EPSON
EXCEED YOUR VISION

S1C17 Series
Small Memory Programming

SEIKO EPSON CORPORATION

Rev.1.0

Evaluation board/kit and Development tool important notice

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

NOTICE

Table of Contents

1. Overview

This document explains how the program generated by the GNU17 uses memory (ROM and RAM) and how to decrease the amount of memory used by the program.

1.1 Environment

This document assumes source codes of the program are written in the C language and compiled and linked by the GNU17.
Example S1C17 instructions of the program are generated by the GNU17 version 2.2.0. The GNU17 version 2.x.x should generate same S1C17 instructions in most cases, but other versions may generate other instructions. Please confirm what instructions are generated for your program by using GNU17.

Example programs of this document are generated to run on the following memory map:

Address	Area
0xFF FFFF 0x03 8000	Reserved
0x03 7FFF 0x00 8000	ROM (192K bytes)
0x00 7FFF 0x00 4000	I/O Registers of Peripheral Devices (16K bytes)
0x00 3FFF 0x00 0000	RAM (16K bytes)

1.2 Memory Used by the GNU17 C Compiler

This section explains how much S1C17 memory is used by functions, variables and constants of the C language. For details of the GNU17 C compiler and the linker, refer to the GNU17 manual, especially “6.4 Compiler Output”.

1.2.1 Functions

The GNU17 C compiler locates functions to the ‘.text’ section. The GNU17 linker and its normal linker script relocate the ‘.text’ section to ROM. Therefore, the GNU17 usually locates functions of the C language to ROM.

C source code	S1C17 instructions
`extern void external_near_function(void);` `extern void external_far_function(void);`	

`void internal_function(void) {`
`  return;`
`}`
	`0x82ee:  ret`

`static void static_function(void) {`
`  return;`
`}`
	`0x82f0:  ret`

`void caller_function(void) {`
`  static_function();`
`  internal_function();`
`  external_near_function();`
`  external_far_function();`
`}`
	`0x82f2:  call  0x3fe`
	`0x82f4:  call  0x3fc`
	`0x82f6:  call  0x3`
	`0x82f8:  ext  0xf`
	`0x82fa:  call  0x282`
	`0x82fc:  ret`

The start address of a function is allocated to 16-bit boundary, because the size of a S1C17 instruction word is 16 bits.
Each function needs a ‘ret’ instruction. This means the least size of a function is 2 bytes.
The caller of a function needs 2 bytes for a ‘call’ instruction at least. The program branches to the address

calculated as PC + 2 + 10-bit signed value specified by the operand of the ‘call’ instruction. When the callee function is not in the 10-bit area, the caller function needs additional 2 bytes for an ‘ext’ instruction to extend the operand in the following ‘call’ instruction.
The number of instructions to call could be decreased by arranging the caller address and the callee address to be near. An easy method to decrease instructions is to define the caller function and the callee function in near lines of a C source file.

1.2.2 Global Variables

The GNU17 C compiler locates global variables without initial values to the ‘.bss’ section.
Global variables with initial values are located to the ‘.data’ section, and their initial values are kept in the ‘data_lma’ area.
The GNU17 linker relocates the ‘.bss’ and ‘.data’ section to RAM and the ‘data_lma’ area to ROM. Therefore, the GNU17 usually locates global variables of the C language to RAM, and keeps their initial values in ROM.

C source code

```
int global_variable_without_value;
int global_variable_with_value_one = 1;
int global_variable_with_value_zero = 0;
```

Mapped address by the GNU17 linker

```
.bss    0x00000000  0x2
        0x00000000      __START_bss=.
 .bss   0x00000000  0x2 variables.o
        0x00000000      global_variable_without_value
        0x00000002      __END_bss=.

.data   0x00000002  0x4 load address 0x000094fe
        0x00000002      __START_data=.
 .data  0x00000002  0x4 variables.o
        0x00000002      global_variable_with_value_one
        0x00000004      global_variable_with_value_zero
        0x00000006      __END_data=.

        0x000094fe      __START_data_lma=__END_rodata
        0x00009502      __END_data_lma=(__END_rodata+(__END_data-__START_data))
```

GDB console

```
(gdb) p &global_variable_without_value
$1 = (int *) 0x0
(gdb) p &global_variable_with_value_one
$2 = (int *) 0x2
(gdb) p &global_variable_with_value_zero
$3 = (int *) 0x4
(gdb) p (void *)&__START_data_lma
$4 = (void *) 0x94fe
(gdb) p (void *)&__END_data_lma
$5 = (void *) 0x9502
(gdb) x/2h &__START_data_lma
0x94fe:  0x0001   0x0000
```

After loading the program to the target device by the GNU17 debugger (GDB), the content of the ‘data_lma’ area could be checked from GDB using debug symbols.

The ‘.bss’ section should be initialized by 0, because the C language specifies that value of a global variable not initialized explicitly is 0.
The size of the ‘data_lma’ area could be decreased by omitting initialization by 0.

C source code

```
int global_variable_without_value;
int global_variable_with_value_one = 1;
int global_variable_with_value_zero;
```

GDB console

```
(gdb) p &global_variable_without_value
$1 = (int *) 0x0
(gdb) p &global_variable_with_value_one
$2 = (int *) 0x4
(gdb) p &global_variable_with_value_zero
$3 = (int *) 0x2
 (gdb) p (void *)&__START_data_lma
$4 = (void *) 0x94fe
(gdb) p (void *)&__END_data_lma
$5 = (void *) 0x9500
(gdb) x/1h &__START_data_lma
```

```
0x94fe:  0x0001
```

Global variables belonging to the same section in a C source file are arranged into the S1C17 address space like the order of their definition.
Following variables belong to the '.bss' section and ordered like their definition.

C source code

```
char global_a_8bit;
long global_b_32bit;
char global_c_8bit;
short global_d_16bit;
short global_e_16bit;
long global_f_32bit;
```

GDB console

```
(gdb) p &global_a_8bit
$1 = 0x0
(gdb) p &global_b_32bit
$2 = (long int *) 0x4
(gdb) p &global_c_8bit
$3 = 0x8 '¥252' <repeats 200 times>...
(gdb) p &global_d_16bit
$4 = (short int *) 0xa
(gdb) p &global_e_16bit
$5 = (short int *) 0xc
(gdb) p &global_f_32bit
$6 = (long int *) 0x10
```

Because each variable are aligned to own boundary depending on their data type, there is some unused memory.

Address	Address + 0	Address + 1	Address + 2	Address + 3
0x000000	global_a_8bit	unused	unused	unused
0x000004	global_b_32bit			
0x000008	global_c_8bit	unused	global_d_16bit	
0x00000c	global_e_16bit		unused	unused
0x000010	global_f_32bit			

The amount of unused memory could be decreased by defining global variables of the same data type in one place.

C source code

```
char global_a_8bit;
char global_c_8bit;
short global_d_16bit;
short global_e_16bit;
long global_b_32bit;
long global_f_32bit;
```

GDB console

```
(gdb) p &global_a_8bit
$1 = 0x0
(gdb) p &global_c_8bit
$2 = 0x1 '¥252' <repeats 200 times>...
(gdb) p &global_d_16bit
$3 = (short int *) 0x2
(gdb) p &global_e_16bit
$4 = (short int *) 0x4
(gdb) p &global_b_32bit
$5 = (long int *) 0x8
(gdb) p &global_f_32bit
$6 = (long int *) 0xc
```

Address	Address + 0	Address + 1	Address + 2	Address + 3
0x000000	global_a_8bit	global_c_8bit	global_d_16bit	
0x000004	global_e_16bit		unused	unused
0x000008	global_b_32bit			
0x00000c	global_f_32bit			

Definition of variables in order from bigger to smaller type could decrease the amount of unused memory further, because the linker may relocate other objects there.

1.2.3 Local Variables

Most local variables have automatic storage duration. The GNU17 C compiler locates these variables to the stack frame of the function or S1C17registers.
Local variables with ‘static’ specifier have static storage duration. The GNU17 C compiler locates these variables without initial values to the ‘.bss’ section.

C source code

```
static short static_local_values(short a) {
    static short static_local_without_value;

    static_local_without_value = a;

    return static_local_without_value;
}
```

S1C17 instructions

```
0x929a:   ld      [0x2],%r0
0x929c:   ret
```

The GNU17 C compiler locates ‘static’ local variables with initial values to the ‘.data’ section, and keeps their initial values in the ‘data_lma’ area.

C source code

```
static short static_local_values(short a) {
    static short static_local_with_value = 10;

    static_local_with_value += a;

    return static_local_with_value;
}
```

S1C17 instructions

```
0x929a:   ld      %r2,[0x14]
0x929c:   add     %r0,%r2
0x929e:   ld      [0x14],%r0
0x92a0:   ret
```

Because local variables of static storage duration are initialized before ‘main’ function, they are not initialized again in the function.
The size of the ‘data_lma’ area could be decreased by omitting initialization by 0.

Local variables on the stack frame are arranged in order of their definition as same as global variables. This means that the size of the stack frame (allocated from RAM) could be decreased by defining local variables of the same data type in one place. But it is necessary to confirm S1C17 instructions to know the size of the stack frame, because some local variables are optimized by the GNU17 C compiler and not stored on the stack frame.

C source code

```
short automatic_local_values(short a) {
    char local_a_8bit;
    long local_b_32bit;
    char local_c_8bit;
    short local_d_16bit;
    short local_e_16bit;
    long local_f_32bit;

    local_a_8bit = sub_function (a);
    local_b_32bit = sub_function (local_a_8bit);
    local_c_8bit = sub_function(local_b_32bit);
    local_d_16bit = sub_function(local_c_8bit);
    local_e_16bit = sub_function(local_d_16bit);
    local_f_32bit = sub_function(local_e_16bit);

    return local_f_32bit;
}
```

S1C17 instructions

```
0x927a:   call    0x3fe
0x927c:   call.d  0x3fd
0x927e:   ld.b    %r0,%r0
0x9280:   ld      %r2,%r0
0x9282:   cv.ls   %r3,%r2
0x9284:   call.d  0x3f9
0x9286:   ld      %r0,%r2
0x9288:   call.d  0x3f7
0x928a:   ld.b    %r0,%r0
0x928c:   call    0x3f5
0x928e:   call    0x3f4
0x9290:   ld      %r2,%r0
0x9292:   cv.ls   %r3,%r2
0x9294:   ld      %r0,%r2
0x9296:   ret
```

1.2.4 Constants

Global constants and 'static' specified constants have static storage duration.
The GNU17 C compiler locates constants of static storage duration to the ‘.rodata’ section. The GNU17 linker relocates the ‘.rodata’ section to ROM.
Therefore, the GNU17 usually locates constants of the C language to ROM.

C source

```
const char global_constant_a_8bit = 0x01;
```

code

```
const short global_constant_b_16bit = 0x2345;

short constant_values(void) {
    static const short local_constant_c_16bit = 0x6789;

    return local_constant_c_16bit;
}

const long global_constant_d_32bit = 0xabcdef01;
const short global_constant_e_16bit = 0x2345;
```

GDB console

```
(gdb) p &global_constant_a_8bit
$1 = 0x9568 "."
(gdb) p &global_constant_b_16bit
$2 = (short int *) 0x956a
(gdb) p &global_constant_d_32bit
$3 = (long int *) 0x9570
(gdb) p &global_constant_e_16bit
$4 = (long int *) 0x9574
(gdb) p &__START_rodata
$5 = (<data variable, no debug info> *) 0x9568
(gdb) p &__END_rodata
$6 = (<variable (not text or data), no debug info> *) 0x9576 "."
(gdb) x/14b &__START_rodata
0x9568:  0x01    0x00    0x45    0x23    0x89    0x67    0x00    0x00
0x9570:  0x01    0xef    0xcd    0xab    0x45    0x23
```

Constants of static storage duration are arranged in order of their definition as same as variables. This means that the size of the '.rodata' section could be decreased by defining constants of the same data type in one place.

C source code

```
const long global_constant_d_32bit = 0xabcdef01;
const short global_constant_b_16bit = 0x2345;
const short global_constant_e_16bit = 0x2345;

short constant_values (void) {
    static const short local_constant_c_16bit = 0x6789;

    return local_constant_c_16bit;
}

const char global_constant_a_8bit = 0x01;
```

GDB console

```
(gdb) p &global_constant_d_32bit
$1 = (long int *) 0x9568
(gdb) p &global_constant_b_16bit
$2 = (short int *) 0x956c
(gdb) p &global_constant_e_16bit
$3 = (short int *) 0x956e
(gdb) p &global_constant_a_8bit
$4 = 0x9572 "."
(gdb) p &__START_rodata
$5 = (<data variable, no debug info> *) 0x9568
(gdb) p &__END_rodata
$6 = (<variable (not text or data), no debug info> *) 0x9574 "."
(gdb) x/12b &__START_rodata
0x9568:  0x01    0xef    0xcd    0xab    0x45    0x23    0x45    0x23
0x9570:  0x89    0x67    0x01    0x00
```

Even if the values of 2 constants are equivalent, the GNU17 generates each object for them. The size of the '.rodata' section could decrease by gathering these constants into a constant.

C source code

```
const long global_constant_d_32bit = 0xabcdef01;
const short global_constant_b_16bit = 0x2345;
#define global_constant_e_16bit      global_constant_b_16bit

short constant_values(void) {
    static const short local_constant_c_16bit = 0x6789;

    return local_constant_c_16bit;
}

const char global_constant_a_8bit = 0x01;
```

GDB

```
(gdb) p &__START_rodata
$1 = (<data variable, no debug info> *) 0x9568
```

console

```
(gdb) p &__END_rodata
$2 = (<variable (not text or data), no debug info> *) 0x9574 "."
(gdb) x/12b &__START_rodata
0x9568:  0x01     0xef     0xcd     0xab     0x45     0x23     0x89     0x67
0x9570:  0x01     0x00     0x00     0x00
```

Constants of automatic storage duration are generated whenever they become effective.
This means that the GNU17 C compiler locates these constants to the stack frame and their initial value to the '.rodata' section. Additionally, the GNU17 C compiler generates S1C17 instructions to set initial value.

C source code

```
short local_constant_values(void) {
    const short constant_array[4] = {
        0x0123, 0x4567, 0x89ab, 0xcdef,
    };

    return sub_function(constant_array);
}
```

S1C17 instructions

```
0x8056:   sub.a   %sp,0x8
0x8058:   ld.a    %r0,%sp
0x805a:   ext     0x12b
0x805c:   ld.a    %r1,0xc
0x805e:   ext     0x2
0x8060:   call.d  0x135    <memcpy>
0x8062:   ld      %r2,0x8
0x8064:   ld.a    %r0,%sp
0x8066:   call    0x3f5
0x8068:   add.a   %sp,0x8
0x806a:   ret
```

The size of the stack frame and S1C17 instructions could decrease by specifying 'static' to a constant to change the storage duration from automatic to static.

C source code

```
short local_constant_values(void) {
    static const short constant_array[4] = {
        0x0123, 0x4567, 0x89ab, 0xcdef,
    };

    return sub_function(constant_array);
}
```

S1C17 instructions

```
0x8056:   ext     0x12a
0x8058:   ld.a    %r0,0x6e
0x805a:   call    0x3fb
0x805c:   ret
```

1.2.5 Stack Area

A part of RAM is used as the stack area to store the call stack of the program.
When the S1C17 executes a 'call' instruction, the value of the PC register is stored to the address hold by the SP register. Because the size of the PC register is 24-bits, it is necessary for the value of the SP register to point out a 32 bit boundary address. And the 2 low-order bits of the SP register are fixed to 0.
Therefore, the GNU17 C compiler adjusts the size of the stack frame of a function to make the SP register points at a 32 bits border. In other words, it means that the size of the stack frame of a function that the GNU17 C compiler generates is a multiple of 4 and could not decreased by 1-3 bytes.

1.3 How to find out the Amount of Used Memory

1.3.1 Used ROM

The GNU17 linker shows the amount of ROM area used by the program in the content of the link map information (*.map) file. Please refer "9.2.2 Output Files" of the GNU17 manual to output the link map information file.

The amount of ROM area used by the program is possible to be calculated as follows:

```
address_of_last_symbol_on_ROM – address_of_first_symbol_on_ROM [bytes]
```

The 'address_of_first_symbol_on_ROM' may be equals 0x008000 and usually the interrupt vector table.
The 'address_of_last_symbol_on_ROM' may be the biggest address of the end of the '.text' section, the '.rodata' section or the 'data_lma' area (initial values of the '.data' section).

