

Cbooklet

reinhardt weber

```
setup_r8c()
(
    prc0 = 1;           /* Protect off */
    cm13 = 1;          /* Xcin Xout */
    cm15 = 1;          /* XCIN-XCOUT Address decoder */
    cm05 = 0;          /* Xcin on */
    cm16 = 0;          /* Main clock = 8MHz */
    cm17 = 0;
    cm06 = 0;          /* CM16 and CM17 enable */
    asm("nop");        /* Waiting for signals */
    asm("nop");        /* Assembler nudge */
    asm("nop");
    asm("nop");
    ocd2 = 0;          /* Main clock divider */
    prc0 = 0;          /* Protect on */
    pd1 = 0x0F;        /* Set Port 1.0-1.3 to output */
)

toggle_leds()
(
while (1)
(
    pl    = 0x00;
```

**Not just for
R8C fans!**

Contents

Foreword	
Why C?	4

C Basics

The structure of a C program	6
The main function	6
Comments in C	6
The #include directive	7
Keywords in C	7

Constants and Variables

Number systems	8
Data types	8
Constants	9
Variables	9

Operators in C

Arithmetic operators	11
Relational operators	11
Logical operators	11
Shortcuts	12

Functions in C

The function concept	13
Declaring a function	13
Calling a function	14

Program Control

If	16
if...else	16
switch	17
for	18
while	19
do...while	20

Appendix

Header file sfr_r813.h	21
Header file math.h	23

Foreword

Why C?

Many electronics hobbyists have used microcontrollers successfully, and they have also written wonderful programs in assembly language. As the size and complexity of an assembly-language program increase, the desire for a more effective programming environment also increases. Anyone who has tried to implement a mathematical function in assembly language, such as $1/x$, $\sin(x)$ or the like, knows the problems. Here a high-level language such as C, which is the industry standard in the microcontroller and microprocessor world, offers decisive advantages. C programs are portable, which means the program structure can be transferred to other types of microcontrollers after it has been written. The only things that have to be modified are the port assignments and the settings for the special function registers.

Professional programmers claim that an assembly-language program that would take 14 days to generate can be generated in 2 to 3 days in C. What's more, an increasing number of semiconductor manufacturers are making highly effective development environments available at no charge. That's another good reason to start using C.

But there's a hitch. As we all know, the gods have ordained that success doesn't come without hard work. An introductory course in the C language and sample programs from technical magazines can help you overcome the initial hurdles, but will take a while before you can write your own programs. You will also have to master a certain amount of specialist vocabulary. As can be seen from contributions to microcontroller forums and questions asked in these forums, that forms a significant problem for many electronics hobbyists.

This booklet is limited to the basic elements of the C language. We have intentionally omitted complex C structures such as pointers, arrays, strings, structures, unions and the like. This booklet is intended to serve as a reference for beginners. It cannot replace a basic course in C, nor is it intended to do so.

C Basics

The structure of a C program

All C programs consist of several parts, such as comments, preprocessor instructions, declarations, definitions, expressions, assignments and functions. The following listing shows a simple example.

```
/* FILE      :my1c.c
/* DATE       :Wed, Nov 23 2005
/* DESCRIPTION :Program toggles leds on port_1
/* CPU TYPE   :R8C

#include "sfr_r813.h"

long t;

setup_r8c()
{
    prc0 = 1; /* Protect off */
    cm13 = 1; /* Xin Xout */
    cm15 = 1; /* XCIN-XCOUT drive capacity: HIGH */
    cm05 = 0; /* Xin on */
    cm16 = 0; /* Main clock = No Division mode */
    cm17 = 0;
    cm06 = 0; /* CM16 and CM17 enable */
    asm("nop"); /* Waiting for stable oscillation */
    asm("nop"); /* Assembly-language code
    asm("nop");
    asm("nop");
    ocd2 = 0; /* Change main clock
    prc0 = 0; /* Protection on */
    pd1 = 0x0F; /* Set ports 1.0-1.3 to output*/
}

toggle_leds()
{
    while (1)
    {
        p1 = 0x00;

        for (t=0; t<150000; t++);

        p1 = 0x0F;

        for (t=0; t<150000; t++);
    }
}

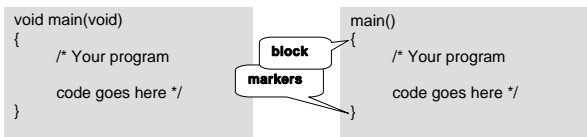
void main(void)
{
    setup_r8c();
    toggle_leds();
}
```

