This application note is an introduction to the RS08 platform, an ultra low-cost 8-bit MCU core, from Freescale Semiconductor.

Section 1 provides information for the user to get started with RS08 and section 2 includes application discussions to demonstrate techniques and concepts, together with working examples.

1 Introduction to RS08

This section covers the RS08 architecture, programming model, and instruction set to help the user to gain a good understanding on the platform. Where necessary, cross references are provided to the popular Freescale HC08 and S08 platforms. In most cases, the MC9RS08KA2 device is used in examples to illustrate concepts.
1.1 RS08 Architecture

The RS08 platform is developed for extremely low cost applications. Its hardware size is optimized and the overall system cost is reduced. The smaller hardware size allows the silicon to fit into a smaller package, such as the 6-pin dual flat no lead package (DFN). The RS08 platform retains a similar programming model as in the popular HC08/S08 platforms to allow easy source code migration between the platforms.

The main features of the RS08 platform are:

- Subset of S08 instruction set
- New instructions for shadow program counter (SPC) — SHA and SLA
- New tiny and short addressing modes for code size optimization
- Maximum 16K-byte accessible memory space
- Eliminated vector fetch mechanism for interrupt and reset service
- Eliminated RAM stacking mechanism for subroutine call
- Single level hardware stacking for subroutine call
- Low power mode supported through the execution of STOP and WAIT instructions
- Stop wakeup through internal or external interrupt trigger
- Illegal address and opcode detection with reset
- Hardware security feature to protect unauthorized access to the non-volatile memory (NVM) area
- Debug and NVM program/erase support using single pin interface

1.1.1 CPU Registers

The RS08 CPU registers include an 8-bit general purpose accumulator (A), 14-bit program counter (PC), 14-bit shadow program counter (SPC), and a 2-bit conditional code register (CCR). The CCR contains two status flags and are tested for conditional branch instructions such as BCS and BEQ. Figure 1-1 shows the RS08 CPU registers.

![Figure 1-1. RS08 CPU Registers](image-url)
The 8-bit general purpose accumulator A provides a primary data register for the RS08 CPU. Data can be read from memory into A with the LDA instruction. The data in A can be written into memory with the STA instruction. The new added exchange instructions, SHA and SLA, allow values to be exchanged between accumulator A and shadow program counter (SPC) high byte and low byte respectively.

The program counter (PC) contains the address of the next instruction or operand to be fetched as in the HC08/S08 platform. However, the PC in RS08 platform is 14-bit long, which means the maximum addressable space is 16K bytes.

In HC08/S08 platform, the return PC value is stacked into RAM during subroutine calls using JSR and BSR instructions. In RS08 platform, RAM stacking mechanism is eliminated, return address is saved into the SPC register. Upon completion of the subroutine, RTS instruction will restore the content of the PC from SPC. SPC only provides a single level of address saving, nested subroutine calls can be performed through software stacking. User firmware can utilize SHA and SLA instructions to swap the high byte and the low byte content of SPC to A, then stack them to RAM.

The status bits (Z and C) in condition code register (CCR) indicates the results of previous arithmetic and other operations. The bit definition is identical as in HC08/S08 platform. Please refer to RS08 Core Reference Manual for their detail definition.

### 1.1.2 Special Registers

In addition to the CPU registers, there are two memory mapped registers that are tightly coupled with the core address generation. They are the indirect data register (D[X]) and the index register (X). These registers are located at $000E$ and $000F$ respectively.

![RS08 Special Registers](image)

Registers D[X] and X together perform indirect data access. The register X contains the address which is used when register D[X] is accessed. Figure 1-3 shows the index addressing scheme. The X and D[X] registers are not part of the CPU internal registers, but they are integrated seamlessly with the RS08 generic instruction set to form a pseudo instruction set.
1.1.3 Generic Addressing Mode

Whenever the MCU reads data from memory or writes data to memory, an addressing mode is used to determine the exact address whether data is read from or write to. Table 1-1 summarizes the generic addressing mode supported by the RS08 platform.

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherent Addressing</td>
<td>CLRA, INCA, SHA, RTS</td>
</tr>
<tr>
<td>Direct Addressing</td>
<td>LDA $20, AND $20</td>
</tr>
<tr>
<td>Relative Addressing</td>
<td>BRA, BCS, BEQ</td>
</tr>
<tr>
<td>Immediate Addressing</td>
<td>LDA #9</td>
</tr>
<tr>
<td>Tiny Addressing</td>
<td>INC &lt;$0D</td>
</tr>
<tr>
<td>Short Addressing</td>
<td>CLR &lt;$1D</td>
</tr>
<tr>
<td>Extended Addressing</td>
<td>JMP, JSR</td>
</tr>
</tbody>
</table>

Figure 1-3. Index Addressing Scheme
1.1.3.1 Addressing Modes Common to HC08/S08 Platforms

The inherent addressing, direct addressing, relative addressing, immediate addressing, and extended addressing modes in RS08 have identical operation as in the HC08/S08 platform. Inherent addressing is used when the CPU inherently knows all the information needed to complete the instruction and no addressing information is supplied in the source code. Relative addressing is used to specify the offset address for branch instructions relative to the program counter. Immediate addressing is used when an explicit value to be used by the instruction is located immediately after the opcode in the instruction stream. Direct addressing is used to access operands located in direct address space ($0000 through $00FF). Extended addressing is used to specify 2-byte operand to the instructions. This addressing mode is only used in JMP and JSR instructions where the 14-bit target address is specified in the operand.

1.1.3.2 Tiny and Short Addressing Modes

Tiny and short addressing modes are introduced in the RS08 platform. These addressing modes have similar operations to direct addressing mode but the addressable space is limited. Only portion of direct address space within $0000–$00FF can be accessed by these addressing modes. However, all instructions associated with these addressing modes are single byte instructions. Maximizing the utilization of these instructions can reduce the overall code size.

Tiny addressing mode is capable of addressing only the first 16 bytes in the address map, from $0000 to $000F. This addressing mode is available for increment (INC), decrement (DEC), add (ADD), and subtract (SUB) instructions. Equivalent instructions are also available in direct addressing mode, 2-byte instructions, where the addressable space is from $0000–$00FF. User should add the less than symbol (\(<\)) before the operand in the source code as shown below, this forces the assembler to use tiny addressing instructions instead.

\[
\begin{align*}
\text{INC} & \quad \text{<$0D} \\
\text{DEC} & \quad \text{<$0D} \\
\text{ADD} & \quad \text{<$0D} \\
\text{SUB} & \quad \text{<$0D}
\end{align*}
\]

Short addressing mode is capable of addressing only the first 32 bytes in the address map, from $0000 to $001F. This addressing mode is available for clear (CLR), load accumulator A (LDA), and store accumulator A (STA) instructions. Similar to tiny addressing instructions equivalent instructions are also available in direct address mode. User should add the less than symbol (\(<\)) before the operand as shown below to force the assembler to use short addressing instructions.

\[
\begin{align*}
\text{CLR} & \quad \text{<$1F} \\
\text{LDA} & \quad \text{<$1F} \\
\text{STA} & \quad \text{<$1F}
\end{align*}
\]

1.1.3.3 Pseudo Addressing Modes

Using the special registers, D[X] and X, the RS08 generic instruction set can be used to emulate some of the accumulator X operations in the HC08/S08 architecture. This emulation is supported by the assembler/compiler and it is done during the time of compilation. When zero offset indexing instructions or register X related operations are involved, user can use the same HC08/S08 coding syntax for RS08 programming. During compilation the assembler will convert the pseudo RS08 instructions to equivalent generic RS08 instructions. This operation is transparent to the user.
Introduction to RS08

Below summarizes the pseudo addressing modes supported by the RS08 architecture.

- **Pseudo inherent addressing** — for example, TSTX, DBNZX — is emulated by equivalent direct addressing operation where the operand is always loaded from register X location ($000F). In some of these operations, such as DECX and INCX, the tiny and short addressing instructions are available. The pseudo instructions become single byte.

- **Pseudo direct addressing** — for example, LDX $20, STX $20 — is emulated by move (MOV) direct-direct operation. LDX operation is equivalent to moving operand to register X ($000F). STX operation is equivalent to move the content of register X to operand targeted address.