Section / Area	Symbol of the end

'.text' section	__END_text
'.rodata' section	__END_rodata
'data_lma' area	__END_data_lma

1.3.2 Used RAM

The GNU17 linker shows the amount of RAM area (except the stack area) allocated for the program in the content of the link map information. Please refer "9.2.2 Output Files" of the GNU17 manual to output the link map information.

The amount of RAM area allocated for the program is able to be calculated as follows:

```
address_of_last_symbol_on_RAM – address_of_first_symbol_on_RAM [bytes]
```

The 'address_of_first_symbol_on_RAM' should be equals 0x000000.
The 'address_of_last_symbol_on_RAM' may be the biggest address of the end of the .bss section or the .data section.

Section	Symbol of the end
'.bss' section	__END_bss
'.data' section	__END_data

1.3.3 Stack area of the Program

The GNU17 linker outputs an initial value of the SP register as the '__START_stack' symbol according to the value set on the GNU17 IDE. Please refer "5.7.9 Editing a Linker Script" of the GNU17 manual to set the initial value of the SP register.
To make the '__START_stack' to the initial value of the SP register, the program should set it to the SP register.

C source code

```
extern unsigned long __START_stack[];
static void boot(void);

#ifdef __POINTER16
#define VECTOR(p)   (p),0
#else
#define VECTOR(p)   (p)
#endif

void * const vector[] = {
    VECTOR(boot),
};
```

```
static void boot(void) {
    asm __volatile__("xld.a %sp, __START_stack");
    asm __volatile__("xjpr main");
}
```

S1C17 instructions

```
0x92a6:    ext      0x1f
0x92a8:    ld.a     %sp,0x40
0x92aa:    jpr      0x5
0x92ac:    ret
```

Mapped address by the GNU17 linker

```
        0x00000fc0        __START_stack=0xfc0

.vector  0x00008000   0x10
        0x00008000        __START_vector=.
 vector.o(.rodata)
 .rodata 0x00008000   0x10 vector.o
        0x00008000        vector
        0x00008010        __END_vector=.

.text    0x00008010 0x155c
        0x00008010        __START_text=.

 vector.o(.text)
 .text   0x000092a6   0x16 vector.o
        0x000092b6        main
```

GDB console

```
(gdb) p &__START_stack
$1 = (<variable (not text or data), no debug info> *) 0xfc0 ".,."
```

```
(gdb) p &vector
$2 = (void *(*)[1]) 0x8000
(gdb) p main
$3 = {int ()} 0x92b6 <main>
(gdb) x/1w 0x8000
0x8000 <vector>:  0x000092a6
```

The size of the stack frame of each function could be calculated from S1C17 instructions generated by the GNU17 C compiler.

C source code

```
short using_stack(short a, short b, short c, short d) {
    short buffer[5] = {0, b, c, d, a};
    short f;

    f = sub_function(buffer);

    return f + buffer[0];
}
```

S1C17 instructions

```
0x9148:   ld.a    -[%sp],%r4
0x914a:   sub.a   %sp,0xc
0x914c:   ld      %r4,0x0
0x914e:   ld      [%sp+0x0],%r4
0x9150:   ld      [%sp+0x2],%r1
0x9152:   ld      [%sp+0x4],%r2
0x9154:   ld      [%sp+0x6],%r3
0x9156:   ld      [%sp+0x8],%r0
0x9158:   ld.a    %r0,%sp
0x915a:   call    0x3f3
0x915c:   add.a   %sp,0xc
0x915e:   ld.a    %r4,[%sp]+
0x9160:   ret
```

In the end of each function, the stack frame has been removed by following instructions:

- An ‘add.a %sp, imm’ instruction removes local variable area from the stack frame.
- Some ‘ld.a %rd, [%sp]+’ instructions remove register save area from the stack frame.
- A ‘ret’ instruction removes return address from the stack frame and goes back to the caller.

The size of the stack frame for a function could be calculated as follows:

```
(Size of local variable area) + (size of register save area) + (return address)
= (‘imm’ of ‘add.a %sp, imm’) + (number of ‘ld.a %rd, [%sp]+’s) * 4 + 4 [bytes]
```

The amount of used stack area at a function could be calculated as follows:

```
“The maximum used stack area among callers of the function” + “the stack frame for the function”
```

The amount of stack area for the program could be calculated as follows:

```
“The maximum used stack area of functions called from the S1C17 reset vector”
 + “the maximum used stack area of functions called from S1C17 interrupt handlers”
```

A rough size of the stack area could be confirmed as following procedures:

- Transfer the program to the target device using the GNU17 GDB.
- Input the ‘c17 fw’ command on the GDB console to fill the stack area (not allocated RAM area by the GNU17 linker) with a value.
- Execute the program and suspend it.
- Confirm the SP register and the stack area. The area changed from the value filled by the ‘c17 fw’ command would be used as the stack.

2. Setting the GNU17 IDE and Tools

This section shows settings of the GNU17 that affects the amount of used memory.
For details of setting the GNU17 IDE and tools, refer to the GNU17 manual, especially “5 GNU17 IDE”.

2.1 Select the Memory Model

The memory model could be selected at creating a new project and it is configurable in the [Properties] dialog box of the project.
The amount of memory used by the middle model is less than the regular model, and the small model is less than the middle model.

- Regular model (24 bits space):
  The address space is not limited but the program needs more memory than other models.

- Middle model (20 bits space):
  The address space used by the program should be limited to 20-bit range. This means that RAM and ROM and all peripheral device registers used by the program should be mapped from 0x000000 to 0x0FFFFF.

C source code

```
int middle_model(int * data, int * array, int offset) {
    int r;
    void * buffer[256];

    memcpy(buffer ,data, sizeof(buffer));
    r = sub_function(buffer);

    return r + array[offset];
}
```

S1C17 instructions (middle model)

```
0x8eae:    ld.a    -[%sp],%r4
0x8eb0:    ld.a    -[%sp],%r5
0x907e:    ld.a    -[%sp],%r4
0x9080:    ld.a    -[%sp],%r5
0x9082:    ld.a    -[%sp],%r6
0x9084:    ld.a    -[%sp],%r7
0x9086:    ext     0x8
0x9088:    sub.a   %sp,0x4
0x908a:    ext     0x8
0x908c:    ld.a    [%sp+0x0],%r0
0x908e:    ld.a    %r4,%r1
0x9090:    ld      %r6,%r2
0x9092:    ld.a    %r0,%sp
0x9094:    ext     0x8
0x9096:    ld.a    %r1,[%sp+0x0]
0x9098:    ext     0x8
0x909a:    ld      %r2,0x0
0x909c:    call    0x152    <memcpy>
0x909e:    ld.a    %r0,%sp
0x90a0:    call    0x3ec
0x90a2:    cv.as   %r6,%r6
0x90a4:    add.a   %r6,%r6
0x90a6:    add.a   %r4,%r6
0x90a8:    ld      %r2,[%r4]
0x90aa:    add     %r0,%r2
0x90ac:    ext     0x8
0x90ae:    add.a   %sp,0x4
0x90b0:    ld.a    %r7,[%sp]+
0x90b2:    ld.a    %r6,[%sp]+
0x90b4:    ld.a    %r5,[%sp]+
0x90b6:    ld.a    %r4,[%sp]+
0x90b8:    ret
```

- Small model (16 bits space):
  The size of the pointer type of the C language is 16-bit, and the address space used by the program should be limited to 16-bit range. This means that RAM and ROM and all peripheral device registers used by the program should be mapped from 0x000000 to 0x00FFFF.
  Because the size of the pointer type becomes 16-bit from 24-bit (32-bit on memory), used memory is less than other models. Moreover, the number of S1C17 instructions is less, because 16-bit pointer makes easy to operate between the pointer type and the integer type.

C source code

```
int small_model(int * data, int * array, int offset) {
    int r;
    void * buffer[256];
```

S1C17 instructions (small model)

```
0x8eae:    ld.a    -[%sp],%r4
0x8eb0:    ld.a    -[%sp],%r5
0x8eb2:    ext     0x4
```

```
    memcpy(buffer ,data, sizeof(buffer));
    r = sub_function(buffer);

    return r + array[offset];
}
```

```
0x8eb4:    sub.a   %sp,0x0
0x8eb6:    ld      %r3,%r0
0x8eb8:    ld      %r5,%r1
0x8eba:    ld      %r4,%r2
0x8ebc:    ld.a    %r0,%sp
0x8ebe:    ext     0x4
0x8ec0:    ld      %r2,0x0
0x8ec2:    call.d  0x113    <memcpy>
0x8ec4:    ld      %r1,%r3
0x8ec6:    ld.a    %r0,%sp
0x8ec8:    call    0x3f0
0x8eca:    sl      %r4,0x1
0x8ecc:    add     %r4,%r5
0x8ece:    ld      %r2,[%r4]
0x8ed0:    add     %r0,%r2
0x8ed2:    ext     0x4
0x8ed4:    add.a   %sp,0x0
0x8ed6:    ld.a    %r5,[%sp]+
0x8ed8:    ld.a    %r4,[%sp]+
0x8eda:    ret
```

Please refer the GNU17 manual “5.4.2 Creating a New Project”, “5.4.11 Project Properties” and “6.3.2 Command-line Options”.

2.2 Locate Sections of the Program

In case of that the callee function exists near the caller address, the GNU17 generates less S1C17 instructions. The caller and the callee may be relocated far away so that the GNU17 IDE links C source files with the alphabetical order of the file name. Creating a section to allocate C source files is necessary to arrange functions at near address. However, the effect of arranging may be small.
Please refer “5.7.9 Editing a Linker Script” of the GNU17 manual to see how to create a new section and edit it.

3. Basic Operations

This section shows what S1C17 instructions is output to basic operations of the C language by the GNU17. For more details, please refer the GNU17 manual, especially “6.4 Compiler Output”.

3.1 Integer Type Operations

3.1.1 8-bit and 16-bit Integer Type Operations

The GNU17 C compiler treats the following types as 8-bit or 16-bit integer of the S1C17.

- ‘char’ 8-bit (equals ‘signed char’)
- ‘unsigned char’ 8-bit
- ‘short’ 16-bit (equals ‘signed short’)
- ‘unsigned short’ 16-bit
- ‘int’ 16-bit (equals ‘singed int’)
- ‘unsigned int’ 16-bit

The S1C17 is able to operate data of these types efficiently.

3.1.1.1 Data Transmission between Memory and Registers in Common Use

The S1C17 transmits an 8-bit or 16-bit value between memory and a register by an ‘ld’ instruction. Procedure that transmits a value from the source memory to the destination memory takes 2 S1C17 instructions because of an ‘ld’ instruction to read from the source and an ‘ld’ instruction to write to the destination.

C source code

```
short transmit_pointed_value(short * a, short * b) {
    short c;

    c = *b;
    *a = c;

    return c;
}
```

S1C17 instructions

```
0x828e:   ld     %r2,[%r1]
0x8290:   ld     [%r0],%r2
0x8292:   ld     %r0,%r2
0x8294:   ret
```

The number of S1C17 instructions for accessing data within the 8KB range of a register value is 2. The GNU17 C compiler uses these instructions to access an offset from a pointer and a member of a structure.

C source code

```
short transmit_offset_value(short * a, short * b) {
    short c;

    c = *(b + 1);
    *(a + 256) = c;

    return c;
}
```

S1C17 instructions

```
0x828e:   ext    0x2
0x8290:   ld     %r2,[%r1]
0x8292:   ext    0x200
0x8294:   ld     [%r0],%r2
0x8296:   ld     %r0,%r2
0x8298:   ret
```

To operate the value in the memory, the S1C17 read the value to a register. Therefore, the value in the memory needs more S1C17 instructions than the value in the register to operate.

C source code

```
void copy_triple(short * a, short * b) {
    *(a + 1) = *(b + 1);
    *(a + 2) = *(b + 1);
    *(a + 3) = *(b + 1);
}
```

S1C17 instructions

```
0x828e:   ext    0x2
0x8290:   ld     %r2,[%r1]
0x8292:   ext    0x2
0x8294:   ld     [%r0],%r2
0x8296:   ext    0x2
0x8298:   ld     %r2,[%r1]
0x829a:   ext    0x4
0x829c:   ld     [%r0],%r2
0x829e:   ext    0x2
```

	0x82a0: ld %r2,[%r1] 0x82a2: ext 0x6 0x82a4: ld [%r0],%r2 0x82a6: ret

The number of S1C17 instructions decreases when the value read from the memory to a register is repeatedly used.

C source code	S1C17 instructions

void copy_triple(short * a, short * b) {
short c;
 	
c = *(b + 1);
*(a + 1) = c;
*(a + 2) = c;
*(a + 3) = c;
}
	0x828e: ext 0x2
	0x8290: ld %r2,[%r1]
	0x8292: ext 0x2
	0x8294: ld [%r0],%r2
	0x8296: ext 0x4
	0x8298: ld [%r0],%r2
	0x829a: ext 0x6
	0x829c: ld [%r0],%r2
	0x829e: ret

The 'ld' instruction is able to increment (or decrement) the value of its destination register. When the C source code is described by the pointer with a postfix increment operator, the number of S1C17 instructions could decrease.

C source code	S1C17 instructions

void copy_triple(short * a, short * b) {
short * p;
short c;
 	
c = *(b + 1);
p = a + 1;
*p++ = c;
*p++ = c;
*p++ = c;
}
	0x828e: ext 0x2
	0x8290: ld %r2,[%r1]
	0x8292: add %r0,0x2
	0x8294: ld [%r0]+,%r2
	0x8296: ld [%r0]+,%r2
	0x8298: ld [%r0],%r2
	0x829a: ret

3.1.1.2 Data Transmission between Array Elements and Registers

When the index of the array is a variable, the GNU17 C compiler calculates the address of the array element by adding the index number to the array (pointer).

C source code	S1C17 instructions

short array_access(short * a, short b) {
return a[b];
}
	0x828e: sl %r1,0x1
	0x8290: add %r1,%r0
	0x8292: ld %r0,[%r1]
	0x8294: ret

The number of S1C17 instructions and used registers may decrease when the index of the array is a constant value.

3.1.1.3 Data Operation

One S1C17 instruction is possible to process some operations (inversion, negation, addition, subtraction, inequality, equality, inequality, AND, OR and exclusive-OR) of the C language about 16-bit integers. The number of S1C17 instructions may decrease if these operations could replace other operations.

The S1C17 is possible to process a left-shift and a right-shift by one instruction about 16-bit integers. However, the shift amount of a S1C17 instruction is limited to 0-4 and 8.
When the shift amount is specified by the variable in the C source code, the GNU17 C compiler outputs S1C17 instructions to call the shift function defined in the GNU17 emulation library (libgcc*.a).

C source code	S1C17 instructions
short shift_variable(short a, short b) { return a << b;	0x8012: ext 0x2 0x8014: call 0x8d <__ashlhi3>

} 	0x8016: ret

When the shift amount is specified by a constant value, the number of S1C17 instructions to shift is 1-3.

C source code	S1C17 instructions

short shift_constant(**short** a) {
  return a << 7;
}
	0x8018: sl %r0,0x4
	0x801a: sl %r0,0x3
	0x801c: ret

The number of S1C17 instructions to shift is different according to the situation, but the shift by constant spends less S1C17 cycles.

The S1C17 is not possible to process multiplication, division and modulus operation by one instruction. To execute these operations, the GNU17 C compiler outputs S1C17 instructions to call functions defined in the GNU17 emulation library.

C source code	S1C17 instructions

short calling_emulation_library(**short** a) {
  return a / 250;
}
	0x801e: ext 0x1
	0x8020: ld %r1,0x7a
	0x8022: ext 0x2
	0x8024: call 0xed <__divhi3>
	0x8026: ret

These operations spend more memory and S1C17 cycles than other operations because of calling instructions and linking functions of the GNU17 emulation library.

3.1.1.4 Type Conversion between 8-bit and 16-bit Integers

The S1C17 executes most operations as 16-bit integer.
The S1C17 operates 8-bit integer as 16-bit integer on the registers, and converts 16-bit integer into 8-bit integer by an 'ld' instruction.
Therefore, when the variable of 8-bit integer type is used to calculate on the C source code, the GNU17 C compiler may output S1C17 instructions to convert 16-bit integer to 8-bit integer.

C source code	S1C17 instructions

char byte_add_and_convert(**char** a, **short** b) {
  return a + b;
}
	0x801e: add %r0,%r1
	0x8020: ld.b %r0,%r0
	0x8022: ret

It is not necessary to convert 16-bit integer into 8-bit integer when all variables are 16-bit integer.

