MOTOROLA
intelligence everywhere™

digital dna™

HCS08

Family Reference Manual

Volume 1

HCS08 Microcontrollers

HCS08RMv1/D
Rev. 1, 6/2003

WWW.MOTOROLA.COM/SEMICONDUCTORS

HCS08RMv1/D

HCS08

Family Reference Manual

Volume I

MOTOROLA
intelligence everywhere™

digital dna™

Important Notice to Users

While every effort has been made to ensure the accuracy of all information in this document, Motorola assumes no liability to any party for any loss or damage caused by errors or omissions or by statements of any kind in this document, its updates, supplements, or special editions, whether such errors are omissions or statements resulting from negligence, accident, or any other cause. Motorola further assumes no liability arising out of the application or use of any information, product, or system described herein: nor any liability for incidental or consequential damages arising from the use of this document. Motorola disclaims all warranties regarding the information contained herein, whether expressed, implied, or statutory, *including implied warranties of merchantability or fitness for a particular purpose*. Motorola makes no representation that the interconnection of products in the manner described herein will not infringe on existing or future patent rights, nor do the descriptions contained herein imply the granting or license to make, use or sell equipment constructed in accordance with this description.

Trademarks

This document includes these trademarks:

Motorola and the Motorola logo are registered trademarks
of Motorola, Inc.

Motorola, Inc., is an Equal Opportunity / Affirmative Action Employer.

This product incorporates SuperFlash® technology licensed from SST.

List of Sections

Table of Contents

Section 1. General Information and Block Diagram

Section 2. Pins and Connections

Section 3. Modes of Operation

Section 4. On-Chip Memory

Section 5. Resets and Interrupts

Section 6. Central Processor Unit (CPU)

Section 7. Development Support

Appendix A. Instruction Set Details

Appendix B. Equate File Conventions

Section 1. General Information and Block Diagram

1.1 Introduction to the HCS08 Family of Microcontrollers

Motorola’s new HCS08 Family of microcontrollers, while containing new instructions to implement rapid debugging and development, is still fully compatible with all legacy code written for the M68HC08 Family. This reference manual uses the MC9S08GB60, the first HCS08 Family member, for describing applications and module behavior. When working with another HCS08 Family MCU, refer to the device data sheet for information specific to that MCU.

Each MCU device in the HCS08 Family consists of the HCS08 core plus several memory and peripheral modules. The HCS08 core consists of:

- HCS08 CPU
- Background debug controller (BDC)
- Support for up to 32 interrupt/reset sources
- Chip-level address decode

The HCS08 CPU executes all HC08 instructions, as well as a background (BGND) instruction and additional addressing modes on the LDHX, STHX, and CPHX instructions to improve compiler efficiency. The maximum clock speed for the CPU is 40 MHz (typically generated from a crystal or internal clock generator). The CPU performs operations at this 40 MHz rate and the maximum bus rate is 20 MHz (half the CPU clock frequency). See **Section 6. Central Processor Unit (CPU)** for more information.

The background debug controller (BDC) is built into the CPU core to allow easier access to address generation circuits and CPU register information. The BDC includes one hardware breakpoint. Other more sophisticated breakpoints are normally included in the separate on-chip debug module. The BDC allows access to internal register and memory locations via a single pin on the MCU. See **Section 7. Development Support** for more information.

The core includes support for up to 32 interrupt or reset sources with separate vectors. The peripheral modules provide local interrupt enable circuitry and flag registers. See **Section 5. Resets and Interrupts** for more information.

Although the exact memory map for each derivative is different, some basic aspects are controlled by decode logic in the HCS08 core which is not expected to change from one HCS08 derivative to another. The registers for input/output (I/O) ports and most control and status registers for peripheral modules are located starting at $0000 and extending for 32, 64, 96, or 128 bytes. The space from the end of these direct page registers to $107F is reserved for static RAM memory. A space starting at $1800 is reserved for high-page registers. These are status and control registers that do not need to be accessed as often as the direct page registers. For example, system setup registers that are written only once after reset may be located in this high-register space to make more room in the direct addressing space for registers and RAM. The remaining space from $1C00 through $FFFF is reserved for FLASH or ROM memory. The last 64 locations ($FFC0–$FFFF) are further classified as vector space (for up to 32 interrupt and reset vectors).

1.2 Programmer's Model for the HCS08 CPU

The programmer's model for the HCS08 CPU shown in **Figure 1-1** includes the same registers as the M68HC08. These include one 8-bit accumulator (A), a 16-bit index register made up of separately accessible upper (H) and lower (X) 8-bit halves, a 16-bit stack pointer (SP), a 16-bit program counter (PC) and an 8-bit condition code register (CCR) which includes five processor status flags (V, H, N, Z, and C) and the global interrupt mask (I).

