1 Introduction

This application note provides a simple way to write RS08 microcontroller codes in C programming language.

C programming is much easier and becoming more popular than assembly in development. This application note provides a simple way to write a pure C program for RS08 MCUs.

The following contents are in this document:

- Schedule RS08 in C technology
- Quick lookup table technology
- Embedded assembly technology

Contents

1 Introduction .................................................. 1
2 RS08 Schedule in C Technology .............................. 2
  2.1 RS08 Interrupt Mechanism ................................ 2
  2.2 RS08 Schedule in C ........................................ 4
  2.3 Performance Promotion ................................. 6
3 Quick Lookup Table Technology ............................ 8
  3.1 Direct Page Lookup .................................... 8
  3.2 High Page Lookup .................................... 11
4 Embedded Assembly Technology ........................... 13
5 Conclusion .................................................... 14
2 RS08 Schedule in C Technology

2.1 RS08 Interrupt Mechanism

2.1.1 RS08 Interrupt Model

The RS08 Family is designed to meet ultra low-end applications. Therefore the CPU is simplified. In the RS08 interrupt model the applications are assumed to occur infrequently. The RS08 CPU is in wait or stop mode most of the time. The CPU wakes up quickly, inquires, and executes the interrupt service routine (ISR) with the software polling mode after an interrupt is detected. After the interrupt service routine finishes, the CPU enters wait or stop mode again. If there are two or more interrupts detected, the execution order is decided by the interrupt priorities.

Figure 1 shows a two-event schedule. Event 1 is a periodic event with the interval $t$. Event 2 is an asynchronous event scheduled prior to Event 1. Event 2 ISR is executed first, after two events occur simultaneously. After event 2 ISR is finished it continues to execute the ISR of event 1. The CPU enters wait or stop mode to wait for the next event.
2.1.2 RS08 Interrupt Schedule

The RS08 MCU has no inhibition of interrupt priority because it has no interrupt vector table. This makes the RS08 MCU have a different schedule in the code. The typical schedule in the RS08 MCU is executed in this order:

1. Enter wait or stop mode.
2. Wake up from wait or stop mode if an interrupt is detected.
3. Inquire and execute the interrupt service routine.
4. Clear interrupt source and execute other tasks in the service routine.
5. After returning from the service routine the CPU goes to 1 if there is no pending interrupt or goes to 3 if there is at least one pending interrupt.

This mechanism ensures the high priority ISR is executed prior to the low one. The priority of interrupt service routine is configurable because the schedule handle is controlled by the software. Figure 2 shows the typical RS08 schedule work flow.
Figure 2. RS08 Schedule Flow
2.1.3 RS08 Schedule in Assembly

Example 1 shows a typical RS08 schedule in assembly. In this example, the CPU enters wait mode first. When an interrupt occurs the CPU wakes and executes the following code one by one. The opcode BRSET is used to check if there are pended interrupts. After an interrupt is detected its ISR is executed immediately. At the end of every interrupt service routine there is an opcode BRA that changes the program flow to the opcode WAIT. If there is a pending interrupt the CPU can not enter wait mode and execute continuously until all polled interrupt service routines are completed.

Example 1. RS08 Schedule in Assembly

```
_Startup:
    MOV #HIGH_6_13(SIP1), PAGESEL

; Polled interrupt code
idle:
    WAIT

; feed watchdog
feed_watchdog
    BRSET SIP1_MTIM, MAP_ADDR_6(SIP1), mtim_ISR
    BRSET SIP1_ACMP, MAP_ADDR_6(SIP1), acmp_ISR
    BRSET SIP1_KBI, MAP_ADDR_6(SIP1), kbi_ISR
    BRSET SIP1_RTI, MAP_ADDR_6(SIP1), rti_ISR
    BRSET SIP1_LVD, MAP_ADDR_6(SIP1), lvd_ISR

; check other interrupts
BRA idle

mtim_ISR: <code>
    BRA idle

acmp_ISR: <code>
    BRA idle

kbi_ISR: <code>
    BRA idle

rti_ISR: <code>
    BRA idle

lvd_ISR: <code>
    BRA idle
```
2.2 RS08 Schedule in C

Because C programming is easy to use and the hardware is independent, it is becoming more popular in the development of RS08 and other MCUs. Writing the RS08 schedule in pure C efficiently is an important progress in development.