The diagram illustrates the structure of a C program with callouts pointing to specific parts of the code:

- compiler directive**: Points to `#include "sfr_r813.h"`
- variable declaration**: Points to `long t;`
- comments**: Points to the multi-line header comments at the top.
- function 1**: Points to the `setup_r8c()` function definition.
- assembly-language code**: Points to the `asm("nop");` statements within the `setup_r8c()` function.
- function 2**: Points to the `toggle_leds()` function definition.
- value assignment**: Points to `p1 = 0x00;` and `p1 = 0x0F;` inside the `toggle_leds()` function.
- endless loop**: Points to the `while (1)` loop structure.
- timing loop**: Points to the `for (t=0; t<150000; t++);` loops used for timing.
- value assignment**: Points to the `for` loop initialization `t=0`.
- main function**: Points to the `void main(void)` function definition.
- function calls**: Points to the `setup_r8c();` and `toggle_leds();` calls within the `main` function.

The main function

Every C program must include at least one function, which is called the `main` function. This is the primary function in a C program, and it is always the first function to be called when the program is run. It's considered good programming style to have the main routine consist primarily or entirely of function calls, instead of containing the entire code of the program. That makes the program a lot easier to understand and maintain, and it allows the programming effort to be divided among several programmers for large projects. The `main` function is declared in the same manner as any other function.



All instructions and functions belonging to `main` are enclosed in curly brackets `{...}`. This is called 'block building'. In the above example, `VOID` means 'empty' and indicates that the `main` function does not require any input parameters and does not return any result after the instructions are executed. The two instances of the `void` keyword can also be omitted if desired.

Comments in C

All text strings and phrases that don't form part of the actual program are called 'comments'. Comments are ignored by the compiler, which means they do not occupy any space in memory. However, they are quite valuable for explaining the program (or important parts of the program) to other people. And of course, they're very useful for the author of the program as well. In many cases, you may not remember why you wrote the code in a particular manner when you look at your program several days later. And no matter how appropriate the saying 'lean is keen' may be in other contexts, it certainly doesn't apply to computer programs.

```
/*
Comments are
enclosed between
diagonal slashes
and asterisks. */
```

```
// This is a single-line comment.
```

Single-line comments begin with two diagonal slashes and end automatically at the end of the line.

A semicolon (`;`) is usually used to designate a comment in assembly-language programming, but in the C language it marks the end of an instruction.

#include

There are many declarations and functions that are not included in the ANSI standard for the C language, even though they may be necessary or very useful. They are commonly 'hidden' in libraries. You have to tell the compiler to include these library files, which are called 'header files', so it can use these declarations and functions when it compiles your C source code. You can recognise header files by the .h file extension.

Examples:

```
#include "stdio.h"
```

Used in C programs intended to be run on a PC.
[Standard input/output; includes the print function printf() .]

```
#include "sfr_r813.h"
```

The Renesas library. The names and bits of the registers of the R8C microcontroller, such as p1, pd1, p1_7 etc., are defined here.