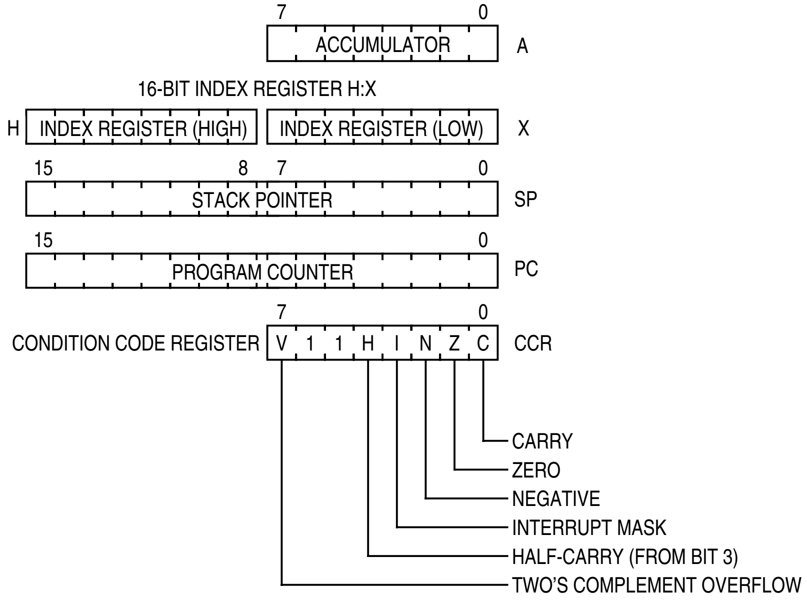


Figure 1-1. CPU Registers

1.3 Peripheral Modules

The combination of peripheral modules included on a specific derivative can vary widely, however there will always be memory for programs and data, and there will always be a clock module and debug module. Some of the peripheral modules in the HCS08 Family include:

- 4K–60K byte FLASH or ROM memory
- 128–4K byte Static RAM
- Asynchronous serial I/O (SCI)
- Synchronous serial I/O (SPI and IIC)
- Timer/PWM modules (TPM)
- Keyboard interrupts (KBI)
- Analog to digital converter (ADC)
- Clock generation modules
  - Full-featured internal clock generator (ICG) capable of operation with no external components (frequency multiplication is accomplished with a frequency-locked loop (FLL) that does not use any external filter components)
  - Traditional Pierce oscillator with no FLL or PLL (OSC)

- Debug module with nine trigger modes and bus capture FIFO (DBG)

Always refer to the appropriate data sheet for more specific information about the features in each HCS08 derivative MCU.

1.4 Features of the MC9S08GB60

The first device in the HCS08 Family is the MC9S08GB60 which is presented here as a representative example of a derivative HCS08 MCU.

1.4.1 Standard Features of the HCS08 Family

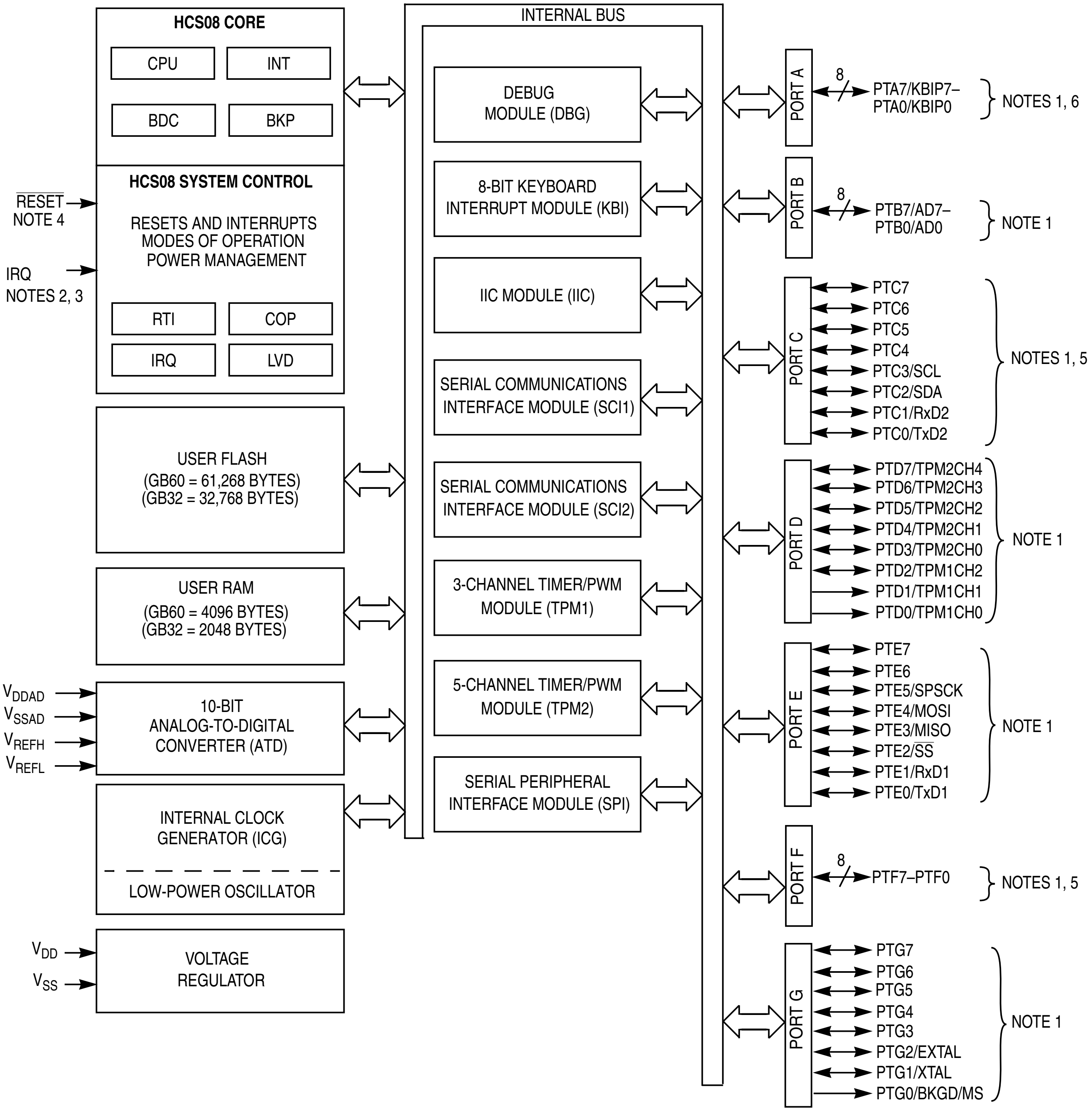
- 40-MHz HCS08 CPU (central processor unit)
- HC08 instruction set with added BGND instruction
- Background debugging system
- Breakpoint capability to allow single breakpoint setting during in-circuit debugging (plus two more breakpoints in on-chip debug module)
- Debug module containing two comparators and nine trigger modes. Eight deep FIFO for storing change-of-flow addresses and event-only data. Debug module supports both tag and force breakpoints.
- Support for up to 32 interrupt/reset sources
- Power-saving modes: wait plus three stops
- System protection features:
  - Optional computer operating properly (COP) reset
  - Low-voltage detection with reset or interrupt
  - Illegal opcode detection with reset
  - Illegal address detection with reset (some devices don’t have illegal addresses)

