

Getting Started with RS08

by: Vincent Ko
Systems Engineering
Microcontroller Division

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.

Contents

1	Introduction to RS08	1
1.1	RS08 Architecture	2
1.2	RS08 Instruction Set	6
1.3	Paging Memory Scheme	15
1.4	MCU Reset	16
1.5	Wait Mode	17
1.6	Stop Mode	17
1.7	Subroutine Call	18
1.8	Interrupt	19
2	Emulated ADC Application Example	22
2.1	Implementation	22
2.2	Calibration	27
2.3	Measurement Result	28
Appendix A		
	Program Listing	30

This document contains information on a new product under development. Freescale reserves the right to change or discontinue this product without notice.

© Freescale Semiconductor, Inc., 2006. All rights reserved.



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.

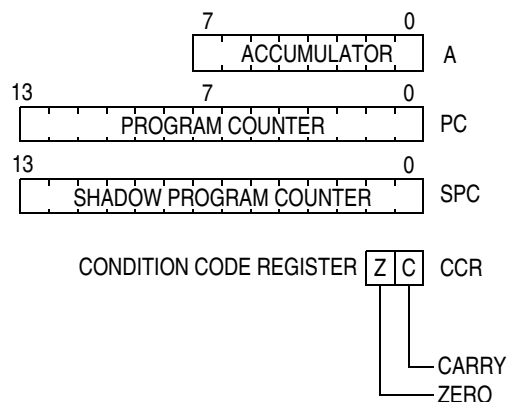


Figure 1-1. RS08 CPU Registers

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.

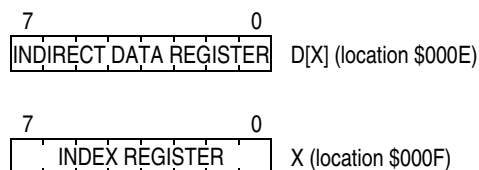


Figure 1-2. RS08 Special Registers

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.

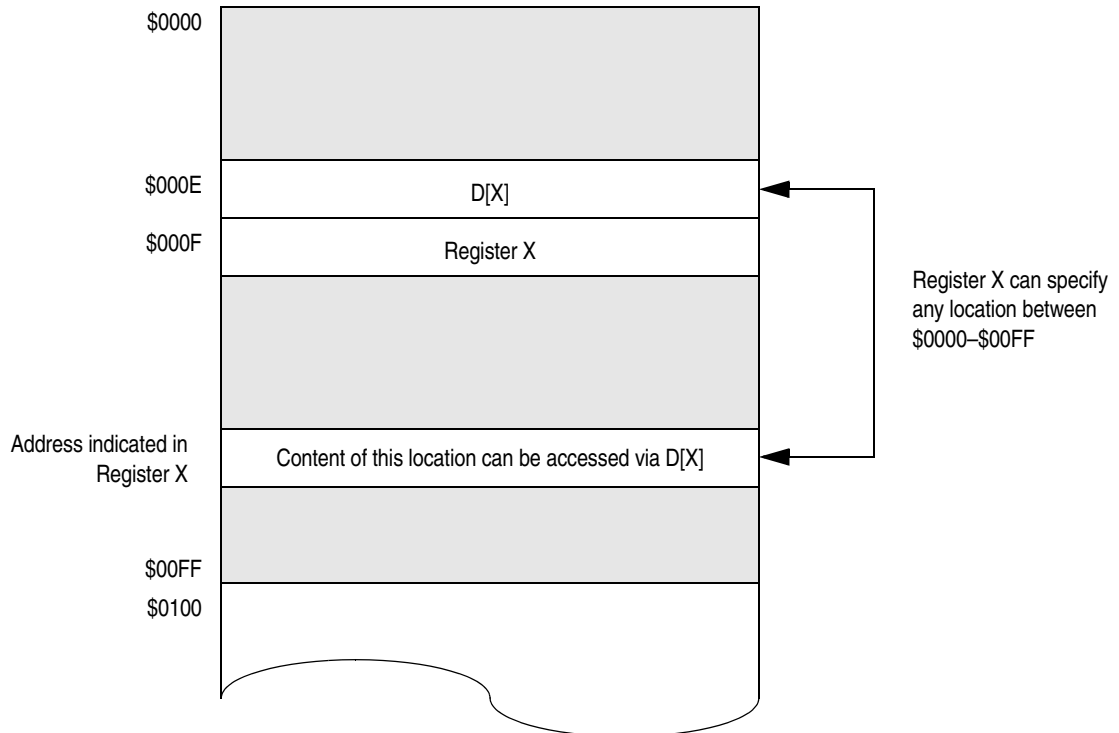


Figure 1-3. Index Addressing Scheme

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 1-1. RS08 Addressing Modes

Addressing Mode	Example
Inherent Addressing	CLRA, INCA, SHA, RTS
Direct Addressing	LDA \$20, AND \$20
Relative Addressing	BRA, BCS, BEQ
Immediate Addressing	LDA #9
Tiny Addressing	INC <\$0D
Short Addressing	CLR <\$1D
Extended Addressing	JMP, JSR

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.

```
INC <$0D
DEC <$0D
ADD <$0D
SUB <$0D
```

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.

```
CLR <$1F
LDA <$1F
STA <$1F
```

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.