- **Pseudo immediate addressing** — for example, LDX #$09 — is emulated by move (MOV) immediate-direct operation. Register X is loaded by explicit data.

- **Pseudo zero offset index addressing** — for example, ADD ,X — is emulated by equivalent direct addressing operation where the operand is always loaded from register D[X] location ($000E). Register D[X] itself holds the indirect data that its address is indicated by register X. Performing operation on register D[X] has equivalent operation as HC08/S08 style zero offset index addressing. RS08 platform preserves the same HC08/S08 style coding syntax which helps user to migrate source code among these platform. Below shows some coding examples.

```plaintext
LDA ,X
ADD ,X
DBNZ ,X, rel
```

**NOTE**

Pseudo instructions are based on emulation, they have equivalent HC08/S08 operations. However in term of CPU cycle count and instruction byte count, they are not the same. Special care is needed for timing critical software before migrating source code from HC08/S08 platform to RS08 platform.

### 1.2 RS08 Instruction Set

The RS08 CPU core can be considered as a reduced version of S08 core. Most arithmetic operations are retained in the RS08 platform such that source code compatibility is maintained as much as possible. However, the RS08 platform is not intended for intensive mathematical calculations, therefore, nibble swap (NSA), multiple (MUL), and divide (DIV) operations were removed from the instruction set.

Since the stacking mechanism is removed, instructions involving the stack pointer (SP) that were in HC08/S08 core were removed from the RS08 core. Code condition register (CCR) contains two status flags, Z-bit and C-bit, only conditional branch instructions involving these bits were included.

Table 1-2 summarizes the difference between RS08 instruction set and S08 instruction set.
### Table 1-2. RS08 and S08 Instruction Set Comparison

<table>
<thead>
<tr>
<th>Description</th>
<th>RS08</th>
<th>S08</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic Operations:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Add with Carry</strong></td>
<td>ADC #opr8, ADC opr8, ADC X^1,2</td>
<td>ADC #opr8, ADC opr8, ADC opr16, ADC opr8,X, ADC opr8,SP, ADC opr16,SP</td>
<td>A ← (A) + (M) + (C), A ← (A) + (X) + (C)^2</td>
</tr>
<tr>
<td><strong>Add without Carry</strong></td>
<td>ADD #opr8, ADD opr8, ADD opr4, ADD X^1,2</td>
<td>ADD #opr8, ADD opr8, ADD opr16, ADD opr8,X, ADD opr8,SP, ADD opr16,SP</td>
<td>A ← (A) + (M), A ← (A) + (X)^2</td>
</tr>
<tr>
<td><strong>Add Immediate Value (Signed) to Stack Pointer</strong></td>
<td>AIS #opr8</td>
<td>SP ← (SP) + (16 « M)</td>
<td></td>
</tr>
<tr>
<td><strong>Add Immediate Value (Signed) to Index Register (H:X)</strong></td>
<td>AIS #opr8</td>
<td>H:X ← (H:X) + (16 « M)</td>
<td></td>
</tr>
<tr>
<td><strong>Arithmetic Shift Left (Same as LSL)</strong></td>
<td>ASLA</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Arithmetic Shift Right</strong></td>
<td>ASR #opr8, ASR X, ASR opr8,SP</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Clear</strong></td>
<td>CLR opr8, CLR opr5, CLR X^1, CLR X^1, CLR, CLR, CLR X, CLR opr8,SP</td>
<td>CLR opr8, CLR X, CLR opr8,SP</td>
<td>M ← $00, A ← $00, X ← $00</td>
</tr>
<tr>
<td><strong>Decimal Adjust Accumulator</strong></td>
<td>DAA</td>
<td></td>
<td>(A)_{10}</td>
</tr>
<tr>
<td><strong>Decrement</strong></td>
<td>DEC opr8, DEC opr4, DEC X^1, DEC X^1</td>
<td>DEC opr8, DEC opr8,X, DEC X, DEC opr8,SP</td>
<td>M ← (M) - $01, A ← (A) - $01, X ← (X) - $01</td>
</tr>
<tr>
<td><strong>Divide</strong></td>
<td>DIV</td>
<td>A ← (H:A)/(X), H ← Remainder</td>
<td></td>
</tr>
<tr>
<td><strong>Increment</strong></td>
<td>INC opr8, INC opr4, INC X^1, INC X^1</td>
<td>INC opr8, INC opr8,X, INC X, INC opr8,SP</td>
<td>M ← (M) + $01, A ← (A) + $01, X ← (X) + $01</td>
</tr>
</tbody>
</table>
### Table 1-2. RS08 and S08 Instruction Set Comparison (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>RS08</th>
<th>S08</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negate</td>
<td>NEG</td>
<td>SBC</td>
<td>( M \leftarrow -(M) = 0000 \text{~} -(M) )</td>
</tr>
<tr>
<td>(Two’s Complement)</td>
<td>Nega</td>
<td>SBC</td>
<td>( A \leftarrow -(A) = 0000 \text{~} -(M) )</td>
</tr>
<tr>
<td></td>
<td>Negx</td>
<td>SBC</td>
<td>( X \leftarrow -(X) = 0000 \text{~} -(X) )</td>
</tr>
<tr>
<td></td>
<td>neg</td>
<td>SBC</td>
<td>( opr8, \text{~}X )</td>
</tr>
<tr>
<td></td>
<td>nega</td>
<td>SBC</td>
<td>( opr8, \text{~}SP )</td>
</tr>
<tr>
<td>Subtract with Carry</td>
<td>SBC opr8</td>
<td>SBC</td>
<td>( A \leftarrow (A) \text{<del>} (M) \text{</del>} (C) )</td>
</tr>
<tr>
<td></td>
<td>SBC opr16</td>
<td>SBC</td>
<td>( A \leftarrow (A) \text{<del>} (X) \text{</del>} (C) )</td>
</tr>
<tr>
<td>Subtract</td>
<td>SUB</td>
<td>SUB</td>
<td>( A \leftarrow (A) \text{~} (M) )</td>
</tr>
<tr>
<td></td>
<td>SUB opr8</td>
<td>SUB</td>
<td>( A \leftarrow (A) \text{~} (X) )</td>
</tr>
<tr>
<td></td>
<td>SUB opr8,X</td>
<td>SUB</td>
<td>( A \leftarrow (A) \text{<del>} (X) \text{</del>} (X) )</td>
</tr>
<tr>
<td></td>
<td>SUB opr8,SP</td>
<td>SUB</td>
<td>( A \leftarrow (A) \text{<del>} (X) \text{</del>} (X) )</td>
</tr>
<tr>
<td>Logical Operations:</td>
<td>Logical AND</td>
<td>AND</td>
<td>( A \leftarrow (A) \text{<del>} (M) \text{</del>} (X) )</td>
</tr>
<tr>
<td></td>
<td>AND opr8</td>
<td>AND</td>
<td>( A \leftarrow (A) \text{<del>} (X) \text{</del>} (X) )</td>
</tr>
<tr>
<td></td>
<td>AND opr8</td>
<td>AND</td>
<td>( A \leftarrow (A) \text{<del>} (X) \text{</del>} (X) )</td>
</tr>
<tr>
<td></td>
<td>AND opr16</td>
<td>AND</td>
<td>( A \leftarrow (A) \text{<del>} (X) \text{</del>} (X) )</td>
</tr>
<tr>
<td></td>
<td>AND opr16,X</td>
<td>AND</td>
<td>( A \leftarrow (A) \text{<del>} (X) \text{</del>} (X) )</td>
</tr>
<tr>
<td></td>
<td>AND opr8,SP</td>
<td>AND</td>
<td>( A \leftarrow (A) \text{<del>} (X) \text{</del>} (X) )</td>
</tr>
<tr>
<td></td>
<td>AND opr16,SP</td>
<td>AND</td>
<td>( A \leftarrow (A) \text{<del>} (X) \text{</del>} (X) )</td>
</tr>
<tr>
<td>Clear Bit ( n ) in Memory</td>
<td>BCLR</td>
<td>BCLR</td>
<td>( Mn \leftarrow 0, \text{~} Xn \leftarrow 0 )</td>
</tr>
<tr>
<td></td>
<td>n,opr8</td>
<td>n,opr8</td>
<td>( Mn \leftarrow 0, \text{~} Xn \leftarrow 0 )</td>
</tr>
<tr>
<td></td>
<td>n,X1,2</td>
<td>n,X1,2</td>
<td>( Mn \leftarrow 0, \text{~} Xn \leftarrow 0 )</td>
</tr>
<tr>
<td>Set Bit ( n ) in Memory</td>
<td>BSET</td>
<td>BSET</td>
<td>( Mn \leftarrow 1, \text{~} Xn \leftarrow 1 )</td>
</tr>
<tr>
<td></td>
<td>n,opr8</td>
<td>n,opr8</td>
<td>( Mn \leftarrow 1, \text{~} Xn \leftarrow 1 )</td>
</tr>
<tr>
<td></td>
<td>n,X1,2</td>
<td>n,X1,2</td>
<td>( Mn \leftarrow 1, \text{~} Xn \leftarrow 1 )</td>
</tr>
<tr>
<td>Complement (One’s Complement)</td>
<td>COMA</td>
<td>COMA</td>
<td>( M \leftarrow (M) = 0000 \text{~} -(M) )</td>
</tr>
<tr>
<td></td>
<td>COMA</td>
<td>COMA</td>
<td>( A \leftarrow -(A) = 0000 \text{~} -(A) )</td>
</tr>
<tr>
<td></td>
<td>COMX</td>
<td>COMX</td>
<td>( X \leftarrow -(X) = 0000 \text{~} -(X) )</td>
</tr>
<tr>
<td></td>
<td>COM opr8</td>
<td>COM opr8</td>
<td>( M \leftarrow (M) = 0000 \text{~} -(M) )</td>
</tr>
<tr>
<td></td>
<td>COM opr8</td>
<td>COM opr8</td>
<td>( A \leftarrow -(A) = 0000 \text{~} -(A) )</td>
</tr>
<tr>
<td></td>
<td>COM opr8</td>
<td>COM opr8</td>
<td>( X \leftarrow -(X) = 0000 \text{~} -(X) )</td>
</tr>
<tr>
<td>Exclusive OR Memory with Accumulator</td>
<td>EOR</td>
<td>EOR</td>
<td>( A \leftarrow (A \text{~} M) )</td>
</tr>
<tr>
<td></td>
<td>#opr8</td>
<td>#opr8</td>
<td>( A \leftarrow (A \text{~} M) )</td>
</tr>
<tr>
<td></td>
<td>#opr8</td>
<td>#opr8</td>
<td>( A \leftarrow (A \text{~} M) )</td>
</tr>
<tr>
<td></td>
<td>#opr8,X</td>
<td>#opr8,X</td>
<td>( A \leftarrow (A \text{~} M) )</td>
</tr>
<tr>
<td></td>
<td>#opr8,SP</td>
<td>#opr8,SP</td>
<td>( A \leftarrow (A \text{~} M) )</td>
</tr>
<tr>
<td></td>
<td>#opr8,SP</td>
<td>#opr8,SP</td>
<td>( A \leftarrow (A \text{~} M) )</td>
</tr>
<tr>
<td>Logical Shift Left (Same as ASL)</td>
<td>LSLA</td>
<td>LSLA</td>
<td></td>
</tr>
</tbody>
</table>
### Table 1-2. RS08 and S08 Instruction Set Comparison (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>RS08</th>
<th>S08</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Logical Shift Right</strong></td>
<td>LSRA</td>
<td>LSR</td>
<td>[0 \rightarrow b_7 \rightarrow b_0]</td>
</tr>
<tr>
<td><strong>Nibble Swap Accumulator</strong></td>
<td>NSA</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Inclusive OR Accumulator and Memory</strong></td>
<td>ORA #opr8</td>
<td>ORA #opr8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ORA opr8</td>
<td>ORA opr8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ORA opr16</td>
<td>ORA opr16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ORA opr8,X</td>
<td>ORA opr8,X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ORA opr16,X</td>
<td>ORA opr16,X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ORA X</td>
<td>ORA X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ORA opr8,SP</td>
<td>ORA opr8,SP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ORA opr16,SP</td>
<td>ORA opr16,SP</td>
<td></td>
</tr>
</tbody>
</table>
| **Rotate Left through Carry** | ROL | ROL | | | \[\begin{array}{c|c|c}
| \hline
| b_7 | b_7 | b_0 | b_0 \\
| \hline
\end{array}\] |
| **Rotate Right through Carry** | ROR | ROR | | | \[\begin{array}{c|c|c}
| \hline
| b_7 | b_7 | b_0 | b_0 \\
| \hline
\end{array}\] |