1.4.2 Features of MC9S08GB60 MCU

- 60K on-chip in-circuit programmable FLASH memory with block protection and security options
- 4K on-chip random-access memory (RAM)
- 8-channel, 10-bit analog-to-digital converter (ATD)
- Two serial communications interface modules (SCI)
- Serial peripheral interface module (SPI)
- Clock source options include crystal, resonator, external clock or internally generated clock with precision NVM trimming
- Inter-integrated circuit bus module to operate up to 100 kbps (IIC)
- One 3-channel and one 5-channel 16-bit timer/pulse width modulator (TPM) modules with selectable input capture, output compare, and edge-aligned PWM capability on each channel. Each timer module may be configured for buffered, centered PWM (CPWM) on all channels (TPMx).
- 8-pin keyboard interrupt module (KBI)
- 16 high-current pins (limited by package dissipation)
- Software selectable pullups on ports when used as input. Selection is on an individual port bit basis. During output mode, pullups are disengaged.
- Internal pullup on $\overline{\text{RESET}}$ and IRQ pin to reduce customer system cost
- 56 general-purpose input/output (I/O) pins, depending on package selection
- 64-pin low-profile quad flat package (LQFP)

1.5 Block Diagram of the MC9S08GB60

Figure 1-2 is an overall block diagram of the MC9S08GB60 MCU showing all major peripheral systems and all device pins. The MC9S08GB60 is a representative device in the HCS08 Family.



NOTES:

1. Port pins are software configurable with pullup device if input port.
2. Pin contains software configurable pullup/pulldown device if IRQ enabled (IRQPE = 1).
3. IRQ does not have a clamp diode to V_{DD}. IRQ should not be driven above V_{DD}.
4. Pin contains integrated pullup device.
5. High current drive
6. Pins PTA[7:4] contain software configurable pullup/pulldown device.

Figure 1-2. MC9S08GB60 Block Diagram

Section 2. Pins and Connections

2.1 Introduction

This section shows basic connections that are common to typical application systems. Additional details are provided for power, oscillator, reset, mode, and background interface connections. The example system uses the MC9S08GB60, which is a representative device in the HCS08 Family.