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.

```
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

Description	RS08	S08	Operation
Arithmetic Operations:			
Add with Carry	ADC #opr8 ADC opr8 ADC opr16 ADC ,X ¹ ADC X ^{1,2}	ADC #opr8 ADC opr8 ADC opr16 ADC opr8,X ADC opr16,X ADC ,X ADC opr8,SP ADC opr16,SP	$A \leftarrow (A) + (M) + (C)$ $A \leftarrow (A) + (X) + (C)^2$
Add without Carry	ADD #opr8 ADD opr8 ADD opr16 ADD ,X ¹ ADD X ^{1,2}	ADD #opr8 ADD opr8 ADD opr16 ADD opr8,X ADD opr16,X ADD ,X ADD opr8,SP ADD opr16,SP	$A \leftarrow (A) + (M)$ $A \leftarrow (A) + (X)^2$
Add Immediate Value (Signed) to Stack Pointer		AIS #opr8	$SP \leftarrow (SP) + (16 \ll M)$
Add Immediate Value (Signed) to Index Register (H:X)		AIX #opr8	$H:X \leftarrow (H:X) + (16 \ll M)$
Arithmetic Shift Left (Same as LSL)	ASLA	ASL opr8 ASLA ASLX ASL opr8,X ASL ,X ASL opr8,SP	
Arithmetic Shift Right		ASR opr8 ASRA ASRX ASR opr8,X ASR ,X ASR opr8,SP	
Clear	CLR opr8 CLR opr5 CLRA CLR ¹ CLR ,X ¹	CLR opr8 CLRA CLR ¹ CLR opr8,X CLR ,X CLR opr8,SP	$M \leftarrow \$00$ $A \leftarrow \$00$ $X \leftarrow \$00$
Decimal Adjust Accumulator		DAA	$(A)_{10}$
Decrement	DEC opr8 DEC opr4 DECA DECX ¹ DEC ,X ¹	DEC opr8 DECA DECX DEC opr8,X DEC ,X DEC opr8,SP	$M \leftarrow (M) - \$01$ $A \leftarrow (A) - \$01$ $X \leftarrow (X) - \$01$
Divide		DIV	$A \leftarrow (H:A)/(X)$ $H \leftarrow \text{Remainder}$
Increment	INC opr8 INC opr4 INCA INCX ¹ INC ,X ¹	INC opr8 INCA INCX INC opr8,X INC ,X INC opr8,SP	$M \leftarrow (M) + \$01$ $A \leftarrow (A) + \$01$ $X \leftarrow (X) + \$01$

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

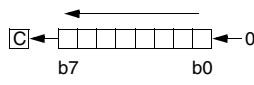
Description	RS08	S08	Operation
Negate (Two's Complement)		NEG <i>opr8</i> NEGA NEGX NEG <i>opr8</i> ,X NEG ,X NEG <i>opr8</i> ,SP	$M \leftarrow \neg(M) = \$00 - (M)$ $A \leftarrow \neg(A) = \$00 - (A)$ $X \leftarrow \neg(X) = \$00 - (X)$
Subtract with Carry	SBC # <i>opr8</i> SBC <i>opr8</i> SBC ,X ¹ SBC X ^{1,2}	SBC # <i>opr8</i> SBC <i>opr8</i> SBC <i>opr16</i> SBC <i>opr8</i> ,X SBC <i>opr16</i> ,X SBC ,X SBC <i>opr8</i> ,SP SBC <i>opr16</i> ,SP	$A \leftarrow (A) - (M) - (C)$ $A \leftarrow (A) - (X) - (C)^2$
Subtract	SUB # <i>opr8</i> SUB <i>opr8</i> SUB <i>opr4</i> SUB ,X ¹ SUB X ^{1,2}	SUB # <i>opr8</i> SUB <i>opr8</i> SUB <i>opr16</i> SUB <i>opr8</i> ,X SUB <i>opr16</i> ,X SUB ,X SUB <i>opr8</i> ,SP SUB <i>opr16</i> ,SP	$A \leftarrow (A) - (M)$ $A \leftarrow (A) - (X)^2$
Logical Operations:			
Logical AND	AND # <i>opr8</i> AND <i>opr8</i> AND ,X ¹ AND X ^{1,2}	AND # <i>opr8</i> AND <i>opr8</i> AND <i>opr16</i> AND <i>opr8</i> ,X AND <i>opr16</i> ,X AND ,X AND <i>opr8</i> ,SP AND <i>opr16</i> ,SP	$A \leftarrow (A) \& (M)$ $A \leftarrow (A) \& (X)^2$
Clear Bit <i>n</i> in Memory	BCLR <i>n</i> , <i>opr8</i> BCLR <i>n</i> ,X ^{1,2} BCLR <i>n</i> ,D[X] ^{1,2}	BCLR <i>n</i> , <i>opr8</i>	$M_n \leftarrow 0$ $X_n \leftarrow 0^2$
Set Bit <i>n</i> in Memory	BSET <i>n</i> , <i>opr8</i> BSET <i>n</i> ,X ^{1,2} BSET <i>n</i> ,D[X] ^{1,2}	BSET <i>n</i> , <i>opr8</i>	$M_n \leftarrow 1$ $X_n \leftarrow 1^2$
Complement (One's Complement)	COMA	COM <i>opr8</i> COMA COMX COM <i>opr8</i> ,X COM ,X COM <i>opr8</i> ,SP	$M \leftarrow \overline{(M)} = \$FF - (M)$ $A \leftarrow \overline{(A)} = \$FF - (M)$ $X \leftarrow \overline{(X)} = \$FF - (M)$
Exclusive OR Memory with Accumulator	EOR # <i>opr8</i> EOR <i>opr8</i> EOR ,X ¹ EOR X ^{1,2}	EOR # <i>opr8</i> EOR <i>opr8</i> EOR <i>opr16</i> EOR <i>opr8</i> ,X EOR <i>opr16</i> ,X EOR ,X EOR <i>opr8</i> ,SP EOR <i>opr16</i> ,SP	$A \leftarrow (A \oplus M)$ $A \leftarrow (A \oplus X)^2$
Logical Shift Left (Same as ASL)	LSLA	LSL <i>opr8</i> LSLA LSLX LSL <i>opr8</i> ,X LSL ,X LSL <i>opr8</i> ,SP	

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

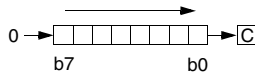
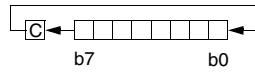
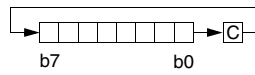
Description	RS08	S08	Operation
Logical Shift Right	LSRA	LSR <i>opr8</i> LSRA LSRX LSR <i>opr8</i> ,X LSR ,X LSR <i>opr8</i> ,SP	
Nibble Swap Accumulator		NSA	$A \leftarrow (A[3:0]:A[7:4])$
Inclusive OR Accumulator and Memory	ORA # <i>opr8</i> ORA <i>opr8</i> ORA ,X ^{1,2} ORA X ^{1,2}	ORA # <i>opr8</i> ORA <i>opr8</i> ORA <i>opr16</i> ORA <i>opr8</i> ,X ORA <i>opr16</i> ,X ORA ,X ORA <i>opr8</i> ,SP ORA <i>opr16</i> ,SP	$A \leftarrow (A) (M)$ $A \leftarrow (A) (X)^2$
Rotate Left through Carry	ROLA	ROL <i>opr</i> ROLA ROLX ROL <i>opr</i> ,X ROL ,X ROL <i>opr</i> ,SP	
Rotate Right through Carry	RORA	ROR <i>opr</i> RORA RORX ROR <i>opr</i> ,X ROR ,X ROR <i>opr</i> ,SP	
Branch Operations:			
Branch if Carry Bit Clear	BCC <i>rel</i>	BCC <i>rel</i>	$PC \leftarrow (PC) + \$0002 + rel ? (C) = 0$
