BG003 : Renesas Technical Training

C Programming Course

Renesas Design Vietnam Co., Ltd.
RVC Training Center
Hung Quoc Pham
Bich Ngoc Thi Nguyen

Apr 22-23, 2015

The Objective of This Course

- To understand techniques for Embedded Programming
  - Difference from PC Programs
  - How to control processor using C Language
- To acquire basic knowledge to be a Professional Programmer
  - Total productivity including design, coding, testing, maintenance
  - Team development

Prerequisites

- Some knowledge of C (or C++, Java, C#) language
- Basic knowledge of CPU
  - Processor, memory, peripherals
  - Registers, Instructions

Contents

I. Quick Review of the C Language

Contents

1. Introduction

- C is a high-level programming language
  - In ’50s, programs were directly written in machine language
  - High-level programming languages are developed in late ’50s.
  - Design goals of high-level programming languages:
    - Standardization
    - Reduction of software development cost
    - Efficient execution

Purpose of Programming Languages

- Description of a job to be executed by a computer (for computers)
  - So you must write precisely, avoiding mistakes.
- Communication medium among programmers (for human beings)
  - Your program will be re-used or maintained.
  - You will forget what you have written after several years.
    So the description must be readable not only to other engineers, but also to yourself.

How do we Review C Language

- Assumption: You already have some experience with C (or other languages such as Java, C++)
- Reviewing the grammar, explanations are added from the following viewpoints:
  - Implementation dependence and how to make programs portable
  - Standard usage to avoid mistakes
  - How to organize data and programs for readability
  - Higher programming techniques

Overview of the C Language

- Compilation is done in the following two steps:
  - Preprocessing: File inclusion, macro expansion, etc. Processing commands are lines starting with “#”
  - Compilation: The result of preprocessing is compiled into an object program.

- Structure of the grammar
  - Basic elements: Basic words such as variables, constants, operators, etc.
  - Expressions
  - Statements
  - Declarations
  - Functions

2. Expressions

- Expressions computes a result based on variables or constants (operands) using operators.
- Each datum has a type, which gives the interpretation of the bits in the data. Every variable must be declared to specify its type.
  e.g. `int a;`
- As C has a rich set of operators, the rules of computation can be very complex.
- Even an assignment (=) is an expression. So expressions can have side-effects.

Basic Data Types

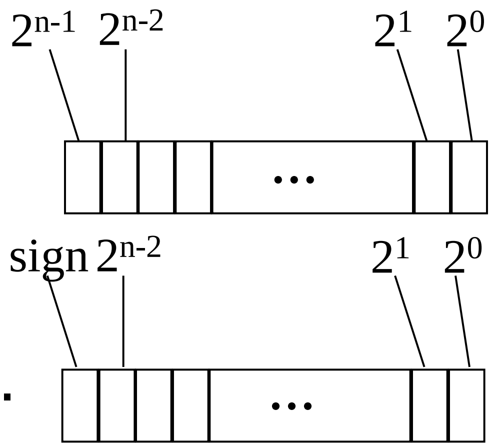
- Integers
  - Signed integers
    - `signed char        8 bit`
    - `short             16 bit`
    - `int               16 or 32 bit`
    - `long              32 bit`
  - Unsigned integers
    - `unsigned char     8 bit`
    - `unsigned short    16 bit`
    - `unsigned          16 or 32 bit`
    - `unsigned long     32 bit`
  - Floating-point numbers
    - `float             32 bit`
    - `double            64 bit`

Basic Data Types

- There are signed and unsigned integers.
- Size of `int` is implementation-dependent.
  It is 16 bit on 16-bit machines, and 32 bit on 32-bit machines.
  - `int` type is the most efficient integer data type on the machine (i.e. the size of machine register).

- The sign of `char` (if there is no keyword `signed` or `unsigned`) is implementation-dependent.
  - So it is not portable to assign values outside the range of 0~127 to `char` type data.

Data Representation



- n-bit unsigned integer represents $0 \sim 2^n-1$
- n-bit signed integer represents $-2^{n-1} \sim 2^{n-1}-1$.
  - The MSB is interpreted as -2^{n-1}.
    e.g. 111....11 is -1 $(-2^{n-1}+(2^{n-2}+2^{n-3}+...+1))=-1$
  - Two's complement representation.
- Floating point data is represented by IEEE 754 Floating Point Standard Format
  - `float`: 1 bit sign, 8 bit exponent, 23 bit mantissa
  - `double`: 1 bit sign, 11 bit exponent, 52 bit mantissa
- **QUIZ**: What is the representation of
  - 255 (unsigned char) ?
  - -255 (short)?

RENESAS

Size of `int`

- For SH, size of `int` is 32 bit (size of registers).
- If data size should be explicit, do not use `int`. Use `long` or `short`.
- Use `int` only when you are sure the data value is small enough (e.g. index of an array).
- Use of `short` may be inefficient on SH (or other 32-bit machines).

Why `short` is Not Efficient on 32-bit Machines

- `short` data must be in the range of short (-2^{15}~$2^{15}-1$ for signed, 0~$2^{16}-1$ for unsigned)
- So after each operation, the instruction to keep the data value in this range must be executed (in SH, `EXTS.W` for signed, `EXTU.W` for unsigned).
- Similar for `char` type.

RENESAS

Arithmetic Operators

- +, − (unary and binary), / has its usual meaning in mathematics.
- *: multiplication, %: remainder
- Operator precedence
  - Unary operators has highest precedence.
  - Multiplicative operators (*, /, %) has higher precedence than additive operators (+, −).
  - Same level operators associates to the left.
- / and % operators for negative integers are not well-defined. So use them only for positive integers.

Example

```
#include <stdio.h>

main()
{
    int a, b;
    double c, d;
    a=5;
    b=3;
    printf("%d %d %d %d %d %d\n", -a, a+b, a-b, a*b, a/b, a%b);
    c=1.0;
    d=2.0;
    printf("%f %f %f %f %f\n", -c, c+d, c-d, c*d, c/d);
}
```

Logical Operators

- `&`: and, `|`: or, `^`: exclusive-or, `~`: negation (unary), `<<`: left shift, `>>`: right shift
- Use logical operators with unsigned data. (because their effect on sign-bit is not well-defined).
- Hexadecimal notation (hexadecimal digits starting with `0x`) is convenient to represent bit patterns.

Example

```
#include <stdio.h>

main()
{
    unsigned int a, b, c;
    a=0x000000ff;
    b=0x00000f0f;
    c=3;
    printf("%x %x %x %x %x %x\n",
           ~a, a&b, a|b, a^b, a<<c, a>>c);
}
```

Operators of C and their precedence (1)

- (Left/Right) shows associativity of the operator.
- Postfix operators: (), [], ++, -- (Highest)
- Unary operators: +, -, *, &, !, ~
- Multiplicative operators (Left): *, /, %
- Additive operators (Left): +, -
- Shift operators (Left): >>, <<
- Relational operators (Left): >, <, >=, <=
- Equational operators (Left): ==, !=

Operators of C and their precedence (2)

- Bitwise and operator (Left): &
- Bitwise exclusive or operator (Left): ^
- Bitwise or operator (Left): |
- Logical and operator (Left): &&
- Logical or operator (Left): ||
- Conditional operator: ? :
- Assignment operator (Right): =, +=, −=, etc.
- Comma operator (Left): , (Lowest)

Precedence and Associativity of Operators

- Precedence and Associativity determines the evaluation order of expressions.
- Higher precedence operator is applied first
  - e.g. `a*b+c` is interpreted as `(a*b)+c`
- In case of same precedence, associativity specifies the operator to be applied first
  - e.g. `a/b*c` is interpreted as `(a/b)*c` (left associative)
  - e.g. `a=b=c` is interpreted as `a=(b=c)` (right associative)
- When you are not sure, use parentheses.
- **QUIZ:** Fully parenthesize the following expressions:
  - `a==b==c` `a&b==c` `a+b<<c`

Arithmetic Promotion

- When operating different type data, arithmetic promotion rules are applied.
- `char`, `short`, `unsigned char`, `unsigned short` types are first converted to `int` type.
- Other types are promoted according to the ordering of the next page to convert both operands into the same type.
- If you are not sure, use cast (explicit type conversion) to avoid mixed-type operations.

Arithmetic Promotion Rules

When `int` is 32 bits

double
↑
float
↑
unsigned long
↗ ↖
long unsigned
↖ ↗
int

When `int` is 16 bits

double
↑
float
↑
unsigned long
↑
long
↑
unsigned
↑
int

You don't have to remember this.

Make sure you don't use mixed-type arithmetic.

Cast Expression

- Cast expression converts an expression to the specified type:

```
long a;
(char)a;
```

Specifies the destination of conversion in parentheses

- Use cast expression to avoid implicit arithmetic promotions.

Arithmetic Overflow

- Unsigned arithmetic is computed modulo 2^{32}. No overflow occurs (guaranteed by the standard)
  - e.g. 0xffffffff+0xffffffff=0xfffffffe

- In signed arithmetic, the result is not guaranteed by the standard (but usual implementation computes modulo 2).
  - e.g. 2147483647+2147483647 -> overflow

Notes on Arithmetic

- Don't mix different types in arithmetic (types with different size, signed/unsigned)
- Assume that signed arithmetic overflow is not guaranteed.
- Don't apply logical operators to signed data.
- >> is not defined for negative signed integers in standard.
- There is a case when signed division overflows (namely, ((-2147483647)-1)/(-1))

Notes on Side Effects

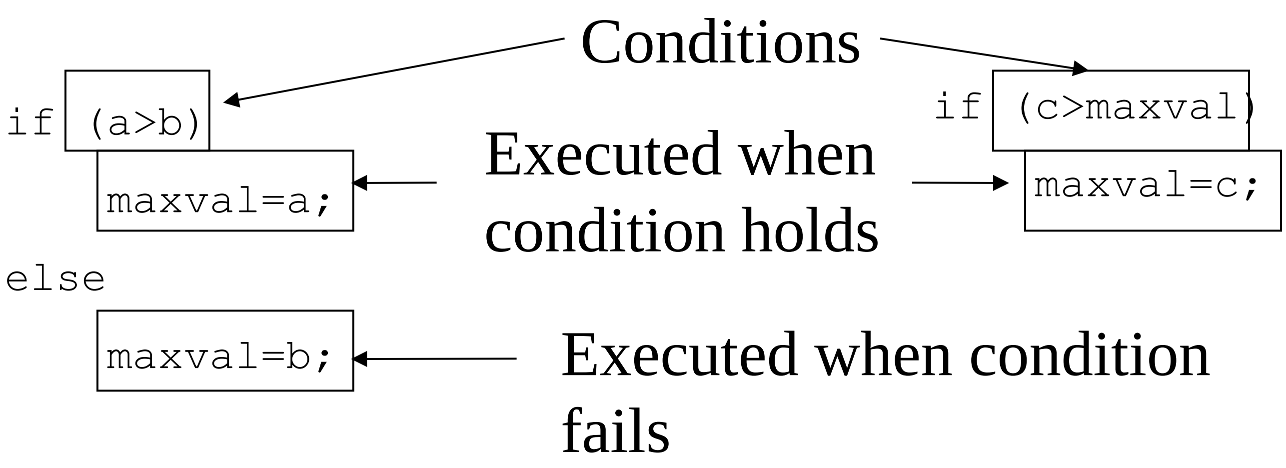
- The side effect occurs in assignment/increment/decrement/function call operations.
- The order of side effects is not specified inside a statement.
- Don't use more than one operations with side effects in one statement.
  - e.g. `a[i++]=b[i++];`

3. Statements

- Statements specifies the control of execution of basic statements.
- There are following kinds of statements:
  - Expression statements: `a=b;`, etc.
  - Conditional statements:
    - if statement, switch statement
  - Iteration statements:
    - while statement, for statement, do statement
  - Compound statement:
    - List of statements enclosed by `{ }`
  - Jump statements:
    - `goto, break, continue` statements.