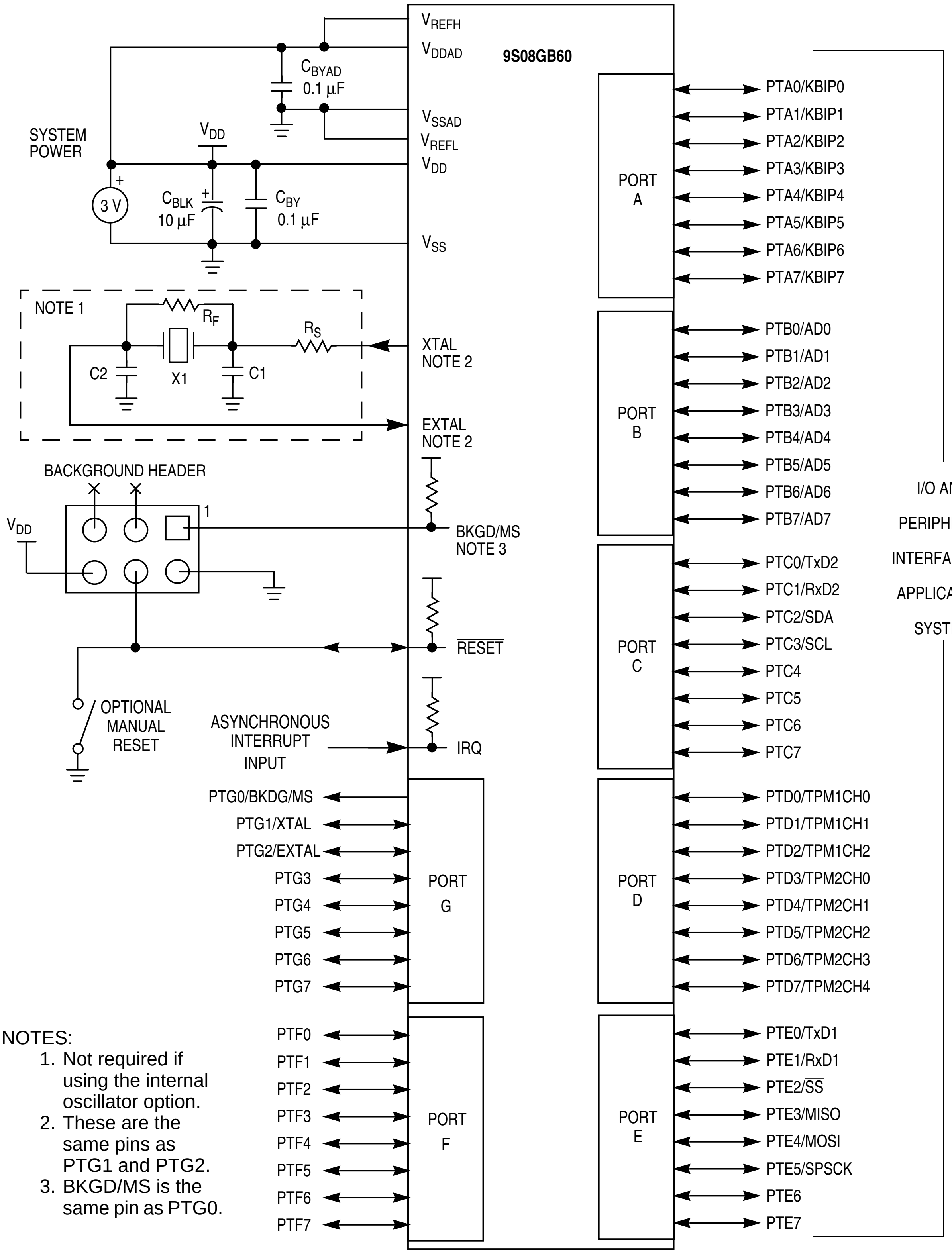
On-chip peripheral systems share pins so that when a peripheral system is not using a pin or pins, those pins may be used as general-purpose input/output (I/O) pins. When planning system connections, the designer should consider the reset condition of these pins, as well as the characteristics of the pins after software has configured them for their application purpose.

For example, a serial TxD pin would have the characteristics of an actively driven CMOS output after the SCI transmitter is enabled. However, between reset and when application software enables the SCI transmitter, the pin will have the characteristics of a high-impedance input. Although floating CMOS inputs are generally considered undesirable, the delay from reset until the pins are reconfigured for other functions is so short that this is almost never a serious concern in most applications. If this is determined to be a problem, the user may need to connect an external pullup resistor to such pins.

2.2 Recommended System Connections

Figure 2-1 shows pin connections that are common to most typical HCS08 application systems. This particular example shows the MC9S08GB60 because it is a representative device in the HCS08 Family. Always refer to the data sheet for a specific derivative to find detailed information about unusual pins.

A more detailed discussion of system connections follows.



NOTES:
1. Not required if using the internal oscillator option.
2. These are the same pins as PTG1 and PTG2.
3. BKGD/MS is the same pin as PTG0.

Figure 2-1. Basic System Connections

The following paragraphs discuss system connections in more detail.

2.2.1 Power

V_{DD} and V_{SS} are the primary power supply pins for the HCS08 MCU. This voltage source supplies power to all I/O buffer circuitry and to an internal voltage regulator. This internal voltage regulator provides regulated 2.5-volt (nominal) power to the CPU and other internal circuitry of the MCU.

Typically, application systems have two separate capacitors across the power pins. In this case, there should be a bulk electrolytic capacitor, such as a 10-µF tantalum capacitor, to provide bulk charge storage for the overall system and a 0.1-µF ceramic bypass capacitor located as close to the MCU power pins as practical to suppress high-frequency noise.

Due to the sub-micron process used, internal logic in the HCS08 MCU uses a lower power supply voltage than earlier MCUs. In addition to allowing the smaller layout geometry, this also has the benefit of lowering overall system power requirements. This implies that an on-chip voltage regulator is used to step down the voltage from the external MCU supply voltage to the internal logic voltage.

V_{DDAD} and V_{SSAD} are the analog power supply pins for the MCU. This voltage source supplies power to the ATD. A 0.1-µF ceramic bypass capacitor should be located as close to the MCU power pins as practical to suppress high-frequency noise.

2.2.2 MC9S08GB60 Oscillator

This section describes the oscillator in the MC9S08GB60. Not all HCS08 derivatives use the same type of oscillator; some have no external oscillator components. Always refer to the data sheet for a particular HCS08 derivative for more details.

The MC9S08GB60 can be operated with no external crystal or oscillator. When this occurs, the MCU uses an internally generated self-clocked rate equivalent to about 8-MHz crystal rate. This frequency source is used during reset startup to avoid the need for a long crystal startup delay.

The oscillator in the MC9S08GB60 is a traditional Pierce oscillator that can accommodate a crystal or ceramic resonator in either of two frequency ranges selected by the RANGE bit in the ICGC1 register. The low range is 32 kHz to 100 kHz and the high range is 1 MHz to 16 MHz.

Rather than a crystal or ceramic resonator, an external oscillator with a frequency up to 40 MHz can be connected to the EXTAL input pin and the XTAL output pin must be left unconnected.