Branch if Carry Bit Set (Same as BLO)	BCS <i>rel</i>	BCS <i>rel</i>	$PC \leftarrow (PC) + \$0002 + rel ? (C) = 1$
Branch if Equal	BEQ <i>rel</i>	BEQ <i>rel</i>	$PC \leftarrow (PC) + \$0002 + rel ? (Z) = 1$
Branch if Greater Than or Equal To (Signed Operands)		BGE <i>opr</i>	$PC \leftarrow (PC) + \$0002 + rel ? (N \oplus V) = 0$
Branch if Greater Than (Signed Operands)		BGT <i>opr</i>	$PC \leftarrow (PC) + \$0002 + rel ? (Z) (N \oplus V) = 0$
Branch if Half Carry Bit Clear		BHCC <i>rel</i>	$PC \leftarrow (PC) + \$0002 + rel ? (H) = 0$
Branch if Half Carry Bit Set		BHCS <i>rel</i>	$PC \leftarrow (PC) + \$0002 + rel ? (H) = 1$
Branch if Higher		BHI <i>rel</i>	$PC \leftarrow (PC) + \$0002 + rel ? (C) (Z) = 0$
Branch if Higher or Same (Same as BCC)	BHS <i>rel</i>	BHS <i>rel</i>	$PC \leftarrow (PC) + \$0002 + rel ? (C) = 0$
Branch if \overline{IRQ} Pin High		BIH <i>rel</i>	$PC \leftarrow (PC) + \$0002 + rel ? \overline{IRQ} = 1$
Branch if \overline{IRQ} Pin Low		BIL <i>rel</i>	$PC \leftarrow (PC) + \$0002 + rel ? \overline{IRQ} = 0$
Branch if Less Than or Equal To (Signed Operands)		BLE <i>opr</i>	$PC \leftarrow (PC) + \$0002 + rel ? (Z) (N \oplus V) = 1$
Branch if Lower (Same as BCS)	BLO <i>rel</i>	BLO <i>rel</i>	$PC \leftarrow (PC) + \$0002 + rel ? (C) = 1$
Branch if Lower or Same		BLS <i>rel</i>	$PC \leftarrow (PC) + \$0002 + rel ? (C) (Z) = 1$

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

Description	RS08	S08	Operation
Branch if Less Than (Signed Operands)		BLT <i>opr</i>	$PC \leftarrow (PC) + \$0002 + rel ? (N \oplus V) = 1$
Branch if Interrupt Mask Clear		BMC <i>rel</i>	$PC \leftarrow (PC) + \$0002 + rel ? (I) = 0$
Branch if Minus		BMI <i>rel</i>	$PC \leftarrow (PC) + \$0002 + rel ? (N) = 1$
Branch if Interrupt Mask Set		BMS <i>rel</i>	$PC \leftarrow (PC) + \$0002 + rel ? (I) = 1$
Branch if Not Equal	BNE <i>rel</i>	BNE <i>rel</i>	$PC \leftarrow (PC) + \$0002 + rel ? (Z) = 0$
Branch if Plus		BPL <i>rel</i>	$PC \leftarrow (PC) + \$0002 + rel ? (N) = 0$
Branch Always	BRA <i>rel</i>	BRA <i>rel</i>	$PC \leftarrow (PC) + \$0002 + rel$
Branch if Bit <i>n</i> in Memory Clear	BRCLR <i>n,opr8,rel</i> BRCLR <i>n,X,rel</i> ^{1,2} BRCLR <i>n,D[X],rel</i> ^{1,2}	BRCLR <i>n,opr8,rel</i>	$PC \leftarrow (PC) + \$0003 + rel ? (Mn) = 0$ $PC \leftarrow (PC) + \$0003 + rel ? (Xn) = 0^2$
Branch Never		BRN <i>rel</i>	$PC \leftarrow (PC) + \$0002$
Branch if Bit <i>n</i> in Memory Set	BRSET <i>n,opr8,rel</i> BRSET <i>n,X,rel</i> ^{1,2} BRSET <i>n,D[X],rel</i> ^{1,2}	BRSET <i>n,opr8,rel</i>	$PC \leftarrow (PC) + \$0003 + rel ? (Mn) = 1$ $PC \leftarrow (PC) + \$0003 + rel ? (Xn) = 1^2$
Branch to Subroutine	BSR <i>rel</i>	BSR <i>rel</i>	For S08: $PC \leftarrow (PC) + \$0002$; push (PCL) $SP \leftarrow (SP) - \$0001$; push (PCH) $SP \leftarrow (SP) - \$0001$ $PC \leftarrow (PC) + rel$ For RS08: $PC \leftarrow (PC) + 2$ Push PC to shadow PC $PC \leftarrow (PC) + rel$
Compare and Branch if Equal	CBEQ <i>opr8,rel</i> CBEQA # <i>opr8,rel</i> CBEQ <i>X,rel</i> ^{1,2} CBEQ <i>,X,rel</i> ^{1,2}	CBEQ <i>opr8,rel</i> CBEQA # <i>opr8,rel</i> CBEQX # <i>opr8,rel</i> CBEQ <i>opr8,X+,rel</i> CBEQ <i>X+,rel</i> CBEQ <i>opr8,SP,rel</i>	For S08: $PC \leftarrow (PC) + \$0003 + rel ? (A) - (M) = \00 $PC \leftarrow (PC) + \$0003 + rel ? (A) - (M) = \00 $PC \leftarrow (PC) + \$0003 + rel ? (X) - (M) = \00 $PC \leftarrow (PC) + \$0003 + rel ? (A) - (M) = \00 $PC \leftarrow (PC) + \$0002 + rel ? (A) - (M) = \00 $PC \leftarrow (PC) + \$0004 + rel ? (A) - (M) = \00 For RS08: $PC \leftarrow (PC) + \$0003 + rel ? (A) - (M) = \00 $PC \leftarrow (PC) + \$0003 + rel ? (A) - (X) = \00^2
Decrement and Branch if Not Zero	DBNZ <i>opr8,rel</i> DBNZA <i>rel</i> DBNZX <i>rel</i> ¹ DBNZ <i>,X,rel</i> ^{1,2}	DBNZ <i>opr8,rel</i> DBNZA <i>rel</i> DBNZX <i>rel</i> DBNZ <i>opr8,X,rel</i> DBNZ <i>X,rel</i> DBNZ <i>opr8,SP,rel</i>	$A \leftarrow (A) - \$0001$ or $M \leftarrow (M) - \$01$ or $X \leftarrow (X) - \$0001$ For S08: $PC \leftarrow (PC) + \$0003 + rel$ if (result) $\neq 0$ for DBNZ direct, IX1 $PC \leftarrow (PC) + \$0002 + rel$ if (result) $\neq 0$ for DBNZA, DBNZX, or IX $PC \leftarrow (PC) + \$0004 + rel$ if (result) $\neq 0$ for DBNZ SP1 For RS08: $PC \leftarrow (PC) + \$0003 + rel$ if (result) $\neq 0$ for DBNZ direct, DBNZX, DBNZ,X $PC \leftarrow (PC) + \$0002 + rel$ if (result) $\neq 0$ for DBNZA
Jump	JMP <i>opr16</i>	JMP <i>opr8</i> JMP <i>opr16</i> JMP <i>opr8,X</i> JMP <i>opr16,X</i> JMP <i>,X</i>	$PC \leftarrow$ Jump Address

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

Description	RS08	S08	Operation