The RS08 schedule in the assembly gives a hint to develop the RS08 schedule in C. The following rules must be followed in C implementation:

- Schedule all actions in a dead loop.
- Check pending interrupts and execute their interrupt service routine.
- Return to the beginning of the schedule if an interrupt service routine is completed.

All initialization must be done before the sentence in C is used to implement a dead loop. A few sentences in C are used to check the pending interrupts. If a pending interrupt is detected its interrupt service routine is called immediately. After the calling, a continued sentence is used to return to the beginning of the loop. This is the beginning of the schedule.

The interrupt service routines are defined as a set of functions in C. Example 2 shows a schedule with three interrupt sources: RTI, MTIM, and ACMP. The RTI interrupt has the highest priority, the MTIM interrupt has the middle priority, and the ACMP interrupt has the lowest priority. The user can modify the priority with an advanced software method and adding any other available interrupt sources in the program.

Example 2. RS08 Schedule in C

```c
void SYS_init(void) {
    ...
}

void RTI_isr(void) {
    ...
}

void MTIM_isr(void) {
    ...
}

void ACMP_isr(void) {
    ...
}

void main(void) {
    SYS_init();
    for(;;) {
        __RESET_WATCHDOG(); /* feeds the dog */
        if(SIP1_RTI) {
            RTI_isr();
            continue;
        }

        if(SIP1_MTIM) {

    ```
2.3 Performance Promotion

2.3.1 C Compiler Inhibition

The RS08 schedule in C introduced in Chapter 2.2, “RS08 Schedule in C” is not as efficient as expected. When the C code is disassembled, the C compiler uses the following instructions to implement the C codes:

- BSR instruction is used to implement the function call.
- RTS instruction is used to implement the function return.
- BRA instruction is used to return to the beginning of schedule.

These compiled results are common in most C compilers. The RS08 has a one-level PC hardware stack (SPC). If another subroutine is needed, a software stack must be used to store the previous PC value in the SPC register. The RS08 CPU does not support hardware stack operation.

In assembly, users can control the program flow from the end of the function call to the beginning of the scheduler. But the C compiler inserts the RTS instruction before the program flow is brought to the beginning of scheduler.

In assembly this is a one-shot only operation.
2.3.2 C Compile Optimization

This section provides two ways to promote the performance of the RS08 schedule in C. The original idea is to use inline code to take the place of a calling operation. Inline code eliminates the branch and calling operation between the main program and subroutines.

CodeWarrior supports a few ways to implement inline code. The following two ways are recommended because they are easy to use and set.

The first way is to use pragma macro pre-processor to indicate the interrupt service routine as inline code. In Example 3 every interrupt service routine has a pre-processor indication. This indicates that every interrupt service routine be used as inline code in the program.
Example 3. RS08 Schedule in C Inline Code

```c
#pragma INLINE
void SYS_init(void) {
  ...
}
#pragma INLINE
void RTI_isr(void) {
  ...
}
#pragma INLINE
void MTIM_isr(void) {
  ...
}
#pragma INLINE
void ACMP_isr(void) {
  ...
}
```

The C compiler makes great progress in the schedule after the inline code is applied. An instruction BRCLR is used to check the pending interrupt and return to the beginning of the schedule. All other routine code is embedded between these two instructions. Example 4 shows the disassembly of the implementation. The label LB is the beginning of the schedule. If the pending interrupt of the RTI is not detected a sentence is skipped and the next sentence is executed. The main body of the interrupt service routine to increment the counter is executed if the pending interrupt of the RTI is detected by the instruction BRCLR. The instruction BRA brings the program flow to the beginning of schedule LB after the interrupt service routine is completed.

Example 4. Disassembly of the RS08 Schedule in C Inline Code