Refer to **Figure 2-1** for the following discussion. R_S (when used) and R_F should be low-inductance resistors such as carbon composition resistors. Wire-wound resistors, and some metal film resistors, have too much inductance. C1 and C2 normally should be high-quality ceramic capacitors that are specifically designed for high-frequency applications.

R_F is used to provide a bias path to keep the EXTAL input in its linear range during crystal startup and its value is not generally critical. Typical systems use 1 MΩ to 10 MΩ. Higher values are sensitive to humidity and lower values reduce gain and (in extreme cases) could prevent startup.

C1 and C2 are typically in the 5-pF to 25-pF range and are chosen to match the requirements of a specific crystal or resonator. Be sure to take into account printed circuit board (PCB) capacitance and MCU pin capacitance when sizing C1 and C2. The crystal manufacturer typically specifies a load capacitance which is the series combination of C1 and C2 which are usually the same size. As a first-order approximation, use 10 pF as an estimate of combined pin and PCB capacitance for each oscillator pin (EXTAL and XTAL).

Normally, R_S is used for the 32-kHz to 100-kHz range. Use up to 10 kΩ or consult the crystal manufacturer for recommendations. R_S is not normally needed for the 1-MHz to 16-MHz range and may be replaced with a direct connection.

2.2.3 Reset

Not all HCS08 derivatives have a reset pin. When there is no reset pin, you can cause a reset by cycling power to force power-on reset (POR), using a background command to write to the BDFR bit in the SBDFR register, or using software to force something like an illegal opcode reset.

In the MC9S08GB60, the reset pin is a dedicated pin with a pullup device built in. It has input hysteresis, a 10-mA output driver, and no output slew rate control. Internal power-on reset and low-voltage reset circuitry typically make external reset circuitry unnecessary. This pin is normally connected to the standard 6-pin background debug connector so a development system can directly reset the MCU system. If desired, a manual external reset can be added by supplying a simple switch to ground (pull reset pin low to force a reset).

Whenever any reset is initiated (whether from an external signal or from an internal system), the reset pin is driven low for about 4.25 μs, released, and sampled again about 4.75 μs later. If reset was caused by an internal source such as low-voltage reset or watchdog timeout, the circuitry expects the reset pin sample to return a logic 1. If the pin is still low at this sample point, the reset is assumed to be from an external source. The reset circuitry decodes the cause of reset and records it by setting a corresponding bit in the reset status register (SRS).

Never connect any significant capacitance to the reset pin because that would interfere with the circuit and sequence that detects the source of reset. If an external capacitance prevents the reset pin from rising to a valid logic 1 before the reset sample point, all resets will appear to be external resets.

2.2.4 Background/Mode Select (BKGD/MS)

The background/mode select (BKGD/MS) pin includes an internal pullup device, input hysteresis, a 2-mA output driver, and no output slew rate control. If nothing is connected to this pin, the MCU will enter normal operating mode at the rising edge of reset. If a debug system is connected to the 6-pin standard background debug header, it can hold

BKGD/MS low during the rising edge of reset which forces the MCU to active background mode.

The BKGD pin is used primarily for background debug controller (BDC) communications using a custom protocol that uses 16 clock cycles of the target MCU's BDC clock per bit time. The target MCU's BDC clock could be as fast as the 20-MHz bus clock rate, so there should never be any significant capacitance connected to the BKGD/MS pin that could interfere with background serial communications.

Although the BKGD pin is a pseudo open-drain pin, the background debug communication protocol provides brief, actively driven, high speedup pulses to ensure fast rise times. Small capacitances from cables and the absolute value of the internal pullup device play almost no role in determining rise and fall times on the BKGD pin.

2.2.5 General-Purpose I/O and Peripheral Ports

Fifty-six pins on the MC9S08GB60 are shared among general-purpose I/O and on-chip peripheral functions such as timers and serial I/O systems. Immediately after reset, all 56 of these pins except PTG0/BKGD/MS are configured as high-impedance general-purpose inputs with internal pullup devices disabled. To avoid extra current drain from floating input pins, the reset initialization routine in the application program should either enable on-chip pullup devices or change the direction of unused pins to outputs so the pins do not float.

For information about controlling these pins as general-purpose I/O pins or, for information about how and when on-chip peripheral systems use these pins, refer to the appropriate section from the data sheet for a particular derivative.

When an on-chip peripheral system is controlling a pin, data direction control bits still determine what is read from port data registers even though the peripheral module controls the pin direction by controlling the enable for the pin's output buffer.

Pullup enable bits for each of the 56 I/O pins control whether on-chip pullup or pulldown devices are enabled whenever the pin is acting as an input even if it is being controlled by an on-chip peripheral module.

Sometimes a pulldown resistor is substituted for the pullup resistor based on control bits, as in the MC9S08GB60 keyboard interrupt pins and IRQ pin. When the PTA7–PTA4 pins are controlled by the KBI module in the MC9S08GB60 and are configured for rising-edge/high-level sensitivity, the pullup enable control bits enable pulldown devices rather than pullup devices. Similarly, when the IRQ input in the MC9S08GB60 and is set to detect rising edges, the pullup enable control bit enables a pulldown device rather than a pullup device.

HCS08 outputs have software controlled slew rate. This feature allows you to effectively choose between two output transistor sizes. When the smaller size is chosen, the output switching slew rate is slower which can result in lower EMI noise. The larger size can be selected where speed of heavy loads are more important.