C source code	S1C17 instructions

short short_add_and_no_convert(**short** a, **short** b) {
  return a + b;
}
	0x8024: add %r0,%r1
	0x8026: ret

3.1.2 32-bit Integer Type Operations

The GNU17 C compiler treats the following types as 32-bit integer of the S1C17.

- 'long' (equals 'signed long')
- 'unsigned long'

There is no S1C17 instruction to operate 32-bit integer type. The GNU17 C compiler outputs S1C17 instructions to combine two S1C17 registers to operate a 32-bit integer.
The operation of 32-bit integer on the GNU17 requires more memory and S1C17 registers than 8-bit and 16-bit operations. Replacing 32-bit operations with 16-bit operations may decrease the size of the program.

3.1.2.1 Data Transmission between Memory and Registers in Common Use

The GNU17 C compiler outputs two 'ld' instructions to transmit a 32-bit integer value between memory and registers. This means that the operation of 32-bit integer needs twice instructions and registers.

C source code

```
void transmit_pointed_long(long * a, long * b) {
    *b = *a;
}
```

S1C17 instructions

```
0x82ac:   ld     %r2,[%r0]
0x82ae:   ext    0x2
0x82b0:   ld     %r3,[%r0]
0x82b2:   ld     [%r1],%r2
0x82b4:   ext    0x2
0x82b6:   ld     [%r1],%r3
0x82b8:   ret
```

The number of S1C17 instructions output by the GNU17 C compiler may decrease by handling the pointer for 32-bit type like the pointer for 16-bit integer type.

C source code

```
void transmit_pointed_long(long * a, long * b) {
    short * c;
    short * d;

    c = (short *)a;
    d = (short *)b;
    *d++ = *c++;
    *d++ = *c++;
}
```

S1C17 instructions

```
0x82ac:   ld     %r2,[%r0]+
0x82ae:   ld     [%r1]+,%r2
0x82b0:   ld     %r2,[%r0]
0x82b2:   ld     [%r1],%r2
0x82b4:   ret
```

3.1.2.2 Data Transmission between Array Elements and Registers

When the index of the array is a 32-bit variable, the GNU17 C compiler calculates the address of the array element by adding the index number to the array address.

C source code

```
short long_array_access(long i, short * a) {
    return a[i];
}
```

S1C17 instructions (middle model)

```
0x9124:   cv.al   %r0,%r1
0x9126:   ld.a    %r0,%r0
0x9128:   add.a   %r0,%r0
0x912a:   add.a   %r2,%r0
0x912c:   ld      %r0,[%r2]
0x912e:   ret
```

The GNU17 C compiler outputs less S1C17 instructions to calculate the address of the array element for the small model program, because its address space is limited to 16-bit range.

C source code

```
short long_array_access(long i, short * a) {
    return a[i];
}
```

S1C17 instructions (small model)

```
0x9124:   ld     %r3,%r0
0x9128:   sl     %r3,0x1
0x912a:   add    %r3,%r2
0x912c:   ld     %r0,[%r3]
0x912e:   ret
```

In case of the index of the array is a 16-bit variable, the GNU17 C compiler outputs less S1C17 instructions and saves S1C17 registers.

C source code

```
short short_array_access(short i, short * a) {
    return a[i];
}
```

S1C17 instructions (small model)

```
0x9130:   sl     %r0,0x1
0x9132:   add    %r0,%r1
0x9134:   ld     %r0,[%r0]
0x9136:   ret
```

3.1.2.3 Data Operation

To calculate 32-bit integers, the GNU17 C compiler outputs S1C17 instructions to call functions defined in the GNU17 emulation library (libgcc*.a). Similarly, the comparison including 32-bit integer needs more S1C17

instructions.

C source code	S1C17 instructions

short compare_long(long a) {
  short b;
 	
  if (a > 0) {
  b = 0;
  } **else** {
  b = 1;
  }
 	
  return b;
}
	0x8076: cmp %r1,0x0
	0x8078: jrgt.d 0x4
	0x807a: cmp %r1,0x0
	0x807c: jrne.d 0x4
	0x807e: cmp %r0,0x0
	0x8080: jrule 0x2
	0x8082: jpr.d 0x2
	0x8084: ld %r0,0x0
	0x8086: ld %r0,0x1
	0x8088: ret

The number of S1C17 instructions that the GNU17 C compiler outputs may be decreased by using 16-bit integer type instead of 32-bit integer type if the value is within the 16-bit range.

C source code	S1C17 instructions

short compare_long(long a) {
  short b;
 	
  b = (**short**)a;
  if (b > 0) {
  b = 0;
  } **else** {
  b = 1;
  }
 	
  return b;
}
	0x8076: cmp %r0,0x0
	0x8078: jrgt.d 0x2
	0x807a: ld %r0,0x0
	0x807c: ld %r0,0x1
	0x807e: ret

3.1.2.4 Type Conversion between 32-bit and Other Integers

The S1C17 is possible to convert a value of 32-bit integer type and other integer types by a few instructions. The number of S1C17 instructions that the GNU17 C compiler outputs may decrease after a value of 32-bit integer type is converted into other types.

C source code	S1C17 instructions

char convert_long_to_char(long a) {
  return (**char**)a;
}
	0x82d0: ld.b %r0,%r0
	0x82d2: ret

short convert_long_to_short(long a) {
  return (**short**)a;
}
	0x82d4: ret

long convert_char_to_long(char a) {
  return a;
}
	0x82d6: ld.b %r0,%r0
	0x82d8: cv.ls %r1,%r0
	0x82da: ret

long convert_short_to_long(short a) {
  return a;
}
	0x82dc: ld %r0,%r0
	0x82de: cv.ls %r1,%r0
	0x82e0: ret

32-bit integer could be converted into 16-bit integer, when it does not affect the operation result.

3.1.3 Pointer Type Operation

Usually the S1C17 address is a 24-bit integer value (24-bit pointer). This means that the GNU17 C compiler treats a pointer of the C language as a 24-bit integer value of the S1C17. However, when the small model is selected on the GNU17 IDE, the GNU17 C compiler treats pointers as 16-bit integer values (16-bit pointer). It is better to select the small mode when all objects and functions of the program are located within 16-bit space (64KB), because 16-bit pointer needs less S1C17 instructions and registers.

This section shows operations of 24-bit pointer when the large model or the middle model is selected on the GNU17 IDE.

3.1.3.1 Data Transmission between Memory and Registers

The S1C17 transmits a 24-bit value between memory and a register by an 'ld' instruction as same as a 16-bit value. Procedure that transmits a value from the source memory to the destination memory takes 2 S1C17 instructions because of an 'ld' instruction to read from the source and an 'ld' instruction to write to the destination.

C source code

```
short * copy_pointed_address(short ** a, short ** b) {
    short * c;

    c = *b;
    *a = c;

    return c;
}
```

S1C17 instructions

```
0x9162:   ld.a    -[%sp],%r4
0x9164:   ld.a    %r3,%r0
0x9166:   ld.a    %r0,[%r1]
0x9168:   ld.a    [%r3],%r0
0x916a:   ld.a    %r4,[%sp]+
0x916c:   ret
```

3.1.3.2 Data Operation

One S1C17 instruction is possible to process some operations (addition, subtraction and comparison) of the C language about 24-bit pointers.
When the index of the array is a variable, the GNU17 C compiler calculates the address of the array element by 24-bit addition.

C source code

```
char array_access(char * a, short b) {
    return a[b];
}
```

S1C17 instructions

```
0x836a:   ld.a    %r2,%r0
0x836c:   cv.as   %r0,%r1
0x836e:   add.a   %r2,%r0
0x8370:   ld      %r0,[%r2]
0x8372:   ret
```

The subtraction of the pointer is calculated as 32-bit integer type like the integral promotions, because the GNU17 C compiler is not able to store the result in 16-bit integer.

C source code

```
short pointer_subtraction(short * a, short * b) {
    return a - b;
}
```

S1C17 instructions

```
0x9180:   ld.a    -[%sp],%r4
0x9182:   ld.a    -[%sp],%r5
0x9184:   ld.a    -[%sp],%r6
0x9186:   ld.a    %r5,%r1
0x9188:   ld.a    %r0,%r0
0x918a:   cv.la   %r1,%r0
0x918c:   ld.a    %r2,%r5
0x918e:   cv.la   %r3,%r2
0x9190:   sub     %r0,%r2
0x9192:   sbc     %r1,%r3
0x9194:   call.d  0x196    <__ashrsi3>
0x9196:   ld      %r2,0x1
0x9198:   ld.a    %r6,[%sp]+
0x919a:   ld.a    %r5,[%sp]+
0x919c:   ld.a    %r4,[%sp]+
0x919e:   ret
```

If two pointers are near, the number of S1C17 instructions may be decreased by calculating the subtraction after converting pointers into 16-bit integers.

C source code

```
short pointer_subtraction(short * a, short * b) {
    return (short)a – (short)b;
}
```

S1C17 instructions

```
0x9180:   ld.a    -[%sp],%r4
0x9182:   sub     %r0,%r1
0x9184:   ld.a    %r4,[%sp]+
0x9186:   ret
```

3.1.3.3 Type Conversions between 24-bit Pointer and Other Integers

The S1C17 is possible to convert a value of 24-bit integer type and other integer types by a few instructions. The number of S1C17 instructions that the GNU17 C compiler outputs may decrease after a value of 24-bit integer type is converted into other types.

3.2 Floating Point Number Operations

There is no S1C17 instruction to operate floating point number directly. The GNU17 C compiler supports the floating point number by expressing 'float' as 4-byte data and 'double' as 8-byte data, and calling functions defined in the GNU17 emulation library to operate them.

The operation of floating point number on the GNU17 requires more memory, S1C17 registers and cycles. Replacing floating point number operations with fixed point number operations may decrease the size of the program.

3.3 Arithmetic Operations

3.3.1 Select Data Type

Please select 16-bit integer type to execute arithmetic operations, if possible.

- A 32-bit integer occupies 4 bytes on memory, but a 16-bit integer occupies 2 bytes.
- The number of S1C17 instructions may be increased by a 32-bit integer, because most operations of 32-bit integer are executed as a function call.
- The size of the stack frame may be increased by a 32-bit integer, because a 32-bit integer uses two registers while a 16-bit integer uses only one register.
- A 32-bit integer may increase the number of S1C17 instructions further to transmit data between the memory and registers.

The amount of used memory for data may decrease, when 8-bit integers are used to store data on the memory.

3.3.2 Simplify Operations

The GNU17 C compiler may replace the multiplication between a variable and a constant by other operations.

C source code

```
short multiple_249(short a) {
    return a * 249;
}
```

S1C17 instructions

```
0x8078:   ld     %r2,%r0
0x807a:   sl     %r0,0x4
0x807c:   sl     %r0,0x1
0x807e:   sub    %r0,%r2
0x8080:   sl     %r0,0x3
0x8082:   add    %r0,%r2
0x8084:   ret
```

For optimization, the GNU17 C compiler executes these replacements of multiplication, division and modulus. In some cases, these optimizations outputs more number of S1C17 instructions than calling a function defined in the GNU17 emulation library.

C source code

```
short multiple_249(short a) {
    extern short __mulhi3(short, short);
    return __mulhi3(a, 249);
}
```

S1C17 instructions

```
0x8078:   ext    0x1
0x807a:   ld     %r1,0x79
0x807c:   ext    0x2
0x807e:   call   0x2f0
0x8080:   ret
```

If the calculation error is acceptable, the number of S1C17 instructions may decrease.

C source code | S1C17 instructions

short multiple_249(short a) {
  return a * 256;
}
	0x8078: sl %r0,0x8
	0x807a: ret

The optimization of the GNU17 C compiler is not so strong.

C source code	S1C17 instructions

short not_optimaized_procedure(short a, **short** b) {
  short c;
 	
  if (a >= 15) {
  c = (a - 15) << 8;
  } **else** {
  c = 0x7000 + (a << 8);
  }
 	
  return c;
}
	0x8044: cmp %r0,0xe
	0x8046: jrle.d 0x4
	0x8048: ld %r2,0x71
	0x804a: add %r0,%r2
	0x804c: jpr.d 0x4
	0x804e: sl %r0,0x8
	0x8050: sl %r0,0x8
	0x8052: ext 0xe0
	0x8054: add %r0,0x0
	0x8056: ret

The number of S1C17 instructions may be decreased by finding out common calculation from the procedure.

C source code	S1C17 instructions

short not_optimized_procedure(short a, **short** b) {
  if (a >= 15) {
  a -= 15;
  } **else** {
  a += 0x70;
  }
  return a << 8;
}
	0x8044: cmp %r0,0xe
	0x8046: jrle.d 0x3
	0x8048: ld %r2,0x71
	0x804a: jpr.d 0x2
	0x804c: add %r0,%r2
	0x804e: add %r0,0x70
	0x8050: sl %r0,0x8
	0x8052: ret

3.3.3 Embed Calculated Data

If the value that is a part of the complex calculation is computable when the GNU17 C compiler compiles the C source code, the number of S1C17 instructions may decrease.

C source code	S1C17 instructions

short multiple_and_add(short a, **short** b) {
  return a * a * a + b;
}
	0x80c4: ld.a -[%sp],%r4
	0x80c6: ld.a -[%sp],%r5
	0x80c8: ld %r4,%r0
	0x80ca: ld %r5,%r1
	0x80cc: ext 0x2
	0x80ce: call.d 0x1ba <__mulhi3>
	0x80d0: ld %r1,%r0
	0x80d2: ext 0x2
	0x80d4: call.d 0x1b7 <__mulhi3>
	0x80d6: ld %r1,%r4
	0x80d8: add %r0,%r5
	0x80da: ld.a %r5,[%sp]+
	0x80dc: ld.a %r4,[%sp]+
	0x80de: ret

The program could be modified to execute the calculation by referring to the value that was calculated and embedded in the table of constants.

C source code	S1C17 instructions

short multiple_and_add(short a, **short** b) {
  static const char calculated_values[] = {
  0 * 0 * 0, 1 * 1 * 1,
  2 * 2 * 2, 3 * 3 * 3,
  4 * 4 * 4, 5 * 5 * 5,
  };
 	
  return calculated_values[a] + b;
}
	0x80c4: ext 0x129
	0x80c6: ld %r2,0x68
	0x80c8: add %r0,%r2
	0x80ca: ld.b %r0,[%r0]
	0x80cc: add %r0,%r1
	0x80ce: ret

However, the number of instructions should decrease more than the size of embedded data.

3.4 Control Statements

3.4.1 Select Data Type

Please select 16-bit integer type to execute comparison, if possible. The S1C17 is not possible to compare 32-bit integer type by one instruction.

C source code

```
short sum_of_array(long a, int * b) {
    long i;
    short r;

    r = 0;
    for (i = 0; i < a; i++) {
        r += *b++;
    }

    return r;
}
```

S1C17 instructions

```
0x812e:   ld.a    -[%sp],%r4
0x8130:   ld.a    -[%sp],%r5
0x8132:   ld.a    -[%sp],%r6
0x8134:   ld      %r6,0x0
0x8136:   sub     %r4,%r4
0x8138:   sub     %r5,%r5
0x813a:   cmp     %r1,%r6
0x813c:   jrgt.d  0x4
0x813e:   ld      %r3,%r2
0x8140:   jrne.d  0xc
0x8142:   cmp     %r0,%r4
0x8144:   jrule   0xa
0x8146:   ld      %r2,[%r3]+
0x8148:   add     %r4,0x1
0x814a:   adc     %r5,0x0
0x814c:   cmp     %r1,%r5
0x814e:   jrgt.d  0x7b
0x8150:   add     %r6,%r2
0x8152:   cmp     %r1,%r5
0x8154:   jrne.d  0x2
0x8156:   cmp     %r0,%r4
0x8158:   jrugt   0x76
0x815a:   ld      %r0,%r6
0x815c:   ld.a    %r6,[%sp]+
0x815e:   ld.a    %r5,[%sp]+
0x8160:   ld.a    %r4,[%sp]+
0x8162:   ret
```

The number of S1C17 instruction may decrease when the comparison between 32-bit integers is replaced by 16-bit integers.