Keywords in C

A total of 32 terms known as 'keywords' are defined in the ANSI standard for the C language. These keywords are reserved for the compiler. All keywords must be written in lower-case characters, and they are not allowed to be used for other purposes (such as naming variables).

```
auto  
break  
case  
char  
const  
continue  
default  
do
```

```
double  
else  
enum  
extern  
float  
for  
goto  
if
```

```
int  
long  
register  
return  
short  
signed  
sizeof  
static
```

```
struct  
switch  
typedef  
union  
unsigned  
void  
volatile  
while
```

Many C compilers add supplementary keywords to those defined in the ANSI specification in order to make best use of the features of the compiler or microcontroller. The terms listed below are also designated as keywords for the R8C family of microcontrollers.

```
_asm  
_far  
_near
```

```
asm  
_Bool  
far
```

```
near  
restrict  
inline
```

Constants and Variables

Number systems

The C language can work with several different number systems (number bases): decimal, octal, and hexadecimal.

Numbers stated without any special identification (notation) are interpreted as decimal numbers by default. Numbers in all other number systems must be specially identified. Octal numbers begin with 0, hexadecimal numbers with 0x, and binary numbers with 0b.

Base	Notation	Available characters	Example
Decimal(10)	-	0123456789	5
Octal(8)	0...	01234567	05
Hexadecimal(16)	0x...	0123456789ABCDEF	0x5
Binary (2)	0b	0 1	0b11110000

The English (US) convention is always used for numerical notation. That means a full stop ('period') is used as the decimal marker for floating-point numbers. In C, the comma is used in as a separator in lists of numbers or variables. A colon marks a range of numbers.

Examples:

USA

3.14159

3,4

0:3

3 and 4

0->3, thus 0, 1, 2, and 3

Data types

In C, the type of a variable must be declared before it can be used. Otherwise the compiler will not know how much memory to allocate for the variable. Basically, you should always select a type that is adequate for the intended purpose and requires the least amount of memory space. The most important data types are listed below.

Type	Memory space	Value range
_Bool	8	0, 1
char	8	0 -> +255
signed char	8	-128 -> +127
int, short	16	-32768 -> +32767
unsigned int	16	0 -> +65535
long	32	-2147483648 -> +2147483647
float	32	-1.17..e-38F -> +3.4..e-38F

Examples:

```
_Bool stop_button // button has only two states: on & off
unsigned int _year // 0 -> 65535 sufficient for year numbers
float _volume // floating-point number for calculations
```

Constants

Constants are numbers that cannot be changed in the program. That also includes all 'normal' numbers. Whole numbers (integers, or `int`) are written without a decimal marker (decimal point). Floating-point numbers (`float`) have a decimal point followed by additional digits. Characters (`char`) are enclosed between single quotation marks (`'`). Constants are declared using the `#define` keyword.

```
#define <label> value
```

No ; because this is only relevant for the compiler

Examples:

```
#define true 1 // 1 = true
#define false 0 // 0 = false
#define pi 3.14159 // the factor
#define letter_1 'A' // 'A' key on the keyboard
```

The names of constants, variables and functions can be freely selected, but they are not allowed to contain any keywords or operator symbols. Basically, only the letters of the English alphabet, numerals and the underscore (`_`) are used for names. You should choose names that give a good indication of the practical meaning of the constant. For instance, `alarm_btn` is much more meaningful than `t1`.

Variables

A variable is an item stored in memory that can be changed in the program. Variables can be numbers, letters, or text strings. In C, all variables must be declared before they can be used. Variables are considered to be 'statements', which means variable declarations must be terminated with a semicolon (`;`). Variables are defined as follows:

```
type <label> ;
```

; because this is a processor instruction

Examples:

```
_Bool keypress ;
long counter ;
float radius ;
```


Variables are assigned values as follows:

```
<label> = value ;
```



;

Examples:

```
keypress = 1 ;           min_val = counter - 50 ;
```

```
keypress = false ;      max_val = counter * counter ;
```

```
counter = 100 ;         _circum = radius * 2 * pi ;
```



!

Commands and instructions for the processor, which are called 'statements', are terminated with a semicolon (;).