Jump to Subroutine	JSR <i>opr16</i>	JSR <i>opr8</i> JSR <i>opr16</i> JSR <i>opr16</i> ,X JSR <i>opr8</i> ,X JSR ,X	For S08: PC ← (PC) + <i>n</i> (<i>n</i> = 1, 2, or 3) Push (PCL); SP ← (SP) – \$0001 Push (PCH); SP ← (SP) – \$0001 PC ← Unconditional Address For RS08: PC ← (PC) + 3 Push PC to shadow PC PC ← Unconditional Address
Return from Subroutine	RTS	RTS	For S08: SP ← SP + \$0001; Pull (PCH) SP ← SP + \$0001; Pull (PCL) For RS08: Pull PC from shadow PC
Data Verification Operations:			
Bit Test		BIT # <i>opr8</i> BIT <i>opr8</i> BIT <i>opr16</i> BIT <i>opr8</i> ,X BIT <i>opr16</i> ,X BIT ,X BIT <i>opr8</i> ,SP BIT <i>opr16</i> ,SP	(A) & (M)
Compare Accumulator with Memory	CMP # <i>opr8</i> CMP <i>opr8</i> CMP ,X ¹ CMP X ^{1,2}	CMP # <i>opr8</i> CMP <i>opr8</i> CMP <i>opr16</i> CMP <i>opr8</i> ,X CMP <i>opr16</i> ,X CMP ,X CMP <i>opr8</i> ,SP CMP <i>opr16</i> ,SP	(A) – (M) (A) – (X) ²
Complement (One's Complement)		CPHX # <i>opr8</i> CPHX <i>opr8</i> CPHX <i>opr16</i> CPHX <i>opr8</i> ,SP	(H:X) – (M:M + \$0001)
Compare Index Register (H:X) with Memory		CPX # <i>opr8</i> CPX <i>opr8</i> CPX <i>opr16</i> CPX ,X CPX <i>opr8</i> ,X CPX <i>opr16</i> ,X CPX <i>opr8</i> ,SP CPX <i>opr16</i> ,SP	(X) – (M)
Test for Negative or Zero	TST <i>opr8</i> ¹ TSTA ¹ TSTX ¹	TST <i>opr8</i> TSTA TSTX TST <i>opr8</i> ,X TST ,X TST <i>opr8</i> ,SP	(A) – \$00 (X) – \$00 (M) – \$00
Data Movement Operations:			
Load Accumulator from Memory	LDA # <i>opr8</i> LDA <i>opr8</i> LDA <i>opr5</i> LDA ,X ¹	LDA # <i>opr8</i> LDA <i>opr8</i> LDA <i>opr16</i> LDA <i>opr8</i> ,X LDA <i>opr16</i> ,X LDA ,X LDA <i>opr8</i> ,SP LDA <i>opr16</i> ,SP	A ← (M)

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

Description	RS08	S08	Operation
Load Index Register (H:X) from Memory		LDHX #opr16 LDHX opr8 LDHX opr16 LDHX LDHX opr8,X LDHX opr16,X LDHX opr8,SP	$H:X \leftarrow (M:M + \$0001)$
Load X (Index Register Low) from Memory	LDX #opr8 ¹ LDX opr8 ¹	LDX #opr8 LDX opr8 LDX opr16 LDX opr8,X LDX opr16,X LDX ,X LDX opr8,SP LDX opr16,SP	$X \leftarrow (M)$
Move	MOV opr8,opr8 MOV #opr8,opr8 MOV D[X],opr8 ¹ MOV opr8,D[X] ¹ MOV #opr8,D[X] ¹	MOV opr8,opr8 MOV opr8,X+ MOV #opr8,opr8 MOV X+,opr8	For S08/RS08: $(M)_{\text{destination}} \leftarrow (M)_{\text{source}}$ For S08 only: $H:X \leftarrow (H:X) + \$001$ in IX+D and DIX+ Modes
Store Accumulator in Memory	STA opr8 STA opr5 STA ,X ¹	STA opr8 STA opr16 STA opr8,X STA opr16,X STA ,X STA opr8,SP STA opr16,SP	$M \leftarrow (A)$
Store H:X (Index Reg.)		STHX opr STHX opr STHX opr,SP	$(M:M + \$0001) \leftarrow (H:X)$
Store X (Index Register Low) in Memory	STX opr8 ¹	STX opr8 STX opr16 STX opr8,X STX opr16,X STX ,X STX opr8,SP STX opr16,SP	$M \leftarrow (X)$
Transfer Accumulator to CCR		TAP	$CCR \leftarrow (A)$
Transfer Accumulator to X (Index Register Low)	TAX ¹	TAX	$X \leftarrow (A)$
Transfer CCR to Accumulator		TPA	$A \leftarrow (CCR)$
Transfer SP to Index Reg.		TSX	$H:X \leftarrow (SP) + \$0001$
Transfer X (Index Reg. Low) to Accumulator	TXA ¹	TXA	$A \leftarrow (X)$
Transfer Index Reg. to SP		TXS	$(SP) \leftarrow (H:X) - \0001
Other Operations:			
Background	BGND	BGND	Enter Background Debug Mode
Clear Carry Bit	CLC	CLC	$C \leftarrow 0$
Clear Interrupt Mask Bit		CLI	$I \leftarrow 0$
No Operation	NOP	NOP	None
Push Accumulator onto Stack		PSHA	Push (A); $SP \leftarrow (SP) - \$0001$
Push H (Index Register High) onto Stack		PSHH	Push (H); $SP \leftarrow (SP) - \$0001$

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

Description	RS08	S08	Operation
Push X (Index Register Low) onto Stack		PSHX	Push (X); $SP \leftarrow (SP) - \$0001$
Pull Accumulator from Stack		PULA	$SP \leftarrow (SP + \$0001)$; Pull (A)
Pull H (Index Register High) from Stack		PULH	$SP \leftarrow (SP + \$0001)$; Pull (H)
Pull X (Index Register Low) from Stack		PULX	$SP \leftarrow (SP + \$0001)$; Pull (X)
Reset Stack Pointer		RSP	$SP \leftarrow \$FF$
Return from Interrupt		RTI	$SP \leftarrow (SP) + \$0001$; Pull (CCR) $SP \leftarrow (SP) + \$0001$; Pull (A) $SP \leftarrow (SP) + \$0001$; Pull (X) $SP \leftarrow (SP) + \$0001$; Pull (PCH) $SP \leftarrow (SP) + \$0001$; Pull (PCL)
Swap Shadow PC High with A	SHA		$A \leftrightarrow SPCH$
Swap Shadow PC Low with A	SLA		$A \leftrightarrow SPCL$
Set Carry Bit	SEC	SEC	$C \leftarrow 1$
Set Interrupt Mask Bit		SEI	$I \leftarrow 1$
Enable \overline{IRQ} pin; Stop Osc.	STOP	STOP	Stop Oscillator I bit $\leftarrow 0$ for S08 only;
Software Interrupt		SWI	$PC \leftarrow (PC) + \$0001$; Push (PCL) $SP \leftarrow (SP) - \$0001$; Push (PCH) $SP \leftarrow (SP) - \$0001$; Push (X) $SP \leftarrow (SP) - \$0001$; Push (A) $SP \leftarrow (SP) - \$0001$; Push (CCR) $SP \leftarrow (SP) - \$0001$; I $\leftarrow 1$ PCH \leftarrow Interrupt Vector High Byte PCL \leftarrow Interrupt Vector Low Byte
Enable Interrupts; Stop Processor	WAIT	WAIT	I bit $\leftarrow 0$ for S08 only;