C source code

```
short sum_of_array(long a, int * b) {
    short i;
    short r;

    r = 0;
    for (i = 0; i < (short)a; i++) {
        r += *b++;
    }

    return r;
}
```

S1C17 instructions

```
0x812e:    ld.a    -[%sp],%r4
0x8130:    ld.a    -[%sp],%r5
0x8132:    ld      %r5,%r1
0x8134:    ld      %r4,%r0
0x8136:    ld      %r0,%r2
0x8138:    ld      %r1,0x0
0x813a:    cmp     %r1,%r4
0x813c:    jrge.d  0x6
0x813e:    ld      %r3,%r1
0x8140:    ld      %r2,[%r0]+
0x8142:    add     %r3,0x1
0x8144:    cmp     %r3,%r4
0x8146:    jrlt.d  0x7c
0x8148:    add     %r1,%r2
0x814a:    ld      %r0,%r1
0x814c:    ld.a    %r5,[%sp]+
0x814e:    ld.a    %r4,[%sp]+
0x8150:    ret
```

3.4.2 Simplify Comparisons

The number of S1C17 instructions and registers necessary for the comparison of variables is more than the comparison between a constant and a variable.
For example, the number of instructions could be decreased by using 0 to the continuation conditional expression.

C source code

```
short sum_of_array(short a, int * b) {
    short r;

    for (r = 0; a > 0; a--) {
        r += *b++;
    }

    return r;
}
```

S1C17 instructions

```
0x812e:   ld      %r3,0x0
0x8130:   cmp     %r0,%r3
0x8132:   jrle    0x6
0x8134:   ld      %r2,[%r1]+
0x8136:   add     %r3,%r2
0x8138:   ld      %r2,0x7f
0x813a:   add     %r0,%r2
0x813c:   cmp     %r0,0x0
0x813e:   jrgt    0x7a
0x8140:   ld      %r0,%r3
0x8142:   ret
```

3.4.3 Change Condition

If the condition is always true, it will be able to change condition to be executed by less instruction. For example, if the counter is not 0, the number of S1C17 instructions could be decreased by omitting the first comparison with 0.

C source code

```
short sum_of_array(short a, int * b) {
    short s;

    s = 0;
    do {
        s += *b++;
    } while (--a > 0);

    return s;
}
```

S1C17 instructions

```
0x8262:   ld      %r3,0x0
0x8264:   ld      %r2,[%r1]+
0x8266:   add     %r3,%r2
0x8268:   ld      %r2,0x7f
0x826a:   add     %r0,%r2
0x826c:   cmp     %r0,0x0
0x826e:   jrgt    0x7a
0x8270:   ld      %r0,%r3
0x8272:   ret
```

In following example, the number of S1C17 instructions has decreased by not using the counter under the continuation condition.

C source code

```
short sum_of_array (int * b) {
    short a;
    short s;

    s = 0;
    do {
        a += *b++;
        s += a;
    } while (a != 0);

    return s;
}
```

S1C17 instructions

```
0x8274:   ld      %r3,0x0
0x8276:   ld      %r2,[%r0]+
0x8278:   cmp     %r2,0x0
0x827a:   jrne.d  0x7d
0x827c:   add     %r3,%r2
0x827e:   ld      %r0,%r3
0x8280:   ret
```

3.4.4 Transform Loops

The number of S1C17 instructions could be decreased by unifying two or more similar loops.

C source code

```
short multi_loops(short * a, short * b) {
    short i;
    short r;

    r = 0;
    for (i = 10; i > 0; i--) {
        r += sub_function(*a++);
    }
    for (i = 10; i > 0; i--) {
        r += sub_function(*b++);
    }

    return r;
}
```

S1C17 instructions

```
0x8284:   ld.a    -[%sp],%r4
0x8286:   ld.a    -[%sp],%r5
0x8288:   ld.a    -[%sp],%r6
0x828a:   ld.a    -[%sp],%r7
0x828c:   ld      %r5,%r0
0x828e:   ld      %r7,%r1
0x8290:   ld      %r6,0x0
0x8292:   ld      %r4,0xa
0x8294:   ld      %r0,[%r5]+
0x8296:   call    0x3f5
0x8298:   ld      %r2,0x7f
0x829a:   add     %r4,%r2
0x829c:   cmp     %r4,0x0
0x829e:   jrgt.d  0x7a
0x82a0:   add     %r6,%r0
```

	0x82a2: ld %r4,0xa 0x82a4: ld %r0,[%r7]+ 0x82a6: call 0x3ed 0x82a8: ld %r2,0x7f 0x82aa: add %r4,%r2 0x82ac: cmp %r4,0x0 0x82ae: jrgt.d 0x7a 0x82b0: add %r6,%r0 0x82b2: ld %r0,%r6 0x82b4: ld.a %r7,[%sp]+ 0x82b6: ld.a %r6,[%sp]+ 0x82b8: ld.a %r5,[%sp]+ 0x82ba: ld.a %r4,[%sp]+ 0x82bc: ret

For example, the number of S1C17 instructions may decrease if two ‘for’ loops are unified in one ‘for’ loop.

C source code	S1C17 instructions

short multi_loops(short * a, short * b) {
  short i;
  short r;
 	
  r = 0;
  for (i = 10; i > 0; i--) {
  r += sub_function(*a++);
  r += sub_function(*b++);
  }
 	
  return r;
}
	0x82be: ld.a -[%sp],%r4
	0x82c0: ld.a -[%sp],%r5
	0x82c2: ld.a -[%sp],%r6
	0x82c4: ld.a -[%sp],%r7
	0x82c6: ld %r7,%r0
	0x82c8: ld %r6,%r1
	0x82ca: ld %r4,0x0
	0x82cc: ld %r5,0xa
	0x82ce: ld %r0,[%r7]+
	0x82d0: call 0x3d8
	0x82d2: add %r4,%r0
	0x82d4: ld %r0,[%r6]+
	0x82d6: call 0x3d5
	0x82d8: ld %r2,0x7f
	0x82da: add %r5,%r2
	0x82dc: cmp %r5,0x0
	0x82de: jrgt.d 0x77
	0x82e0: add %r4,%r0
	0x82e2: ld %r0,%r4
	0x82e4: ld.a %r7,[%sp]+
	0x82e6: ld.a %r6,[%sp]+
	0x82e8: ld.a %r5,[%sp]+
	0x82ea: ld.a %r4,[%sp]+
	0x82ec: ret

3.5 Accessing I/O Registers

3.5.1 Select Data Type

Please access I/O registers by the bit width specified by the hardware specification. However, most I/O registers of embedded into the S1C17 are accessible both the 8-bit width and 16-bit width.

Even if the bit field is define as a 16-bit integer type, fields with 8 or less bit width will be accessed as an 8-bit integer type by the GNU17 C compiler.

C source code	S1C17 instructions

typedef union {
  unsigned short word;
  unsigned char byte[2];
  struct {
  unsigned short a: 1;
  unsigned short b: 7;
  unsigned short c: 3;
  unsigned short d: 5;
  } bit;
} short_fields;
 	
#define device16 (*(volatile short_fields *)0x4000)	

unsigned short get_device_a(void) {
  return device16.bit.a;
}
	0x8fc0: ext 0x80
	0x8fc2: ld.b %r0,[0x0]
	0x8fc4: and %r0,0x1

	0x8fc6: ret

When the I/O register is arranged in consecutive 4 bytes, this I/O register could be accessed as a 32-bit integer. However, the number of S1C17 instructions does not decrease because the S1C17 executes the reading and writing of 32-bit integer as two 'ld' instructions.

C source code	S1C17 instructions

typedef union {
  unsigned long dword;
  void * pointer;
  struct {
  short_fields low;
  short_fields high;
  } word;
} long_fields;
 	
#define device32 (*(volatile long_fields *)0x4000)	

void set_long_device(unsigned long value) {
  device32.dword = value;
}
	0x8fc8: ext 0x80
	0x8fca: ld [0x0],%r0
	0x8fcc: ext 0x80
	0x8fce: ld [0x2],%r1
	0x8fd0: ret

void set_low_high(unsigned int l, unsigned int h) {
  device32.word.low.word = l;
  device32.word.high.word = h;
}
	0x8fd2: ext 0x80
	0x8fd4: ld [0x0],%r0
	0x8fd6: ext 0x80
	0x8fd8: ld [0x2],%r1
	0x8fda: ret

When the small model is not selected and a pointer is 24-bit integer value, the I/O register arranged in consecutive 4 bytes could be accessed as a 24-bit value. In this case, these I/O registers should be accessible with 32-bit width.

C source code	S1C17 instructions

void set_24bit_pointer(void * pointer) {
  device32.pointer = pointer;
}
	0x91be: ext 0x80
	0x91c0: ld.a [0x0],%r0
	0x91c2: ret

Please note that S1C17 instructions generated by the GNU17 C compiler from this C source code does not act as expected in case of the small mode.

3.5.2 Set a Value to the I/O Register

In order to change some bits of the value held on the memory, the GNU17 C compiler generates the following steps:

- Read a value in memory to a S1C17 register.
- Modify some bits of the read value.
- Write the modified value to the memory.

When the I/O register is defined as bit fields, the GNU17 C compiler generates similar instructions for writing the I/O register.

C source code	S1C17 instructions

void initialize_device_field(void) {
  device16.bit.a = 0;
  device16.bit.b = 1;
  device16.bit.c = 7;
  device16.bit.d = 2;
}
	0x91aa: ld.a -[%sp],%r4
	0x91ac: ext 0x80
	0x91ae: ld.a %r3,0x0
	0x91b0: ld.b %r2,[%r3]
	0x91b2: and %r2,0x7e
	0x91b4: ld.b [%r3],%r2
	0x91b6: ld.b %r2,[%r3]
	0x91b8: and %r2,0x1
	0x91ba: or %r2,0x2
	0x91bc: ld.b [%r3],%r2
	0x91be: ext 0x80
	0x91c0: ld.b %r2,[0x1]

	0x91c2: or %r2,0x7 0x91c4: ext 0x80 0x91c6: ld.b [0x1],%r2 0x91c8: ext 0x80 0x91ca: ld.b %r2,[0x1] 0x91cc: and %r2,0x7 0x91ce: or %r2,0x10 0x91d0: ext 0x80 0x91d2: ld.b [0x1],%r2 0x91d4: ld.a %r4,[%sp]+ 0x91d6: ret

However, the GNU17 C compiler generates less S1C17 instructions, if all bits could be overwritten at the same time.

C source code	S1C17 instructions

void initialize_device_field(void) {
  short_fields value;
 	
  value.bit.a = 0;
  value.bit.b = 1;
  value.bit.c = 7;
  value.bit.d = 2;
 	
  device16.word = value.word;
}
	0x91aa: ext 0x2e
	0x91ac: ld %r2,0x2
	0x91ae: ext 0x80
	0x91b0: ld [0x0],%r2
	0x91b2: ret

In the following cases, all bits of the I/O register will be able to be written at the same time:

- There is only one valid bit field in the 16-bit (or 8-bit) I/O register.
- All bits of the I/O register are updated for initialization etc.
- The value of bits except the bit field to modify is known beforehand.

3.5.3 Get a Value from the I/O Register

When some bit fields are contained in a 16-bit (or 8-bit) I/O register, the GNU17 C compiler generates two or more 'ld' instructions to read each bit field.

C source code	S1C17 instructions

short check_device_flag(void) {
  if ((device16.bit.a == 1) && (device16.bit.b == 3)) {
  return 1;
  }
  return 0;
}
	0x91e2: ld.a -[%sp],%r4
	0x91e4: ext 0x80
	0x91e6: ld.a %r3,0x0
	0x91e8: ld.b %r2,[%r3]
	0x91ea: ld %r0,%r2
	0x91ec: and %r0,0x1
	0x91ee: cmp %r0,0x1
	0x91f0: jrne 0x5
	0x91f2: ld.b %r2,[%r3]
	0x91f4: ld.ub %r2,%r2
	0x91f6: sr %r2,0x1
	0x91f8: cmp %r2,0x3
	0x91fa: jreq 0x1
	0x91fc: ld %r0,0x0
	0x91fe: ld.a %r4,[%sp]+
	0x9200: ret

When some bit fields in the same I/O register are referred at the same time, the GNU17 C compiler may generate less S1C17 instructions by operating some bits together.

C source code	S1C17 instructions

short check_device_flag(void) {
  unsigned char flag;
 	
  flag = device16.byte[0];
  if (flag== ((3 << 1) | (1 << 0))) {
  return 1;
  }
	0x91e2: ext 0x80
	0x91e4: ld.b %r2,[0x0]
	0x91e6: ld.ub %r2,%r2
	0x91e8: cmp %r2,0x7
	0x91ea: jreq.d 0x2
	0x91ec: ld %r0,0x1
	0x91ee: ld %r0,0x0

`    return 0;` `}`	`0x91f0:  ret`

4. Function Design

This section shows how to select arguments and returns of the C function to decrease the number of S1C17 instructions.
For more details, please refer the GNU17 manual, especially “6.4.3 Method of Using Registers” and “6.4.4 Function Call”.

4.1 Arguments

4.1.1 Number of Arguments

The calling convention of the GNU17 C compiler is that the caller passes the first 4 arguments to the callee function in the S1C17 registers R0-R3 and other arguments are passed on the stack area. Each 8-bit, 16-bit and 24-bit (pointer of the regular or large model) argument uses one S1C17 register. This means that the maximum number of arguments unless using the stack area is 4.

C source code	S1C17 instructions

int argument_4_callee(**int** a, **int** b, **int** c, **int** d) {
  return a + b + c + d;
}
	0x8010: add %r0,%r1
	0x8012: add %r0,%r2
	0x8014: add %r0,%r3
	0x8016: ret

int argument_4_caller(**int** a[]) {
  return argument_4_callee(a[0], a[1], a[2], a[3]);
}
	0x8018: ld %r3,%r0
	0x801a: ld %r0,[%r0]
	0x801c: ext 0x2
	0x801e: ld %r1,[%r3]
	0x8020: ext 0x4
	0x8022: ld %r2,[%r3]
	0x8024: ext 0x6
	0x8026: ld %r3,[%r3]
	0x8028: call 0x3f3
	0x802a: ret

Because 5th argument is passed on the stack area, the caller and callee functions need larger stack frames and more S1C17 instructions to retrieve stacked values.

C source code	S1C17 instructions

int argument_5_callee(**int** a, **int** b, **int** c, **int** d, **int** e) {
  return a + b + c + d + e;
}
	0x802c: add %r0,%r1
	0x802e: add %r0,%r2
	0x8030: add %r0,%r3
	0x8032: ld %r2,[%sp+0x4]
	0x8034: add %r0,%r2
	0x8036: ret

int argument_5_caller(**int** a[]) {
  return argument_5_callee(a[0], a[1], a[2], a[3], a[4]);
}
	0x8038: sub.a %sp,0x4
	0x803a: ld %r3,%r0
	0x803c: ext 0x8
	0x803e: ld %r2,[%r0]
	0x8040: ld [%sp+0x0],%r2
	0x8042: ld %r0,[%r0]
	0x8044: ext 0x2
	0x8046: ld %r1,[%r3]
	0x8048: ext 0x4
	0x804a: ld %r2,[%r3]
	0x804c: ext 0x6
	0x804e: ld %r3,[%r3]
	0x8050: call 0x3ed
	0x8052: add.a %sp,0x4
	0x8054: ret

4.1.2 Order of Arguments

The GNU17 C compiler assigns the R0 register as the 1st argument and the R1 register as the 2nd argument and so on. The R0 and R1 registers are also used to store returned values.
When a caller function passes arguments to callee functions, more S1C17 instructions are necessary to replace the S1C17 register of disordered arguments.

C source code	S1C17 instructions

int argument_disordering(int a, **int** b, **int** c, **int** d) {
  return argument_4_callee(d, c, b, a);
}
	0x8056: ld.a -[%sp],%r4
	0x8058: ld.a -[%sp],%r5
	0x805a: ld %r5,%r0
	0x805c: ld %r4,%r1
	0x805e: ld %r0,%r3
	0x8060: ld %r1,%r2
	0x8062: ld %r2,%r4
	0x8064: call.d 0x3d5
	0x8066: ld %r3,%r5
	0x8068: ld.a %r5,[%sp]+
	0x806a: ld.a %r4,[%sp]+
	0x806c: ret

If the order of arguments is same between the caller and the callee function, the GNU17 C compiler generates less S1C17 instructions.

C source code	S1C17 instructions

int argument_ordering(int a, **int** b, **int** c, **int** d) {
  return argument_4_callee(a, b, c, d);
}
	0x806e: call 0x3d0
	0x8070: ret

4.1.3 Parameter Set

At each call of the function, the GNU17 C compiler generates S1C17 instructions which set the value to arguments. Even if the number of arguments and the order of arguments are common between 2 or more functions, S1C17 registers for arguments are set again, because the callee function could change the value of R0-R3 registers according to the calling convention.