`if` Statement

- The condition (inside `()`) is evaluated, and one of two statements is evaluated.
- If else `part` does not exist, no processing is done if the condition fails.



Example

```
#include <stdio.h>

int max(int a, int b, int c)
{
    int maxval;
    if (a>b)
        maxval=a;
    else
        maxval=b;
    if (c>maxval)
        maxval=c;
    return maxval;
}

main()
{
    printf("%d\n", max(3, 4, 5));
}
```

Combination of `if` statements

- `if` statement itself can be a part of another `if` statement.
- Following construct is useful for multi-way selection.

```
if (a==0)
  zero();
else if (a==1)
       one();
     else
       others();
```

Nested `if` statement

Data Type of Conditions

- There is no special data type for conditions. They are just `int`.
- 0 represents false, and all the other data represents true.
- Relational, equational, and other logical operators (`!`, `&&`, `||`) returns 0 for false, and 1 for true.

Notes on writing sub-statements

- A sequence of statements can be handled as one statement by enclosing them by `{}`.

The range of `if` statement

```
if (c>maxval)
    maxval=c;
    maxval1=c1;
```

```
if (c>maxval){
    maxval=c;
    maxval1=c1;
}
```

This statement is not dominated by `if`, and always executed (indentation is not the part of syntax).

switch Statement

- Switch statement selects according to the value.

```
switch (c){
case 'Y':
    printf("OK\n");
    break;
case 'N':
    printf("NG\n");
    break;
default:
    printf("Illegal Character\n");
    break;
}
```

Value to be tested

case labels specify where to start the execution

break statement exits the switch statement

default means that no case label matches

Example

```
#include <stdio.h>

main()
{
    int c;
    c=getchar();
    switch (c){
    case 'Y':
        printf("OK\n");
        break;
    case 'N':
        printf("NG\n");
        break;
    default:
        printf("Illegal Character\n");
        break;
    }
}
```

`switch` Statement

- `switch` statement evaluates the value, and starts execution from the matching `case` label in the compound statement.
- When no case label matches, the execution starts with `default` label. If `default` label does not exist, no processing is done.
- `switch` statements ends when the execution reaches end of the compound statement, or it encounters `break` statement.

Notes on `switch` Statement

- More than one case label can be specified in the same place.

```
switch (c){
case 0: case 1: case 2:
     small();
     break;
case 3: case 4: case 5:
     large();
     break;
default:
     break;
}
```

Notes on `switch` Statement

- If you don't write `break` statement, the execution fall through the `case` label.

```
switch (c){
case 0:
    a=1;
case 1:
    a=2;
    break;
default:
    break;
}
```

Quiz:
When c == 0, what is the value of a after execution.

Such a program is difficult to understand. Write `break` statement corresponding to each case label.

Notes on `switch` Statement

- Don't omit the `default` label, even if there is nothing to do in the default case.
- It is a good idea to check errors for unexpected cases.
- It is a good practice to explicitly specify that you do nothing in the "default" case.

`for` Statement

- for statement specifies a loop with initialization, termination condition, and what to do at the end of each step.

The loop continues while this condition holds.

Initialization

Step processing, executed after each step of the loop.

```
for ( i=0 ; i<10 ; i++ )
    printf("%d %d %d\n", i, i*i, i*i*i);
```

`i++` is increment operator, and has the same effect as `i=i+1`.

Loop body, repeated while the condition holds.

Example

```
#include <stdio.h>

main()
{
    int i;
    for (i=0; i<10; i++)
        printf("%d %d %d\n", i, i*i, i*i*i);
}
```

`while` Statement

- while statement specifies a loop with a termination condition.

Loop continues while this condition holds.

```
while ((c=getchar())!=EOF){
    if (c>='a' && c<='z')
        c=c-'a'+'A';
    putchar(c);
}
```

Body of the loop

Example

```c
#include <stdio.h>

main()
{
    int c;
    while ((c=getchar())!=EOF){
        if (c>='a' && c<='z')
            c=c-'a'+'A';
        putchar(c);
    }
}
```

`break` statement

- `break` statement is also used to exit `for`/`while` loops.

```
for (i=0; i<100; i++){
  if (a[i]==0)
    break;
  ...
}
```

exits the loop

RENESAS

Other Statements

- There are less frequently used other statement constructs:
- `do` <statement> `while (`<condition>`);`
  - Repeats <statement> while <condition> holds.
  - <statement> is executed at least once.
- `continue;`
  - Restarts the loop (goes back to the beginning of iteration).
- `goto` <label>`;`
  - Jumps to the program point specified by <label>
  - Don't use `goto` statements.

4. Data Structures

- Arrays define a homogeneous (same type) set of data.
- Structures define a heterogeneous (different type) set of data.
- Unions define a set of data sharing the same memory location.
- Pointers allow links between data.

Arrays

- `char buf[81]` declares an array of `char` whose number of elements (array size) is 81.
- Array index is 0-based, i.e. the first member is `buf[0]` and the last member is `buf[80]`.
- An array element is referenced as `buf[i]`, where `i` is the index of the array.
- Make sure that array index is within the array range.

Example-1

```
#include <stdio.h>

char buf[81];

void getline(void)
{
    int i, c;
    i=0;
    while ((c=getchar())!='\n'){
        buf[i]=c;
        i++;
    }
    buf[i]='\0';
}
/* to be continued */
```

Example-2

```
/* continued */
main()
{
    int i;
    getline();
    i=0;
    while (buf[i]!='\0'){
        putchar(buf[i]);
        i++;
    }
    putchar('\n');
}
```

Memory Allocation of an Array

- Array elements are allocated consecutively in memory.

```
int a[81];
```

a[0]	a[1]	a[2]	...	a[80]

Multidimentional Arrays

- `int a[5][10]` declares an two-dimentional (5*10) array of `int`.
- Multidimentional array is referenced like `a[i][j]`.
- This declaration is interpreted as "array of 5 (array of 10 `int`)".

Multidimentional Array Allocation

- `int a[5][10]` is allocated in the following way.
- The last index changes the fastest (Row Major)

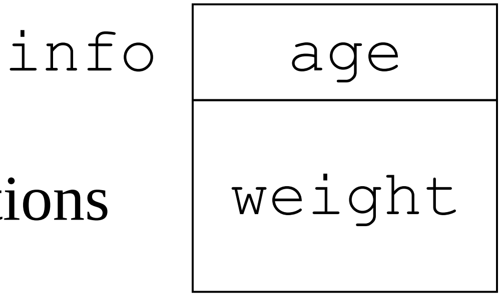
a[0][0]	a[0][1]	a[0][2]	...	a[0][9]
a[1][0]	a[1][1]	a[1][2]	...	a[1][9]
:				
a[4][0]	a[4][1]	a[4][2]	...	a[4][9]

Structures

- Structure declares a type with a set of component members (`struct { ... }` specifies a type, just like `int`).
- The struct members are allocated consecutively (except gap).
- Struct members are referenced like `info.age`, `info.weight`, etc.

```
struct{
    int age;
    double weight;
} info;
```

Member declarations



Example

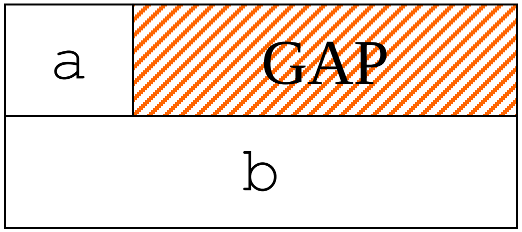
```
#include <stdio.h>
struct{
    int age;
    double weight;
} info;

main()
{
    int a;
    double w;
    scanf("%d", &a);
    while (a>=0){
        scanf("%lf", &w);
        info.age=a;
        info.weight=w;
        printf("age: %d weight: %f\n", info.age, info.weight);
        scanf("%d", &a);
    }
}
```

Data Alignment

- Some data types should be aligned.
- For SH, `short` must be aligned to 2-byte boundary, and `int`, `long`, `float`, `double` must be aligned to 4-byte.
- This may cause data gaps in structures.
- Consider gaps to reduce data size.

```
struct{
    char a;
    int b;
} info;
```

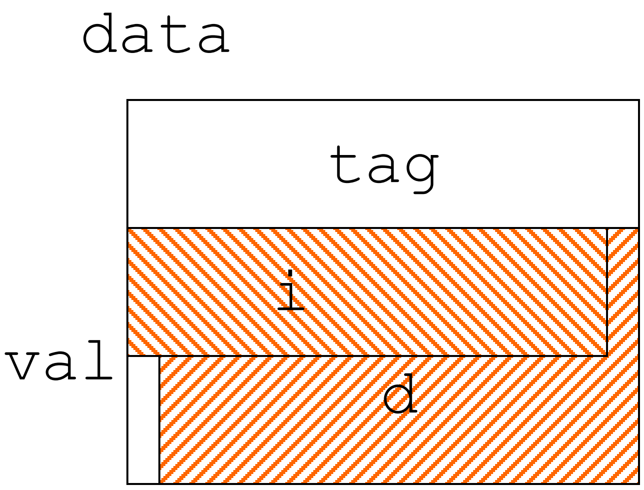


Unions

- Unions is like a structure, but the members are allocated at the same address and shares memory area.

```
struct{
    int tag;
    union{
        int i;
        double d;
    } val;
} data;
```

Member declarations



Example-1

```
#include <stdio.h>
struct{
    int tag;
    union{
        int i;
        double d;
    } val;
} data;

/* to be continued */
```

Example-2

```
main()    /* continued */
{
    char c;
    scanf("%c", &c);
    while (c=='i' || c=='d'){
        if (c=='i'){
            data.tag=0;
            scanf("%d", &data.val.i);
        }
        else{
            data.tag=1;
            scanf("%lf", &data.val.d);
        }
        if (data.tag==0)
            printf("int value: %d\n", data.val.i);
        else
            printf("double value: %f\n", data.val.d);
        scanf(" %c", &c);
    }
}
```

Notes on Union

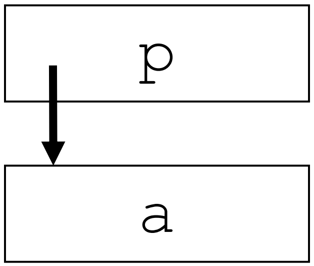
- The union data should be references with the same member name as it is assigned. Don't access union members through a different member name.
- It is recommended to have some member that remembers which member of the union has been stored (`tag` in this case).

RENESAS

Pointers

- Pointer data holds memory address, but it must be declared to point to some specific data type.
- You can take address of any data in memory by `&` operator.
- You get the contents of the pointer by `*` operator.

`int *p;`	Declares that `p` points `int`.
`int a;`	
`p=&a;`	`p` points `a`.
`a=1;`	
`printf("%d\n", *p);`	Prints the value of `a` .