Operators in C

Arithmetic operators

The symbols for arithmetic operators in C correspond to the familiar symbols used on pocket calculators:

+	addition	// examples:	y = x + 3 ;
-	subtraction	//	y = x - b ;
*	multiplication	//	y = a * b ;
/	division	//	Y = a / b ;

The equal sign has a different meaning in C than in ordinary mathematics. In C, it is called the 'assignment operator'. That means the expression to the right of the equal sign is computed and the result is assigned to the variable to the left of the equal sign. The following expressions are thus allowed in C, but not in normal mathematics:



```
x = x+y ;    // compute x + y and store the result in x
x = -x ;     // change the sign of the variable x
```

Relational operators

Relational operators are used to compare variables. They return the result **true** or **false**, depending on the event.

>	greater than	==	equal
>=	greater than or equal	!=	not equal
<	less than		
<=	less than or equal		

Logical operators

The logical operators AND, OR and NOT can be used to execute the familiar operations of digital logic.

b	a	a && b	AND	a b	OR	!a	NOT
0	0	0	0	0	1	1	
0	1	0	0	1	0	0	
1	0	0	0	1	1	1	
1	1	1	1	1	0	0	

Example:

```
if(_price <= max_price && _account > 1000)
    _buy();
```

```
/* The function _buy() will only be called if _price is less
than or equal to max_price and _account contains more than
1000 pounds */
```

The Renesas C compiler for the R8C has several additional logical operators that can be used for bitwise operations on variables:

&	for bitwise AND operations	b	a	a^b	
		0	0	0	
	for bitwise OR operations		0	1	0
		1	0	0	
^	for bitwise XOR operations	1	1	1	

Examples:

```
a = 10011010
b = 11000011
a&b = 10000010
```

```
a = 10011010
b = 11000011
a|b = 11011011
```

```
a = 10011010
b = 11000011
a^b = 01011001
```

Shortcuts

Americans are masters at inventing shortcuts. That's especially true for Dennis Ritchie and Brian Kernighan, the inventors of the C language. Many programmers type in their programs using the 'hunt-and-peck' system, so they try to avoid any unnecessary typing effort.

Shortcut	Normal	Shortcut	Normal
a*=b	a = a*b	a<<=b	a = a<<b
a/=b	a = a/b	a>>=b	a = a>>b
a+=b	a = a+b	a&=b	a = a&b
a-=b	a = a-b	a =b	a = a b
a%=b	a = a%b	a^=b	a = a^b
a++	a = a+1 (increment)		
a--	a = a-1 (decrement)		

Example:

```
for(t=0, t<100000, t++); /* timer loop */
```

Functions in C

The function concept

Functions are the essence of the C programming language. A function can be called from the main routine or from any other function. Every C program must include at least one function, which is called `main()`. It is automatically called when the program is started.

Functions are individual program segments (blocks) that perform specific activities (operations), such as the familiar operations performed by pocket calculators:

`CE` clears the input memory of a pocket calculator. In C, it would be described as a function that does not require any numerical input (parameters) and does not return any sort of number.

`1/x` expects to receive a number as input in order to calculate its inverse value. In C, this is described as a function with an input parameter.

`+`, `-`, `*`, `/`, by contrast, requires two parameters as input, and `O`, the summation function, requires several parameters.

`sin` and `log`, on the other hand, are functions that require one input parameter.

Declaring a function

The general form of a C function is:

```
type function_name(type var1,type var2,type var3,...) ;
```

return type

input parameter with type declaration

Examples:

A function with no input or output parameters:

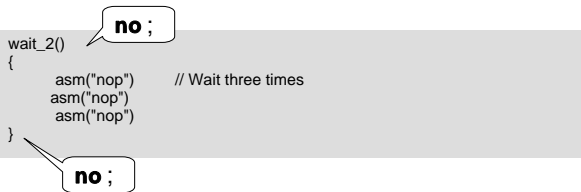
```
void wait_1(void)
asm("nop"); // Call no operation in assembly language
```

The word `VOID` tells the compiler that the function `wait_1` does not require any input parameters and does not return any result. The `VOID` keywords can also be omitted:

```
wait_1()
asm("nop");
```