NOTES:

- ¹ This is pseudo-instruction, the CPU cycle count and the instruction byte count may not be the same as the S08 equivalent instruction.
- ² This emulated operation do not have an equivalent operation in S08 instruction set.

1.2.1 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.

Table 1-3. RS08 Tiny and Short Addressing Mode Instructions

Description	Tiny/Short Instruction	Addressable Space	Coding Example
Load Accumulator from Memory	LDA <i>opr5</i>	\$0000 to \$001F	LDA <\$1F LDA <\$00
Store Accumulator in Memory	STA <i>opr5</i>	\$0000 to \$001F	STA <\$1F STA <\$00
Clear	CLR <i>opr5</i>	\$0000 to \$001F	CLR <\$1F CLR <\$00
Add without Carry	ADD <i>opr4</i>	\$0000 to \$000F	ADD <\$0F ADD <\$00
Subtract	SUB <i>opr4</i>	\$0000 to \$000F	SUB <\$0F SUB <\$00
Increment	INC <i>opr4</i>	\$0000 to \$000F	INC <\$0F INC <\$00
Decrement	DEC <i>opr4</i>	\$0000 to \$000F	DEC <\$0F DEC <\$00

1.2.2 Pseudo Instructions

Using register X located in \$000F and register D[X] 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.

Table 1-4. Pseudo Instructions in RS08 Platform

Operation	Pseudo Instruction	Emulation	Description	Bytes	Cycles
Add with Carry	ADC ,X ADC X	ADC \$0E ADC \$0F	$A \leftarrow (A) + (M) + (C)$ $A \leftarrow (A) + (X) + (C)$	2 2	3 3
Add without Carry	ADD ,X ADD X	ADD <\$0E ADD <\$0F	$A \leftarrow (A) + (M)$ $A \leftarrow (A) + (X)$	1 1	3 3
Logical AND	AND ,X AND X	AND \$0E AND \$0F	$A \leftarrow (A) \& (M)$ $A \leftarrow (A) \& (X)$	2 2	3 3
Clear Bit n in Memory	BCLR <i>n</i> ,D[X] BCLR <i>n</i> ,X	BCLR <i>n</i> , \$0E BCLR <i>n</i> , \$0F	$M_n \leftarrow 0$ $X_n \leftarrow 0$	2 2	5 5
Branch if Bit n in Memory Clear	BRCLR <i>n</i> ,D[X], <i>rel</i> BRCLR <i>n</i> ,X, <i>rel</i>	BRCLR <i>n</i> , \$0E, <i>rel</i> BRCLR <i>n</i> , \$0F, <i>rel</i>	$PC \leftarrow (PC) + \$0003 + rel ? (M_n) = 0$ $PC \leftarrow (PC) + \$0003 + rel ? (X_n) = 0$	3 3	5 5
Branch if Bit n in Memory Set	BRSET <i>n</i> ,D[X], <i>rel</i> BRSET <i>n</i> ,X, <i>rel</i>	BRSET <i>n</i> , \$0E, <i>rel</i> BRSET <i>n</i> , \$0F, <i>rel</i>	$PC \leftarrow (PC) + \$0003 + rel ? (M_n) = 1$ $PC \leftarrow (PC) + \$0003 + rel ? (X_n) = 1$	3 3	5 5
Set Bit n in Memory	BSET <i>n</i> ,D[X] BSET <i>n</i> ,X	BSET <i>n</i> , \$0E BSET <i>n</i> , \$0F	$M_n \leftarrow 1$ $X_n \leftarrow 1$	2 2	5 5
Compare and Branch if Equal	CBEQ ,X, <i>rel</i> CBEQ X <i>rel</i>	CBEQ \$0E, <i>rel</i> CBEQ \$0F, <i>rel</i>	$PC \leftarrow (PC) + \$0003 + rel ? (A) - (M) = \00 $PC \leftarrow (PC) + \$0003 + rel ? (A) - (X) = \00	3 3	5 5

Table 1-4. Pseudo Instructions in RS08 Platform (continued)

Operation	Pseudo Instruction	Emulation	Description	Bytes	Cycles
Clear	CLR ,X	CLR <\$0E	$M \leftarrow \$00$	1	2
	CLR X	CLR <\$0F	$X \leftarrow \$00$	1	2
Compare Accumulator with Memory	CMP ,X	CMP \$0E	$(A) - (M)$	2	3
	CMP X	CMP \$0F	$(A) - (X)$	2	3
Decrement and Branch if Not Zero	DBNZ ,X,rel	DBNZ \$0E, rel	$M \leftarrow (M) - \$01$	3	6
	DBNZ X rel	DBNZ \$0F, rel	$X \leftarrow (X) - \$01$ $PC \leftarrow (PC) + \$0003 + rel$ if (result) $\neq 0$	3	6
Decrement	DEC ,X	DEC <\$0E	$M \leftarrow (M) - \$01$	1	4
	DEC X	DEC <\$0F	$X \leftarrow (X) - \$01$	1	4
Exclusive OR Memory with Accumulator	EOR ,X	EOR \$0E	$A \leftarrow (A \oplus M)$	2	3
	EOR X	EOR \$0F	$A \leftarrow (A \oplus X)$	2	3
Increment	INC ,X	INC <\$0E	$M \leftarrow (M) + \$01$	1	4
	INC X	INC <\$0F	$X \leftarrow (X) + \$01$	1	4
Load Accumulator from Memory	LDA ,X	LDA <\$0E	$A \leftarrow (M)$	1	3
Load X (Index Register Low) from Memory	LDX #opr8	MOV #opr8, \$0F	$X \leftarrow (M)$	3	4
	LDX opr8	MOV opr8, \$0F		3	5
Inclusive OR Accumulator and Memory	ORA ,X	ORA \$0E	$A \leftarrow (A) (M)$	2	3
	ORA X	ORA \$0F	$A \leftarrow (A) (X)$	2	3
Subtract with Carry	SBC ,X	SBC \$0E	$A \leftarrow (A) - (M) - (C)$	2	3
	SBC X	SBC \$0F	$A \leftarrow (A) - (X) - (C)$	2	3
Store Accumulator in Memory	STA ,X	STA <\$0E	$M \leftarrow (A)$	1	2
Store X (Index Register Low) in Memory	STX opr8	MOV \$0F, opr8	$M \leftarrow (X)$	3	5
Subtract	SUB ,X	SUB <\$0E	$A \leftarrow (A) - (M)$	1	3
	SUB X	SUB <\$0F	$A \leftarrow (A) - (X)$	1	3
Transfer Accumulator to X (Index Register Low)	TAX	STA <\$0F	$X \leftarrow (A)$	1	2
Test for Negative or Zero	TST opr8	MOV opr8, opr8	$(M) - \$00$	3	5
	TSTA	ORA #00	$(A) - \$00$	2	2
	TSTX	MOV X, X	$(X) - \$00$	3	5
Transfer X (Index Reg. Low) to Accumulator	TXA	LDA <\$0F	$A \leftarrow (X)$	1	3

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.

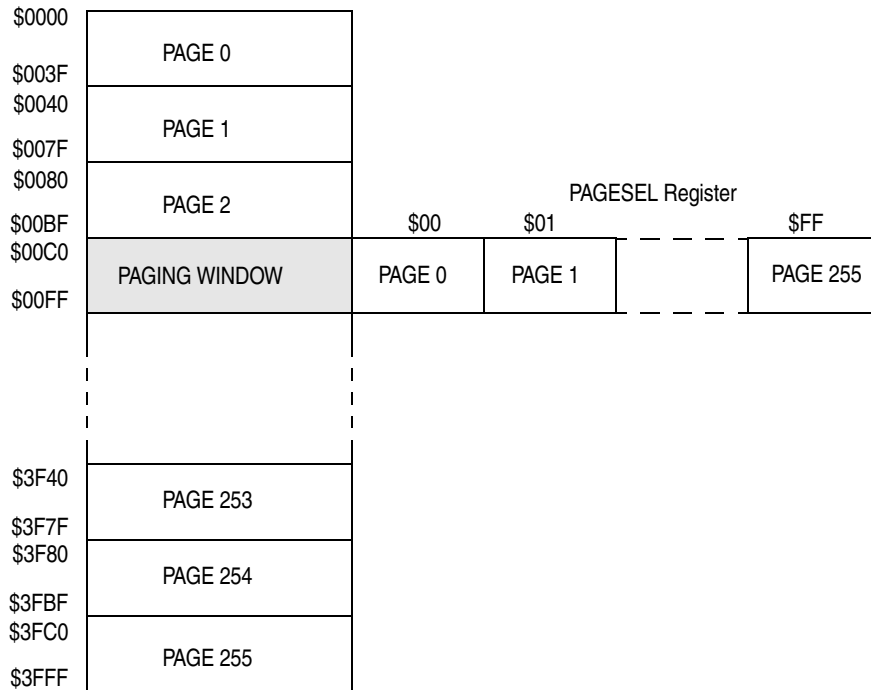


Figure 1-4. RS08 Paging Scheme

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 be programmed into the locations \$3FFD-\$3FFF. The operand indicates the user defined location to start user program execution.