Example-1

```
#include <stdio.h>
#include <stlib.h>
struct node{
  int val;
  struct node *next;
} *root;

/* to be continued */
```

Example-2

```
/* continued */
main()
{
    int i;
    struct node *p;
    root=NULL;
    scanf("%d\n", &i);
    while (i>=0){
        p=malloc(sizeof(struct node));
        p->val=i;
        p->next=root;
        root=p;
        scanf("%d", &i);
    }
    p=root;
    while (p!=NULL){
        printf("%d\n", p->val);
        p=p->next;
    }
}
```

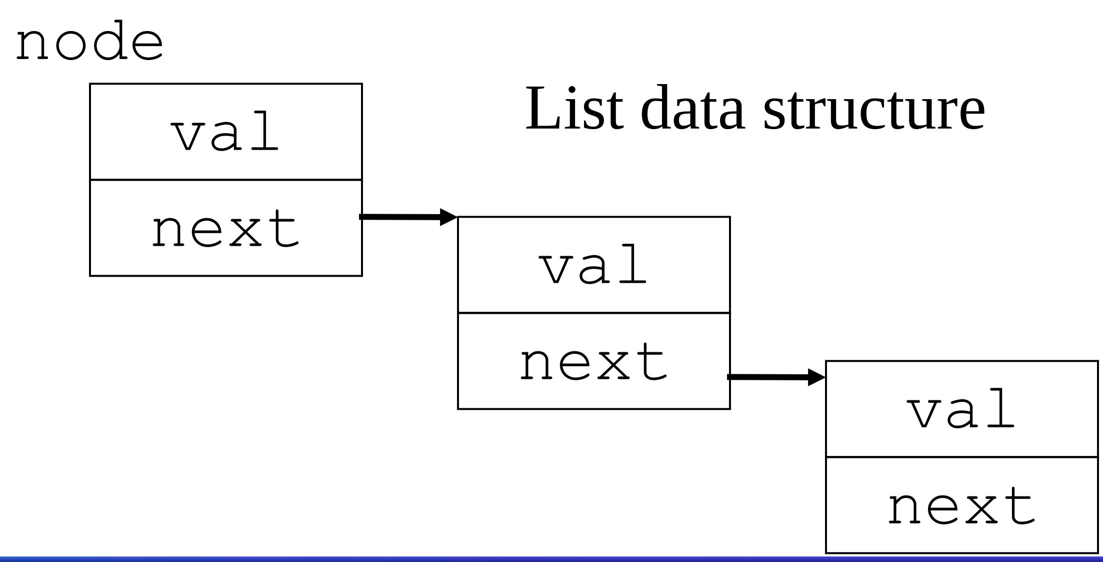
Notes on Pointers

- Pointers must be initialized with some address, otherwise, it points to indefinite location. Don't use uninitialized pointers.
- `NULL` is the pointer constant with value 0 (defined in `<stdlib.h>`: basic library). We have a convention that "`NULL` points to nothing".
- Don't de-reference `NULL` pointers.
- Misuse of a pointer may destroy unexpected memory area.

Self Referencing Structures

- Structure cannot include the same structure, but can include the pointer to the same structure. This is used to define useful data types.
- To include the reference to itself, the structure needs the name.

```
struct node{
  int val;
  struct node *next;
} *root;
```



List data structure

Structure Names

- To define self-referencing structures, structure must be named.
- `struct node{ ... }` gives the name `node` to the structure.
- Once you give a name to a structure, you can declare the same structure without specifying members.
- Structure name is necessary when you declare self-referencing structure.

```
struct s{
    int x, y;
};
```

Declaration of a structure

```
struct s a, b;
```

Declaration of variables with the type `struct s`.

Operator ->

- Operator `->` refers to the member pointed by a pointer to a structure. This operator is very convenient when handling pointers to structures.
- `a->b` can be considered as an abbreviation of `((*a).b)`.

More Data Structure Example

```
struct tree_node{
  int val;
  struct tree_node *first, *second;
} *root;
```

Quiz:

Figure to illustrate the data structure ???

Pointer Arithmetic

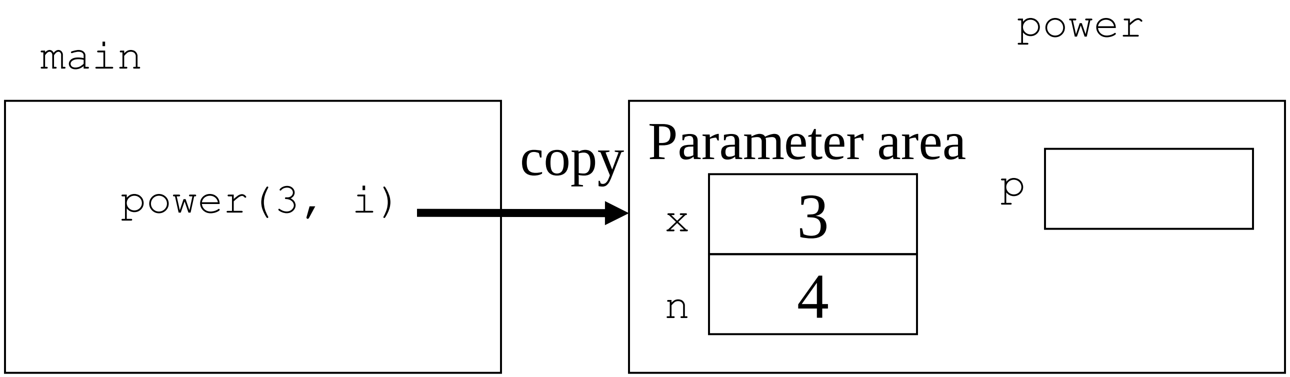
- When adding/subtracting integer to pointer, the integer is multiplied by the size of the pointed data. e.g. `long *p; p+1` adds 4 to the address value `p`.
- When pointer to the same type is subtracted, the difference is divided by the size of pointed data.
- Thus, `p[i]` is an abbreviation of `*(p+i)`.
- The same rule applies to `++` and `--`.
- **QUIZ**: If "`p`" with declaration "`long *p;`" has the value 0x1000, what is the value of "`p+20`"?

5. Function Interface

- Function parameters are copied and passed to the function.
- Modifying parameters doesn't affect the original parameters.

Function Execution Model

- Function parameters are copied (to registers or stack) and passed to the function.
- A function's local data are allocated to the stack, and the area is deallocated when the function exits.



Example

```
#include <stdio.h>

int power(int x, int n)
{
    int p;
    p=1;
    while (n>0){
        p=p*x;
        n--;
    }
    return p;
}

main()
{
    int i;
    i=4;
    printf("%d\n", power(3, i));
}
```

Pointer Parameters

- Use pointer parameters in the following cases:
  - Data to be passed is too big and copying is not efficient.
  - You want to change the data in the calling function by the called function.
  - You want to pass a function as a parameter (next example).

Example

```c
#include <stdio.h>

void swap(int *p, int *q)
{
	int temp;
	temp=*p;
	*p=*q;
	*q=temp;
}

main()
{
	int i, j;
	i=1;
	j=2;
	swap(&i, &j);
	printf("%d %d\n", i, j);
}
```

Function Parameters

- Pointer to functions can be declared as data, or passed as parameters.
- The function pointed can be called by using `*` operator (e.g. `(*p)(i)`).

Example

```
#include <stdio.h>

int square(int n)
{
    return n*n;
}

int sum(int (*p)(int))
{
    int i, s;
    s=0;
    for (i=0; i<10; i++)
        s=s+(*p)(i);
    return s;
}

main()
{
    printf("%d\n", sum(&square));
}
```

Function Prototypes

- Function prototype is a declaration of a function (without definition) specifying parameter types.
  e.g. `int f(int, int *);`
- Declare function prototypes to check function caller-callee interface.
- When calling a function before it is defined, insert function prototype declaration before its use.
  - C compiler gives the default type to the function, and it may conflict with the prototype declaration.

Homework

Exercise 1:

What is the value of y after executing the following code?

```
  int x, y;
  x=2;
  y=1;
  switch (x){
  case 1: x++;
  case 2: x++;
  case 3: y++;
  }
```

Exercise 2:

What is the size of the following struct?

```
struct {
  char a, b;
  int c;
  float d;
}info;
```

(suppose that int type has 2-byte size and
the program is executed on 32-bits machine (4 bytes alignment))

6. Compilation Units and Preprocessing Directives

- Compilation Units consists of data or function declarations and definitions.

- Preprocessing directives specifies textual operations (macro expansion, file inclusion, etc) to the source program.

Declarations and Definitions

- Declarations give types to variables or functions.
  - Variable declaration:
    - `extern int a;`
  - Function declaration:
    - `extern int f(int);`
- Definitions define the contents of variables or functions.
  - Variable definition:
    - `int a = 10;`
  - Function declaration:
    -
```
int f(int){
  return x;
}
```

Notes on Declarations and Definitions

- The keyword "`extern`" specifies a declaration. The contents of data or function should be defined elsewhere (maybe in a different compilation unit).

- There should be only one "definition" of a variable or a function in a programming project.

- The keyword "`static`" specifies that the definition is local to the compilation unit, and cannot be used outside its compilation unit.
  - e.g. `static int a;`

`static` Declarations

- When declared with `static` keyword (instead of `extern`), the function or variable is local to the file, and cannot be accessed from other files.
- Use `static` to define local variables and functions.

Preprocessing Directives

- Preprocessing directives gives a command to the compiler.
- `#define` defines a macro.
- `#ifdef` selects the compiled portion of the program.
- `#include` allows file inclusion.
- The grammar of directives are independent of C language syntax, and C syntax applies after the directives are processed.
- Compiler directives are processed line by line. The line starting with `#` introduces a compiler directive.

Macro Definitions

- `#define` defines a macro with or without parameters.
- `#define` name text defines a parameterless macro, which indicates that text replaces name.
- `#define` name`(`parameters`)` text defines a macro with parameters, and parameter names in the text is replaced by the actual parameters in the macro call.

Example

```
#include <stdio.h>

#define PI 3.14

#define AREA(r) (PI*(r)*(r))

main()
{
    printf("%f\n", AREA(2.0));
}
```

Macro Replacement

```
AREA(2.0)
   ↓          Macro AREA is applied
(PI*(2.0)*(2.0))
   ↓          Macro PI is applied
(3.14*(2.0)*(2.0))
```

Notes on Macros

- When declaring macros with parameters, （ (the opening parenthesis starting macro parameters) must be written immediately after the macro name.
- Enclose macro parameters and replacement text by parentheses to avoid unexpected interpretation

```
#define X 1+2
…
Y = X*3
…
```

Quiz:
What is the value of Y ?

Notes on Macros

- Don't specify an expression with side effects in a macro call.
- The macro definition might use the parameter more than once, in which case, the side effect might be executed more than once.
- The number of side effects might vary when the macro definition is modified.

```
#define MAX(x, y) ((x)>(y)?(x):(y))
…
return MAX(a++, 1);
…
```

Expanded into
`((x++)>(1)?(x++):(1))`
`x++` might be executed more than once.

Macros vs Functions

```
int abs(int x){
  return x>=0 ? x : -x;
}
f(){
  a=abs(b);
  c=abs(d);
}
```

→

```
#define abs(x) \
  ((x)>=0 ? (x) : -(x))
f(){
  a=abs(b);
  c=abs(c);
}
```

Macros don't have function call overhead.

But extensive use of macros make your program size very large.

C++ and C99 provides `inline` function declarations.

Conditional Compilation

- Conditional compilation can select the part of the program to be compiled.
- The part enclosed by `#ifdef` name ... `#endif` is compiled only if the name is defined.
- Useful to keep track with software versions, or insert debugging statements.

