
```

Stores data for the first argument in r0 to r3 registers, and data for the second argument in the r4 register.

sub. a secures the area required for data in the stack.
Stores arguments in the secured stack area.
As the return value exceeds 32 bits data, it is stored in the stack.
Stores the storage location address in the r0 register and deliver it to the function.

The return value can be obtained from the address specified by r0, where the value has been stored.
Release the secured stack area at the final step.

If the argument size is 64 bits or more upon calling the function, the argument is stored in the stack prior to the delivery. The stack area for 10 bytes of the argument is secured by sub.a **%sp**,0xc. Note that the address that SP can specify is the 31-bit boundary.
For details on the argument delivery, refer to the register usage description in the S5U1C17001C Manual.
If the return value exceeds 32 bits, specify in the R0 register the storage location address of the return value.
Be sure to release the secured stack area after using it.

4. PRECAUTIONS ON CREATING ASSEMBLER PROGRAMS

This chapter describes precautions to be noted when creating assembler programs.

4.1 Using “ext”

The immediate value definable by a 16-bit fixed length instruction code must be specified using 7-bit or 10-bit bit field depending on the instruction. Use the ext instruction to extend the immediate value size.
The ext instruction should be used in combination with a data transfer instruction, operation instruction, or a branch instruction, placed just before the instruction you want to extend its immediate value. Write the instruction with the “ext imm13” format (immX is an unsigned X-bit immediate value). A single ext instruction can extend an immediate value to 13 bits. Up to two ext instruction can be written continuously for further extension.
The ext instruction is valid only for the following instruction with an extendable immediate value, and invalid for others. If you write three or more ext instructions continuously, only the last two are valid, and others are neglected.
If the following instruction is not compatible with the ext instruction for extension, the ext instruction is executed as a nop instruction.
The next page shows an example of immediate value addressing.

●**Extending an immediate value to 16 bits, 20 bits or 24 bits**

"add" is a 16 bits addition instruction Extend a 7 bits immediate value by 0 and add it to the Rd register. The immediate value is unsigned 7 bits.

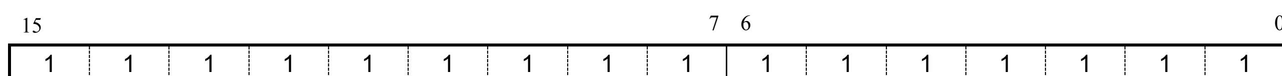
• Extending to 16 bits immediate value

To extend an immediate value to 16 bits, place a single ext instruction before the target instruction.

List 4.1 Example of 16 bits Extension

```
ext     0x1ff
add     %rd,0x7f                    ;=xadd    %rd, 0xffff
```

Extended immediate value



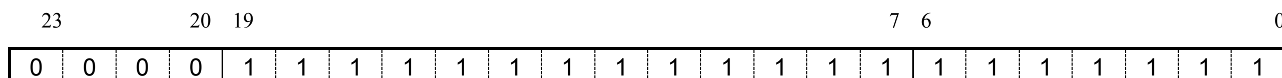
• Extending to 20 bits immediate value

To extend an immediate value to 20 bits, place a single ext instruction before the target instruction.

List 4.2 Example of 20 bits Extension

```
ext     0x1fff
add.a   %rd,0x7f                    ;=xadd.a  %rd, 0xfffff
```

Extended immediate value



Bits 23 to 20 are filled with 0 (Zero extension)

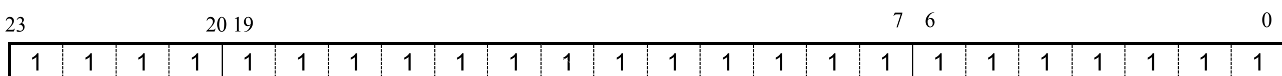
• Extending to 24 bits immediate value

To extend an immediate value to 24 bits, place two ext instructions before the target instruction.

List 4.3 Example of 24 bits Extension