If a function contains more than one instruction, the instructions must be grouped into a block by enclosing them in curly brackets ({ }). In that case the semicolon at the end must be omitted.

```
wait_2()
{
    asm("nop") // Wait three times
    asm("nop")
    asm("nop")
}
```



A function with an input parameter but no return parameters:

```
int t;

wait_3(int several_times)
    for(t=0;,t<= several_times;t++);
```

/* The for loop increases the value of t stepwise (increments t) starting from 0 until it reaches the value of the input several_times (time delay) */

A function with input parameters and a return parameter:

```
float _volume(float length, float width, float height)
    return length*width*height;
```

/* This function expects three inputs, which are stored in the variables length, width and height. The product of these three variables is then calculated and returned to the caller as a floating-point number */.

Calling a function

Functions are called by simply stating their names. This can be done at any location in the program. After the function has been processed, which can be recognised by the ; in case of a function containing only one instruction or the curly bracket } in case of a function containing several instructions, a return to the calling location occurs automatically. The keyword return has a different meaning in C than in assembly language. In C, it designates the return value instead of the end of a subroutine or function.

Multiple-level function nesting is allowed. 'Function nesting' means that one function calls a second function, which in turn calls a third function, and so on.

Examples:

Calling a function from the main routine without any input or return parameters:

```
void main(void)
{
    wait_1();
}
```

Calling a function from the main routine with an input parameter but no return parameters:

```
void main(void);
{
    wait_3(100);
}
```

// The constant 100 is passed to function wait_3

Calling a function from the main routine with input and return parameters:

```
void main(void);
{
    no_of_litres = _volume(a,b,c);
}
```

/* The values of the variables a, b, and c are passed to the function _volume. The function then calculates the volume of a body and returns the result to the variable no_of_litres.

Program Control

if

It frequently happens that an instruction or block of instructions should only be executed if a certain condition is satisfied. A condition is satisfied if a test of the condition returns the value `true`. Every number except zero is regarded as `true`; zero is regarded as `false`.

The general form is:

```
if (condition) statement ;
```

;

If several instructions are to be executed when the condition is satisfied, they must be grouped into a block.

```
if (condition)
{
    statement_1
    statement_2
    statement_3
    // ...
}
```

no ;

no ;

Examples:

```
if(button == 3)
    red_led = _on;
```

```
if(button == 3)
{
    grn_led = _off;
    red_led = _on;
}
```

if...else

If one instruction or block of instructions is to be executed when a condition is satisfied, while another instruction or block of instructions is to be executed when the condition is not satisfied, an `if...else` conditional statement is used.

The general form is:

```
if (condition) statement_1 ; else statement_2 ;
```

;

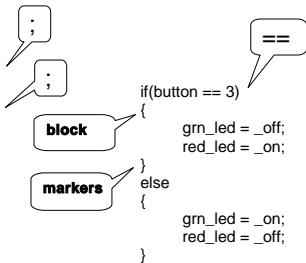
;

If several instructions are to be executed, they must be grouped into a block.

Examples:

```
if(button == 3)
    red_led = _on;
else
    grn_led = _on;
```

```
if(button == 3)
{
    grn_led = _off;
    red_led = _on;
}
else
    grn_led = _on;
```



switch

If a conditional statement has more than one or two possible outcomes, it is very tedious to implement it using the `if...else` structure. In that case it's better to use the `switch /case` structure with multiple alternatives. This type of program control can be compared to a rotary selector switch with multiple positions (cases).

The general form is:

```
switch (variable)
{
    case constant_1 ;
        instruction_1;
        break;

    case constant_2;
        instruction_2
        break;

    case constant_3;
        instruction_3

    case // ...
        break;

    default instruction_x;
}
no ;
```

The `switch` function compares the content of `variable` to the value of a constant (`constant_x`) for each of the defined cases (`case`). If the result of the comparison is positive, the corresponding instruction (`instruction_x`) or

block of instructions is executed. Program control returns to the caller of the `case` statement when the `break` keyword is reached. If none of the cases listed in the `case` statement is found, the instruction following `default` is executed. The `default` portion can be omitted if it is not necessary.