RENESAS

Example

```
#include <stdio.h>

//#define DEBUG

sub(int i)
{
#ifdef DEBUG
    if (i<0)
        printf("Bad Argument: %d\n", i);
#endif
    printf("%d\n", i*i);
}
```

Quiz:

Which above statements will be compiled ?

File Inclusion

- `#include "filename"` includes user-defined file (searches from the same directory as the compiled file).
- `#include <filename>` includes system-defined file (searches from the system directory).

Example

file1: def.h

```c
extern int a;
extern void sub(void);
```

file2: main.c

```c
#include "def.h"
int a;

main()
{
     sub();
}
```

file3: sub.c

```c
#include <stdio.h>
#include "def.h"

void sub(void)
{
     a=1;
     printf("%d\n", a);
}
```

Dividing a Project into Files

- Keyword `extern` specifies a declaration without defining an object/function.
- Put `extern` declaration in a common header (`.h`) file.
- Each C (`.c`) file includes the common header file.
- Make sure that each object/function is defined in exactly one of the C files.

7. Advanced Topics

- Declaration of complex data types
- `const` and `volatile`
- `enum`

How to Declare Pointer to a Function

- `int (*p)(int)` declares a pointer to a function (which receives `int` parameter).
- This can be interpreted as follows:
  - `int X(int)` declares `X` to be a function receiving `int` parameter.
  - Replacing `X` by `(*p)` (note the precedence) declares `*p` to be a function receiving `int` parameter.
  - This shows that `p` is a pointer to a function receiving `int` parameter.

Other Complex Declarations

- `int *p(int)` (interpreted as `int *(p(int))`) declares `p` as a function (receiving `int` parameter) which returns a pointer to `int`.
- `int (*p)[5]` declares `p` as a pointer to an array of 5 `int`'s.
- `int *p[5]` declares `p` as an array of 5 pointers to `int`.
- All these interpretations can be verified by considering how `p` is used in an expression.

Declaration of Types

- Types are recursively defined as follows:
  - Basic types (int, long, short, char, float, double, struct, union, etc.) are types.
  - Let T be a type, then "pointer to T" is a type.
  - Let T be a type, then "n element array of T" is a type.
  - Let T be a type, then "function returning T" is a type (actually, parameters must be considered, but here we use simple version).
- Example:

  10 element array of (pointer to (function returning `int`))

Declaration and type

- Type declaration can be recursively derived as follows:
  - If T is declared as ... T ..., then PT: "pointer to T" is declared as ... (*PT) ...
  - If T is declared as ... T ..., then AT: "n element array of T" is declared as ... (AT[n]) ...
  - If T is declared as ... T..., then FT: "function returning T" is declared as ... (FT()) ...
- Example:
  - FI: function returning int: `int FI();`
  - PFI: pointer to (function returning int): `int (*PFI)();`
  - APFI: 10 element array of (pointer to (function returning int)):

```
int (*(APFI[10]))();
```


RENESAS

Declaration of types: Example

- Another way to find declaration of complex type is consider what expression gives you the basic type from the type to be declared.
- To declare a variable `p` to be of type 10 element array of (pointer to (function returning `int`))
  - `p` is 10 element array of (pointer to (function returning `int`))
  - `p[i]` is pointer to (function returning `int`)
  - `*p[i]` (`*(p[i])` when fully paranthesized) is function returning `int`
  - `*p[i]()` is `int`
  - `p` is declared as `int *p[10]();`
- Apply type constructor to the declared name in the reverse order of its occurrence in the description.

Construction of Type Declaration with `typedef`

- You can do type construction step-by-step using `typedef`

```
typedef int FI(void);     /* FI is function returning int */
typedef FI *PFI;          /* PFI is pointer to FI        */
typedef PFI APFI[10];     /* APFI is 10 element array of  */
                          /* PFI                          */
APFI p;
```

`const` and `volatile`

- `const` data cannot be assigned.
  - Usually used for ROM data, but also can be used for parameters/local variables/structure members, so that compiler can check if they are modified.
- `volatile` data are guaranteed to be loaded and stored from/to memory whenever they appear in the program (i.e. compiler doesn't optimize load/store).
  - Used for I/O registers, and data which might be modified by interrupt processing.
- A data can be `const` and `volatile` at the same time (e.g. timer register)

`const` (`volatile`) pointers

- `const int i=10;` declares `i` to be read-only `int` data.
- `const int *pci;` declares `pci` to be a pointer to `const int`.
  - `pci` itself can be modified
  - `*pci` (the data which `pci` points to) cannot be modified.
- To declare non-modifiable pointer-to-`int` data, use `int * const cpi=&i;`
  - "`* const`" is a type constructor to build constant pointers.
- `volatile` and `const volatile` has similar syntax.

`enum` types

- Enumeration declares enumerated data type.
  - `enum DAY {sun, mon, tue, wed, thu, fri, sat};`
- Enumeration members can be used whenever `int` can be used.
- Each member are assigned consecutive value starting from 0 (`sun`=0, `mon`=1, ..., `sat`=6, etc.).
- Use `enum` type for enumerated items for which assignment of value is arbitrary (like `enum DAY` above).
- Usage of `enum` is more readable than using small numbers (0, 1, 2, ...) directly.

II. Some notes in C embedded programming

Contents

1. Introduction

- Basically, C programming in PC is similar to C programming in embedded software, except a few additional necessary knowledge;

  - Correspondence between C program and ROM/RAM
  - Program Startup and Interrupt Handling
  - Accessing Hardware
  - Linkage with assembler program

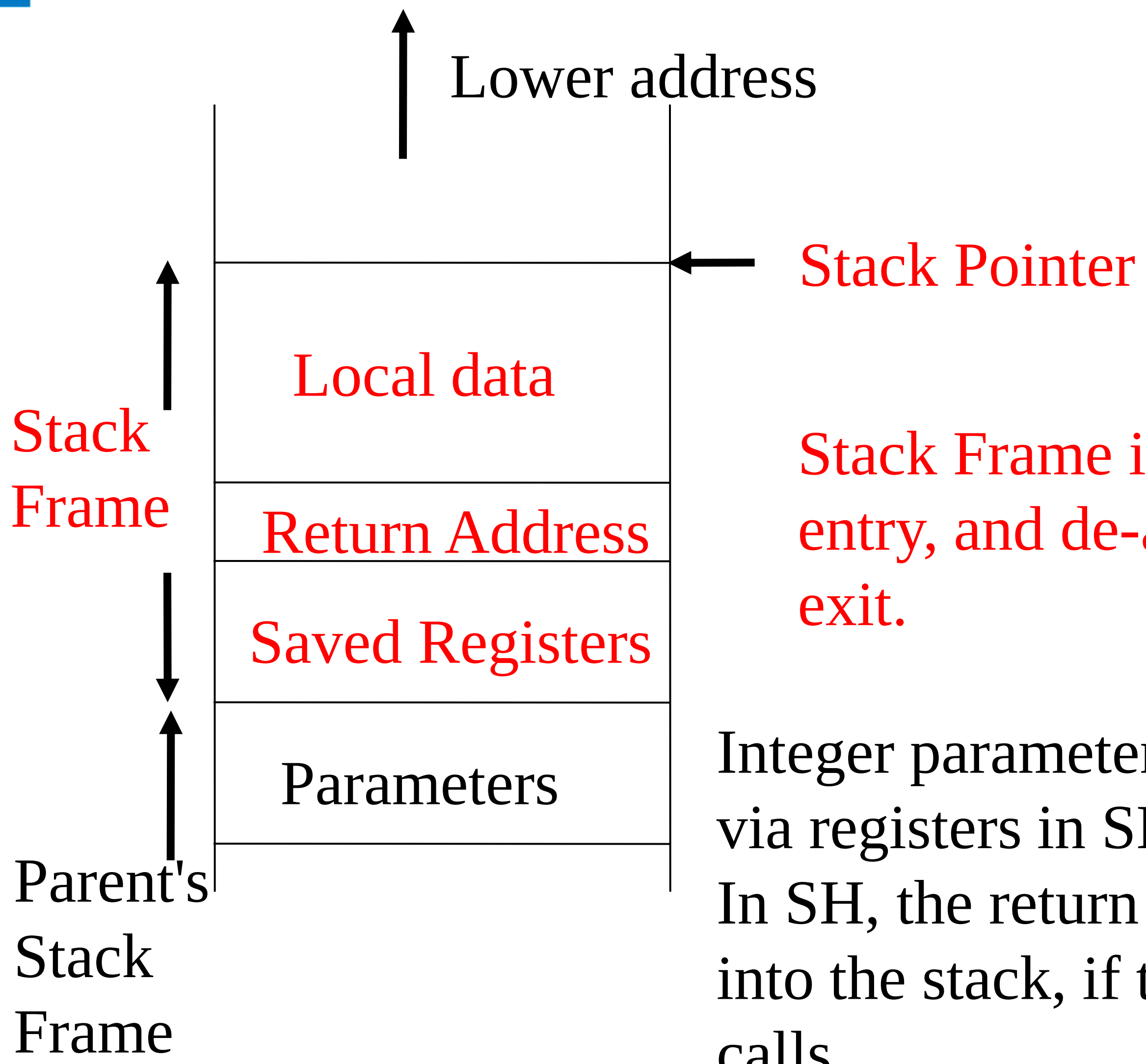
Difference of Embedded Programs from PC Programs

- Explicit memory configuration (ROM, RAM, I/O registers)
- Initialization (from RESET to main)
- No stdio (unless you write it yourself)
- Processing is driven by interrupts
- Program should run permanently (higher quality, error recovery required)

2. C Program Memory Model

- C program uses following kinds of memory areas:
  - Program code (initialized, read only)
  - Constant data (initialized, read only)
  - Initialized data (initialized, read/write)
  - Uninitialized data (uninitialized, read/write)
  - Stack (used for function-call interface, parameters, and local data)
  - Heap (managed by library functions: malloc, etc.)

Usage of Stack



Local data are accessed using R15 (Stack Pointer) as a base register.

Stack Frame is built on function entry, and de-allocated on function exit.

Integer parameters (up to 4) are passed via registers in SH.
In SH, the return address may not be pushed into the stack, if there is no further function calls

3. Sections

- Sections are re-locatable (i.e. can be placed anywhere in the memory) unit of program or data.
- Each C program compilation unit generates 4 kinds of sections.
- The same kind of sections from several compilation unit are linked together in a contiguous memory area.

Section Attributes

	Section name	Attribute	Initialization	Memory
`int a;` →	BSS	R/W	Zero	RAM
`int b=1;` →	Data	R/W	Initialized	RAM
`const int c=1;` →	Const	R	Initialized	ROM
`main(){ ... }` →	Text	R	Initialized	ROM

BSS is an abbreviation of "Block Storage Segment"

`const` type

- `const` keyword specifies that the data cannot be assigned.
- The data with `const` type can be placed in ROM.
- `const int *p;` declares that `p` points to constant area, but `p` can be assigned. To declare const pointer, use declaration

```
int *const p;
```

Data Section and its Initialization

- Data section has its initial value, but the variables in data section can be modified.

- To implement this, data section must be allocated in RAM, but its initial value must be copied from ROM at program startup.