```

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


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 exits 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 exits 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.

```

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

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

PSH_SPC: MACRO
    ; NOTE: Destructive to X content
    ; 20 CPU cycles, 14 bytes code
    ldx        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
    ; NOTE: Destructive to X content
    ; 22 CPU cycles, 14 byte code
    ldx        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

```

```

;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
; Subroutine A
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
SubA:
    PSH_SPC                ; Stack SPC
    ;... <Subroutine Content> ...
    bsr SubB               ; Multi-level subroutine call
    ;... <Subroutine Content> ...
    PUL_SPC                ; Unstack SPC
    rts
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
; Subroutine B
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
SubB:
    PSH_SPC                ; Stack SPC
    ;... <Subroutine Content> ...
    PUL_SPC                ; Unstack SPC
    rts
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
; Main
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
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 1-5. Interrupt Servicing Priority Example

Highest	—————▶				Lowest
MTIM	KBI	ACMP	RTI	LVD	

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> ...

```

```

ACMP_ISR:      bra      InfLoop
                ;... <ISR coding> ...
                bra      InfLoop
KBI_ISR:      ;... <ISR coding> ...
                bra      InfLoop
RTI_ISR:      ;... <ISR coding> ...
                bra      InfLoop
LVD_ISR:      ;... <ISR coding> ...
                bra      InfLoop

```

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

Interrupt	Latency (μ s)
MTIM	0.8
ACMP	1.3 ¹
KBI	1.8 ¹
RTI	2.3 ¹
LVD	2.8 ¹

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.

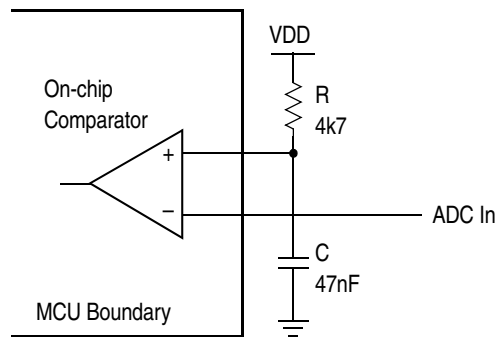


Figure 2-5. Emulated ADC Schematic

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 $3.9\mu\text{s}$ (the function is given in Equation 1) and is rounded up to $4\mu\text{s}$. Maximum timer overflow is assumed, then overflow period becomes 255 times $4\mu\text{s}$, i.e. 1.02ms .

$$\text{TimerResolution} = \frac{\text{ChargeUpTime}}{2^n - 1} \quad \text{Eqn. -1}$$

2. **Define RC time constant.** The RC charging profile follows [Equation 2](#).

$$V = V_{DD} \left(1 - e^{-\frac{t}{RC}} \right) \quad \text{Eqn. -2}$$

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.

$$RC = \frac{\text{TimerOverflowPeriod}}{4.61} \quad \text{Eqn. -3}$$

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 Ω 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 μ 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:

$$\text{Step} = \frac{V_{DD}}{255} \quad \text{Eqn. -4}$$

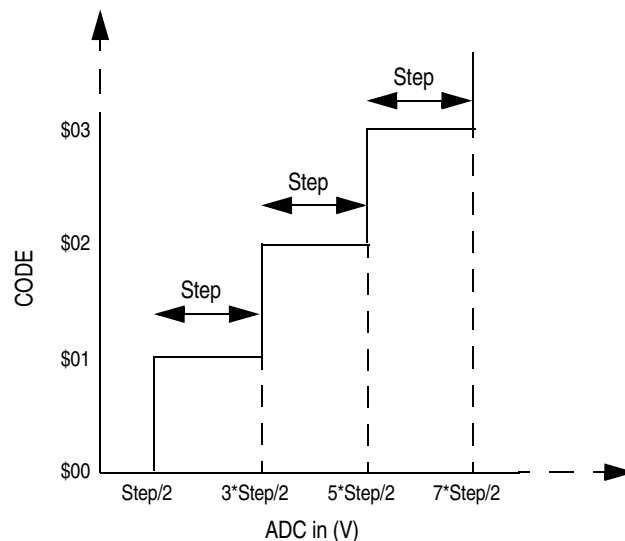


Figure 2-6. ADC Quantization Diagram

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:

$$Code = \begin{cases} \left(\frac{ADCin - \frac{Step}{2}}{Step} \right) + 1 & ; (ADCin \geq \frac{Step}{2}) \\ 0 & ; (ADCin < \frac{Step}{2}) \end{cases} \quad \text{Eqn. -5}$$