Example:

```
switch (_button)
{
  case 1:
    red_led = _on;
    break;

  case 2:
    yel_led = _on;
    break;

  case 3:
    grn_led = _on;

  default:
    blu_led = _on;
}
```

block (points to the opening curly brace)

markers (points to the closing curly brace)

may be omitted (points to the default case)

```
/* Enable the red LED if the value of _button = 1, the yellow LED if it is 2, and the green LED if it is 3. If the variable _button does not contain 1, 2, or 3 (e.g. 4), enable the blue LED */
```

for

A `for` loop is used if part of a program must be executed multiple times.

The general form is:

```
for(start_value; end_condition; step_size)
  instruction_1;
```

;

When the `for` loop is called, `start_value` is assigned to a previously defined count variable. The count variable is then incremented or decremented by the value of `step_size` each time the loop is executed, until the test of `end_condition` yields the logical value `true`.

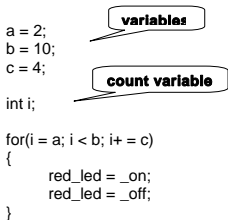
If several instructions are to be executed, curly brackets (`{ }`) must be used to group them into a function block.

Examples:

```
int t;  
  
for(t=0, t < 10, t++)  
    blink_led();
```

/* Integer variable t is assigned the value 0 when the for loop is entered. Next, the function blink_led() is called. After the first pass through the loop, the variable t is incremented (t++) to the value 1. As 1 is less than 10, the process continues until t = 9. The loop is thus executed ten times. */

```
a = 2;  
b = 10;  
c = 4;  
  
int i;  
  
for(i = a; i < b; i += c)  
{  
    red_led = _on;  
    red_led = _off;  
}
```



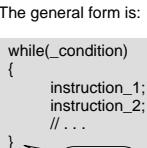
/* This loop is executed only two times. */

while

A **while** loop is used when execution is tied to a condition.

The general form is:

```
while(_condition)  
{  
    instruction_1;  
    instruction_2;  
    // ...  
}
```




When the **while** loop is called, the value of the condition (`_condition`) is first tested. If the result is positive (true), the instructions (`instruction_x`) are executed repeatedly until the result of the test is false.

Example:

```
#define true 1

while(i == true)
{
    i = button_pressed();
    blink_led();
}
```






/* The function blink_led is executed until the function button_pressed no longer returns the value 1 to variable i.

do...while

A while loop will not be executed if the condition is not satisfied at the beginning of the loop. If it is necessary for instructions to be executed at least once, the condition must be tested at the end instead. A do...while loop is used in such cases.

The general form is:


```
do
{
    instruction_1;
    instruction_2;
    instruction_3;
    // ...
}
while(_condition);
```



Example:

```
#define true 1

do
{
    i = button_pressed();
    blink_led();
}
while(i == true);
```



/* The function blink_led is executed at least once */

Appendix

Header file sfr_r813.h

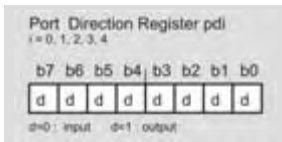
The header file `sfr_r813.h` provides access to the special function registers (SFRs) of the R8C microcontroller. These registers contain the basic settings for the microcontroller, such as the port directions (in/out), timer settings, A/D converter settings, UART settings, and so on.

The R8C microcontroller has more than 50 SFRs. That means we have to limit ourselves here to a selection of the most important SFRs. Refer to the **R8C/13 Group Hardware Manual** for detailed information on all of the SFRs.

Port registers (P0, P1, P2, P3 and P4)

A port is a memory location that is connected to the pins of the microcontroller and is thus externally accessible. Ports are used for inputting and outputting data. A port can be either an input or an output. When the microcontroller is started up, all ports are configured as inputs by default. The direction (input or output) can be changed using the Port Direction (PD) registers.