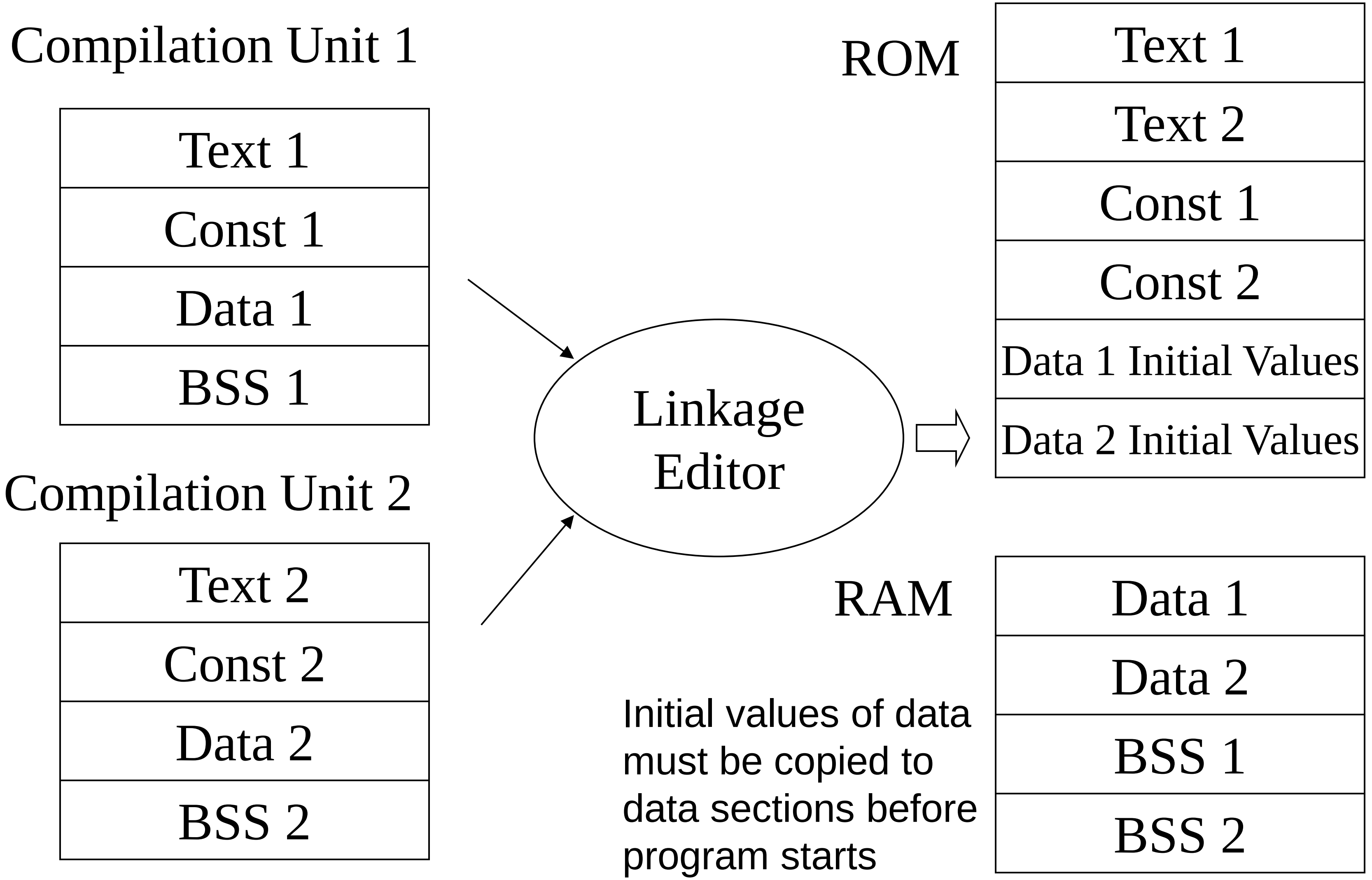
Functions of Linkage Editor

- The linkage editor collects the sections of the same name from several compilation units, and allocates them in a contiguous area.

- The linkage editor resolves the references to variables/functions by the allocated addresses.

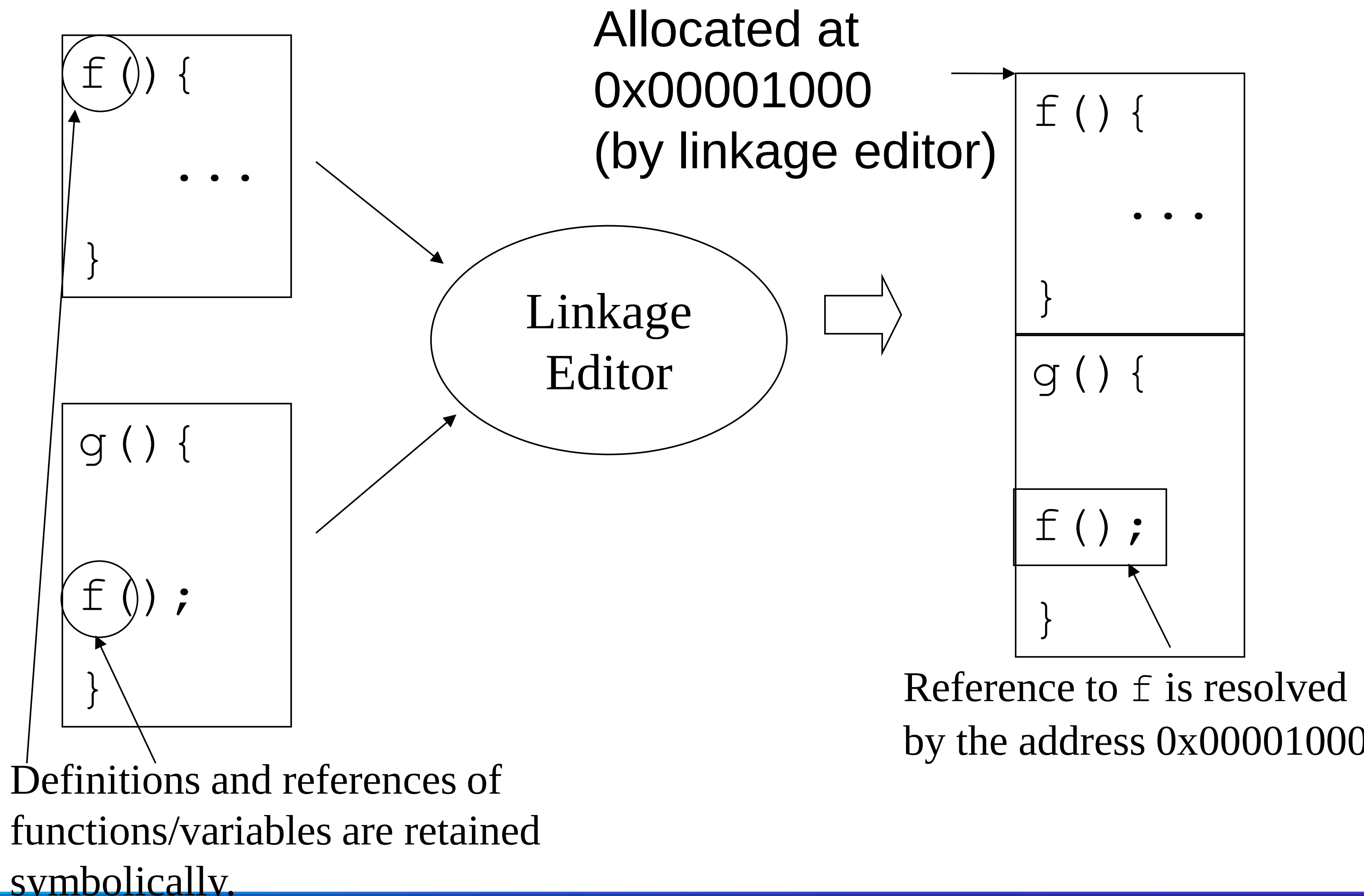
RENESAS

Allocation of Sections



Initial values of data must be copied to data sections before program starts

Resolving References



Assignment of Absolute Address to Sections

- To start up a program, you need to assign fixed absolute address to the entry point of the program.
- Linkage editors can assign absolute address to a section (e.g. `START sec1(100)` allocates the section `sec1` to the address 100 in Renesas Linkage Editor).
- Locate the entry point at the beginning of a compilation unit.
- Link the module including the entry point as the first member of a section.
- Use linkage editor command to allocate the section to the specific address.

RENESAS

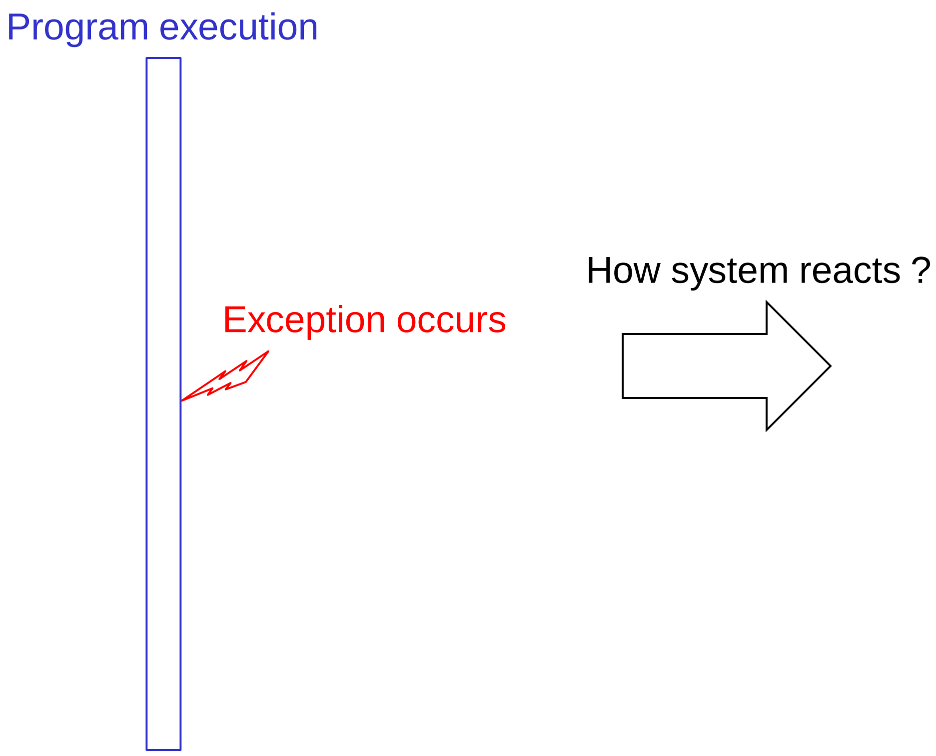
4. Program Startup and Interrupt Handling

- The following assumes SH4 architecture.

- Program starts at the address 0xa0000000.
  - Check exception event, and jump to appropriate routine.
  - The exception event is power-on-reset, go to overall initialization.

- When interrupt, the program goes to the address VBR+0x600.
  - Check interrupt event and call appropriate interrupt handler.

Reset Handler: concept



Check cause of exception

↓

Get reset handler address (vector)

↓

Calculate the appropriate addr.