**Branch Operations:**

- **Branch if Carry Bit Clear**
  - BCC rel
  - BCC rel
  - \[PC \leftarrow (PC) + 0002 + rel ? (C) = 0\]

- **Branch if Carry Bit Set (Same as BLO)**
  - BCS rel
  - BCS rel
  - \[PC \leftarrow (PC) + 0002 + rel ? (C) = 1\]

- **Branch if Equal**
  - BEQ rel
  - BEQ rel
  - \[PC \leftarrow (PC) + 0002 + rel ? (Z) = 1\]

- **Branch if Greater Than or Equal To (Signed Operands)**
  - BGE opr
  - BGE opr
  - \[PC \leftarrow (PC) + 0002 + rel ? (N \oplus V) = 0\]

- **Branch if Greater Than (Signed Operands)**
  - BGT opr
  - BGT opr
  - \[PC \leftarrow (PC) + 0002 + rel ? (Z) \mid (N \oplus V) = 0\]

- **Branch if Half Carry Bit Clear**
  - BHCC rel
  - BHCC rel
  - \[PC \leftarrow (PC) + 0002 + rel ? (Z) = 0\]

- **Branch if Half Carry Bit Set**
  - BHCS rel
  - BHCS rel
  - \[PC \leftarrow (PC) + 0002 + rel ? (H) = 0\]

- **Branch if Higher**
  - BHI rel
  - BHI rel
  - \[PC \leftarrow (PC) + 0002 + rel ? (C) \mid (Z) = 0\]

- **Branch if Higher or Same (Same as BCC)**
  - BHS rel
  - BHS rel
  - \[PC \leftarrow (PC) + 0002 + rel ? (C) = 0\]

- **Branch if IRQ Pin High**
  - BIH rel
  - BIH rel
  - \[PC \leftarrow (PC) + 0002 + rel ? \text{IRQ} = 1\]

- **Branch if IRQ Pin Low**
  - BIL rel
  - BIL rel
  - \[PC \leftarrow (PC) + 0002 + rel ? \text{IRQ} = 0\]

- **Branch if Less Than or Equal To (Signed Operands)**
  - BLE opr
  - BLE opr
  - \[PC \leftarrow (PC) + 0002 + rel ? (Z) \mid (N \oplus V) = 1\]

- **Branch if Lower (Same as BCS)**
  - BLO rel
  - BLO rel
  - \[PC \leftarrow (PC) + 0002 + rel ? (C) = 1\]

- **Branch if Lower or Same**
  - BLS rel
  - BLS rel
  - \[PC \leftarrow (PC) + 0002 + rel ? (C) \mid (Z) = 1\]
### Introduction to RS08