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

Table 2-7. Non-Linearity Compensation Lookup Table

Time (μs)	ADC Input (V) (Equation 2)	Timer Count	Linear ADC Code (Equation 5)
0	0	0	0
4	0.09	1	5
8	0.18	2	10
12	0.26	3	14
16	0.35	4	18
20	0.43	5	23
and so on...			
1012	4.95	253	253
1016	4.95	254	253
1020	4.95	255	253

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.


```

;=====
; Application Definition
;=====
RC          equ      PTAD_PTAD0
mRC        equ      mPTAD_PTAD0
TableStart equ      $3E00
           org      Tiny_RAMStart
; variable/data section
SensorReading ds.b 1
ADCOut      ds.b 1

```

- 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:

```

;-----
; 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

```

- c) 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|ACMP_OUTPUT_RAISING), ACMP1SC
                                                         ; Enable ACMP, start RC rise
           bset     ACMP1SC_ACF, ACMP1SC                      ;Clear ACMP Flag
           wait
           mov      MTIM1CNT, SensorReading
           brclr   ACMP1SC_ACF, ACMP1SC, NoReading

```

Emulated ADC Application Example

```
    bset    ACMP1SC_ACF, ACMP1SC          ;Clear ACMP Flag
    clr     ACMP1SC                       ;disable ACMP
    mov     #(mMTIM1SC_TSTP|mMTIM1SC_TRST), MTIM1SC ;mask int and clear flag
    rts
NoReading:
    mov     #$FF, SensorReading           ;Biggest Number
    clr     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    ;                             ;Extract 2 MSB
    rola    ;
    rola    ;
    and     #$03                          ;Mask all other bits
    add     #(TableStart>>6)              ;Add to Lookup table page
    sta     PAGESEL                        ;High page
    lda     SensorReading                  ;
    and     #$3F                          ;Extract 6 LSB
    add     #$c0                          ;Index to paging window
    tax
    lda     ,x                             ;Read upper memory
    sta     ADCOut                        ;Store lookup table content
    mov     #(HREG), PAGESEL              ;Return to register page
    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   222,223,223,224,224,225,225,226,226,227,228,228,228,229,229,230
    dc.b   230,231,231,232,232,233,233,233,234,234,235,235,235,236,236,236
    dc.b   237,237,237,238,238,238,239,239,239,240,240,240,240,241,241,241
    dc.b   241,242,242,242,242,243,243,243,243,244,244,244,244,244,245,245
    dc.b   245,245,245,246,246,246,246,246,246,247,247,247,247,247,247,247
    dc.b   248,248,248,248,248,248,248,249,249,249,249,249,249,249,249,249
    dc.b   250,250,250,250,250,250,250,250,250,250,251,251,251,251,251,251
    dc.b   251,251,251,251,251,251,252,252,252,252,252,252,252,252,252,252
    dc.b   252,252,252,252,252,252,253,253,253,253,253,253,253,253,253,253
```

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.

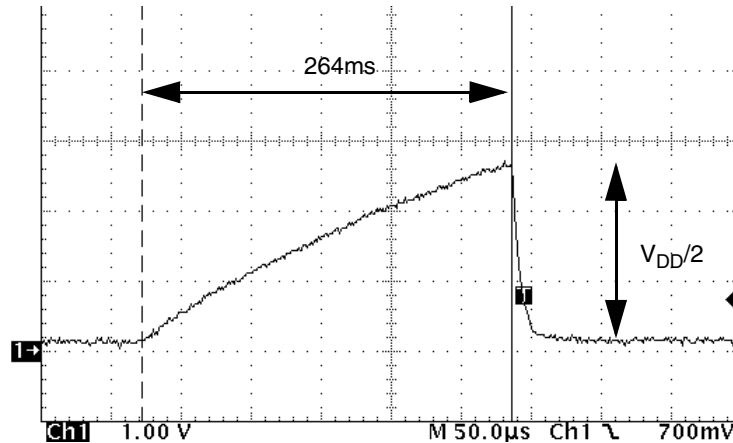


Figure 2-7. 2.5V Input R-C Charging Profile

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, $264\mu\text{s}$ in this case. The expected rise time based on the previous calculation listed in Table 2-7 is $152\mu\text{s}$, 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 ($152\mu\text{s}$ 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 $4\mu\text{s}$ based on the previous calculation with 2MHz bus frequency, 38 timer counts are expected to reach $V_{DD}/2$ voltage level. So, with $264\mu\text{s}$ measured rise time, new timer resolution should be $264\mu\text{s}$ divided by 38, i.e. $6.94\mu\text{s}$. With a divide-by-8 prescaler option selected for the timer clock source, compensated bus period should be $6.94\mu\text{s}$ 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.

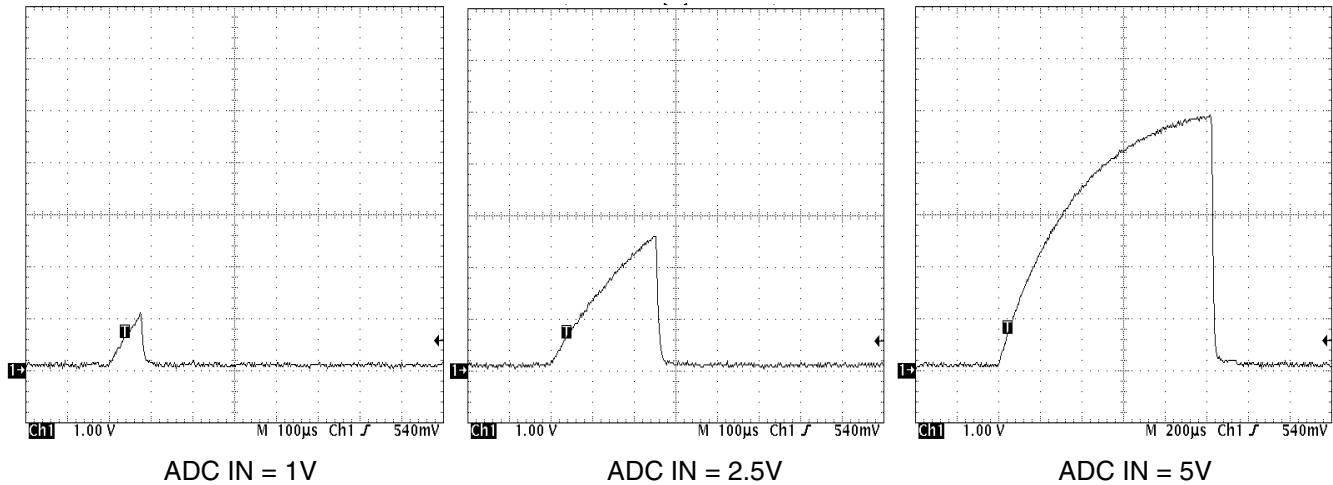


Figure 2-8. RC Charging Profile Against Different ADC Input Voltages

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

Table 2-8. Emulated ADC Performance

ADC Input Voltage (V)	Expected ADC code (Decimal)	Measured ADC code (Decimal)
1	51	50
1.5	76	75
2	102	99
2.5	127	123
3	153	150
3.5	178	175
4	204	202
4.5	229	234

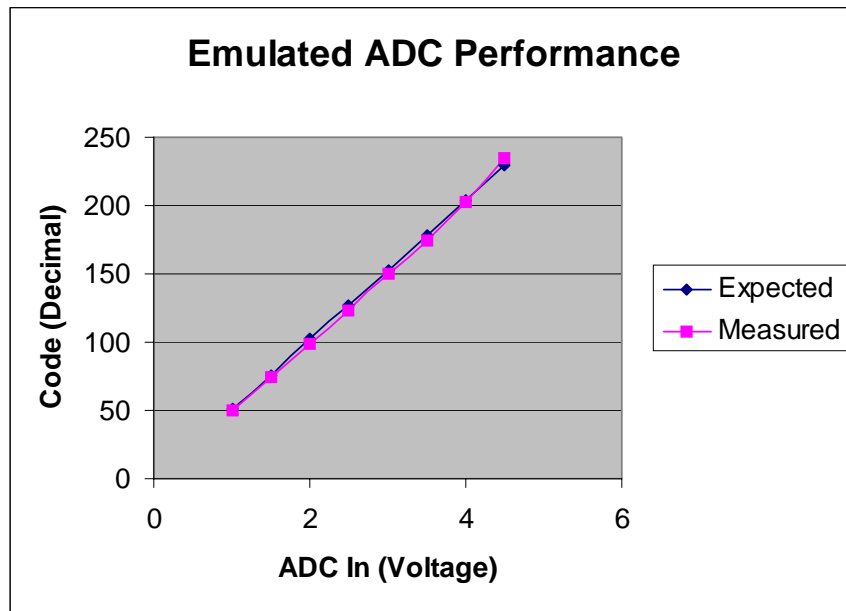


Figure 2-9. Emulated ADC Performance

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_RAISING equ      $01
ACMP_OUTPUT_BOTH   equ      $03
;=====
; RTI Definition
;=====
RTI_DISABLE        equ      $00
RTI_8MS            equ      $01