↓

Jump to “exception” processing program

Reset Handler

```
_ResetHandler:                          ; Located at 0x80000000
mov.l   #EXPEVT,r0                      ; save the cause of exception
mov.l   @r0,r0
shlr2   r0
shlr    r0
mov.l   #_RESET_Vectors,r1              ; get reset vector
add r1,r0
mov.l   @r0,r0                          ; calculate "appropriate address"
        mov.l   #_INIT_SP,r15           ; Initialize Stack Pointer
jmp @r0                                 ; jump to "appropriate address"
nop
```

This program gets the cause of exception (stored at the label EXPEVT, and jump to appropriate address using the table _RESET_Vectors

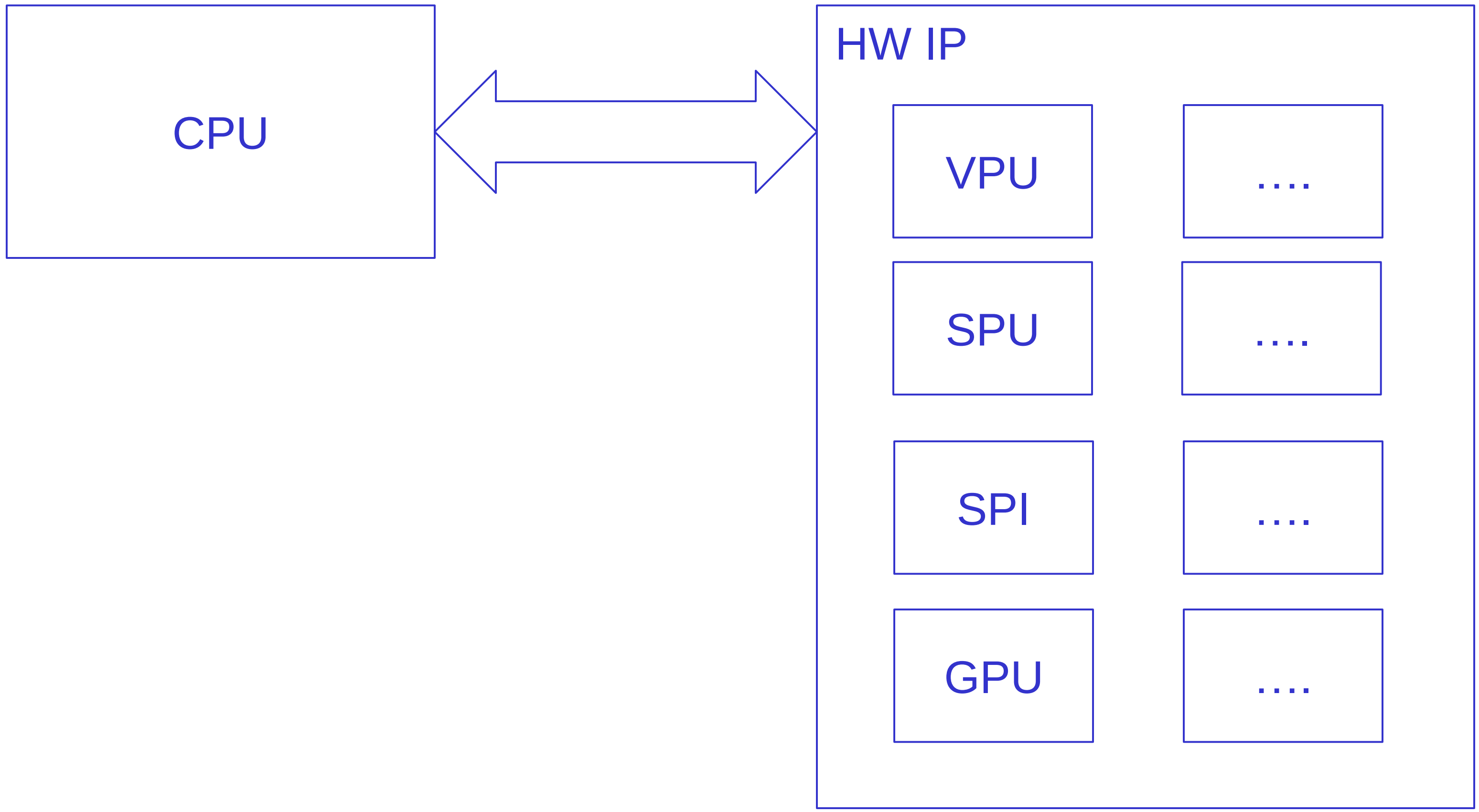
Initialization Program

```
void PowerON_Reset(void)
{
       /* Set Vector Base Register */
set_vbr((void *)((_UINT)INTHandlerPRG - INT_OFFSET));
_INITSCT();
_INIT_IOLIB();     /* Initialize library */
    HardwareSetup();   /* Setup Hardware     */
set_cr(SR_Init);        /* Set CR (be user mode) */
        nop();
main();                 /* Initialize applications */
sleep();                /* Sleep to wait interrupt */
}
```

5. Accessing Hardware

- Memory-mapped registers (I/O ports) can be accessed via pointers using their absolute addresses.

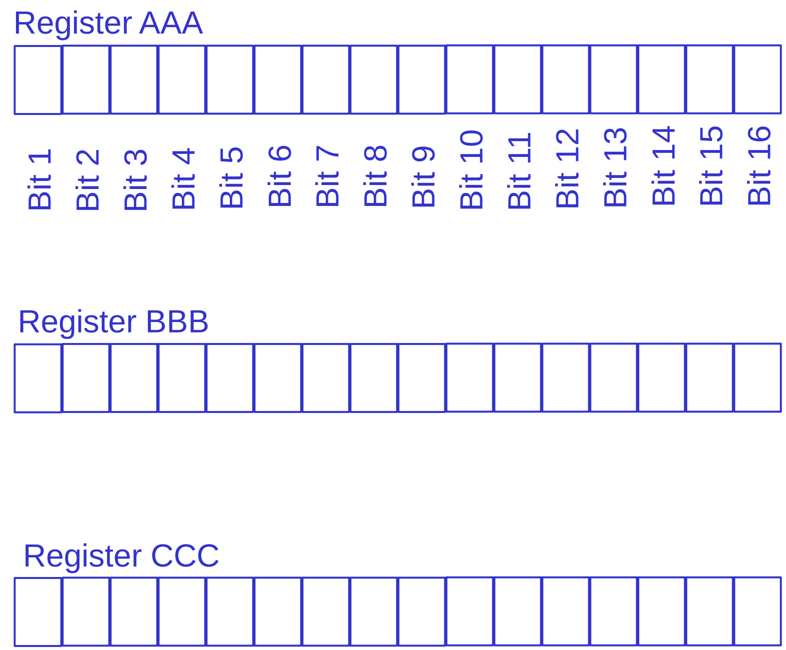
Concept: HW structure



Concept: HW IP

VPU

VPU HW IP has many registers.
In order to control VPU HW operation as our expectation, we need to read and/or set each bit in each register.
Each bit will have its own purpose and meaning, which were described in HW manual.



…

So, our need-to-learn thing is how to set and/or read each bit by C programming.

I/O Port Definitions

```
/* Timer Register                                      */
struct tcsr{
        unsigned short OVF:  1;    /* OVF  bit         */
        unsigned short WTIT: 1:    /* WTIT bit         */
        unsigned short :     3;    /* don't care       */
        unsigned short CSK2: 1;    /* CSK2 bit         */
        unsigned short CSK1: 1;    /* CSK1 bit         */
        unsigned short :     9;    /* don't care       */
};
#define TCSR_FRT (*(volatile struct tcsr *) 0x5FFFFB8)
...
```

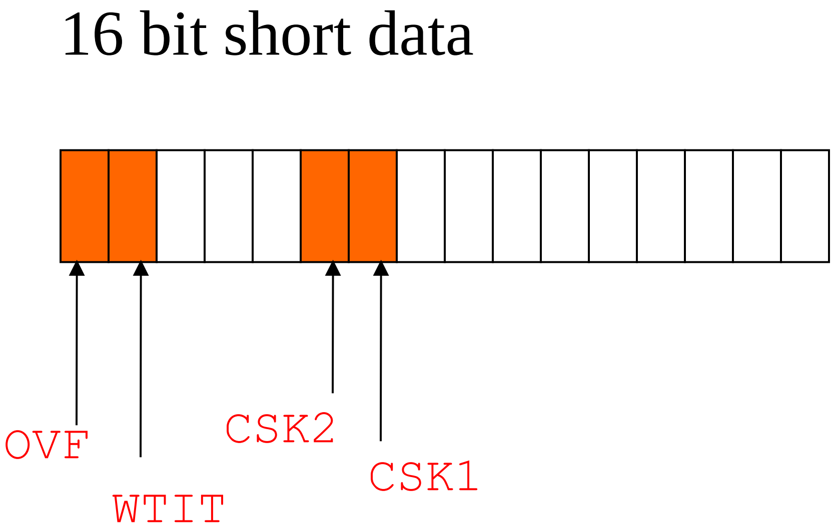
This struct definition gives the data structure of I/O port.

#define gives the name to the (absolute) port address.

Bit Fields

- Structure members declared with `:d` (d is the number of bits) are bit fields, and consecutive members are packed into its data size.
- Unnamed member indicates a gap in the bits.

```
struct tcsr{
        unsigned short OVF:  1;
        unsigned short WTIT: 1:
        unsigned short :     3;
        unsigned short CSK2: 1;
        unsigned short CSK1: 1;
        unsigned short :     9;
};
```



`volatile` type

- The keyword `volatile` defines a data attribute (type) which requests that the compiler don't optimize the variable.
  e.g.
```
volatile int a;
a=1;
a=2;
```
  Compiler can eliminate the first assignment as an optimization. But such an optimization is not allowed for volatile type variables.

- `volatile` must be specified to declare I/O registers.

Using Absolute Address to Access Hardware Registers

- `x=TSCR_FRT.OVF` reads `OVF` bit.
- `TSCR_FRT.OVF=1` sets `OVF` bit to 1.

- The expression
  `*(volatile struct tcsr *)0x5FFFFB8`
  can be interpreted as "convert absolute address 0x5FFFFB8 to a pointer to `volatile struct tcsr`, and access its contents".

- So the macro TSCR_FRT behaves just like a variable with type `volatile struct tcsr`.

III. Structured Program Design

Contents

1. Module Decomposition
2. Implementing Modules in C
3. Good Programming Practice

1. Module Decomposition

When developing a large scale program, first decompose the program into functional unit.