C source code	S1C17 instructions

int argument_4_callee(int a, **int** b, **int** c, **int** d) {
  return a + b + c + d;
}
	0x8010: add %r0,%r1
	0x8012: add %r0,%r2
	0x8014: add %r0,%r3
	0x8016: ret

int argument_with_parameters(int a, **int** b, **int** c) {
  int sum;
 	
  sum = argument_4_callee(a, b, c, 0);
  return argument_4_callee(a, b, c, sum);
}
	0x8074: ld.a -[%sp],%r4
	0x8076: ld.a -[%sp],%r5
	0x8078: ld.a -[%sp],%r6
	0x807a: ld %r4,%r0
	0x807c: ld %r5,%r1
	0x807e: ld %r3,0x0
	0x8080: call.d 0x3c7
	0x8082: ld %r6,%r2
	0x8084: ld %r3,%r0
	0x8086: ld %r0,%r4
	0x8088: ld %r1,%r5
	0x808a: call.d 0x3c2
	0x808c: ld %r2,%r6
	0x808e: ld.a %r6,[%sp]+
	0x8090: ld.a %r5,[%sp]+
	0x8092: ld.a %r4,[%sp]+
	0x8094: ret

These load of function call become heavy along with the number of arguments.

When a caller generates a parameter set and pass its address to callee functions, the number of S1C17 instructions may decreases. It takes more S1C17 instructions to initialize a parameter set and retrieve parameters from it, but the number of instructions to call functions becomes less.

C source code	S1C17 instructions

int argument_set_callee(int a, **int** set[]) {
  return a + set[0] + set[1] + set[2];
}
	0x8096: ld %r2,[%r1]
	0x8098: add %r0,%r2
	0x809a: ext 0x2
	0x809c: ld %r2,[%r1]
	0x809e: add %r0,%r2
	0x80a0: ext 0x4
	0x80a2: ld %r2,[%r1]

	0x80a4: add %r0,%r2 0x80a6: ret

int **argument_with_set**(int a, int b, int c) {
  int set[3] = {
  a, b, c,
  };
  int sum;
 	
  sum = argument_set_callee(0, set);
  return argument_set_callee(sum, set);
}
	0x80a8: sub.a %sp,0x8
	0x80aa: ld [%sp+0x0],%r0
	0x80ac: ld [%sp+0x2],%r1
	0x80ae: ld [%sp+0x4],%r2
	0x80b0: ld %r0,0x0
	0x80b2: ld.a %r1,%sp
	0x80b4: call 0x3f0
	0x80b6: ld.a %r1,%sp
	0x80b8: call 0x3ee
	0x80ba: add.a %sp,0x8
	0x80bc: ret

4.1.4 32-bit and Other Big Values

When one of the parameter is a 32-bit value, the maximum number of arguments unless using the stack area is less than 4, because a pair of S1C17 registers is used to store a 32-bit (long or float) argument.

- If the 1st argument is a 32-bit value, the R0 register is lower 16 bits and the R1 register is higher 16 bits.
- If the 2nd (or 3rd) argument is a 32-bit value, the R2 register is lower 16 bits and the R3 register is higher 16 bits.

C source code	S1C17 instructions

long **argument_value32_callee**(long a, long b) {
  return (long)((short)a + (short)b);
}
	0x80be: add %r2,%r0
	0x80c0: ld %r0,%r2
	0x80c2: cv.ls %r1,%r0
	0x80c4: ret

long **argument_value32_caller**(void) {
  return argument_value32_callee(1L, 2L);
}
	0x80c6: ld %r0,0x1
	0x80c8: sub %r1,%r1
	0x80ca: ld %r2,0x2
	0x80cc: sub %r3,%r3
	0x80ce: call 0x3f7
	0x80d0: ret

It is better to pass the 16-bit value or the address of the 32-bit value than pass the value itself.

C source code	S1C17 instructions

long **argument_value16_callee**(short a, short b) {
  return a + b;
}
	0x80d2: add %r0,%r1
	0x80d4: ld %r0,%r0
	0x80d6: cv.ls %r1,%r0
	0x80d8: ret

long **argument_value16_caller**(void) {
  return argument_value16_callee(1, 2);
}
	0x80da: ld %r0,0x1
	0x80dc: call.d 0x3fa
	0x80de: ld %r1,0x2
	0x80e0: ret

If the size of the structure is 64-bit or smaller and its members are able to store in 4 registers, the GNU17 C compiler assign the R0-R3 registers to pass the structure to the callee function by value.

C source code	S1C17 instructions

struct member4 {
  int r0;
  int r1;
  int r2;
  int r3;
};	

int **member4_value_caller**(void) {
  struct member4 m4 = {
  0, 1, 2, 3,
  };
  int r;
 	
  r = value_callee1(m4);
  return r + value_callee2(m4);
}
	0x80fa: ld.a -[%sp],%r4
	0x80fc: sub.a %sp,0x8
	0x80fe: ld.a %r0,%sp
	0x8100: ext 0x103
	0x8102: ld %r1,0x16
	0x8104: call.d 0x40 <memcpy>
	0x8106: ld %r2,0x8
	0x8108: ld %r0,[%sp+0x0]
	0x810a: ld %r1,[%sp+0x2]

	0x810c: ld %r2,[%sp+0x4] 0x810e: ld %r3,[%sp+0x6] 0x8110: call 0x3e8 0x8112: ld %r4,%r0 0x8114: ld %r0,[%sp+0x0] 0x8116: ld %r1,[%sp+0x2] 0x8118: ld %r2,[%sp+0x4] 0x811a: ld %r3,[%sp+0x6] 0x811c: call 0x3e8 0x811e: add %r4,%r0 0x8120: ld %r0,%r4 0x8122: add.a %sp,0x8 0x8124: ld.a %r4,[%sp]+ 0x8126: ret

Generally, the number of S1C17 instructions may decrease when the structure is passed to the callee by reference.

C source code	S1C17 instructions

int member4_reference_caller(void) {
  static const struct member4 m4 = {
  0, 1, 2, 3,
  };
  int r;
 	
  r = reference_callee1(&m4);
  return r + reference_callee2(&m4);
}
	0x8158: ld.a -[%sp],%r4
	0x815a: ext 0x103
	0x815c: ld %r0,0x1e
	0x815e: call 0x3e4
	0x8160: ld %r4,%r0
	0x8162: ext 0x103
	0x8164: ld %r0,0x1e
	0x8166: call 0x3ec
	0x8168: add %r4,%r0
	0x816a: ld %r0,%r4
	0x816c: ld.a %r4,[%sp]+
	0x816e: ret

Bigger structures than 64-bit should be passed by reference, because the GNU17 C compiler always stores them in the stack area.
And arguments of 64-bit (long long or double) type are always stored in the stack area also. It is better to call by reference than call by value for them, too.

4.2 Returns

According to the calling convention, the GNU17 C compiler generates S1C17 instructions that the callee returns a value in the R0 register.

C source code	S1C17 instructions

int return_callee(void) {
  return 0;
}
	0x8176: ld %r0,0x0
	0x8178: ret

int return_caller(int a) {
  return a + return_callee();
}
	0x817a: call 0x3fd
	0x817c: add %r0,0x1
	0x817e: ret

4.2.1 Big Values

Two S1C17 registers (the R0 register and the R1 register) are used to return a 32-bit value such as ‘long’. And bigger values are always returned in the stack area.

C source code	S1C17 instructions

struct big {
  short a;
  short b;
  short c;
  short d;
};	

struct big return_big_callee(void) {
  struct big r;
 	
  r.a = 1;
  r.b = 2;
	0x8180: ld.a -[%sp],%r4
	0x8182: sub.a %sp,0x8
	0x8184: ld %r4,%r0
	0x8186: ld %r2,0x1
	0x8188: ld [%sp+0x0],%r2
	0x818a: ld %r2,0x2

```
    return r;
}
```

```
0x818c:   ld      [%sp+0x2],%r2
0x818e:   ld.a    %r1,%sp
0x8190:   call.d  0x1d     <memcpy>
0x8192:   ld      %r2,0x8
0x8194:   ld      %r0,%r4
0x8196:   add.a   %sp,0x8
0x8198:   ld.a    %r4,[%sp]+
0x819a:   ret
```

```
long return_long_callee(void) {
    return 0L;
}
```

```
0x819c:   sub     %r0,%r0
0x819e:   sub     %r1,%r1
0x81a0:   ret
```

```
int return_big_caller(void) {
    struct big r;

    r = return_big_callee();
    return r.a + r.b + (short)return_long_callee();
}
```

```
0x81a2:   ld.a    -[%sp],%r4
0x81a4:   sub.a   %sp,0x8
0x81a6:   ld.a    %r0,%sp
0x81a8:   call    0x3eb
0x81aa:   ld      %r4,[%sp+0x0]
0x81ac:   ld      %r2,[%sp+0x2]
0x81ae:   call.d  0x3f6
0x81b0:   add     %r4,%r2
0x81b2:   add     %r4,%r0
0x81b4:   ld      %r0,%r4
0x81b6:   add.a   %sp,0x8
0x81b8:   ld.a    %r4,[%sp]+
0x81ba:   ret
```

Because the big object is always returned by reference, the callee function should take the address to store the return value as an argument.

C source code	S1C17 instructions

`struct big * return_pointer_callee(struct big * r) {`
`    r->a = 1;`
`    r->b = 2;`
 	
`    return r;`
`}`
	`0x81bc:  ld      %r3,0x1`
	`0x81be:  ld      [%r0],%r3`
	`0x81c0:  ld      %r3,0x2`
	`0x81c2:  ext     0x2`
	`0x81c4:  ld      [%r0],%r3`
	`0x81c6:  ret`

`int return_pointer_caller(void) {`
`    struct big r;`
 	
`    (void)return_pointer_callee(&r);`
`    return r.a + r.b + (short)return_long_callee();`
`}`
	`0x81c8:  ld.a    -[%sp],%r4`
	`0x81ca:  sub.a   %sp,0x8`
	`0x81cc:  ld.a    %r0,%sp`
	`0x81ce:  call    0x3f6`
	`0x81d0:  ld      %r4,[%sp+0x0]`
	`0x81d2:  ld      %r2,[%sp+0x2]`
	`0x81d4:  call.d  0x3e3`
	`0x81d6:  add     %r4,%r2`
	`0x81d8:  add     %r4,%r0`
	`0x81da:  ld      %r0,%r4`
	`0x81dc:  add.a   %sp,0x8`
	`0x81de:  ld.a    %r4,[%sp]+`
	`0x81e0:  ret`

4.2.2 Two Values

Because the function returns only one object and big objects are always returned in the stack area, the call by reference is used to return 2 or more parameters.

C source code	S1C17 instructions

`int return_error_callee(int * out) {`
`    *out = 0;`
`    return 0;`
`}`
	`0x81e2:  ld      %r2,%r0`
	`0x81e4:  ld      %r0,0x0`
	`0x81e6:  ld      [%r2],%r0`
	`0x81e8:  ret`

`int return_error_caller(void) {`
`    int r;`
`    int error;`
 	
`    error = return_error_callee(&r);`
 	
`    return error;`
`}`
	`0x81ea:  sub.a   %sp,0x4`
	`0x81ec:  ld.a    %r0,%sp`
	`0x81ee:  call    0x3f9`
	`0x81f0:  add.a   %sp,0x4`
	`0x81f2:  ret`

If the size of the structure is 32-bit or smaller and the number of its members is 2, the GNU17 C compiler could

assign the R0-R1 registers to return the structure by value.

C source code	S1C17 instructions

union value32 {
  long v;
  struct value32_m {
  short e;
  short r;
  } m;
};	

union value32 return_value32_callee(void) {
  union value32 v;
 	
  v.m.e = 0;
  v.m.r = 0;
 	
  return v;
}
	0x81f4: sub %r0,%r0
	0x81f6: sub %r1,%r1
	0x81f8: ld %r1,%r0
	0x81fa: ret

int return_value32_caller(void) {
  union value32 v;
 	
  v = return_value32_callee();
 	
  return v.m.e;
}
	0x81fc: call 0x3fb
	0x81fe: ret

If the bit-width of each value is less than 16, 2 parameters could be returned by a 16-bit value.

C source code	S1C17 instructions

union value16 {
  short v;
  struct value16_m {
  char e;
  char r;
  } m;
};	

union value16 return_value16_callee(void) {
  union value16 v;
 	
  v.m.e = 0;
  v.m.r = 0;
 	
  return v;
}
	0x8200: ld %r0,0x0
	0x8202: ret

int return_value16_caller(void) {
  union value16 v;
 	
  v = return_value16_callee();
 	
  return v.m.e;
}
	0x8204: call 0x3fd
	0x8206: ld.b %r0,%r0
	0x8208: ret

4.2.3 Parameter of the Next Function

The R0 register is used to store the returned value and the 1st argument. When the variable is updated by the returned value of a function and passed as the 1st argument to the next function, the GNU17 C compiler optimizes S1C17 instructions and may assign the R0 register to the variable.

C source code	S1C17 instructions

int return_next_caller(int a) {
  int r = a;
 	
  r = return_next_callee1(r);
  r = return_next_callee2(r);
  r = return_next_callee3(r);
 	
  return r;
}
	0x8210: call 0x3fc
	0x8212: call 0x3fc
	0x8214: call 0x3fc
	0x8216: ret

4.3 Dividing into More Functions

When various processing is implemented in one function, the number of variables in the function increases. The stack frame is created at the function entry, not enhanced during the function, removed at the end of the function. Therefore, when the number of variables increases, the size of the stack frame grows.

C source code	S1C17 instructions

int function_too_big_stack(void) {
  unsigned char a[16];
  unsigned char b[16];
  int r;
 	
  r = sub_function_a(a);
  r += sub_function_b(b);
 	
  return r;
}
	0x8178: ld.a -[%sp],%r4
	0x817a: sub.a %sp,0x20
	0x817c: ld.a %r0,%sp
	0x817e: call 0x3f8
	0x8180: ld %r4,%r0
	0x8182: ld.a %r0,%sp
	0x8184: call.d 0x3f7
	0x8186: add %r0,0x10
	0x8188: add %r4,%r0
	0x818a: ld %r0,%r4
	0x818c: add.a %sp,0x20
	0x818e: ld.a %r4,[%sp]+
	0x8190: ret

The amount of stack area required by the program may be decreased by dividing a big function into two or more functions to which size of the stack frame is almost equal.

C source code	S1C17 instructions

int part_function_a(void) {
  unsigned char a[16];
 	
  return sub_function_a(a);
}
	0x8192: sub.a %sp,0x10
	0x8194: ld.a %r0,%sp
	0x8196: call 0x3ec
	0x8198: add.a %sp,0x10
	0x819a: ret

int part_function_b(void) {
  unsigned char b[16];
 	
  return sub_function_b(b);
}
	0x819c: sub.a %sp,0x10
	0x819e: ld.a %r0,%sp
	0x81a0: call 0x3e9
	0x81a2: add.a %sp,0x10
	0x81a4: ret

int function_divided_stack(void) {
  int r;
 	
  r = part_function_a();
  r += part_function_b();
 	
  return r;
}
	0x81a6: ld.a -[%sp],%r4
	0x81a8: call 0x3f4
	0x81aa: call.d 0x3f8
	0x81ac: ld %r4,%r0
	0x81ae: add %r4,%r0
	0x81b0: ld %r0,%r4
	0x81b2: ld.a %r4,[%sp]+
	0x81b4: ret

5. Data Structure Design

This section shows how to define data structure and data itself to decrease the data storage.
For more details, please refer the GNU17 manual, especially “6.4.2 Data Representation”.

5.1 Constants

5.1.1 Storage Duration

The GNU17 C compiler locates constants of static storage duration to the ‘.rodata’ section, and the linker usually relocates the ‘.rodata’ section to ROM.
Constants of automatic storage duration are located to the stack frame on RAM and need the ‘.rodata’ section on ROM to store their initial value. These types of constants are almost same as automatic variables with initial value and usually wastes memory. Don’t forget specifying ‘static’ to local constants to make their storage duration static.

5.1.2 Variable with Initial Value

Variables with initial values need both RAM and ROM. It is better to use constants instead of variables with initial values if possible.

The size of a complex variable (ex. an instance of an array type or a structure type) with initial value may decrease if some part of the variable could be defined as constants.

C source code

```
struct all_member_is_variable {
        int current_value;
        int initial_value;
        char message[16];
} all_variable[4] = {
        {0, 0, "1st Message"},
        {1, 1, "2nd Message"},
        {2, 2, "3rd Message"},
        {3, 3, "4th Message"},
};

const struct constant_part {
        int initial_value;
        char message[16];
} part_constant[4] = {
        {0, "1st Message"},
        {1, "2nd Message"},
        {2, "3rd Message"},
        {3, "4th Message"},
};

struct variable_part {
        int current_value;
        const struct constant_part * constants;
} part_variable[4] = {
        {0, part_constant + 0},
        {1, part_constant + 1},
        {2, part_constant + 2},
        {3, part_constant + 3},
};
```

GDB console

```
(gdb) p sizeof(struct all_member_is_variable) * 4
$1 = 80
(gdb) p sizeof(struct constant_part) * 4
$2 = 72
(gdb) p sizeof(struct variable_part) * 4
$3 = 16
```

The variable part may increase 1-4 bytes to indicate the constant part. Therefore, the constant part should be larger than 4 bytes.
The number of S1C17 instructions may increase to refer to the constant part.