#### Table 1-2. RS08 and S08 Instruction Set Comparison (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>RS08</th>
<th>S08</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch if Less Than (Signed Operands)</td>
<td>BLT opr</td>
<td>PC ← (PC) + $0002 + rel ? (N ⊕ V) = 1</td>
<td></td>
</tr>
<tr>
<td>Branch if Interrupt Mask Clear</td>
<td>BMC rel</td>
<td>PC ← (PC) + $0002 + rel ? (I) = 0</td>
<td></td>
</tr>
<tr>
<td>Branch if Minus</td>
<td>BMI rel</td>
<td>PC ← (PC) + $0002 + rel ? (N) = 1</td>
<td></td>
</tr>
<tr>
<td>Branch if Interrupt Mask Set</td>
<td>BMS rel</td>
<td>PC ← (PC) + $0002 + rel ? (I) = 1</td>
<td></td>
</tr>
<tr>
<td>Branch if Not Equal</td>
<td>BNE rel</td>
<td>PC ← (PC) + $0002 + rel ? (Z) = 0</td>
<td></td>
</tr>
<tr>
<td>Branch if Plus</td>
<td>BPL rel</td>
<td>PC ← (PC) + $0002 + rel ? (N) = 0</td>
<td></td>
</tr>
<tr>
<td>Branch Always</td>
<td>BRA rel</td>
<td>PC ← (PC) + $0002 + rel</td>
<td></td>
</tr>
<tr>
<td>Branch if Bit n in Memory Clear</td>
<td>BRCLR n, opr8 rel</td>
<td>PC ← (PC) + $0003 + rel ? (Mn) = 0</td>
<td></td>
</tr>
<tr>
<td>Branch Never</td>
<td>BRN rel</td>
<td>PC ← (PC) + $0002</td>
<td></td>
</tr>
<tr>
<td>Branch if Bit n in Memory Set</td>
<td>BRSET n, opr8 rel</td>
<td>PC ← (PC) + $0003 + rel ? (Mn) = 1</td>
<td></td>
</tr>
<tr>
<td>Branch to Subroutine</td>
<td>BSR rel</td>
<td>BSR rel</td>
<td></td>
</tr>
<tr>
<td>Compare and Branch if Equal</td>
<td>CBEQ opr8 rel</td>
<td>PC ← (PC) + $0002; push (PCL)</td>
<td></td>
</tr>
<tr>
<td>Decrement and Branch if Not Zero</td>
<td>DBNZ opr8 rel</td>
<td>PC ← (PC) + $0003 + rel ? (A) – (M) = $00</td>
<td></td>
</tr>
<tr>
<td>Jump</td>
<td>JMP opr16</td>
<td>PC ← Jump Address</td>
<td></td>
</tr>
</tbody>
</table>

For RS08:

- Push PC to shadow PC
- PC ← (PC) + rel

For S08:

- Push (PCL) to (SP)
- SP ← (SP) + $0001
- PC ← (PC) + rel

For RS08:

- Push PC to shadow PC
- PC ← (PC) + rel

For S08:

- SP ← (SP) – $0001
- PC ← (PC) + $0001
- PC ← (PC) + rel

For RS08:

- Push PC to shadow PC
- PC ← (PC) + rel

- PC ← (PC) + $0002

For S08:

- Push (PCH) to (SP)
- SP ← (SP) – $0001
- PC ← (PC) + rel

For RS08:

- Push PC to shadow PC
- PC ← (PC) + rel

For S08:

- Push (PCH) to (SP)
- SP ← (SP) – $0001
- PC ← (PC) + rel

For RS08:

- Push PC to shadow PC
- PC ← (PC) + rel

For S08:

- Push (PCH) to (SP)
- SP ← (SP) – $0001
- PC ← (PC) + rel

For RS08:

- Push PC to shadow PC
- PC ← (PC) + rel

For S08:

- Push (PCH) to (SP)
- SP ← (SP) – $0001
- PC ← (PC) + rel

For RS08:

- Push PC to shadow PC
- PC ← (PC) + rel

For S08:

- Push (PCH) to (SP)
- SP ← (SP) – $0001
- PC ← (PC) + rel

For RS08:

- Push PC to shadow PC
- PC ← (PC) + rel

For S08:

- Push (PCH) to (SP)
- SP ← (SP) – $0001
- PC ← (PC) + rel

For RS08:

- Push PC to shadow PC
- PC ← (PC) + rel
**Table 1-2. RS08 and S08 Instruction Set Comparison (continued)**

<table>
<thead>
<tr>
<th>Description</th>
<th>RS08</th>
<th>S08</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jump to Subroutine</td>
<td>JSR opr16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Return from Subroutine</td>
<td>RTS</td>
<td>RTS</td>
<td></td>
</tr>
</tbody>
</table>

**Data Verification Operations:**

<table>
<thead>
<tr>
<th>Bit Test</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare Accumulator with Memory</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complement (One’s Complement)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compare Index Register (H:X) with Memory</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test for Negative or Zero</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Data Movement Operations:**

| Load Accumulator from Memory |                       |                      |           |

---

For RS08:
- PC ← (PC) + 3
- Push PC to shadow PC
- PC ← Unconditional Address

For S08:
- PC ← (PC) + n (n = 1, 2, or 3)
- Push (PCL): SP ← (SP) – $0001
- Push (PCH): SP ← (SP) – $0001
- PC ← Unconditional Address

For RS08:
- SP ← SP + $0001; Pull (PCH)
- SP ← SP + $0001; Pull (PCL)
- Pull PC from shadow PC

For S08:
- SP ← SP + $0001; Pull (PCH)

---

1. Instruction available only on 16-bit addressing modes.
2. Implemented as a special case of the 8-bit instructions.
### Table 1-2. RS08 and S08 Instruction Set Comparison (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>RS08</th>
<th>S08</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Index Register (H:X) from Memory</td>
<td>LDHX #opr16&lt;br&gt;LDHX opr8&lt;br&gt;LDHX opr16&lt;br&gt;LDHX&lt;br&gt;LDHX opr8,X&lt;br&gt;LDHX opr16,X&lt;br&gt;LDHX opr8,SP</td>
<td></td>
<td>H:X ← (M:M + $00001)</td>
</tr>
<tr>
<td>Load X (Index Register Low) from Memory</td>
<td>LDX #opr8 1&lt;br&gt;LDX opr8 1&lt;br&gt;LDX opr8&lt;br&gt;LDX&lt;br&gt;LDX opr8,SP&lt;br&gt;LDX opr16,SP</td>
<td></td>
<td>X ← (M)</td>
</tr>
<tr>
<td>Move</td>
<td>MOV opr8,opr8&lt;br&gt;Mov #opr8,opr8&lt;br&gt;Mov D[X],opr8 1&lt;br&gt;Mov opr8,D[X] 1&lt;br&gt;Mov #opr8,D[X] 1</td>
<td>MOV opr8,opr8&lt;br&gt;Mov #opr8,opr8&lt;br&gt;Mov opr8,X+&lt;br&gt;Mov opr8,opr8&lt;br&gt;Mov X+,opr8</td>
<td>For S08/RS08: (M)destination ← (M)source&lt;br&gt;For S08 only: H:X ← (H:X) + $001 in IX+D and DIX+ Modes</td>
</tr>
<tr>
<td>Store Accumulator in Memory</td>
<td>STA opr8&lt;br&gt;STA opr5&lt;br&gt;STA ,X 1&lt;br&gt;STA opr8,X&lt;br&gt;STA opr8,SP&lt;br&gt;STA opr16,SP</td>
<td></td>
<td>M ← (A)</td>
</tr>
<tr>
<td>Store H:X (Index Reg.)</td>
<td>STHX opr&lt;br&gt;STHX opr&lt;br&gt;STHX opr&lt;br&gt;STHX opr,SP</td>
<td></td>
<td>(M:M + $0001) ← (H:X)</td>
</tr>
<tr>
<td>Store X (Index Register Low) in Memory</td>
<td>STX opr8 1&lt;br&gt;STX opr8&lt;br&gt;STX opr8&lt;br&gt;STX opr8,SP&lt;br&gt;STX opr16,SP</td>
<td></td>
<td>M ← (X)</td>
</tr>
<tr>
<td>Transfer Accumulator to CCR</td>
<td>TAP</td>
<td></td>
<td>CCR ← (A)</td>
</tr>
<tr>
<td>Transfer Accumulator to X (Index Register Low)</td>
<td>TAX 1</td>
<td></td>
<td>X ← (A)</td>
</tr>
<tr>
<td>Transfer CCR to Accumulator</td>
<td>TPA</td>
<td></td>
<td>A ← (CCR)</td>
</tr>
<tr>
<td>Transfer SP to Index Reg.</td>
<td>TSX</td>
<td></td>
<td>H:X ← (SP) + $0001</td>
</tr>
<tr>
<td>Transfer X (Index Reg. Low) to Accumulator</td>
<td>TXA 1</td>
<td></td>
<td>A ← (X)</td>
</tr>
<tr>
<td>Transfer Index Reg. to SP</td>
<td>TXS</td>
<td></td>
<td>(SP) ← (H:X) – $0001</td>
</tr>
</tbody>
</table>

**Other Operations:**

- Background: BGND
- Clear Carry Bit: CLC
- Clear Interrupt Mask Bit: CLI
- No Operation: NOP
- Push Accumulator onto Stack: PSHA
- Push H (Index Register High) onto Stack: PSHH
- Enter Background Debug Mode
- None
- Push (A); SP ← (SP) – $0001
- Push (H); SP ← (SP) – $0001
Tiny and Short Addressing Mode Instructions

Tiny and short addressing mode instructions are single byte instructions. Maximizing the use of these instructions can efficiently improve the overall code density. Given the limited addressable space for these instructions, careful planning to allocate the most frequently used variables to be located within the tiny and short addressable area is recommended. Table 1-3 summarizes the tiny and short instructions support for the RS08 platform.
1.2.2 Pseudo Instructions

Using register X located in $000F and register D[0] located in $000E, most HC08/S08 zero offset index addressing instructions and accumulator instructions can be emulated. This index addressing can be performed on virtually all direct addressing mode instructions. Table 1-4 summarizes all of the pseudo instructions supported in RS08 platform and their operations.