Some HCS08 output pins have high-current drivers capable of sourcing or sinking on the order of 10 mA each (subject to a total chip I/O current.

Section 3. Modes of Operation

3.1 Introduction

This section discusses stop and wait power-saving modes, as well as run mode versus the active background mode. Entry into each mode, exit from each mode, and functionality while in each of the modes are described.

An on-chip voltage regulator is a new feature of MCUs in Motorola’s HCS08 Family. The primary function of this regulator is to produce an internal 2.5-volt logic power supply from the MCU’s V_{DD} power supply. This regulator has standby, passthrough, and power-down modes, which are used to place an 9S08GB/GT into stop1, stop2, and stop3 modes. These modes and the related functions and registers are discussed in this section. Since registers and control bits may not be identical for all HCS08 derivatives, always refer to the data sheet for a specific derivative for more information.

3.2 Features

- Run mode for normal user operation
- Active background mode for code development
- Wait mode:
  - CPU shuts down to conserve power
  - System clocks running
  - Full voltage regulation maintained
- Stop modes:
  - System clocks stopped; voltage regulator in standby
  - Stop1 — Full power down of internal circuits for maximum power savings

  - Stop2 — Partial power down of internal circuits, RAM contents retained
  - Stop3 — All internal circuits powered for fast recovery
  - Separate periodic wakeup clock can stay running in stop2, stop3
  - Oscillator can be left on to reduce crystal startup time in stop3

3.3 Run Mode

This is the normal operating mode for the 9S08GB/GT. This mode is selected when the BKGD/MS pin is high at the rising edge of reset. In this mode, the CPU executes code from internal memory with execution beginning at the address fetched from memory at $FFFE:$FFFF after reset.

3.4 Active Background Mode

The active background mode functions are managed through the background debug controller (BDC) in the HCS08 core. The BDC, together with the on-chip debug module (DBG), provide the means for analyzing MCU operation during software development.

Active background mode is entered in any of five ways:

- When the BKGD/MS pin is low at the rising edge of reset
- When a BACKGROUND command is received through the BKGD pin
- When a BGND instruction is executed
- When encountering a BDC breakpoint
- When encountering a DBG breakpoint

Once in active background mode, the CPU is held in a suspended state waiting for serial background commands rather than executing instructions from the user's application program.

Background commands are of two types:

- Non-intrusive commands, defined as commands that can be issued while the user program is running. Non-intrusive commands can be issued through the BKGD pin while the MCU is in run mode; non-intrusive commands can also be executed when the MCU is in the active background mode. Non-intrusive commands include:
  - Memory access commands
  - Memory-access-with-status commands
  - BDC register access commands
  - The BACKGROUND command
- Active background commands, which can only be executed while the MCU is in active background mode. Active background commands include commands to:
  - Read or write CPU registers
  - Trace one user program instruction at a time
  - Leave active background mode to return to the user's application program (GO)

The active background mode is used to program a bootloader or user application program into the FLASH program memory before the MCU is operated in run mode for the first time. When the 9S08GB/GT is shipped from the Motorola factory, the FLASH program memory is erased by default unless specifically noted so there is no program that could be executed in run mode until the FLASH memory is initially programmed. The active background mode can also be used to erase and reprogram the FLASH memory after it has been previously programmed.

Users may choose to use some other communication channel such as the on-chip serial communications interface (SCI) to erase and reprogram the FLASH memory. Typically, the user would program a bootloader into the upper address locations of the FLASH. This bootloader could allow execution of normal user application programs. When some special sequence of characters is received through the SCI or some special combination of I/O signals is detected, control can be passed to the bootloader to allow FLASH erase and programming or other debug operations.

The user decides the operation of the bootloader program because the operation is not written and preprogrammed into the MCU by Motorola. The user is free to write this program to do anything within the MCU's capability. The function of this bootloader or other application programs is primarily limited by the imagination of the programmer.

For additional information about the active background mode, refer to **Section 7. Development Support**.

3.5 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 I bit in CCR is cleared when the CPU enters the wait mode, enabling interrupts. When an interrupt request occurs, the CPU exits the wait mode and resumes processing, beginning with the stacking operations leading to the interrupt service routine. Peripheral modules can be disabled to conserve power in wait mode but a peripheral must be enabled to be the source of an interrupt that will wake the MCU from wait.

Only the BACKGROUND command and memory-access-with-status commands are available when the MCU is in wait mode. The memory-access-with-status commands do not allow memory access, but they report an error indicating that the MCU is in either stop or wait mode. The BACKGROUND command can be used to wake the MCU from wait mode and enter active background mode.

3.6 Stop Modes

One of three stop modes is entered upon execution of a STOP instruction when the STOPE bit in the system option register is set. In all stop modes, all internal clocks are halted. If the STOPE bit is not set when the CPU executes a STOP instruction, the MCU will not enter any of the stop modes and an illegal opcode reset is forced. The stop modes are selected by setting the appropriate bits in the system power management status and control 2 register (SPMSC2).

Table 3-1 summarizes the behavior of the MCU in each of the stop modes.