C source code	S1C17 instructions

char * **message1**(**struct** all_member_is_variable * v) {
  return v->message;
}
	0x81bc: add %r0,0x4
	0x81be: ret

const char * **message2**(**struct** variable_part * v) {
  return v->constants->message;
}
	0x81ce: ext 0x2
	0x81d0: ld %r0,[%r0]
	0x81d2: add %r0,0x2
	0x81d4: ret

The influence of additional instructions is a little, if the number of expressions they refer to the constant part from the variable part is a little.

C source code	S1C17 instructions

char * **message_of_index1**(**int** i) {
  return all_variable[i].message;
}
	0x81c0: ld %r2,%r0
	0x81c2: sl %r0,0x2
	0x81c4: add %r0,%r2
	0x81c6: sl %r0,0x2
	0x81c8: ld %r3,0x4
	0x81ca: add %r0,%r3
	0x81cc: ret

const char * **message_of_index2**(**int** i) {
  return part_constant[i]. message;
}
	0x81d6: ld %r2,%r0
	0x81d8: sl %r0,0x3
	0x81da: add %r0,%r2
	0x81dc: sl %r0,0x1
	0x81de: ext 0x105
	0x81e0: ld %r3,0x3a
	0x81e2: add %r0,%r3
	0x81e4: ret

5.1.3 Small Integer Value

The S1C17 data transfer instructions could take the immediate value. The size of immediate value is usually 7-bit, and expandable to 24-bit.

C source code	S1C17 instructions

int **imm0**(**void**) {
  int r = 0;
 	
  return r;
}
	0x81be: ld %r0,0x0
	0x81c0: ret

void * **imm0x876543** (**void**) {
  void * r = (**void** *)0x876543;
 	
  return r;
}
	0x81c2: ext 0x8
	0x81c4: ext 0xeca
	0x81c6: ld.a %r0,0x43
	0x81c8: ret

static const int internal_value = 0x1234;	

int **const_internal_value**(**void**) {
  return internal_value;
}
	0x81d0: ext 0x24
	0x81d2: ld %r0,0x34
	0x81d4: ret

The amount of used ROM area may be decreased by using immediate values embedded in the instruction, instead of defining small integer constants.

C source code	S1C17 instructions
extern const int external_value;	

int **const_external_value**(**void**) {
  return external_value;
}
	0x81ca: ext 0x106
	0x81cc: ld %r0,[0x50]
	0x81ce: ret

GDB console	
	(gdb) p sizeof(external_value) $1 = 2

Following values are suitable for the immediate value and the GNU17 C compiler embeds them into the S1C17 instructions:

- The address of the I/O register

- Bit pattern to compare with the I/O register value
- Bit pattern to write into the I/O register
- The number of elements in the table (size of an array)
- The maximum number of repeats
- Initial value of counters and other temporary variables
- Status codes

Embedded immediate values are better than defining integer constants if the width of the value is 24-bit or less. However, the operation of the program in which immediate values are embedded could not be changed dynamically.

5.2 Alignment of Structure Members

5.2.1 Order of Members

Members of the structure are arranged into the memory in the order in which they are declared.

C source code

```
struct disordered_structure {
    char a_8bit;
    long b_32bit;
    char c_8bit;
    short d_16bit;
    short e_16bit;
    long f_32bit;
};

struct disordered_structure dv;
```

GDB console

```
(gdb) p &(dv.a_8bit)
$1 = 0x0
(gdb) p &(dv.b_32bit)
$2 = (long int *) 0x4
(gdb) p &(dv.c_8bit)
$3 = 0x8 '¥252' <repeats 200 times>...
(gdb) p &(dv.d_16bit)
$4 = (short int *) 0xa
(gdb) p &(dv.e_16bit)
$5 = (short int *) 0xc
(gdb) p &(dv.f_32bit)
$6 = (long int *) 0x10
```

Because each member of a structure is aligned to own boundary depending on their data type, the order of members cause some unused memories.

Address	Address + 0	Address + 1	Address + 2	Address + 3
0x000000	dv.a_8bit	unused	unused	unused
0x000004	dv.b_32bit			
0x000008	dv.c_8bit	unused	dv.d_16bit	
0x00000c	dv.e_16bit		unused	unused
0x000010	dv.f_32bit			

The arrangement method is same as the order of variables or constants belonging to the same section. Even if structures are located in any kind of section, members of them are arranged in same way.
The amount of unused memories in the structure could be decreased by declaring members of the same data type in one place.

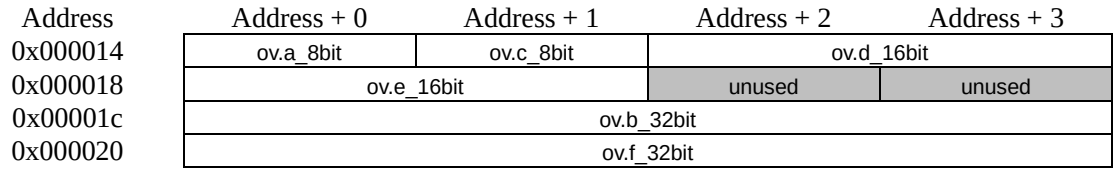
C source code

```
struct ordered_structure {
    char a_8bit;
    char c_8bit;
    short d_16bit;
    short e_16bit;
    long b_32bit;
    long f_32bit;
};
```

```
struct ordered_structure ov;
```

GDB console

```
(gdb) p &(ov.a_8bit)
$7 = 0x14 '¥252' <repeats 200 times>...
(gdb) p &(ov.c_8bit)
$8 = 0x15 '¥252' <repeats 200 times>...
(gdb) p &(ov.d_16bit)
$9 = (short int *) 0x16
(gdb) p &(ov.e_16bit)
$10 = (short int *) 0x18
(gdb) p &(ov.b_32bit)
$11 = (long int *) 0x1c
(gdb) p &(ov.f_32bit)
$12 = (long int *) 0x20
```



Definition of members in order from bigger to smaller type could decrease the amount of unused memory further, because the linker may relocate other objects there.

5.2.2 Dividing into More Structures

There may be some unnamed paddings within a structure, because of the alignment. When the structure with paddings is an element of an array, the amount of unused memories is multiplied by the number of elements.

C source code

```
struct ordered_structure {
    char a_8bit;
    char c_8bit;
    short d_16bit;
    short e_16bit;
    long b_32bit;
    long f_32bit;
};

struct ordered_structure array_all[16];
```

GDB console

```
(gdb) p sizeof(array_all)
$1 = 256
```

Dividing the structure with padding could declare new structures without padding. The amount of memory used by arrays of new structures is less than the array of the original structure.

C source code

```
struct ordered_structure_a {
    long b_32bit;
    long f_32bit;
};

struct ordered_structure_b {
    short d_16bit;
    short e_16bit;
    char a_8bit;
    char c_8bit;
};

struct ordered_structure_a array_a[16];
struct ordered_structure_b array_b[16];
```

GDB console

```
(gdb) p sizeof(array_a)
$2 = 128
(gdb) p sizeof(array_b)
$3 = 96
(gdb) p &(array_a[0].b_32bit)
$4 = (long int *) 0x124
(gdb) p &(array_a[0].f_32bit)
$5 = (long int *) 0x128
(gdb) p &(array_a[1].b_32bit)
```

```
$6 = (long int *) 0x12c
(gdb) p &(array_a[1].f_32bit)
$7 = (long int *) 0x130
(gdb) p &(array_b[0].d_16bit)
$8 = (short int *) 0x1a4
(gdb) p &(array_b[0].e_16bit)
$9 = (short int *) 0x1a6
(gdb) p &(array_b[0].a_8bit)
$10 = 0x1a8 '¥252' <repeats 200 times>...
(gdb) p &(array_b[0].c_8bit)
$11 = 0x1a9 '¥252' <repeats 200 times>...
(gdb) p &(array_b[1].d_16bit)
$12 = (short int *) 0x1aa
(gdb) p &(array_b[1].e_16bit)
$13 = (short int *) 0x1ac
(gdb) p &(array_b[1].a_8bit)
$14 = 0x1ae '¥252' <repeats 200 times>...
(gdb) p &(array_b[1].c_8bit)
$15 = 0x1af '¥252' <repeats 200 times>...
```

Address	Address + 0	Address + 1	Address + 2	Address + 3
0x000124	array_a[0].b_32bit			
0x000128	array_a[0].f_32bit			
0x00012c	array_a[1].b_32bit			
0x000130	array_a[1].f_32bit			

Address	Address + 0	Address + 1	Address + 2	Address + 3
0x0001a4	array_b[0].d_16bit		array_b[0].e_16bit	
0x0001a8	array_b[0].a_8bit	array_b[0].c_8bit	array_b[1].d_16bit	
0x0001ac	array_b[1].e_16bit		array_b[1].a_8bit	array_b[1].c_8bit

However, the number of S1C17 instructions may increase to operate two or more structures.

C source code

```
int member_sum_array_all(int index) {
    int r;

    r = (int)array_all[index].b_32bit;
    r += array_all[index].d_16bit;

    return r;
}
```

S1C17 instructions

```
0x81d2:   sl      %r0,0x4
0x81d4:   ld      %r2,0x24
0x81d6:   add     %r0,%r2
0x81d8:   ext     0x8
0x81da:   ld      %r2,[%r0]
0x81dc:   ext     0x2
0x81de:   ld      %r3,[%r0]
0x81e0:   add     %r2,%r3
0x81e2:   ld      %r0,%r2
0x81e4:   ret
```

C source code

```
int member_sum_partial_array(int index) {
    int r;

    r = (int)array_a [index].b_32bit;
    r += array_b [index].d_16bit;

    return r;
}
```

S1C17 instructions

```
0x81e6:   ld      %r2,%r0
0x81e8:   sl      %r2,0x3
0x81ea:   ext     0x2
0x81ec:   ld      %r3,0x24
0x81ee:   add     %r2,%r3
0x81f0:   ld      %r3,[%r2]
0x81f2:   ld      %r2,%r0
0x81f4:   sl      %r2,0x1
0x81f6:   add     %r2,%r0
0x81f8:   sl      %r2,0x1
0x81fa:   ext     0x3
0x81fc:   ld      %r1,0x24
0x81fe:   add     %r2,%r1
0x8200:   ld      %r2,[%r2]
0x8202:   add     %r3,%r2
0x8204:   ld      %r0,%r3
0x8206:   ret
```

5.2.3 First Member

Each member of a structure is usually accessed via offset from the address (lowest address) of the structure. When a S1C17 register keeps the address of a structure, data transfer between a S1C17 register and a member of a structure on memory needs 2 instructions.

C source code	S1C17 instructions

short member_read_e(struct ordered_structure_b * p) {
  return p->e_16bit;
}
	0x820c: ext 0x2
	0x820e: ld %r0,[%r0]
	0x8210: ret

Because the address of a structure points its 1st member, the number of instructions to transfer data between a register and 1st member is one.

C source code	S1C17 instructions

short member_read_d(struct ordered_structure_b * p) {
  return p->d_16bit;
}
	0x8208: ld %r0,[%r0]
	0x820a: ret

It may decrease the number of S1C17 instructions that arranging the member referred most frequently to the 1st member of the structure.

5.3 Static Data Definition

To shorten the computing time in the following cases, the result of following calculations are generated by the C compiler and stored as constants.

- The calculation is complex and takes time for execution.
- The calculation is frequently executed.

If the calculation is complex, the amount of used memory grows in general.

C source code	S1C17 instructions

short calc(short n) {
  short p;
  short r;
 	
  if (n < 5) {
  p = n * n;
  r = (((7 * 6 - p) * p - 7 * 6 * 5 * 4) * p * n);
  r = ((n * 7 * 6 * 5 * 4 * 3 * 2 * 1) + r) / (7 * 3 * 5 * 3);
  } **else** {
  r = 0;
  }
 	
  return r;
}
	0x8170: ld.a -[%sp],%r4
	0x8172: ld.a -[%sp],%r5
	0x8174: cmp %r0,0x4
	0x8176: jrgt.d 0x1c
	0x8178: ld %r5,%r0
	0x817a: call.d 0x10b <__mulhi3>
	0x817c: ld %r1,%r0
	0x817e: ld %r4,%r0
	0x8180: ld %r0,0x2a
	0x8182: sub %r0,%r4
	0x8184: call.d 0x106 <__mulhi3>
	0x8186: ld %r1,%r4
	0x8188: ext 0x1f9
	0x818a: ld %r1,0x38
	0x818c: add %r0,%r1
	0x818e: call.d 0x101 <__mulhi3>
	0x8190: ld %r1,%r4
	0x8192: call.d 0xff <__mulhi3>
	0x8194: ld %r1,%r5
	0x8196: ld %r3,%r5
	0x8198: sl %r3,0x2
	0x819a: add %r3,%r5
	0x819c: ld %r2,%r3
	0x819e: sl %r2,0x4
	0x81a0: sl %r2,0x2
	0x81a2: sub %r2,%r3
	0x81a4: sl %r2,0x4
	0x81a6: ext 0x2
	0x81a8: ld %r1,0x3b
	0x81aa: call.d 0xce <__divhi3>
	0x81ac: add %r0,%r2
	0x81ae: jpr 0x1
	0x81b0: ld %r0,0x0
	0x81b2: ld.a %r5,[%sp]+
	0x81b4: ld.a %r4,[%sp]+
	0x81b6: ret

Therefore, the size of the calculation result may become smaller than the size of the function that executes

complex calculations.

C source code	S1C17 instructions
`#define C1(n) (((7*6-n*n)*n*n-7*6*5*4)*n*n*n)` `#define C2(n) ((n*7*6*5*4*3*2*1)+C1(n))/(7*3*5*3)`	

`static const short calc_result[5] = {`
`  C2(0), C2(1), C2(2), C2(3), C2(4),`
`};`
	`0x8326: 0x0000`
	`0x8328: 0x000d`
	`0x832a: 0x000e`
	`0x832c: 0x0001`
	`0x832e: 0xffea`

`short calc(short n) {`
`  return (n < 5) ? calc_result[n]: 0;`
`}`
	`0x8170: cmp %r0,0x4`
	`0x8172: jrgt.d 0x7`
	`0x8174: ld %r2,%r0`
	`0x8176: sl %r2,0x1`
	`0x8178: ext 0x106`
	`0x817a: ld %r3,0x26`
	`0x817c: add %r2,%r3`
	`0x817e: ld %r0,[%r2]`
	`0x8180: jpr 0x1`
	`0x8182: ld %r0,0x0`
	`0x8184: ret`

The calculation including floating point numbers and 64-bit integer (“long long”) calls the function of the GNU17 emulation library even if it is easy. The amount of used memory will grow by linked functions of the emulation library.

C source code

```
float convert_to_float(int n) {
    return (float)n;
}
```

Mapped address by the GNU17 linker

```
convert_to_float.o(.text)
 .text   0x00008186    0x convert_to_float.o
         0x00008186       convert_to_float

libgcc.a(.text)
 .text   0x00008308   0x54 libgcc.a(_ashlsi3.o)
         0x00008308        __ashlsi3
 .text   0x0000835c   0x50 libgcc.a(lshrdi3.o)
         0x0000835c        __lshrsi3
 .text   0x000083ac   0x74 libgcc.a(addsf3.o)
         0x000083ac        __floatsisf
.text   0x00008420   0x7a libgcc.a(_c17_emulib_private_func_scan16.o)
         0x00008420       _c17_emulib_private_func_scan16
         0x0000844a       _c17_emulib_private_func_scan32
         0x00008468       _c17_emulib_private_func_scan64
```

The amount of used memory may be decreased by store calculated values in the table instead of calling these functions.