The following example shows how to change the port direction:

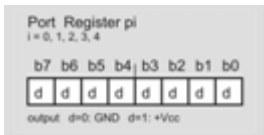


Examples:

```
pd1 = 0x0F;  
/* port1, bits 0:3 = output  
bits 4:7 = input */
```

```
pd2_3 = 1;  
/* port2,bit3 = output */
```

... and the following example shows how to output data via a port:



Examples:

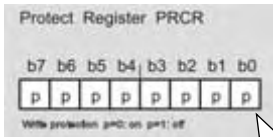
```
p1 = 0x0F;  
/* port1, bits 0:3 = 1  
bits 4:7 = 0 */
```

```
pd2_3 = 0;  
/* port2,bit3 = 0 */
```

/* If a 1 is written to a port register, the supply voltage (e.g., +5 V) will be present on the corresponding pin. A 0 causes the pin to be at ground potential (0 V) */

Protection registers (PRCR)

The PRCR registers can be used to protect the contents of other important registers against overwriting (if the program goes out of control, for instance).



Examples:

```
prc0 = 1;
/* write protection disabled */
```

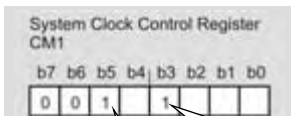
```
prc0 = 0;
/* write protection enabled */
```

Bit b0 is the write-protect bit for CM0, CM1, OCD, HR0 and HR1

System Clock Control registers (CM1, CM2 and OSD)

The R8C microcontroller has two oscillators to provide the clock for the CPU. One oscillator is internal and is called 'on-chip oscillator', while the other is external and is called 'main clock'. The 'main clock' oscillator uses a quartz crystal connected to the Xin and Xout pins.

The CM registers determine how the microcontroller clock signal is generated. In addition, a prescaler can be enabled and configured to reduce the frequency of the processor clock.



prescaler setting

external crystal oscillator mode

oscillator gain

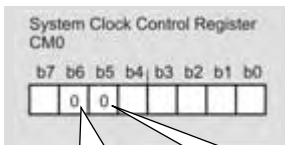
Examples:

```
cm13 = 1;
/* Xin/Xout on ports p46 and p47
= ext. xtal */
```

```
cm15 = 1;
/* Xin/Xout driver high */
```

```
cm16 = 0;
/* prescaler for CPU clock */
```

```
cm17 = 0;
/* prescaler for CPU clock */
```



CM1 prescaler bits 6 & 7 set

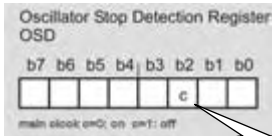
crystal oscillator enabled

Examples:

```
cm05 = 0;
/* enable Xin/Xout on ports p4.6
and p4.7 */
```

```
cm06 = 0;
/* enable prescaler */
```

The Oscillator Stop Detection (OSD) register is used together with the other registers to select the clock source, and it is also responsible for monitoring the clock signal.



Examples:

```
osd2 = 0;
/* enable external xtal osc. as
CPU clock source */
```

```
osd2 = 1;
/* disable external xtal osc. */
```

Header file math.h

The header file `math.h` contains the library of mathematical functions for the R8C. Here we describe some of the most important functions.

Functions of type double
`f_name(double x);`

`sin();`
`cos();`
`tan();`

`asin();`
`acos();`
`atan();`

`sinh();`
`cosh();`
`tanh();`

`sqrt();`

`exp();`
`log();`
`log10();`

`mod();`

`fabs();`
`floor();`
`ceil();`

Functions of type float
`f_name(float x);`

`sinf();`
`cosf();`
`tanf();`

`asinf();`
`acosf();`
`atanf();`

`sinhf();`
`coshf();`
`tanhf();`

`sqrtf();`
`powf();`

`expf();`
`logf();`
`log10f();`

`fabsf();`
`floorf();`
`ceilf();`

Functions of type double
`f_name(double x, double y);`

`pow();`
`fmod();`
`atan2();`
`fmod();`

Functions of type float
`f_name(float x, float y);`

`powf();`
`atan2f();`
`fmodf();`