NOTE
Instruction translation is done during time of compilation by the assembler, and is transparent to the user.
### 1.3 Paging Memory Scheme

The RS08 instruction set does not include extended addressing capability. There is a 64-byte window, known as the paging window, from location $00C0$ to $00FF$, in the direct page reserved for paging access. A page selection (PAGESEL) register ($001F$) determines the corresponding 64-byte block in the memory map for the paging window access. Upper memory access can be done by direct-page access through the paging window area.

The entire accessible memory space for RS08 is 16K-bytes, and divided into 256 pages of 64-byte memory. Programming the PAGESEL register ($001F$) defines the page to be accessed through the paging window. Figure 1-4 illustrates the paging memory scheme.
The PAGESEL register defines the memory page to be accessed, the register X indicates the corresponding location in the paging window that points to the desired upper memory location, CPU access through register D[X] and the paging window can index to the corresponding upper memory location. Most pseudo instructions can utilize this scheme to perform index addressing to the upper memory locations.

NOTE
Accessing any unimplemented location through the paging window will generate an illegal address reset.

Figure 1-4. RS08 Paging Scheme

1.4 MCU Reset

MCU reset provides a way to restart the MCU to a known set of initial conditions. An MCU reset forces most control and status registers to their initial values and the program counter (PC) is started from $3FFD. In the RS08 platform there is no vector lookup mechanism, a JMP instruction (opcode $BC) with a 2-byte operand must programmed into the locations $3FFD–$3FFF. The operand indicates the user defined location to start user program execution.

```assembly
; Reset Vector
org $3FFC
Security:
dc.b $FF ; SECD=1 is unsecured, SECD=0 is secured
jmp main
```

Getting Started with RS08, Rev. 1
Similar to the HC08/S08 devices, RS08 has seven sources for reset:

- External pin reset (PIN) — enabled using RSTPE and SOPT
- Power-on reset (POR)
- Low-voltage detect (LVD)
- Computer operating properly (COP) timer
- Illegal opcode detect (ILOP)
- Illegal address detect (ILAD)
- Background debug forced reset via BDC command BDC_RESET

The system reset status register (SRS) located in $0200 includes read-only status flags to indicate the source of the most recent reset.

1.5 **Wait Mode**

Wait mode is entered by executing a WAIT instruction. Upon execution of the WAIT instruction, the CPU enters a low-power state in which it is not clocked. The program counter (PC) is halted at the position following the WAIT instruction where it is executed. Exit from wait is done by asserting any reset and any type of interrupt sources that has been enabled. When an interrupt request occurs:

1. MCU exists wait mode and resumes processing.
2. Fetches the following instruction and program flow continues.

It is the responsibility of the user program to probe the corresponding interrupt source that woke the MCU because no vector fetching process is involved.

1.6 **Stop Mode**

Stop mode is entered upon execution of a STOP instruction when the STOPE bit in the system option register is set. In STOP mode all internal clocks to the CPU and the modules are halted. Exit from stop is done by asserting any reset, any asynchronous interrupt such as KBI that has been enabled, or the real-time interrupt. When the requests occurs:

1. MCU clock module is enabled.
2. MCU exists stop mode and resumes processing.
3. Fetches the following instruction and program flow continues.

It is the responsibility of the user program to poll the corresponding interrupt source that woke the MCU, because no vector fetching process is involved.

There are options to enable various modules, such as the internal clock source (ICS) and analog comparator (ACMP), during stop mode. Please refer to the specific device data sheet for more details.

**NOTE**

If the STOPE bit is not set when the CPU executes a STOP instruction, the MCU will not enter stop mode and an illegal opcode reset is forced.
1.7 Subroutine Call

The RS08 platform provides only a single level of hardware stacking. When the instruction, JSR or BSR, is executed, current program counter (PC) value is uploaded to the shadow program counter (SPC) register before the PC is modified with a new location. In the case when the program encounters the instruction RTS, the saved PC value is restored from the SPC register. Program execution resumes at the address that was just restored from SPC register.

Single level of subroutine call may not be sufficient for some applications, multi-level software stacking can be emulated with the help of SHA/SLA instructions. These instructions exchange the high byte and the low byte of SPC register with accumulator A respectively. Software stacking can be implemented that place the SPC content for each level of subroutine call in RAM.

The following code shows how software stacking can be implemented in macro format. In this example, location $00 is arbitrarily chosen for the stack pointer (STACKPTR) variable and the stack content is placed from address $4F downwards. The code shown provides no stack overflow checking.

```assembly
SPInit equ $4F ; Stack block allocation
FLASHSTART equ $3800 ; For MC9RS08KA2

RESETSP: MACRO
  mov #SPInit, STACKPTR ; Init Stack pointer
ENDM

PSH_SPC: MACRO ; 20 CPU cycles, 14 bytes code
  ; NOTE: Destructive to X content
  lda STACKPTR ; Load Stack pointer
  sha ; Swap SPC high byte
  sta ,X ; Push high byte to stack
  sha ; Resume A content
  decx ; update stack pointer
  sla ; Swap SPC low byte
  sta ,X ; Push low byte to stack
  sla ; Resume A content
  decx ; update stack pointer
  stx STACKPTR ; Save stack pointer
ENDM

PUL_SPC: MACRO ; 22 CPU cycles, 14 byte code
  ; NOTE: Destructive to X content
  lda STACKPTR ; Load Stack pointer
  incx ; Update stack pointer
  sla ; Swap SPC low byte
  lda ,X ; Pull low byte
  sla ; Resume A and SPCL content
  incx ; Update stack pointer
  sha ; Swap SPC high byte
  lda ,X ; Pull high byte
  sha ; Resume A and SPCH content
  stx STACKPTR ; Save stack pointer
ENDM

org TINY_RAM
STACKPTR ds.b 1 ; Stack pointer location
org FLASHSTART
```
SubA:
PSH_SPC ; Stack SPC
;... <Subroutine Content> ...
brr SubB ; Multi-level subroutine call
PUL_SPC ; Unstack SPC
rts
SubB:
PSH_SPC ; Stack SPC
;... <Subroutine Content> ...
PUL_SPC ; Unstack SPC
rts
Main:
RESETSP
;... <Software Content> ...
jsr SubA
;... <Software Content> ...
jsr SubB
;... <Software Content> ...

Three macros are defined here. RESETSP is used to reset the stack pointer to the initial position. PSH_SPC pushes shadow PC (SPC) content to stack and decrements STACKPTR variable accordingly. Similarly, PUL_SPC pulls the SPC content from stack and increments STACKPTR variable accordingly. Calling PSH_SPC at the beginning of each subroutine and PUL_SPC before executing RTS would stack up and retrieve the return address (shadow PC) for each level of subroutine calls accordingly.

NOTE
Both PSH_SPC and PUL_SPC macro are destructive to register X. If X content requires to carry across subroutine calls, enhancements to the macros are required.

1.8 Interrupt
RS08 platform is targeted for small applications where usually intensive interrupt servicing is not required. The interrupt request in the RS08 platform is designed to wake the MCU from either wait or stop mode. At the same time the corresponding interrupt flags will also be set to indicate the interrupt events that had happened. If multiple events had happened, it is up to the software to decide the priority of servicing. When the MCU is operating in run mode or active background debug mode (BDM), interrupt events will not affect the software flow. Users can check the interrupt events on a regular basis by polling the corresponding interrupt flag and determine if interrupt service is required.