```
LB:
__RESET_WATCHDOG(); /* feeds the dog */
3f00        CLR       %MAP_ADDR_6(_SRS)
...
if(SIP1_RTI) {
  030004   BRCLR      #1,%MAP_ADDR_6(_SIP1),L14
  3c00     INC       counter
  RTI_isr();
  continue;
  30f7     BRA       LB
L14:
}
```
Quick Lookup Table Technology

In the above disassembly the inline C code has the same efficiency as assembly because of the elimination of the function calling and branch. The implementation does not use JSR and RTS instructions. This allows a subroutine call in the pseudo interrupt service routine without the software stack support.

The other important way is to set inline in compiler options. Checking the inlining option in the RS08 Compiler Option Settings dialog works the same as the inline macro pre-processor. Figure 3 shows the compiler option.

![Figure 3. Setting Inline Options in C Compiler](image)

3 Quick Lookup Table Technology

3.1 Direct Page Lookup

3.1.1 Indexed Addressing Mode

The RS08 MCU has two special mapped registers. They are the indexed date register (D[X]) and the index register (X). These registers are located at 0x000E and 0x000F. The indexed data register allows the user to access the data in the direct page address space indexed by X. The index register allows the user to index or address any location in the direct page address space.
3.1.2 Direct Page Lookup Implementation

In the RS08 MCU, the shortcut to acquire a quick lookup in C is to use the index register X and the indexed data register D[X]. Example 5 shows a typical lookup program in traditional C style programming. In this example the function Lookup Table takes a loop to find the matched value in a table that is defined in the direct page.

Example 5. Lookup Table in Traditional C Style Programming

```c
#define MAX_TABLE_NUM 16
...
unsigned char table[MAX_TABLE_NUM];
...
unsigned char LookupTable(unsigned char target) {
    unsigned char i;

    for ( i = 0 ; i < MAX_TABLE_NUM ; i++ ) {
        if ( table[i] == target ) {
            break;
        }
    }

    return i;
}
```

In this example a loop is used to lookup a matched value in an array that has 16 data. Although the C code looks good unfortunately the object codes run on low efficiency. As shown in Example 6, every loop takes 29 CPU cycles. This is an excellent result for the C compiler. The code work can be driven more efficiently if the idea of the indexed data register and index register is used.
Example 6. Disassemble of the Lookup Table in Traditional C Style Programming

0000 b700 STA __OVL_OldLookupTable_p0
41: 42:  unsigned char i;
43: 44:  for ( i = 0 ; i < MAX_TABLE_NUM ; i++ ) {
0002 3f00 CLR __OVL_OldLookupTable_i ; 3 cycles
0004 L4:
45: 46:   if (target == table[i])
0004 b600 LDA __OVL_OldLookupTable_i ; 3 cycles
0006 ab00 ADD #table ; 2 cycles
0008 ef TAX ; 2 cycles
0009 ce LDA D[X] ; 3 cycles
000a b100 CMP __OVL_OldLookupTable_p0 ; 3 cycles
000c 3708 BEQ L16 ; 3 cycles
000e 3c00 INC __OVL_OldLookupTable_i ; 5 cycles
0010 b600 LDA __OVL_OldLookupTable_i ; 3 cycles
0012 a110 CMP #16 ; 2 cycles
0014 35ee BCS L4 ; 3 cycles

Almost all instructions are involved in the memory access that reduces the performance of the CPU. To get a significant performance promotion the way is to use both registers in the code. Both registers must be defined in the C code. They are not defined in any associated files. The index method using X register and D[X] register is the entry to introduce both registers into the code. The C compiler can optimize this code with these registers. Example 7 shows the optimized C code to complete quick lookup with the index register and the indexed data register.

Example 7. Quick Lookup Table Technology in C

#define MAX_TABLE_NUM   16

unsigned char table[MAX_TABLE_NUM];
volatile unsigned char d @0x0E;
unsigned char * x @0x0F;

unsigned char LookupTable(unsigned char target) {
    for ( x = &table[0] ; x < &table[MAX_TABLE_NUM] ; x++ ) {
        if ( d == target ) {
            break;
        }
    }

    return x - &table[0];
}
As shown in Example 8, there is no need to access the data through the indirect addressing mode because variables x and d are assigned to the index register and the indexed data register. The instruction elimination saves at least five cycles. The second improvement, is using the variable x to take the place of the loop counter reducing 8 to 21 cycles. This is a great improvement without any embedded assembly support.

Example 8. Disassemble of the Quick Lookup Table Technology in C