```

```

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

;=====
; Application Definition
;=====
RC                equ      PTAD_PTAD0
mRC               equ      mPTAD_PTAD0
DATAOUT           equ      PTAD_PTAD5
mDATAOUT          equ      mPTAD_PTAD5

TableStart        equ      $3E00

        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 ; SOPT, 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

```

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  
    mov     MTIM1CNT, SensorReading  
    brclr   ACMP1SC_ACF, ACMP1SC, NoReading  
    bset    ACMP1SC_ACF, ACMP1SC                    ;Clear ACMP Flag  
    clr     ACMP1SC                                  ;disable ACMP  
    mov     #(mMTIM1SC_TSTP|mMTIM1SC_TRST), MTIM1SC ;mask int and clear flag  
    rts  
NoReading:  
    mov     #$FF, SensorReading                      ;Biggest Number  
    clr     ACMP1SC                                  ;disable ACMP  
    mov     #(mMTIM1SC_TSTP|mMTIM1SC_TRST), MTIM1SC ;mask int and clear flag  
    rts  
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
; 8bit Table Lookup  
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
TableLookup:  
    lda     SensorReading                            ;  
    rola                                       ;Extract 2 MSB  
    rola                                       ;  
    rola                                       ;  
    and     #$03                                  ;Mask all other bits  
    add     #(TableStart>>6)                    ;Add to Lookup table page  
    sta     PAGESEL                               ;High page  
    lda     SensorReading                            ;  
    and     #$3F                                  ;Extract 6 LSB  
    add     #$c0                                  ;Index to paging window  
    tax  
    lda     ,x                                    ;Read upper memory
```



```

        sta      ADCOut          ;Store lookup table content
        mov      #(HREG), PAGESEL ;Return to register page
        rts
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
; Serial Data dump
; <LOW><b7><b6><b5><b4><b3><b2><b1><b0><HIGH>
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
DataDump:
        mov      #8,          BitCount
        lda      ADCOut

        bclr     DATAOUT, PTAD ;5 Start bit
        bclr     DATAOUT, PTAD ;5 dummy
        cmp      0             ;3 dummy
        nop
NextBit:
        lsla
        bcc      ClrPort       ;3
        bset     DATAOUT, PTAD ;5
        bra      BitEnd        ;3
ClrPort:
        bclr     DATAOUT, PTAD ;5
        bra      BitEnd        ;3
BitEnd:
        dbnz    BitCount, NextBit ;6
ByteEnd:
        bset     DATAOUT, PTAD ;5 End bit
        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    222,223,223,224,224,225,225,226,226,227,228,228,228,229,229,230
        dc.b    230,231,231,232,232,233,233,233,234,234,235,235,235,236,236,236
        dc.b    237,237,237,238,238,238,239,239,239,240,240,240,240,241,241,241
        dc.b    241,242,242,242,242,243,243,243,243,244,244,244,244,244,245,245
        dc.b    245,245,245,246,246,246,246,246,246,247,247,247,247,247,247,247
        dc.b    248,248,248,248,248,248,248,249,249,249,249,249,249,249,249,249
        dc.b    250,250,250,250,250,250,250,250,250,250,251,251,251,251,251,251
        dc.b    251,251,251,251,251,251,252,252,252,252,252,252,252,252,252,252
        dc.b    252,252,252,252,252,252,253,253,253,253,253,253,253,253,253,253
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
; Reset Vector
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        org      $3ffc
Security:
        dc.b    $FF
        jmp     main

```

How to Reach Us:

Home Page:

www.freescale.com

E-mail:

support@freescale.com

USA/Europe or Locations Not Listed:

Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

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)
support@freescale.com

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.

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>.

Document Number: AN3266
Rev. 1
5/2006