Similar to the HC08/S08 platform, in RS08 each interrupt source is associated with a corresponding interrupt flag and an interrupt enable bit. The wait/stop wakeup capability of an interrupt source can only
be enabled when the corresponding interrupt enable bit is set. When the MCU wakes up from wait/stop mode, the program flow is resumed from where it was stopped. At this point, software can determine which interrupt had occurred by polling the interrupt flags and then jump to the service subroutine accordingly.

The interrupt flags from individual modules are scattered in several register locations, therefore it is not efficient for the software to poll the corresponding flag among several registers. The RS08 platform implements a system interrupt pending (SIP1) register where it provides a central location for the interrupt sources notification. If hardware interrupt is enabled, the corresponding flag in SIP1 register will be set when the interrupt event occurs. For example, if keyboard interrupt is required, it can be enabled by setting the KBIE bit in KBISC register. When KBI event occurs, KBF flag in KBISC register and KBI flag in SIP1 register are both set. User has a choice to poll either of these bits to determine of the event existence. Writing a logic 1 to KBACK bit in KBISC register will clear both KBF in KBISC and KBI flag in SIP1.

### 1.8.1 Interrupt Handling Coding Example

The interrupt sources associated with the MC9RS08KA2 are shown below:

- Low voltage detect (LVD)
- Real timer interrupt (RTI)
- Modulo timer overflow (MTIM)
- Analog comparator (ACMP)
- Keyboard interrupt (KBI)

First, the priority of servicing should be defined based on the application need. In general, the interrupt that requires the shortest latency should have the highest priority. To illustrate the idea the servicing priority is arbitrarily defined as follows:

<table>
<thead>
<tr>
<th>Highest</th>
<th>Lowest</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTIM</td>
<td>LVD</td>
</tr>
<tr>
<td>KBI</td>
<td>RTI</td>
</tr>
<tr>
<td>ACMP</td>
<td></td>
</tr>
</tbody>
</table>

For many interrupt driven applications the interrupt event period is unknown to the application; most of the time the MCU is in idle state and waiting for an event to trigger. Once it happens, the MCU will wakeup and performs a defined task then returns to its idle state. With the priority table defined in Table 1-5, the interrupt servicing loop can be written as follows:

```
InfLoop:    sta    SRS      ;Bump COP
            wait
Priority1:  brset  SIP1_MTIM, SIP1, MTIM_ISR ;5 bus cycles
Priority2:  brset  SIP1_ACMP, SIP1, ACMP_ISR ;5 bus cycles
Priority3:  brset  SIP1_KBI, SIP1, KBI_ISR  ;5 bus cycles
Priority4:  brset  SIP1_RTI, SIP1, RTI_ISR  ;5 bus cycles
Priority5:  brset  SIP1_LVD, SIP1, LVD_ISR  ;5 bus cycles
            bra    InfLoop
MTIM_ISR:   ;... <ISR coding> ...
```
The above example illustrates the software priority handling technique. In the example the MCU enters wait mode during the application idle state. RS08 CPU requires typically three bus cycles to wakeup from wait mode, the interrupt latency is mainly due to the software execution time. Assuming a bus frequency of 10MHz (bus period is 100ns) the corresponding latencies are summarized in Table 1-6. User is free to customize the software loop and minimize the interrupt latency according to the application requirement.

Table 1-6. Interrupt Latency based on 10MHz Bus Clock

<table>
<thead>
<tr>
<th>Interrupt</th>
<th>Latency (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTIM</td>
<td>0.8</td>
</tr>
<tr>
<td>ACMP</td>
<td>1.3</td>
</tr>
<tr>
<td>KBI</td>
<td>1.8</td>
</tr>
<tr>
<td>RTI</td>
<td>2.3</td>
</tr>
<tr>
<td>LVD</td>
<td>2.8</td>
</tr>
</tbody>
</table>

NOTES:

1. Additional delay (typically 2 bus clock cycles) may exist to synchronize the asynchronous interrupt source to the bus clock.

**NOTE**

In the above example COP is refreshed before entering wait mode. In order to avoid a COP reset, at least one interrupt event is expected within the COP timeout period.

In many applications the interrupt period is much longer, it would be wise to put the MCU in stop mode to minimize the power consumption, particularly in battery operated applications. Because the RS08 CPU can only be waked up from stop by asynchronous interrupt source such as KBI, ACMP, etc., all synchronous interrupt events checking such as MTIM can be eliminated from the interrupt servicing loop. For MC9RS08KA2, all interrupt sources except MTIM has stop wakeup capability (refer to MC9RS08KA2 data sheet for more details). On top of the software execution time the interrupt latency from stop must include the MCU stop recovery time that allows the system clock and internal regulator to wakeup from their standby mode. The stop recovery time varies among product families, it depends on the clock module and internal regulator technology used.
2 Emulated ADC Application Example

In this section the analog comparator module in the MC9RS08KA2 is used to implement an 8-bit analog-to-digital (ADC).

In many applications, precise ADC operation is not needed. With a timer module and a low cost high performance analog comparator module built into the MCU, an ADC can be emulated. The emulated ADC resolution depends on the resolution of the timer. In the case of MC9RS08KA2 an 8-bit modulo timer (MTIM) is included, hence an 8-bit ADC operation can easily be emulated. Comparing with a dedicated ADC module the trade-off is the sampling time and the dynamic range. Emulated ADC usually has longer sampling time, narrower dynamic range, and rail-to-rail operation is not feasible.

![Figure 2-5. Emulated ADC Schematic](image)

Figure 2-5 shows the schematic of a simple emulated ADC. The positive terminal of the comparator is connected to a RC network and the negative terminal is the ADC input. Before the comparator function is enabled, both terminals are general I/O ports. The positive terminal is initially set to output low to discharge the RC. When ADC function is required, the comparator is then enabled. The ADC function is emulated by comparing the ADC input to the voltage across the C. Timer is used to monitor the time it takes for the RC to charge up to the ADC input voltage. Since the RC charging profile is not linear, if the ADC dynamic range is small, the timer reading can be used as it is. In general it is more desirable to convert the timer reading back to linear scale using a simple lookup table.

2.1 Implementation

The following is the procedure to use the MC9RS08KA2 to perform the emulated ADC function. The complete program is listed in Appendix A.

1. **Define the sampling time and timer resolution.** The sampling time is the time for the RC to charge up to the maximum ADC input voltage (dynamic range). In this example one millisecond is arbitrarily chosen. When 8-bit timer is used (n=8), the timer resolution is 3.9 µs (the function is given in Equation 1) and is rounded up to 4 µs. Maximum timer overflow is assumed, then overflow period becomes 255 times 4 µs, i.e. 1.02 ms.

   \[
   \text{TimerResolution} = \frac{\text{ChargeUpTime}}{2^n - 1}
   \]

   \textbf{Eqn. -1}
2. **Define RC time constant.** The RC charging profile follows Equation 2.

\[
V = V_{DD}\left(1 - e^{-\frac{t}{RC}}\right)
\]

*Eqn. -2*

The capacitor charge level reaches 99% when the time, \(t\), reaches about 4.6 times of the RC constant. To maximize the measurement range, the timer overflow period is expected to be longer than or equal to this value. In this example, with 1.02ms timer overflow period RC constant becomes 2.21E-4.

\[
RC = \frac{\text{TimerOverflowPeriod}}{4.61}
\]

*Eqn. -3*

The value of the resistor, \(R\), is defined by the port sinking capability. Referring to the data sheet of MC9RS08KA2, the sinking current can keep in around 1mA level so that the initial discharged voltage level can maintain to be close to 0V. Assuming \(V_{DD}\) of 5V is used, 4700\(\Omega\) resistor \(R\) is chosen. Then, given 2.21E-4 time constant, capacitor \(C\) becomes 47nF. Please note this sinking current will contribute to the overall system \(I_{DD}\) consumption. If the ADC function is not used, or before the MCU enters stop mode, it is recommended to configure the port back to input or high impedance to avoid current leakage.