```
0000 b700     STA            _OVL_LookupTable_p0
 98: 0002 3e0000 MOV            #table,x ; 4 cycles
     0005 3006 BRA            LD ; 3 cycles
     0007 L7: 0007 c0 LDA            d ; 3 cycles
     0008 b100 CMP            _OVL_LookupTable_p0 ; 3 cycles
     000a 3706 BEQ            L12 ; 3 cycles
     000c 20 INC            x ; 4 cycles
     000d L7: 000d c0 LDA            x ; 3 cycles
     000e a110 CMP            #table:16 ; 2 cycles
     0010 35f5 BCS            L7 ; 3 cycles
     0012 L12: 101: 0012 c0 LDA            x
     102:       0013 a000 SUB            #table
     103:       104: 0015 be RTS
105:  return x - table;
```

3.2 High Page Lookup

3.2.1 High Page Accessing

The RS08 MCU can access the first 256-byte addresses. The page window must be used to access high page addresses. Every 64-byte address aligned data can be mapped to the page window by setting the register PAGESEL. Page window locates are from 0x00C0 to 0x00FF. Therefore, mapped high page data can be accessed at this address range. The first 256-byte data is called direct page. They can be accessed alone as well as by the page window.
The page select register (PAGESEL) is used to set which page content is mapped in the page window. This register resides at the memory mapped location 0x001F. The page select register allows the user to access all memory locations in the entire 16 KB address space through the page window.

3.2.2 High Page Lookup Implementation

The RS08 MCUs can access the first 256-byte addresses with efficient performance. The high page addresses that exceed 0x00FF must be accessed by the page window. A macro PAGE_OFF is introduced to have a quick look-up to high page tables and to get the mapped address of tables in the page window. Example 9 shows the implementation in C.
Example 9. Quick Lookup Tables at the High Page in C

```c
#define MAX_TABLE_NUM 16

#define SET_PAGE(addr) \  
    { char *__paged addr1 = (char *)__paged(&addr); 
      PAGESEL = *((unsigned char *)&addr1); }

#define PAGE_OFF(addr) (unsigned char *)(unsigned char)(unsigned char *)__paged)(unsigned char *)&addr)

const unsigned char tablehigh[MAX_TABLE_NUM] = {
    0,1,4,9,16,25,36,49,64,81,100,121,144,169,196,225
};

unsigned char LookupTableHighPage(register unsigned char target) {
    unsigned char * start = PAGE_OFF(tablehigh[0]);
    unsigned char * end = PAGE_OFF(tablehigh[MAX_TABLE_NUM]);

    SET_PAGE(tablehigh);

    for ( x = start ; x < end ; x++ ) {
        if (target == d) {
            break;
        }
    }

    return x - start;
}
```

High page accessing must change only the linear address of the table high into the mapped address. The high page lookup has the same performance with direct page lookup. Every loop takes 21 cycles because the high page lookup operates the direct page.

This technology is the quick lookup in the RS08 programming because the page window can map any contents in the whole memory map. This means this code can be used to access the direct page contents. The only change is to set a correct value to the page select register.

4 Embedded Assembly Technology

Some users need higher performance and do not like to write in assembly. The technology to embed assembly in C is a good solution. ANSI-C and most of the other C versions support embedded C with the keyword asm. This indicates the asm sentences in C sources. The other two non-standard keywords, _asm and __asm are both supported in the RS08 C compiler. There are a few ways to embed assembly in C with these keywords. Example 10 shows the syntax.
The embedded assembly support can become more efficient with the lookup work above. Compared to the initial 29 cycles the minimum lookup can get 16 cycles for every loop, reduced by 13 cycles. This is because almost every opcode can be written manually. The space consumption can be kept at a low level. Example 11 shows the list of quick lookup with embedded assembly support.

**Example 11. Quick Lookup with Embedded Assembly Support**

```c
unsigned char i @0x0D;
...
unsigned char LookupTableAsm(register unsigned char target){
    __asm {
        MOV #MAX_TABLE_NUM, i;
        LDX #table;
        LOOP:
            CBEQ ,X, LOOPEND ; 5 cycles
            INCX ; 4 cycles
            DBNZ i, LOOP ; 7 cycles
        LOOPEND:
    }
    return x - table;
}
```
5 Conclusion

This article introduces C programming to the RS08 MCU development. C programming has a couple of advantages: hardware independent, shorter development cycle, and easy to port. It is becoming more popular these days. With considered optimizations code can be written in C with the same efficiency as assembly in the RS08 development.
How to Reach Us:

Home Page: www.freescale.com

Web Support: http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-900-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals", must be validated for each customer application by customer's technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

RoHS-compliant and/or Pb-free versions of Freescale products have the functionality and electrical characteristics as their non-RoHS-compliant and/or non-Pb-free counterparts. For further information, see http://www.freescale.com or contact your Freescale sales representative.

For information on Freescale’s Environmental Products program, go to http://www.freescale.com/epp.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners. © Freescale Semiconductor, Inc. 2008. All rights reserved.