Table 3-1. Stop Mode Behavior

Mode	CPU, Digital Peripherals, FLASH	RAM	Clock Module	ATD	KBI	Regulator	I/O Pins	RTI
Stop1	Off	Off	Off	Disabled	Off	Off	Reset	Off
Stop2	Off	Standby	Off	Disabled	Off	Standby	States held	Optionally on
Stop3	Standby	Standby	Standby (1)	Disabled	Optionally on	Standby	States held	Optionally on

1. Crystal oscillator can be configured to run in stop3. Please see the ICG registers.

Normally, the interrupt input paths for the IRQ and keyboard interrupt inputs pass through clocked synchronization logic. Since there are no clocks when the MCU is in stop mode, these synchronizers are bypassed in stop mode so asynchronous inputs to IRQ for all stop modes and keyboard interrupt inputs for stop3 can wake the MCU from stop.

Table 3-2 summarizes the configuration and exit conditions for stop1, stop2, and stop3.

Table 3-2. Stop Mode Selection and Source of Exit

Mode	SPMC2 Configuration		Source of Exit	Condition Upon Exit[1]
	PDC	PPDC		
Stop1	1	0	IRQ or reset	POR
Stop2	1	1	IRQ or reset, RTI	POR (PPDF bit set in SPMSCR)
Stop3	0	Don't care	IRQ or reset, RTI, KBI	Either reset or normal operation continues from the interrupt vector

[1] POR is valid exit in all cases.

3.6.1 Stop1 Mode

Stop1 mode provides the lowest possible standby power consumption by causing the internal circuitry of the MCU to be powered down. To select entry into stop1 mode, the PDC bit in SPMSC2 must be set and the PPDC bit in SPMSC2 must be clear upon execution of a STOP instruction.

When the MCU is in stop1 mode, all internal circuits that are powered from the voltage regulator are turned off. The voltage regulator is in a low-power standby state, as is the ATD.

Exit from stop1 is done by asserting either of the wake-up pins on the MCU: $\overline{\text{RESET}}$ or IRQ. IRQ is always an active low input when the MCU is in stop1, regardless of how it was configured before entering stop1.

Entering stop1 mode automatically asserts LVD. Stop1 cannot be exited until $V_{DD} > V_{LVDH/L}$ rising (V_{DD} must rise above the LVI rearm voltage).

Upon wake-up from stop1 mode, the MCU will start up as from a power-on reset (POR). The CPU will take the reset vector.

3.6.2 Stop2 Mode

Stop2 mode provides very low standby power consumption and maintains the contents of RAM and the current state of all of the I/O pins. To select entry into stop2, the user must execute a STOP instruction while the PPDC and PDC bits in SPMSC2 are set.

Before entering stop2 mode, the user can save the contents of the I/O port registers, as well as any other memory-mapped registers which they want to restore after exit of stop2, to locations in RAM. Upon exit of stop2, these values can be restored by user software before pin driver latches are opened.

When the MCU is in stop2 mode, all internal circuits that are powered from the voltage regulator are turned off, except for the RAM. The voltage regulator is in a low-power standby state, as is the ATD. Upon entry into stop2, the states of the I/O pins are latched. The states are held while in stop2 mode and after exiting stop2 mode until a logic 1 is written to PPDACK in SPMSC2.

Exit from stop2 is done by asserting either of the wake-up pins: $\overline{\text{RESET}}$ or IRQ, or by an RTI interrupt. IRQ is always an active low input when the MCU is in stop2, regardless of how it was configured before entering stop2. When the RTI is used to cause a wakeup event, a separate self-clocked source (≈1 kHz) for the real-time interrupt allows a wakeup from stop2 or stop3 mode with no external components. When RTIS2:RTIS1:RTIS0 = 0:0:0, the real-time interrupt function and this 1-kHz source are disabled. Power consumption is lower when the 1-kHz source is disabled.

Upon wake-up from stop2 mode, the MCU will start up as from a power-on reset (POR) except pin states remain latched. The CPU will take the reset vector. The system and all peripherals will be in their default reset states and must be initialized.

After waking up from stop2, the PPDF bit in SPMSC2 is set. This flag may be used to direct user code to go to stop2 recovery routine. PPDF remains set and the I/O pin states remain latched until a logic 1 is written to PPDACK in SPMSC2.

To maintain I/O state for pins that were configured as general-purpose I/O, the user must restore the contents of the I/O port registers, which have been saved in RAM, to the port registers before writing to the PPDACK bit. If the port registers are not restored from RAM before writing to PPDACK, then the register bits will assume their reset states when the I/O pin latches are opened and the I/O pins will switch to their reset states.

For pins that were configured as peripheral I/O, the user must reconfigure the peripheral module that interfaces to the pin before writing to the PPDACK bit. If the peripheral module is not enabled before writing to PPDACK, the pins will be controlled by their associated port control registers when the I/O latches are opened.

3.6.3 Stop3 Mode

Upon entering stop3 mode, all of the clocks in the MCU, including the oscillator itself, are halted. The clock module (ICG on the MC9S08GB/GT) enters its standby state, as does the voltage regulator and the ATD. The states of all of the internal registers and logic, as well as the RAM content, are maintained. The I/O pin states are not latched at the pin as in stop2. Instead they are maintained by virtue of the states of the internal logic driving the pins being maintained.

Exit from stop3 is done by asserting $\overline{\text{RESET}}$, an asynchronous interrupt pin, or through the real-time interrupt. The asynchronous interrupt pins are the IRQ or KBI pins.

If stop3 is exited by means of the $\overline{\text{RESET}}$ pin, then the MCU will be reset and operation will resume after taking the reset vector. Exit by means of an asynchronous interrupt or the real-time interrupt will result in the MCU taking the appropriate interrupt vector.