3. **Construct the lookup table.** Given the timer resolution, 4\(\mu\)s in this example, it is possible to construct a lookup table to compensate for the nonlinearity of the charging profile based on Equation 2. The step size for a linear 8-bit ADC is given as:

\[
Step = \frac{V_{DD}}{255}
\]

*Eqn. -4*

**Figure 2-6. ADC Quantization Diagram**

A linear ADC is expected to quantize the input voltage at step boundary starting from step/2 input voltage as shown in Figure 2-6. The conversion function becomes:
Getting Started with RS08, Rev. 1

Emulated ADC Application Example

The lookup table that converts the timer count to linear ADC code is shown in Table 2-7.

<table>
<thead>
<tr>
<th>Time (µs)</th>
<th>ADC Input (V) (Equation 2)</th>
<th>Timer Count</th>
<th>Linear ADC Code (Equation 5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0.09</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>0.18</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>0.26</td>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>16</td>
<td>0.35</td>
<td>4</td>
<td>18</td>
</tr>
<tr>
<td>20</td>
<td>0.43</td>
<td>5</td>
<td>23</td>
</tr>
<tr>
<td>1012</td>
<td>4.95</td>
<td>253</td>
<td>253</td>
</tr>
<tr>
<td>1016</td>
<td>4.95</td>
<td>254</td>
<td>253</td>
</tr>
<tr>
<td>1020</td>
<td>4.95</td>
<td>255</td>
<td>253</td>
</tr>
</tbody>
</table>

4. **Define bus frequency.** There is software overhead to enable the timer and the comparator before taking measurements. To avoid software latency error, it is recommended to choose a bus frequency which is at least five times the timer clock frequency. In this example, a 2MHz bus frequency is initially chosen, then timer prescaler is set to divide-by-8 option which gives 250kHz timer clock frequency, i.e. 4µs resolution. In applications where the choice of bus frequency cannot be chosen freely, the lookup table can be rebuilt to compensate for the software latency.

5. **RS08 coding.** The software code can be divided into four parts: declaration, initialization, ADC read, and table lookup.

   a) First, declare the variables required and the lookup table location. The most frequently used variables should be allocated on the tiny addressable RAM area, i.e. $0000 to $000D, such that the single byte tiny/short instructions can be used for data manipulation. Hence, code density is greatly improved. Lookup table is located in the upper memory, there is no restriction on where to put the table, in this example $3E00 is arbitrarily chosen. All upper memory access is done through the 64-byte paging window located on the first page.
Emulated ADC Application Example

Emulated ADC Application Example

Emulated ADC Application Example

Emulated ADC Application Example

Emulated ADC Application Example

Emulated ADC Application Example

Emulated ADC Application Example

Emulated ADC Application Example

Emulated ADC Application Example

Emulated ADC Application Example

Emulated ADC Application Example

Emulated ADC Application Example

Emulated ADC Application Example

Emulated ADC Application Example

Emulated ADC Application Example

Emulated ADC Application Example

Emulated ADC Application Example

Emulated ADC Application Example

Emulated ADC Application Example

Emulated ADC Application Example

Emulated ADC Application Example

b) Comparator positive terminal must be initialized as output low, so that RC network will start up at a completely discharged state. Coding is shown below:

```assembly
; Init RAM
clr SensorReading ; Single byte instruction
clr ADCOut ; Single byte instruction

; Config GPIO
; RC - init L
mov #(mDATAOUT), PTAD ; RC Initial low
mov #(mRC | mDATAOUT), PTADD ; Set Output pins

; In the ADCRead subroutine, the timer is initialized and started to run before enabling the comparator. Once the comparator is enabled, both of its terminals become analog inputs and the RC network starts to charge up. The MCU then enters wait mode and waiting for interrupt events to trigger. Both timer (MTIM) overflow interrupt and comparator interrupt are enabled since either of these events will wake the MCU up from wait mode. When an interrupt triggers the software flow continues and the following instruction is executed. The timer counter value is read out immediately and save in SensorReading variable. The comparator flag is then checked. If it is clear, it indicates no comparator event occurred. The ADC input could be out of range and the saved SensorReading value is flushed. Otherwise the comparator is disabled, the positive terminal returns to output low and discharges the RC network.

; Read Sensor (ADC) Value
; Timer prescalar=8 -> Timer clk~250kHz
; Bus = 2MHz
; Max OF period = 1.02ms
; Timer resolution = 4us
ADCRead:
  mov  # (MTIM_BUS_CLK | MTIM_DIV_8), MTIM1CLK ; Change Timer resolution
  mov  # 255, MTIM1MOD ; OF period
  mov  # (mMTIM1SC_TRST | mMTIM1SC_TOIE), MTIM1SC ; Reset and Start Timer
  mov  # (mACMP1SC_ACME | mACMP1SC_ACIE | ACMP1SC_OUTPUT_RAISING), ACMP1SC ; Enable ACMP, start RC rise
  bset  ACMP1SC_ACF, ACMP1SC ; Clear ACMP Flag
  wait
  mov  MTIM1CNT, SensorReading
  brclr ACMP1SC_ACF, ACMP1SC, NoReading
```

Getting Started with RS08, Rev. 1
Emulated ADC Application Example

bset ACMP1SC_ACF, ACMP1SC ;Clear ACMP Flag
cir ACMP1SC ;disable ACMP
mov #(mMTIM1SC_TSTP|mMTIM1SC_TRST), MTIM1SC ;mask int and clear flag
rts

NoReading:
mov #$FF, SensorReading ;Biggest Number
cir ACMP1SC ;disable ACMP
mov #(mMTIM1SC_TSTP|mMTIM1SC_TRST), MTIM1SC ;mask int and clear flag
rts

d) In TableLookup subroutine the two most significant bits (MSB) of the variable SensorReading are extracted and added to the page number that holds the lookup table. The corresponding lookup table content is mapped to the 64-byte paging window, $00C0 to $00FF. Then the six least significant bits (LSB) of the variable SensorReading is used as an index to read out the upper memory content directly from the paging window.

;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
; 8bit Table Lookup
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
TableLookup:

lda SensorReading
rola
rola
rola
and #$03
add #(TableStart>>6)
sta PAGESEL
lda SensorReading
and #$3F
add #$c0
tax
lda ,x
sta ADCOut
mov #(HREG), PAGESEL
rts

;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
; ADC Lookup Table - RC charging profile
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
org TableStart
dc.b 0,  5, 10, 14, 18, 23, 27, 31, 35, 39, 43, 47, 50, 54, 58, 61
dc.b 65, 68, 71, 75, 78, 81, 84, 87, 90, 93, 96, 99,102,105,107,110
dc.b 113,115,118,120,123,125,127,130,132,134,136,138,141,143,145,147
dc.b 149,150,152,154,156,158,160,161,163,165,166,168,169,171,173,174
dc.b 175,177,178,180,181,182,184,185,186,188,189,190,191,192,193,195
dc.b 196,197,198,199,200,201,202,203,204,205,206,206,207,208,209,210
dc.b 211,211,212,213,214,215,215,216,217,217,218,219,219,220,221,221
dc.b 222,223,223,224,224,225,225,226,226,227,228,228,229,229,230
...
2.2 Calibration

The emulated ADC performance depends highly on the RC network time constant accuracy. If the actual RC component values deviates from their specified values, the RC charging profile will be shifted and the timer capture will be inaccurate. In addition, variations in parasitic loading on the PCB layout will also contribute to RC time constant error. Simple calibration can be performed to compensate for the change in RC constant.