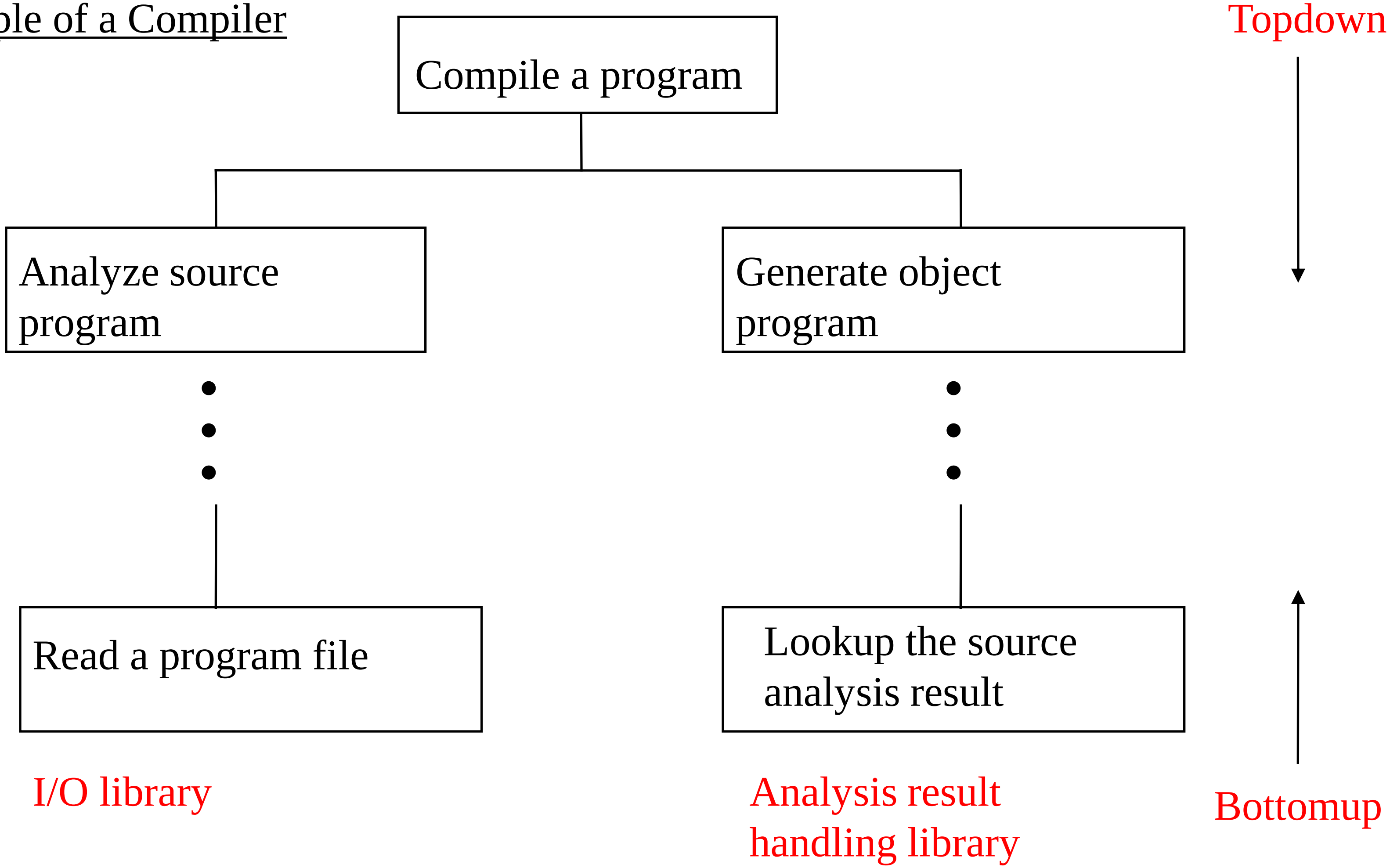
(1) A "Functional Unit" is a unit which you can state its function in one sentence. You should not decompose a program by execution sequence until you start detailed design.

(2) The "Top-down Design" decomposes the total program into smaller functional unit.

(3) The "Bottom-up Design" designs basic set of functions, like I/O and handling of common data structures.

Sample Module Decomposition

Example of a Compiler



Topdown Design (Example)

Prepare a header file for each module.
(Don't put all the declaration in a single header file)

main.c

```
#include "defs.h"
#include "anlyze.h"
#include "gen.h"
main(){
  anlyze();
  generate();
}
```

defs.h

Declares variables common to all the modules

anlyze.h

```
void anlyze(void);
```

generate.h

```
void generate(void);
```

anlyze.c

Definition of `analyze`

generate.c

Definition of `generate`

Sample Bottomup Design

anlyze.c

```
#include "io.h"
#include "tbl.h"
...
read_line();
...
enter_tbl();
```

Module calling library functions

I/O library

io.h

```
void read_line(void);
void write_line(void);
```

io.c

Definitions of `read_line`, `write_line`

Table handling library

tbl.h

```
void enter_tbl(void);
void remove_tbl(void);
```

tbl.c

Definitions of `enter_tbl`, `remove_tbl`

2. Implementing Modules in C

When programming in C, header (`".h"`) files (interface modules) and `".c"` files (definition modules) are considered as modules.
Some modules don't have definition (e.g. modules with only #define directives).

Structure of Modules

(1) Header files (include only the declarations required to use the module)
  (a) `#define` directives required to use the module.
  (b) Type declarations (`struct`, `union`, `typedef`) required to use the module.
  (c) `extern` declarations of variables required to use the module.
  (d) Function declarations required to use the module.
(2) ".c" files
  (a) `#include` of the header files used in the module.
  (b) `#include` of the header of the module itself.
  (c) Local `#define`, `struct`, `union`, `typedef`s of the module.
  (d) `static` declarations of its local variables, and functions.
  (e) Definitions of its variables and functions.

Sample Module Implementation

stack.h

```
#define STK_SIZE 256

extern void init(void);
extern void push(int);
extern int pop(void);
```

Declare all the internal variables and functions as static. Header files should include minimum interface.

stack.c

```
#include "stack.h"

static int sp;

static int stack[STK_SIZE];

void init(void){
   sp=0;
}

void push(int i){
   ...
}
...
```

Independence of Modules

A large scale program includes a lot of definitions and references to them. To implement efficient development process, it is important to:

(1) Make definitions and references local to modules.
(2) Minimize references between different modules.
i.e. make modules as independent as possible.

Quiz: Why do we need to make modules as independent as possible ?

<u>Recommended Practice</u>

(1) Decompose program so that a module implements a function which can be stated in one sentence.
(2) Declare functions and variables local to the module with `static` keyword.
(3) Make a header file for each module. Don't put everything in one header file.
(4) Don't include unnecessary header files.
(5) Design re-usable library modules.

RENESAS

Module Interface (Header Files)

Header files have two roles.

(1) A "specification" to define module interface.
(2) A mechanism with which a compiler checks the interfaces between modules.

Recommended Practice

(1) First write down the header file before writing the definition module.
(2) Use header files to review module interface.
(3) Don't change header file without reviewing with other team members.
(4) Refer to external functions/variables only through declarations in the header file.

Problems of Bypassing Header File Definitions

def.h

```
extern int x;
```

def.c

```
int x;
```

Reference using header file declaration (Recommended)

```
#include "def.h"

x=1;
```

Reference bypasssing header file declaration (bad)

```
extern int x;

x=1;
```

When the type of x has been changed, the reference will be illegal. The compiler cannot check the type incompatibility.

3. Good Programming Practice

- Programs should be easy to develop, easy to understand, easy to debug, and easy to maintain.
- Simple is the best.
- Don't try optimizations before making your program correct.
- Documentation is very important.

Comments

- Comment at the beginning of every file (header comment), describing:
  - The project name
  - The name and the function of the module
  - Author
  - Revision history
- Comment on every data, describing its meaning.
- Comment on every function, describing:
  - The name of the function.
  - The function of the "function".
  - Meaning of its input (parameters) and return value.
  - The files and tables used or modified by the function.
- Comment on what you do, not on how (how you do it must be clearly expressed by the program itself).

Example of a Header Comment

```
/*****************************************************************/
/* Copyright (c) 2005 by Renesas Technology Corp.,              */
/* All rights reserved.                                          */
/* Project name: SH C Compiler.                                  */
/* Module name:  gencode                                         */
/* Function:     Generate SH object code from intermediate*/
/*               language.                                       */
/* Author:       Yugo Kashiwagi                                  */
/* History:                                                      */
/* Aug. 01, 2005: Version 1.0                                    */
/* Aug. 30, 2005: Version 1.1                                    */
/*                Fixed register allocation bugs.                */
/* Sep. 22, 2005: Version 2.0                                    */
/*                Added new optimization.                        */
/*****************************************************************/
```

Example of Function Comment

```
/************************************************************/
/* Function name:     search_table                         */
/* Function:          Search data in the table.            */
/* Input parameters: int key;        key data for search   */
/* Return value:      index of the table with the key      */
/* Table used:        key_table                            */
/* Table modified:    (None)                               */
/************************************************************/
```

Naming Convention

- Use descriptive names for global variable/function names.
  e.g. `read_one_line`,
  `symbol_table_index`
- You can use simple names for local or temporary variables.
  e.g. `i`, `j`
- Use consistent naming convention throughout one project.
- Don't write magic constants in a program. `#define` constants as a macro.

Layout and Indentation

- Use spaces and blank lines to make program easy to understand.
- Indent program according to its structure.

```
char line_buffer[81];                 /* input line              */
int  buffer_index;                    /* index into line_buffer */
int  c;                               /* input character         */
int read_one_line(void){
    c=getchar();
    while (c!='\n'){
        line_buffer[buffer_index++]=c;
        c=getchar();
    }
}
```

Debugging

- Don't write more than two statement in a single line (because debugger usually steps by line).

BAD!!!

```
for (i=1; i<100; i++){c=getchar(); buf[i]=c;}
```

You cannot break inside this loop using debuggers.

- Don't optimize (by machine or by hand) unless your program is running correctly.
- Use `#ifdef DEBUG` ~ `#endif` to insert debugging statements (e.g. checking consistency of input data, etc.).

Size of a Function

- If a function size is too large (more than one page), consider breaking it into smaller functions.

- If a function has more than 7 local variables (if you cannot remember them all), consider breaking it into smaller functions.

Appendix

RENESAS

IV. Writing Reliable Code (self-reference)

Contents

1. Common Mistakes

- How to avoid common mistakes
  - Obey coding rules (MISRA C, etc.)
  - Review and proofread programs
  - Turn on highest error-checking level of the compiler
  - Use header files to keep consistency of the declarations in the program
  - Use program checker to detect mistakes

- Comments -

- Don't forget to close a comment.

  e.g.

```
/* Comment
      a++;
/* Comment */
      b++;
```

This statement is ignored.
No error message issued.

  - // comment (comment which ends at the end of line is available in C++ and C99, but not portable among traditional C compilers.

- False Indentation -

- Indent according to the program structure.

e.g.

```
if (a==0)
        a++;
        b++;
```

This statement is outside the `if` statement.

- Use `{}` to enclose sub-statements even if it consists of only one statement.

- Mistakes in conditions -

- What is the problem of following program ?

  e.g.

```
int a = 2;
int b = 7;
int c = 8;
if (a=0){
    int sum;
    sum = b + c;
}
```

- De-referencing Uninitialized Pointers -

- Don't access using uninitialized pointers

  e.g.

```
f(){
    int *p;
    return *p;
}
```

Quiz: This is very common mistake of engineers.

What is the big problem when we use pointer as above program ?

Dereferencing `NULL` Pointers

- Don't access through a `NULL` pointer.

  e.g.

```
f(){
    int *p=NULL;
    *p=0;
}
```

Address 0 is accessed.

System memory area might be clobbered.

Using Uninitialized Variables

- ALWAYS initialize variable before it is used

e.g.

```
f()
{
   int i;
   if (i==0){
      ...
   }
}
```

The value of `i` is undefined here

- Exceeding Array Bounds -

- Don't access an array with out-of-bound index.
  e.g.

```
char a[10];
f(){
    int i;
    for (i=0; i<=10; i++)
        a[i]=0;
}
```

a[10] is out of bound

Quiz: What happens ?

Forgetting `return` statement

- Don't forget to write `return` statement of a function.

  e.g.

```
int sq(int x){
  int result;
  result=x*x;
}
void g(void){
  a=sq(10);
}
```

`return result;`
should be added.

By historical reason, some C compilers don't check out this as an error.

- Cheating Types -

- Don't cheat types.