C source code	S1C17 instructions

`static const float int_float[16] = {`
`  0.0, 1.0, 2.0, 3.0,`
`  4.0, 5.0, 6.0, 7.0,`
`  8.0, 9.0, 10.0, 11.0,`
`  12.0, 13.0, 14.0, 15.0,`
`};`
	`0x833c: 0x00000000 0x3f800000`
	`0x8344: 0x40000000 0x40400000`
	`0x834c: 0x40800000 0x40a00000`
	`0x8354: 0x40c00000 0x40e00000`
	`0x835c: 0x41000000 0x41100000`
	`0x8364: 0x41200000 0x41300000`
	`0x836c: 0x41400000 0x41500000`
	`0x8374: 0x41600000 0x41700000`

`float convert_to_float(int n) {`
`  union {`
`  float f;`
`  long l;`
`  } r;`
 	
`  if (n < 16) {`
`  r.f = int_float[n];`
`  } else {`
`  r.l = 0x7f900000; /* NaN */`
`  }`
 	
`  return r.f;`
	`0x8170: cmp %r0,0xf`
	`0x8172: jrgt.d 0x9`
	`0x8174: ld %r2,%r0`
	`0x8176: sl %r2,0x2`
	`0x8178: ext 0x106`
	`0x817a: ld %r3,0x3c`
	`0x817c: add %r2,%r3`
	`0x817e: ld %r0,[%r2]`
	`0x8180: ext 0x2`
	`0x8182: ld %r1,[%r2]`
	`0x8184: jpr 0x3`
	`0x8186: sub %r0,%r0`
	`0x8188: ext 0xff`

```
}
```

```
0x818a:   ld      %r1,0x10
0x818c:   ret
```

5.4 Dynamic Data Generation

5.4.1 Calculation on Demand

Programs are able to calculate values dynamically while executing. And programs are also able to keep same values as constants which were calculated by the compiler.

C source code

```
const unsigned long calculated_exponent2[64] = {
    0 * 0, 1 * 1, 2 * 2, 3 * 3,
    4 * 4, 5 * 5, 6 * 6, 7 * 7,
    8 * 8, 9 * 9, 10 * 10, 11 * 11,
    12 * 12, 13 * 13, 14 * 14, 15 * 15,
    …
};

unsigned long exponent2(unsigned long n) {
    return (n < 32) ? calculated_exponent2[n]: ULONG_MAX;
}
```

Mapped address by the GNU17 linker

```
exponent2.o(.text)
 .text   0x000081d4   0x24 exponent2.o
         0x000081d4        exponent2

 exponent2.o(.rodata)
 .rodata 0x000083a0  0x100 exponent2.o
         0x000083a0        calculated_exponent2
```

When too big data is stored statically, the amount of used memory may be decreased by generating data responding to a demand.

C source code

```
unsigned long exponent2(unsigned long n) {
    return n * n;
}
```

Mapped address by the GNU17 linker

```
dynamic.o(.text)
 .text   0x000081d4    0x8 dynamic.o
         0x000081d4        exponent2

 libgcc.a(.text)
 .text   0x00008326   0x64 libgcc.a(_mulhi3.o)
         0x00008326        __mulhi3
 .text   0x0000838a   0x40 libgcc.a(_mulsi3.o)
         0x0000838a        __mulsi3
```

5.4.2 Decompression

Generally the expression of data is redundant. The amount of used memory could be decreased by compressing big redundant data.

C source code

```
extern const struct constant_part *part_constant;

struct variable_part2 {
    int current_value[4];
    char index[4];
} part_variable2 = {
    {0, 1, 2, 3},
    {0, 1, 2, 3},
};
```

GDB console

```
(gdb) p sizeof(struct constant_part) * 4
$1 = 72
(gdb) p sizeof(struct variable_part) * 4
$2 = 16
(gdb) p sizeof(struct variable_part2)
$3 = 12
```

If the processing of compression and decompression is easy, the amount of used memory becomes small.

C source code

```
const char * message_of_index3(int i) {
    int index = part_variable2.constant_index[i];
    return part_constant[index].message;
}
```

S1C17 instructions

```
0x81f8:   ext     0x4
0x81fa:   ld      %r2,0xc
0x81fc:   add     %r0,%r2
0x81fe:   ld.b    %r2,[%r0]
0x8200:   ld      %r0,%r2
0x8202:   sl      %r0,0x3
0x8204:   add     %r0,%r2
0x8206:   sl      %r0,0x1
0x8208:   ext     0x109
0x820a:   ld      %r2,[0x3a]
0x820c:   add     %r0,%r2
0x820e:   add     %r0,0x2
0x8210:   ret
```

When the amount of used memory grows, and the project selects the middle model of the GNU17, the size of the S1C17 pointer type becomes 32-bit.
If the size of the target RAM is 64KB or less and the variable always points an address on RAM, the effective data in the variable is 16-bits or less. In this case, it is possible to compress a 32-bit address into a 16-bit data by rounding down higher 16-bits and to decompress a 16-bit data oppositely.

5.5 Memory Sharing

5.5.1 Heap Area

The GNU17 ANSI library 'libc.a' supports heap functions 'malloc', 'calloc', 'realloc' and 'free'.
For usage of heap functions of the GNU17, refer to the GNU17 manual, especially "7.3.3 Declaring and Initializing Global Variables".
There is little advantage to use the heap area in the small memory program, because the small heap area is used up at once.

5.5.2 Stack Area

Functions called from the same function will assign each stack frame from the same address (value of the stack pointer). These functions share the same stack area. And 'auto' variables (variables of automatic storage duration) of these functions share same addresses at other times.
Therefore, the amount of used RAM area could be decreased by specifying automatic assigning duration for variables used in the function. A variable which is declared without the storage class specifier 'static' in the function has automatic storage duration.

C source code

```
void * stack_share_callee1(int n) {
    char test[16];
    void * r;

    memset(test, n, sizeof(test));
    r = (void *)test;

    return r;
}

void * stack_share_callee2(int n) {
    char test[16];
    void * r;

    memset(test, n, sizeof(test));
    r = (void *)test;

    return r;
}

void * stack_share_callee3(int n) {
    char test[16];
    void * r;
```

```
    memset(test, n, sizeof(test));
    r = (void *)test;

    return r;
}

void * stack_share_caller() {
    void * stack;
    void * r;

    r = stack = stack_share_callee1(1);
    stack = stack_share_callee2(2);
    if (stack < r) {
        r = stack;
    }
    stack = stack_share_callee3(3);
    if (stack < r) {
        r = stack;
    }

    return r;
}
```

Mapped address by the GNU17 linker

```
stack_share.o(.text)
 .text   0x0000833c 0x52 stack_share.o
        0x0000833c     stack_share_callee1
        0x0000834c     stack_share_callee2
        0x0000835c     stack_share_callee3
        0x0000836c     stack_share_caller
```

GDB console

```
(gdb) set $pc = stack_share_caller
(gdb) set $sp = 0x400
(gdb) next 4
(gdb) p r
$1 = (void *) 0x3e8
(gdb) p $sp - (int)r
$2 = 20
(gdb) x/20xb r
0x3e8:   0x03     0x03     0x03     0x03     0x03     0x03     0x03     0x03
0x3f0:   0x03     0x03     0x03     0x03     0x03     0x03     0x03     0x03
0x3f8:   0x82     0x83     0x00     0x00
```

This effect requires that the same stack area is used by most functions. When some functions require big stack frame, the effect becomes small, because the stack area is not fully shared.

C source code

```
void * stack_share_callee2(int n) {
    char test[32];        // size of 'test' is changed.
    void * r;

    memset(test, n, sizeof(test));
    r = (void *)test;

    return r;
}
```

GDB console

```
(gdb) set $pc = stack_share_caller
(gdb) set $sp = 0x400
(gdb) next 7
(gdb) p r
$1 = (void *) 0x3d8
(gdb) p $sp - (int)r
$2 = 36
(gdb) x/36xb r
0x3d8:   0x02     0x02     0x02     0x02     0x02     0x02     0x02     0x02
0x3e0:   0x02     0x02     0x02     0x02     0x66     0x83     0x00     0x00
0x3e8:   0x03     0x03     0x03     0x03     0x03     0x03     0x03     0x03
0x3f0:   0x03     0x03     0x03     0x03     0x03     0x03     0x03     0x03
0x3f8:   0x82     0x83     0x00     0x00
```

5.5.3 Union

One memory area could be used as some variables by defining a union type including some members which should not be used at the same time.

C source code

```
short mode_a_variables[8];
short mode_b_variables[16];
```

Mapped address by the GNU17 linker

```
mode_variables.o(.bss)
 .bss    0x00000204 0x30 mode_variables.o
        0x00000204     mode_a_variables
        0x00000214     mode_b_variables
```

Please define union type when a part of global variables or some members of the structure are used according to the state of the program.

C source code

```
struct global_parameter {
    short current_mode;
    union mode_variable {
        short a_variables[8];
        short b_variables[16];
    } mode;
} global_parameter;
```

Mapped address by the GNU17 linker

```
mode_variables.o(.bss)
 .bss    0x00000204 0x22 mode_variables.o
        0x00000204     global_parameter
```

6. Combine Procedures and Data

This section shows how to decrease the number of S1C17 instructions by using functions for common procedures, and shows how to decrease the data storage by sharing common values.

6.1 Use Standard Library Functions

The GNU17 provides the ANSI C standard library 'libc.a'. This library is written by EPSON and not under the GPL. For more detail of the library function, refer to the GNU17 manual, especially "7.3 ANSI library".

Standard character functions ('memcmp', 'memcpy' and so on) declared in 'string.h' are useful to operate stream data in the memory. Source codes of GNU17 character functions are written by the S1C17 assembler to decrease the number of S1C17 instructions.

C source code

```
short clear_and_call (void) {
        short work[16];
        int i;

        for (i = 0; i < 16; i++) {
                work[i] = 0;
        }

        return output_some_value(work);
}
```

S1C17 instructions

```
0x8192:   sub.a   %sp,0x20
0x8194:   ld      %r0,0x0
0x8196:   ld      %r2,%r0
0x8198:   sl      %r2,0x1
0x819a:   ld.a    %r3,%sp
0x819c:   add     %r2,%r3
0x819e:   ld      %r3,0x0
0x81a0:   add     %r0,0x1
0x81a2:   cmp     %r0,0xf
0x81a4:   jrle.d  0x78
0x81a6:   ld      [%r2],%r3
0x81a8:   ld.a    %r0,%sp
0x81aa:   call    0x3f1
0x81ac:   add.a   %sp,0x20
0x81ae:   ret
```

C source code

```
short clear_and_call (void) {
        short work[16];

        memset(work, 0, sizeof(work));

        return output_some_value(work);
}
```

S1C17 instructions

```
0x8192:   sub.a   %sp,0x20
0x8194:   ld.a    %r0,%sp
0x8196:   ld      %r1,0x0
0x8198:   call.d  0x115    <memset>
0x819a:   ld      %r2,0x20
0x819c:   ld.a    %r0,%sp
0x819e:   call    0x3f7
0x81a0:   add.a   %sp,0x20
0x81a2:   ret
```

All source codes of the GNU17 library are contained in the GNU17 'utility/lib_src' folder.

The GNU17 C compiler generates S1C17 instructions which call 'memcpy' function from the C source code which copies arrays and structures.

C source code

```
struct copy_struct {
        short buffer[16];
};
short copy_and_call(struct copy_struct * value) {
        struct copy_struct work;

        work = *value;

        return output_some_value(work.buffer);
}
```

S1C17 instructions

```
0x81b0:   sub.a   %sp,0x20
0x81b2:   ld      %r1,%r0
0x81b4:   ld.a    %r0,%sp
0x81b6:   call.d  0x10d    <memcpy>
0x81b8:   ld      %r2,0x20
0x81ba:   ld.a    %r0,%sp
0x81bc:   call    0x3e8
0x81be:   add.a   %sp,0x20
0x81c0:   ret
```

When 'memcpy' function has been called by the GNU17 C compiler, the amount of used memory does not increase even if additional C source code calls 'memcpy'.

6.2 Define New Function for Same Procedures

The amount of used memory may be decreased by defining a new function from same procedures.

C source code	S1C17 instructions

short partly_same_procedure_1(**int** n) {
  int i;
  short buffer[8];
 	
  if (8 < n) {
  n = 8;
  }
  for (i = 0; i < n; i++) {
  buffer[i] = i;
  }
 	
  return similar_function_1(buffer, n);
}
	0x81dc: sub.a %sp,0x10
	0x81de: ld %r2,0x8
	0x81e0: cmp %r0,%r2
	0x81e2: jrle.d 0x2
	0x81e4: ld %r1,%r0
	0x81e6: ld %r1,%r2
	0x81e8: ld %r3,0x0
	0x81ea: cmp %r3,%r1
	0x81ec: jrge.d 0x9
	0x81ee: ld %r2,%r3
	0x81f0: sl %r2,0x1
	0x81f2: ld.a %r0,%sp
	0x81f4: add %r2,%r0
	0x81f6: ld [%r2],%r3
	0x81f8: add %r3,0x1
	0x81fa: cmp %r3,%r1
	0x81fc: jrlt.d 0x79
	0x81fe: ld %r2,%r3
	0x8200: ld.a %r0,%sp
	0x8202: call 0x3e8
	0x8204: add.a %sp,0x10
	0x8206: ret

short partly_same_procedure_2(**int** n) {
  int i;
  short buffer[4];
 	
  if (4 < n) {
  n = 4;
  }
  for (i = 0; i < n; i++) {
  buffer[i] = i;
  }
 	
  return similar_function_2(buffer, n);
}
	0x8208: sub.a %sp,0x8
	0x820a: cmp %r0,0x4
	0x820c: jrle.d 0x2
	0x820e: ld %r1,%r0
	0x8210: ld %r1,0x4
	0x8212: ld %r3,0x0
	0x8214: cmp %r3,%r1
	0x8216: jrge.d 0x9
	0x8218: ld %r2,%r3
	0x821a: sl %r2,0x1
	0x821c: ld.a %r0,%sp
	0x821e: add %r2,%r0
	0x8220: ld [%r2],%r3
	0x8222: add %r3,0x1
	0x8224: cmp %r3,%r1
	0x8226: jrlt.d 0x79
	0x8228: ld %r2,%r3
	0x822a: ld.a %r0,%sp
	0x822c: call 0x3d5
	0x822e: add.a %sp,0x8
	0x8230: ret

However, as described by Section 1.2.1 “Functions”, the GNU17 C compiler adds procedures to calling, beginning and ending new function.

C source code	S1C17 instructions

short partly_same_procedure_1(**int** n) {
  short buffer[8];
 	
  if (8 < n) {
  n = 8;
  }
  same_procedure(buffer, n);
 	
  return similar_function_1(buffer, n);
}
	0x81dc: ld.a -[%sp],%r4
	0x81de: sub.a %sp,0x10
	0x81e0: ld %r2,0x8
	0x81e2: cmp %r0,%r2
	0x81e4: jrle.d 0x2
	0x81e6: ld %r4,%r0
	0x81e8: ld %r4,%r2
	0x81ea: ld.a %r0,%sp
	0x81ec: call.d 0x16
	0x81ee: ld %r1,%r4
	0x81f0: ld.a %r0,%sp
	0x81f2: call.d 0x3f0
	0x81f4: ld %r1,%r4
	0x81f6: add.a %sp,0x10
	0x81f8: ld.a %r4,[%sp]+
	0x81fa: ret

short partly_same_procedure_2(**int** n) { **short** buffer[4];	0x81fc: ld.a -[%sp],%r4 0x81fe: sub.a %sp,0x8

```
    if (4 < n) {
        n = 4;
    }
    same_procedure(buffer, n);

    return similar_function_2(buffer, n);
}
```

```
0x8200:   cmp     %r0,0x4
0x8202:   jrle.d  0x2
0x8204:   ld      %r4,%r0
0x8206:   ld      %r4,0x4
0x8208:   ld.a    %r0,%sp
0x820a:   call.d  0x7
0x820c:   ld      %r1,%r4
0x820e:   ld.a    %r0,%sp
0x8210:   call.d  0x3e3
0x8212:   ld      %r1,%r4
0x8214:   add.a   %sp,0x8
0x8216:   ld.a    %r4,[%sp]+
0x8218:   ret
```

```
void same_procedure(short * buffer, int n) {
    int i;

    for (i = 0; i < n; i++) {
        buffer[i] = i;
    }
}
```

```
0x821a:   ld.a    -[%sp],%r4
0x821c:   ld      %r3,0x0
0x821e:   cmp     %r3,%r1
0x8220:   jrge.d  0x8
0x8222:   ld      %r2,%r3
0x8224:   sl      %r2,0x1
0x8226:   add     %r2,%r0
0x8228:   ld      [%r2],%r3
0x822a:   add     %r3,0x1
0x822c:   cmp     %r3,%r1
0x822e:   jrlt.d  0x7a
0x8230:   ld      %r2,%r3
0x8232:   ld.a    %r4,[%sp]+
0x8234:   ret
```

It is necessary to approve the following relation to decrease the size of the program actually:

A * N > A + B + C * N

A: The number of S1C17 instructions of the procedure
N: The number of locations where the procedure is executed
B: The number of additional S1C17 instructions in the new function
C: The number of additional S1C17 instructions to prepare arguments and call the function

Here assumes following relations, because B and C vary with the number of arguments:

B = b1 * NA + b2

b1 * NA: The number of S1C17 instructions to begin the function
b2: The number of S1C17 instructions to end the function
NA: The number of arguments for the function

C = c1 * NA + c2

c1 * NA: The number of S1C17 instructions to prepare arguments
c2: The number of S1C17 instructions to call the function

b1, b2, c1 and c2 are small values on the GNU17. If they are assumed 3 instructions, it is necessary to approve the following relation to decrease the size of the program:

A > 3 * (NA + 1) * (N + 1) / (N - 1)

N <= 1: The size of the program could not become small.
N == 2: A > 9 * (NA + 1)

N == 3: A > 6 * (NA + 1)
N == 4: A > 5 * (NA + 1)
N is big enough:
A > 3 * (NA + 1)

When the number of S1C17 instructions for the procedure is smaller, the procedure should be defined as a macro function.