![Figure 2-7. 2.5V Input R-C Charging Profile](image)

To measure the actual RC constant the charging profile must be recorded. This can be done by applying a $V_{DD}/2$ voltage to the ADC input. The charging profile is recorded as in Figure 2-7. The time taken for the RC network to reach $V_{DD}/2$ voltage level, 264µs in this case. The expected rise time based on the previous calculation listed in Table 2-7 is 152µs, which is equivalent to 38 timer counts. There are several ways to do the calibration:

- From Equation 2 it is possible to deduce the actual RC constant and rebuild the lookup table.
- Instead of using fixed value R or C, variable R or C component can be used. Adjusting the R or the C until the rise time is reduced to the expected value (152µs in this case).
- Compensation can be done by adjusting the timer resolution. MC9RS08KA2 and many Freescale MCUs include a software programmable clock source (ICS), bus frequency can be fine-tuned by simply reprogramming the content of the TRIM register. In this example, timer resolution is 4µs based on the previous calculation with 2MHz bus frequency, 38 timer counts are expected to reach $V_{DD}/2$ voltage level. So, with 264µs measured rise time, new timer resolution should be 264µs divided by 38, i.e. 6.94µs. With a divide-by-8 prescaler option selected for the timer clock source, compensated bus period should be 6.94µs divided by 8, which is 868ns, i.e. 1.15MHz bus frequency. Therefore, if a 1.15MHz bus frequency is used, no hardware adjustment nor lookup table modification is required. For MC9RS08KA2, bus frequency can be changed by reprogramming the TRIM register and bus frequency divider bits in the ICSC2 register. (Refer to MC9RS08KA2 data sheet for more details.)
2.3 Measurement Result

When ADCRead subroutine is executed, the RC network starts the charging process. Once the ADC input voltage matches the RC voltage, the timer counter value is read out and the comparator is disabled. RC network returns to the discharged state. Figure 2-8 shows the charging and discharging process with various ADC input voltages.

![Figure 2-8. RC Charging Profile Against Different ADC Input Voltages](image)

With bus frequency adjusted to 1.15MHz the emulated ADC performance for $V_{DD}=5\text{ V}$ is shown in Table 2-8 and Figure 2-9.

<table>
<thead>
<tr>
<th>ADC Input Voltage (V)</th>
<th>Expected ADC code (Decimal)</th>
<th>Measured ADC code (Decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>51</td>
<td>50</td>
</tr>
<tr>
<td>1.5</td>
<td>76</td>
<td>75</td>
</tr>
<tr>
<td>2</td>
<td>102</td>
<td>99</td>
</tr>
<tr>
<td>2.5</td>
<td>127</td>
<td>123</td>
</tr>
<tr>
<td>3</td>
<td>153</td>
<td>150</td>
</tr>
<tr>
<td>3.5</td>
<td>178</td>
<td>175</td>
</tr>
<tr>
<td>4</td>
<td>204</td>
<td>202</td>
</tr>
<tr>
<td>4.5</td>
<td>229</td>
<td>234</td>
</tr>
</tbody>
</table>
Figure 2-9. Emulated ADC Performance
Emulated ADC Application Example

Appendix A  Program Listing

;******************************************************************************
;
; (c) copyright Freescale Semiconductor, Inc. 2006.
; ALL RIGHTS RESERVED
;
;******************************************************************************
;******************************************************************************
;* Emulated ADC Coding for MC9RS08KA2
;*
;* Author:  Vincent Ko
;* Date:    Jan 2006
;*
;* PTA0/KBI0/ACMP+       RC network
;* PTA1/KBI1/ACMP-       ADCIN
;* PTA5/KBI5            DATAOUT
;*
;******************************************************************************

; include derivative specific macros
XDEF Entry

include "MC9RS08KA2.inc"

;==============================================
; ICS Definition
;==============================================
ICS_DIV_1 equ $00
ICS_DIV_2 equ $40
ICS_DIV_4 equ $80
ICS_DIV_8 equ $c0

;==============================================
; MTIM Definition
;==============================================
MTIM_DIV_1 equ $00
MTIM_DIV_2 equ $01
MTIM_DIV_4 equ $02
MTIM_DIV_8 equ $03
MTIM_DIV_16 equ $04
MTIM_DIV_32 equ $05
MTIM_DIV_64 equ $06
MTIM_DIV_128 equ $07
MTIM_DIV_256 equ $08
MTIM_BUS_CLK equ $00
MTIM_XCLK equ $10
MTIM_TCLK_FALLING equ $20
MTIM_TCLK_RISING equ $30

;==============================================
; ACMP Definition
;==============================================
ACMP_OUTPUT_FALLING equ $00
ACMP_OUTPUT_RISING equ $01
ACMP_OUTPUT_BOTH equ $03

;==============================================
; RTI Definition
;==============================================
RTI_DISABLE equ $00
RTI_8MS equ $01

Getting Started with RS08, Rev. 1
Emulated ADC Application Example

RTI_32MS equ $02
RTI_64MS equ $03
RTI_128MS equ $04
RTI_256MS equ $05
RTI_512MS equ $06
RTI_1024MS equ $07

ORG Tiny_RAMStart
; variable/data section
SensorReading ds.b 1
ADCOut ds.b 1
BitCount ds.b 1

ORG Z_RAMStart
; variable/data section

ORG ROMStart
; code section

main:
Entry:
;---------------------------------------------------------------------
; Config ICS
; Device is pre-trim to 18.4MHz ICLK frequency
; TRIM value are stored in $3FFA:$3FFB
;---------------------------------------------------------------------
mov #$FF, PAGESEL
mov $FB, ICSSC ; $3FFB
mov $FA, ICSTRIM ; $3FFA
mov #(ICS_DIV_8), ICSC2 ; Use 1.15MHz bus

; Config System
;---------------------------------------------------------------------
mov #HREG, PAGESEL ; Init Page register
mov #(mSOPT_COPE|mSOPT_COPT|mSOPT_STOPE), SOPT ; SOFT, COP enabled
mov #(mSPMSC1_LVDE|mSPMSC1_LVDRE), SPMSC1 ; LVI enable
mov #(RTI_128MS|mSRTISC_RTIE), SRTISC ; 128ms RTI

; Init RAM
---------------------------------------------------------------------
clr SensorReading ; Single byte instruction
clr ADCOut ; Single byte instruction

; Config GPIO
; RC - init L
;---------------------------------------------------------------------
mov #(mDATAOUT), PTAD ; RC Initial low
mov #(mRC|mDATAOUT), PTADD ; Set Output pins

Getting Started with RS08, Rev. 1
Emulated ADC Application Example

; Application Loop
; 1) Wakeup every 128ms
; 2) Read ADC input
; 3) Dump code to serially output port (DATAOUT)
;-----------------------------------------

InfLoop:
  wait
  bset SRTISC_RTIACK, SRTISC
  bsr ReadSensor ; Read Charge up time data
  bsr TableLookup ; Decode 8bit level
  bsr DataDump ; Dump ADC code
  sta SRS ; Bump COP
  bra InfLoop

;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
; Read Sensor (ADC) Value
; Timer prescalar=8 -> Timer clk~250kHz
; Bus = 2MHz
; Max OF period = 1.02ms
; Timer resolution = 4us
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ADCRead:
  mov # (MTIM_BUS_CLK|MTIM_DIV_8), MTIM1CLK ; Change Timer resolution
  mov # 255, MTIM1MOD ; OF period
  mov # (mMTIM1SC_TRST|mMTIM1SC_TOIE), MTIM1SC ; Reset and Start Timer
  mov # (mACMP1SC_ACME|mACMP1SC_ACIE|ACMP_OUTPUT_RAISING), ACMP1SC ; Enable ACMP, start RC rise
  bset ACMP1SC_ACF, ACMP1SC ; Clear ACMP Flag
  wait
  brcir ACMP1SC_ACF, ACMP1SC, NoReading
  cir ACMP1SC ; disable ACMP
  mov # (mMTIM1SC_TSTP|mMTIM1SC_TRST), MTIM1SC ; mask int and clear flag
  rts

NoReading:
  mov # $FF, SensorReading ; Biggest Number
  cir ACMP1SC ; disable ACMP
  mov # (mMTIM1SC_TSTP|mMTIM1SC_TRST), MTIM1SC ; mask int and clear flag
  rts

;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
; 8bit Table Lookup
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

TableLookup:
  lda SensorReading ; Extract 2 MSB
  rola
  rola
  rola
  and # $03 ; Mask all other bits
  add # (TableStart>>6) ; Add to Lookup table page
  sta PAGESEL ; High page
  lda SensorReading ; Extract 6 LSB
  and # $3F
  add # $c0 ; Index to paging window
  tax
  lda ,x ; Read upper memory

Getting Started with RS08, Rev. 1
Emulated ADC Application Example

Getting Started with RS08, Rev. 1

Freescale Semiconductor 33
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 use, 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.

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. 2006. All rights reserved.

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.