- Type cheating can be done in the following ways:
  - Through union members.
  - Through pointers (different type pointers pointing to the same area)
  - Through function parameters and return values

- Cheating Types (union) -

```
union{
    long x;
    float y;
} U;
float f(){
    U.x=1;
    return U.y;
}
```

- Cheating Types (pointer) -

```
long *pl=(long *)100000;
float *pf=(float *)100000;
float f(){
    *pl=100;
    return *pf;
}
```

-Cheating Types (inconsistent declarations) -

file 1

```
float f(float x){
   ...
}
```

file 2

```
long f(long x);

g(){
   long x=f(100);
}
```

Endianness

- Cheating types causes serious problem when you port a program between machines with different "Endianness".

- Endianness determines how a word/long word is stored in memory.

Big Endian vs Little Endian

31 0

A	B	C	D

long word data on a register

Big Endian
68000, IBM, etc.

Memory

Little Endian
x86, etc.

Big Endian	
Address 4n	A
4n+1	B
4n+2	C
4n+3	D

Little Endian	
Address 4n	D
4n+1	C
4n+2	B
4n+3	A

Applications and Endianness

- x86 (Pentium, etc.) is little endian machine, so PC-related application requires little endian

- Network Protocol assumes big endian, so big endian machines are more efficient for network application

- SH is originally a big-endian machine. But SH3 and SH4 supports both endianness to support these application areas

Different Endian Gives Different Program Behavior

```
union{
  unsigned long l;
  unsigned char a[4];
} u;

...
  u.l=0x12345678;
  x=u.a[0];
```

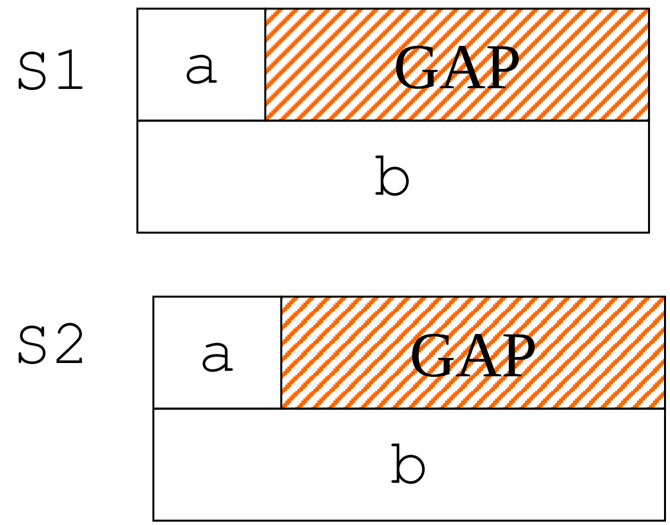
Quiz: What is the value of x in the following cases:
* Big endian ?
* Little endian ?

Comparison of `structs` as a Memory Areas

- Structures have gaps between members.
- So **comparing `structs` as memory areas (using `memcmp` library functions, etc.) might not be correct.**

```
struct{
    char a;
    int b;
} S1, S2;
```

The contents of gaps are not guaranteed



2. Testing

- Unit testing is a testing method to test programs according to the program structure (white-box testing) (c.f. black-box testing, testing from outside the program).

- Sometimes black-box testing cannot test boundary conditions for each functions.

Test Coverage

- C0 Coverage:
  - Coverage of Statements
    - No. of executed statements/No. of all the statements
- C1 Coverage:
  - Coverage of Branches
    - No. of branches executed/No. of all the branches
    - For each branch, both of taken one and not taken one are counted
- Make sure that you cover 100% of C0 and C1 in unit test
  - You must design test cases to cover these requirements
- Use coverage to measure the progress of system test and compare with number of bugs

3. Error Handling

- Embedded systems don't have "RESET" button. Error conditions should be handled inside your program.
- Detect all the unexpected errors while debugging/testing.
- Design how to handle all the expected errors inside your program.

V. Writing Efficient Code (self-reference)

Contents

1. Tuning-up Strategy

- First, reconsider the algorithm before tuning up.
- Measure the performance of the program to determine where to tune-up
- Add comment about what you have done. Retain the original code as a comment.
- Rely on compiler optimization whenever possible.

[Tune-up] means:

When you change the setting of particular parts of an engine, especially slightly, so that it works as well as possible

2. Data Structures
- Use 4-byte local variables -

```
int f(void)
{
  char a=10;
  int c=0;
  for (; a>0; a--)
  c+= a;
  return(c);
}
```

→

```
int f(void)
{
  long a=10;
  int c=0;
  for (; a>0; a--)
  c += a;
  return(c);
}
```

Local variables/parameters are usually allocated to 4-byte registers. Declaring them as 4-byte data eliminates `EXTU/EXTS` instructions.

Specific to 32-bit CPUs

- Sign of Global Variables -

```
unsigned short a;
unsigned short b;
int c;
void f(void)
{
  c=b+a;
}
```

→

```
short a;
short b;
int c;
void f(void)
{
  c=b+a;
}
```

For 1 or 2 byte global data, prefer signed data type to unsigned data type. SH automatically sign-extends these data. Unsigned data requires `EXTU` instruction.

Specific to SH

- Put Related Data in a `struct` -

```
int a, b, c;
void f(void)
{
  a=1;
  b=2;
  c=3;
}
```

```
struct s{
  int a;
  int b;
  int c;
} s1;
void f(void)
{
  struct s *p=&s1;
  p->a=1;
  p->b=2;
  p->c=3;
}
```

Global variable requires 4-byte address to access. Structuring them reduces the usage of 4-byte addresses. This also improves data locality (better cache usage).

- Put Important Members at the Beginning of a `struct` -

```
struct{
{
  char buf[80];
  int key;
}
```

→

```
struct s{
  int key;
  char buf[80];
}
```

If the struct member is near the beginning, the offset to access the member is smaller, and more efficient code can be generated.

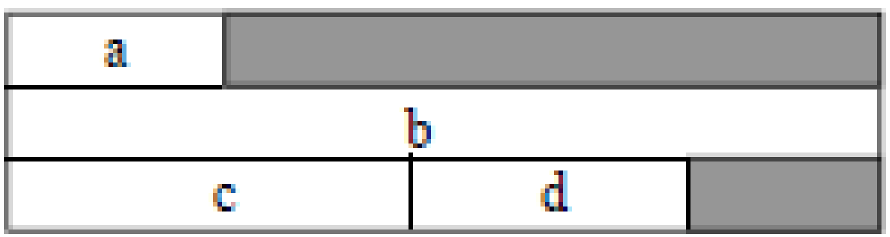
RENESAS

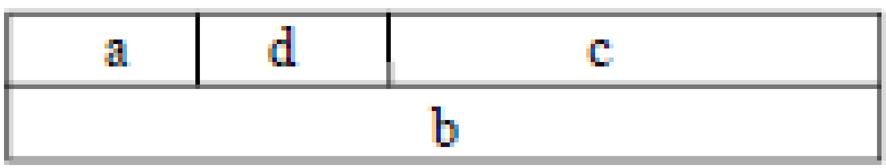
- Consider Data Alignment -

```
struct{
  char a;
  int b;
  short c;
  char d;
}
```

→

```
struct{
  char a;
  char d;
  short c;
  int b;
}
```





Reduce alignment gaps by declaring smaller data first.

- Use `const` Data -

```
char a[]={
  1, 2, 3, 4, 5
};
```

→

```
const char a[]={
  1, 2, 3, 4, 5
};
```

ROM data are less expensive than RAM data.
Declaration without `const` requires both RAM area and ROM area for initial values.

- Prefer Local Data to Global Data -

```
int i;
void f(void)
{
  for (i=0; i<10; i++)
    ;
}
```

```
void f(void)
{
  int i;
  for (i=0; i<10; i++)
    ;
}
```

Don't declare local data as global variables.

Local variables can be allocated on registers.

- Use Pointers to Access Array Elements -

```c
int f1(int data[],
       int count)
{
  int ret=0, i;
  for (i=0; i<count; i++)
    ret+=data[i]*i;
  return ret;
}
```

```c
int f2(int *data,
       int count)
{
  int ret=0, i;
  for (i=0; i<count; i++)
    ret+=*data++ *i;
  return ret;
}
```

Using pointer may reduce the time of array element address calculation.

- Prefer Using Smaller Constants -

```
int i;
void f(void)
{
  i=0x10000;
}
```

→

```
int i;
void f(void)
{
  i=0x01;
}
```

Smaller constants require smaller code..

3. Function calls
- Put Related Functions in a File -

```
extern g(void);
int f(void)
{
  g();
}
```

→

```
int g(void)
{
     ...
}
int f(void)
{
  g();
}
```

Put related functions in a single file, so that compiler can optimize function call instruction (JSR -> BSR).

- Use Function Table instead of `switch` statement -

```
extern void A(void);
extern void B(void);
extern void C(void);
void f(int a)
{
  switch (a){
  case 0:
    A(); break;
  case 1:
    B(); break;
  case 2:
    C(); break;
  }
}
```

```
extern void A(void);
extern void B(void);
extern void C(void);
static int (*tbl[3])()={
  A, B, C};
void f(int a)
{
  (*tbl[a])();
}
```

`switch` statements has overhead of checking switch value. function table doesn't check input value.

- Pass a Pointer to `struct` instead of Many Parameters -

```
int f(int, int, int,
      int, int);
void g(void){
  f(1, 2, 3, 4, 5);
}
```

→

```
struct b{
  int a, b, c, d, e;
} b1={1, 2, 3, 4, 5};
int f(struct b *p);
void g(void)
{
  f(&b1);
}
```

Keep the number of parameters small so that all the parameters are passed through registers.
If not, consider passing parameters as a pointer to a `struct`.

RENESAS

4. Operations
- Pre-compute Constant Expressions in a Loop -

```
extern int a[100],
           b[100];
void f(void)
{
  int i, j;
  j=5;
  for (i=0; i<100; i++)
    a[i]=b[j];
}
```

→

```
extern int a[100],
           b[100];
void f(void)
{
  int i, j, tmp;
  j=5;
  tmp=b[j];
  for (i=0; i<100; i++)
    a[i]=tmp;
}
```

Pre-compute expressions which remains constant in the loop, before entering into the loop..

- Loop Unrolling -

```
extern int a[100];
void f(void)
{
  int i;
  for (i=0; i<100; i++)
    a[i]=0;
}
```

```
extern int a[100];
void f(void)
{
  int i;
  for (i=0; i<100; i+=2){
    a[i]=0;
    a[i+1]=0;
  }
}
```

Reduce the number of loops by unrolling loops reduces the number of branch instructions.

- Use a table instead of (Simple) `switch` Statement -

```
int f(int i)
{
  int ch;
  switch (i)
  {
  case 0: ch='a'; break;
  case 1: ch='x'; break;
  case 2: ch='b'; break;
  }
}
```

→

```
char tbl[]={
  'a', 'x', 'b'
};
int f(int i){
  return (tbl[i]);
}
```

If a `switch` statement has a simple structure, consider using a table.

- Prefer comparison with zero -

```c
int f(int x)
{
  if (x>=1)
    return 1;
  else
    return 0;
}
```

→

```c
int f(int x)
{
  if (x>0)
    return 1;
  else
    return 0;
}
```

Usually, comparing with zero is expanded into simpler instructions.

- Put Error Processing in else Clause -

```
int x(int a)
{
  if (a==0)
    error_proc();
  else
    g(a);
}
```

→

```
int x(int a)
{
  if (a!=0)
    g(a);
  else
    error_proc();
}
```

Putting normal processing in `if` clause (instead of `else` clause), you can save one branch instruction in the normal processing. Don't sacrifice the speed of normal processing for error checking.

Renesas Design Vietnam Co., Ltd.