6.3 Combine Similar Functions

If some procedures are similar but not same, it is not possible to define a new function without any changing. When the size of the common part is large to the different part, the size of the entire program may become small even if the procedure becomes complex than before. There are some approaches to design.

6.3.1 Define Common Functions

New functions to execute the common part should be defined. Similar procedures are modified to call new functions.

C source code

```
char p0_get(void) {
    static const char convert[8] = {
        0, 0, 0, 1, 0, 1, 1, 1,
    };
    int bit, byte, sample, length;

    t16.ctl = 0x0103;
    length = 8;
    byte = 0;

    do {
        bit = 0;
        sample = 3;
        do {
            while (t16.intf == 0);
            t16.intf = 0x0001;
            bit <<= 1;
            bit |= p0.dat & 1;
        } while (--sample > 0);

        byte <<= 1;
        byte |= convert[bit];
    } while (--length > 0);

    t16.ctl = 0x0000;
    t16.intf = 0x0001;

    return (char)byte;
}
```

C source code

```
char p1_get(void) {
    static const char convert[8] = {
        0, 0, 0, 1, 0, 1, 1, 1,
    };
    int bit, byte, sample, length;

    t16.ctl = 0x0103;
    length = 8;
    byte = 0;

    do {
        bit = 0;
        sample = 3;
        do {
            while (t16.intf == 0);
            t16.intf = 0x0001;
            bit <<= 1;
            bit |= p1.dat & 1;
        } while (--sample > 0);

        byte <<= 1;
        byte |= convert[bit];
    } while (--length > 0);

    t16.ctl = 0x0000;
    t16.intf = 0x0001;

    return (char)byte;
}
```

The before and behind part of the different part are defined as new functions.

C source code

```
void t16_wait(void) {
    while (t16.intf == 0);
    t16.intf = 0x0001;
}
```

C source code

```
void t16_stop(void) {
    t16.ctl = 0x0000;
    t16.intf = 0x0001;
}
```

C source code

```
int convert_bit(int byte, int bit) {
    static const char convert[8] = {
        0, 0, 0, 1, 0, 1, 1, 1,
    };
    byte <<= 1;
    byte |= convert[bit];

    return byte;
}
```

C source code

```
char p0_get(void) {
    int bit, byte, sample, length;

    t16.ctl = 0x0103;
    length = 8;
    byte = 0;

    do {
        bit = 0;
        sample = 3;
        do {
            t16_wait();
            bit <<= 1;
            bit |= p0.dat & 1;
        } while (--sample > 0);
        byte = convert_bit(byte, bit);
    } while (--length > 0);

    t16_stop();

    return (char)byte;
}
```

```
char p1_get(void) {
    int bit, byte, sample, length;

    t16.ctl = 0x0103;
    length = 8;
    byte = 0;

    do {
        bit = 0;
        sample = 3;
        do {
            t16_wait();
            bit <<= 1;
            bit |= p1.dat & 1;
        } while (--sample > 0);
        byte = convert_bit(byte, bit);
    } while (--length > 0);

    t16_stop();

    return (char)byte;
}
```

The number of functions has increased by 3 for this example.

6.3.2 Define New Parameter to Select Difference

A new function to execute different parts by the condition branching should be defined. A parameter of the function specifies which part should be executed. Similar procedures are changed to call the new function.

C source code

```
char port_get(int channel) {
    static const char convert[8] = {
        0, 0, 0, 1, 0, 1, 1, 1,
    };
    int bit, byte, sample, length;

    t16.ctl = 0x0103;
    length = 8;
    byte = 0;

    do {
        bit = 0;
        sample = 3;
        do {
            while (t16.intf == 0);
            t16.intf = 0x0001;
            bit <<= 1;
            switch (channel) {
            case 1:
                bit |= p1.dat & 1;
                break;
            case 0:
            default:
                bit |= p0.dat & 1;
                break;
            }
        } while (--sample > 0);

        byte <<= 1;
        byte |= convert[bit];
    } while (--length > 0);

    t16.ctl = 0x0000;
    t16.intf = 0x0001;

    return (char)byte;
}
```

C source code

```
char p0_get(void) {
    return port_get(0);
}
```

C source code

```
char p1_get(void) {
    return port_get(1);
}
```

The number of functions has increased by 1 for this example.

6.3.3 Define Different Functions

New functions to execute each different part and another new function to execute common parts should be defined. A parameter of the common part function specifies which different part should be executed. Similar procedures are changed to call the common part function.

C source code

```
char port_get(int (*get_dat)(void)) {
    static const char convert[8] = {
        0, 0, 0, 1, 0, 1, 1, 1,
    };
    int bit, byte, sample, length;

    t16.ctl = 0x0103;
    length = 8;
    byte = 0;

    do {
        bit = 0;
        sample = 3;
        do {
            while (t16.intf == 0);
            t16.intf = 0x0001;
            bit <<= 1;
            bit |= get_dat();
        } while (--sample > 0);
```

```
            byte <<= 1;
            byte |= convert[bit];
        } while (--length > 0);

        t16.ctl = 0x0000;
        t16.intf = 0x0001;

        return (char)byte;
}
```

C source code

```
int p0_dat(void) {
        return p0.dat & 1;
}
```

C source code

```
int p1_dat(void) {
        return p1.dat & 1;
}
```

C source code

```
char p0_get(void) {
        return port_get(p0_dat);
}
```

C source code

```
char p1_get(void) {
        return port_get(p1_dat);
}
```

The number of functions has increased by 3 for this example.
Because there is less location where new different functions are called, the amount of used memory does not become small in this approach compared with other approaches.

6.4 Share Same Values

Not only the variable but also the constant becomes another object even if the value is the same, and they are aligned to other addresses.

C source code

```
const int zero = 0;
const int initial = 0;
const char test_message[] = "TEST:";
const char work_message[] = "TEST:";
```

Mapped address by the GNU17 linker

```
samevalues.o(.rodata)
 .rodata 0x00008704 0x10 samevalues.o
        0x00008704      zero
        0x00008706      initial
        0x00008708      test_message
        0x0000870e      work_message
```

GDB console

```
(gdb) p zero
$1 = 0
(gdb) p initial
$2 = 0
(gdb) p test_message
$3 = "TEST:"
(gdb) p work_message
$4 = "TEST:"
```

When some constants are in the C source code and are pointer to the same string literal, they indicate the same address. When they are in different C source codes, they indicate different address.

C source code

```
const char * const test_message = "TEST:";
const char * const work_message = "TEST:";
```

C source code

```
const char * const external_message = "TEST:";
```

Mapped address by the GNU17 linker

```
externvalues.o(.rodata)
 .rodata 0x000086b8  0x8 externvalues.o
        0x000086be      external_message
 samevalues.o(.rodata)
 .rodata 0x0000870c  0xa samevalues.o
        0x00008712      test_message
        0x00008714      work_message
```

GDB console

```
(gdb) p test_message
$1 = 0x870c "TEST:"
(gdb) p work_message
$2 = 0x870c "TEST:"
```

```
(gdb) p external_message
$3 = 0x86b8 "TEST:"
```

When the string literal is defined in the C header file as a macro, same string literals are generated in each C source code that uses the macro.
The amount of used memory may be decreased, when the constant is generated in a specific C source code and other source codes refer to the address of the constant.

C source code

```
const char * error_message(short code) {
	static const struct error_message_t {
		short code;
		const char * string;
	} error_messages[] = {
		{0x0001, "Failed."},
		{0x0000, "Succeeded."},
		{-1, "Unknown Error."},
	};
	const struct error_message_t * message;

	message = error_messages;
	while (message->code != code) {
		if (message->code == -1) {
			break;
		}
		message++;
	}

	return message->string;
}
```

7. Remove Unused Procedures and Data

This section shows how to detect unused functions, variables and constants to remove the unused part of the program.

7.1 Analyze C Source Codes

7.1.1 GNU17 C Compiler

If '-Wall' is specified for the GNU17 C compiler command-line option, the compiler outputs warnings to unused functions and variables of static storage duration, and unused variables of automatic storage duration.
For details of the GNU17 C compiler command-line option, refer to the GNU17 manual, especially "6.3.2 Command-line Options".

The GNU17 IDE usually specifies '-Wall' for the C compiler to generate object file from a C source code. Therefore, the compiler outputs warnings to unused functions and variables, when the GNU17 IDE builds the program.
Warnings are reflected in the console view and the problem view of the GNU17 IDE.

C source code

```
static void unused_internal_function(void) {
    return;
}
```

console view of IDE

```
unused_function.c:8: warning: `unused_internal_function' defined but not used
```

Warned functions and variables also exist in the memory of the built program. The amount of used memory could be decreased by removing warned functions and variables.
After warned objects are removed and the program is built again, the C compiler may warn again that some functions and variables are unused. Please remove newly warned unused objects, too.

However, the GNU17 C compiler does not warn of unused constants of static storage duration.

7.1.2 GNU17 IDE Function

The GNU17 C compiler does not warn unused functions and variables they are defined as external linkage. The function that has no storage class specifier has external linkage.

C source code

```
void unused_external_function(void) {
    return;
}
```

Unused functions and variables with external linkage also exist in the memory of the built program.
When "Unused Functions" detecting is enabled and the project is built on the GNU17 IDE, Splint outputs warnings if there is unused function in the project. Warnings are reflected in the problem view of the GNU17 IDE.
For details to enable the detecting, refer to the GNU17 manual, especially "5.7.11 Detecting Unused Functions".

Sometimes, Splint fails to parse a C source code and gives up analysis the project after reporting 'parse error'. In this case, insert 'S_SPLINT_S' macro into the C source code to discard the failed part while Splint analysis.

C source code

```
static void boot(void) {
#ifndef S_SPLINT_S
    asm __volatile__("xld.a %ps, __START_stack ");
    asm __volatile__("xjpr main");
#endif
}
```

The amount of used memory could be decreased by removing warned functions. After warned functions are

removed and the program is built again, Splint may warn again that some functions are unused. Please remove newly warned unused functions, too.

7.1.3 Eclipse CDT Indexer

Eclipse CDT (base of the GNU17 IDE) contains indexer of C source codes.
Some indexers are selectable from the GNU17 IDE menu [Window] > [Preferences]. Please select "Full C/C++ Indexer" from [C/C++] > [Indexer] section of the Preferences dialog to analyze accurately.
The indexer analyses the project by the following procedure:

- Build the project.
- Open the context menu on the C/C++ projects view.
- Select [Index] > [Freshen All Files] of the context menu.
- Select name to analyze on the C source code.
- Press Ctrl + Alt + H key to open the call hierarchy.

The result is output to the call hierarchy view. If any caller does not exist, selected function, variable or constant is not used in C source codes. However, it may be used from the S1C17 assembler source code (including inline assembler).

The original caller of functions is the interrupt vector table in case of the S1C17 program. When the table is described by the assembler, functions registered in the table are original caller of others.
Please confirm the referrer of variables and constants, when address of the function is set to variables and constants.

Detecting unused constants is possible by this method.

7.2 Analyze Object Files

If '—cref' is specified for the GNU17 linker command-line option, the linker outputs the cross reference table. For details to specify additional command-line option to the linker, refer to the GNU17 manual, especially "5.7.5 Setting Linker Options".

When the project is built, the linker outputs the following cross reference table to the tail of the map (*.map) file:

Cross Reference Table by the GNU17 linker

```
Cross Reference Table

Symbol                                  File
__START_stack                           vector.o
__START_bss                             vector.o
__START_data                            vector.o
__START_data_lma                        vector.o
__END_bss                               vector.o
__END_data                              vector.o
main                                    main.o
                                        vector.o
memcpy                                  libc.a(memcpy.o)
                                        vector.o
memset                                  libc.a(memset.o)
                                        vector.o
vector                                  vector.o
```

The left side of the table (labeled 'Symbol') is global symbols, and the right side (labeled 'File') is object files which refer the symbol. Global symbols are external linkage.
The symbol referred by two or more object files is used. The symbol referred by only one object file may be unused. The symbol referred by one should not be external linkage. Please change the symbol to internal linkage by specifying 'static'. If the symbol is not actually used, the GNU17 C compiler outputs warnings after that.

7.3 Remove Functions Depending on Environment

7.3.1 Debug and Test Functions

When the program embedded in the product is built from the project including the function to debug and test, the GNU17 IDE should compile source codes excluding the function not to embed.
For this case, please implement C source codes to exclude the function not to embed by defining a macro.
Following example defines 'NDEBUG' macro while non-debugging:

C source code

```
#ifdef NDEBUG
#define ASSERT(c)
#else
#define ASSERT(c)     ((c) ? TRUE: assert_output(__FILE__, __LINE__, #c), FALSE)
#endif
```

C source code

```
#ifdef NDEBUG
#define PACKET_LOG(b, n)
#else
extern void packet_log_add(const char * packet, int length);
#define PACKET_LOG(b, l)          packet_log_add((b), (l))
#endif
```

C source code

```
int packet_get(char * packet) {
    int length;
    int error;

    length = sio_read(packet, 16);
    ASSERT(length <= 16);

    if (length <= 0) {
        error = 0;
    } else {
        PACKET_LOG(packet, length);
        if (0 == checksum(packet, length)) {
            error = length;
        } else {
            error = ERROR_PACKET_CHECKSUM;
        }
    }

    return error;
}
```

The status of defining a macro is changed in the GNU17 C compiler command-line option. Please refer the GNU17 manual "5.7.3 Setting Compiler Options" [Symbols] section.

7.3.2 Non-S1C17 Functions

When the GNU17 C compiler compiles the C source code developed for PC, it is necessary to exclude the function not to use with the S1C17.
For this case, please implement C source codes to exclude PC functions by the GNU17 pre-defined macro.

C source code

```
#ifdef __c17
// for GNU17
extern int spi0_read(char * buffer, int length);
#define sio_read(b, l)   spi0_read((b), (l));
#else
// for PC
#include <stdio.h>
extern FILE * sio_in;
#define sio_read(b, l)   fread((b), 1, (l), sio_in)
#endif
```

'__c17' is a pre-defined macro of the GNU17 C compiler, and is defined whenever the GNU17 C compiler compiles C source code.

Revision History

Attachment-1

Rev. No.	Date	Page	Category	Contents
Rev 1.0	2013/12/01	All	new	

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.
7F, Jinbao Bldg., No.89 Jinbao St.,
Dongcheng District,
Beijing 100005, CHINA
Phone: +86-10-8522-1199 FAX: +86-10-8522-1125

SHANGHAI BRANCH
7F, Block B, Hi-Tech Bldg., 900 Yishan Road,
Shanghai 200233, CHINA
Phone: +86-21-5423-5577 FAX: +86-21-5423-4677

SHENZHEN BRANCH
12F, Dawning Mansion, Keji South 12th Road,
Hi-Tech Park, Shenzhen 518057, CHINA
Phone: +86-755-2699-3828 FAX: +86-755-2699-3838

EPSON HONG KONG LTD.
Unit 715-723, 7/F Trade Square, 681 Cheung Sha Wan Road,
Kowloon, Hong Kong.
Phone: +852-2585-4600 FAX: +852-2827-4346

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
5F, 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.
MICRODEVICES OPERATIONS DIVISION

IC Sales & Marketing Department
421-8, Hino, Hino-shi, Tokyo 191-8501, JAPAN
Phone: +81-42-587-5814 FAX: +81-42-587-5117

Document Code: 412684000
First Issue Dec., 2013 in JAPAN