```
ext     0xf
ext     0x1fff
add.a   %rd,0x7f                    ;=xadd.a  %rd, 0xffffff
```

Extended immediate value



For details, refer to the description of addressing mode with ext in the S1C17 Core Manual.

4.2 Extension Instruction

The assembler “as” supports extended instructions as explained in the following.
An extension instruction can usually include the content written by multiple instructions including the ext instruction. The requisite minimum basic instructions are extracted from the extension instruction depending on required capabilities and an operand immediate value.
The following explains the extension instruction by using the data transfer instruction between the stack and register.

・Symbols used for the explanation
immX An unsigned X-bit immediate value
(x,y) Bit field from bit X to bit Y.

Table 4.1 Types and Functions of Extension Instructions

Extension instruction	Capability	Spread format
sld.b %rd,[%sp+imm20]	%rb ← B[%sp+imm20] (sign extension)	(1)
sld.ub %rd,[%sp+imm20]	%rb ← B[%sp+imm20] (Zero extension)	(1)
sld %rd,[%sp+imm20]	%rb ← W[%sp+imm20]	(1)
sld.a %rd,[%sp+imm20]	%rb ← B[%sp+imm20](23:0), Neglect←A[%sp+imm20](31:24)	(1)
sld.b [%sp+imm20],%rs	B[%sp+imm20] ←%rs(7:0)	(1)
sld [%sp+imm20],%rs	W[%sp+imm20] ←%rs(15:0)	(1)
sld.a [%sp+imm24],%rs	A[%sp+imm20](23:0) ←%rs(23:0),A[%sp+imm20](31:24) ←0	(1)
xld.b %rd,[%sp+imm24]	%rb ← B[%sp+imm24] (sign extension)	(2)
xld.ub %rd,[%sp+imm24]	%rb ← B[%sp+imm24] (Zero extension)	(2)
xld %rd,[%sp+imm24]	%rb ← W[%sp+imm24]	(2)
xld.a %rd,[%sp+imm24]	%rb ← B[%sp+imm24](23:0), Neglect←A[%sp+imm24](31:24)	(2)
xld.b [%sp+imm24],%rs	B[%sp+imm24] ←%rs(7:0)	(2)
xld [%sp+imm24],%rs	W[%sp+imm24] ←%rs(15:0)	(2)
xld.a [%sp+imm24],%rs	A[%sp+imm24](23:0) ←%rs(23:0),A[%sp+imm24](31:24) ←0	(2)

* Each alphabetical character in the above table indicates as follows.
B (byte) → 8 bits
W (word) → 16 bits
A (address data) → 32 bits (with 0 written to higher 8 bits)

For address data, refer to the data format description in the S5U1C17001C Manual.

• Basic instruction after spreading

sld.b	xld.b	ld.b instruction
sld.ub	xld.ub	ld.ub instruction
sld	xld	ld instruction
sld.a	xld.a	ld.a instruction

• Spreading format

Omitting imm20 and imm24 indicates that [%sp+0x0] is specified for spreading the instruction.

(1) sld.a %rd, [%sp+imm20]
sld.a [%sp+imm20], %rs

Example: sld.a %rd,[%sp+imm20]

imm20≤0x7f	0x7f<imm20
ld.a %rd,[%sp+imm20(6:0)]	ext imm20(19:7) ld.a %rd,[%sp+imm20(6:0)]

(2) xld.a %rd, [%sp+imm24]
xld.a [%sp+imm24], %rs

Example: xld.a %rd,[%sp+imm24]

imm24≤0x7f	0x7f<imm24≤0xfffff	0xfffff<imm24
ld.a %rd,[%sp+imm24(6:0)]	ext imm24(19:7) ld.a %rd,[%sp+imm24(6:0)]	ext imm24(23:20) ext imm24(19:7) ld.a %rd,[%sp+imm24(6:0)]

For details, refer to the extension instruction description in the S5U1C17001C Manual.

4.3 Memory Models

Tool start command options and linked libraries can be switched according to the CPU type and memory space size used for the application system you will develop. We recommend you therefore select an appropriate memory model.
The memory model should be configured when creating a new project. It can be changed in the later stages.

There are three types of memory models: REGULAR, MIDDLE and SMALL.
Although the address space increases when the SMALL, MIDDLE and REGULAR memory models are used in this sequence, their coding efficiency is dropped. Select the memory address suitable for your application.

Table 4.2 Memory Models and Address Size

Memory model	Address size	Address space
REGULAR	24 bits	16M bytes
MIDDLE	20 bits	1M bytes
SMALL	16 bits	64K bytes

REVISION HISTORY

Attachment-1

Rev. No.	Date	Page	Category	Contents
Rev 1.0	2008/09/16	All	New	New establishment
Rev 1.1	2017/12/13	P11, 12	Error correction	The jpeq instructions in Lists 2.4 and 2.5 were corrected to jreq.

EPSON

International Sales Operations

AMERICA

EPSON ELECTRONICS AMERICA, INC.
214 Devcon Drive,
San Jose, CA 95112, USA
Phone: +1-800-228-3964 FAX: +1-408-922-0238

EUROPE

EPSON EUROPE ELECTRONICS GmbH
Riesstrasse 15, 80992 Munich,
GERMANY
Phone: +49-89-14005-0 FAX: +49-89-14005-110

ASIA

EPSON (CHINA) CO., LTD.
4th Floor, Tower 1 of China Central Place, 81 Jianguo Road, Chaoyang District, Beijing 100025 China
Phone: +86-10-8522-1199 FAX: +86-10-8522-1120

SHANGHAI BRANCH
Room 1701 & 1704, 17 Floor, Greenland Center II,
562 Dong An Road, Xu Hui District, Shanghai, CHINA
Phone: +86-21-5330-4888 FAX: +86-21-5423-4677

SHENZHEN BRANCH
Room 804-805, 8 Floor, Tower 2, Ali Center,No.3331
Keyuan South RD(Shenzhen bay), Nanshan District, Shenzhen 518054, CHINA
Phone: +86-10-3299-0588 FAX: +86-10-3299-0560

EPSON TAIWAN TECHNOLOGY & TRADING LTD.
14F, No. 7, Song Ren Road,
Taipei 110, TAIWAN
Phone: +886-2-8786-6688 FAX: +886-2-8786-6660

EPSON SINGAPORE PTE., LTD.
1 HarbourFront Place,
#03-02 HarbourFront Tower One, Singapore 098633
Phone: +65-6586-5500 FAX: +65-6271-3182

SEIKO EPSON CORP.
KOREA OFFICE
19F, KLI 63 Bldg., 60 Yoido-dong,
Youngdeungpo-Ku, Seoul 150-763, KOREA
Phone: +82-2-784-6027 FAX: +82-2-767-3677

SEIKO EPSON CORP.
SALES & MARKETING DIVISION

Device Sales & Marketing Department
421-8, Hino, Hino-shi, Tokyo 191-8501, JAPAN
Phone: +81-42-587-5816 FAX: +81-42-587-5116

Document Code: 411574601
First Issue September 2008
Revised December 2017 in JAPAN Ⓛ