A separate self-clocked source (≈1 kHz) for the real-time interrupt allows a wakeup from stop2 or stop3 mode with no external components. When RTIS2:RTIS1:RTIS0 = 0:0:0, the real-time interrupt function and this 1-kHz source are disabled. Power consumption is lower when the 1-kHz source is disabled, but in that case the real-time interrupt cannot wake the MCU from stop.

3.6.4 Active BDM Enabled in Stop Mode

Entry into the active background mode from run mode is enabled if the ENBDM bit in BDCSCR is set. This register is described in the **Section 7. Development Support** section of this reference manual. If ENBDM is set when the CPU executes a STOP instruction, the system clocks to the background debug logic remain active when the MCU enters stop mode so background debug communication is still possible. In addition, the voltage regulator does not enter its low-power standby state but maintains full internal regulation. If the user attempts to enter either stop1 or stop2 with ENBDM set, the MCU will instead enter stop3.

Most background commands are not available in stop mode. The memory-access-with-status commands do not allow memory access, but they report an error indicating that the MCU is in either stop or wait mode. The BACKGROUND command can be used to wake the MCU from stop and enter active background mode if the ENBDM bit is set. Once in background debug mode, all background commands are available. The table below summarizes the behavior of the MCU in stop when entry into the background debug mode is enabled.

Table 3-3. BDM Enabled Stop Mode Behavior

Mode	PDC	PPDC	CPU, Digital Peripherals, FLASH	RAM	Clock Module	ATD	Regulator	I/O Pins	RTI
Stop3	Don’t care	Don’t care	Standby	Standby	Active	Disabled	Active	States held	Optionally on

3.6.5 OSCSTEN Bit Set

When the oscillator is enabled in stop mode (OSCSTEN = 1), the individual clock generators are enabled but the clock feed to the rest of the MCU is turned off. This option is provided to avoid long oscillator startup times if necessary.

3.6.6 LVD Enabled in Stop Mode

The LVD system is capable of generating either an interrupt or a reset when the supply voltage drops below the LVD voltage. If the LVD is enabled in stop by setting the LVDE and the LVDSE bits in SPMSC1 when the CPU executes a STOP instruction, then the voltage regulator remains active during stop mode. If the user attempts to enter either stop1 or stop2 with the LVD enabled for stop (LVDSE = 1), the MCU will instead enter stop3. The table below summarizes the behavior of the MCU in stop when the LVD is enabled.

Table 3-4. LVD Enabled Stop Mode Behavior

Mode	PDC	PPDC	CPU, Digital Peripherals, FLASH	RAM	Clock Module	ATD	Regulator	I/O Pins	RTI
Stop3	Don’t care	Don’t care	Standby	Standby	Standby	Disabled	Active	States held	Optionally on

3.6.7 On-Chip Peripheral Modules in Stop Modes

When the MCU enters any stop mode, system clocks to the internal peripheral modules are stopped. Even in the exception case (ENBDM = 1), where clocks are kept alive to the background debug logic, clocks to the peripheral systems are halted to reduce power consumption. Refer to **3.6.1 Stop1 Mode**, **3.6.2 Stop2 Mode**, and **3.6.3 Stop3 Mode** for specific information on system behavior in stop modes. The information provided here applies to the MC9S08GB60. Consult the device-specific data sheet for information about another MCU.

I/O Pins

- All I/O pin states remain unchanged when the MCU enters stop3 mode.
- If the MCU is configured to go into stop2 mode, all I/O pins states are latched before entering stop.
- If the MCU is configured to go into stop1 mode, all I/O pins are forced to their default reset state upon entry into stop.

Memory

The contents of the FLASH memory are non-volatile and are preserved in any of the stop modes.

- All RAM and register contents are preserved while the MCU is in stop3 mode.
- All registers will be reset upon wake-up from stop2, but the contents of RAM are preserved and pin states remain latched until the PPDACK bit is written. The user may save any memory-mapped register data into RAM before entering stop2 and restore the data upon exit from stop2.
- All registers will be reset upon wake-up from stop1 and the contents of RAM are not preserved. The MCU must be initialized as upon reset.

ICG — In stop3 mode, the ICG enters its low-power standby state. Either the oscillator or the internal reference may be kept running when the ICG is in standby by setting the appropriate control bit (OSCSTEN). In both stop2 and stop1 modes, the ICG is turned off. Neither the oscillator nor the internal reference can be kept running in stop2 or stop1, even if enabled within the ICG module. Upon exit from stop1 or stop2, the ICG must be initialized as if from a POR. The digitally controlled oscillator (DCO) in the ICG preserves previous frequency settings, allowing fast frequency lock when recovering from stop3 mode.

CPU — On entry to stop mode, the CPU clocks are stopped and CPU operation is halted. If the voltage regulator was not configured to go into power-down mode and an interrupt wakes the CPU from stop, CPU clocks are restored and the CPU resumes processing with the stacking operation leading to the interrupt service routine. When an RTI instruction is executed to return from this interrupt, the return address takes the CPU back to the instruction that immediately follows the STOP instruction. If the voltage regulator was powered down or a reset was used to wake the MCU from stop mode, processing resumes by fetching the reset vector.

TPM — When the MCU enters stop mode, the clock to the TPM1 and TPM2 modules stop. The modules halt operation. If the MCU is configured to go into stop2 or stop1 mode, the TPM modules will be reset upon wake-up from stop and must be reinitialized.

ATD — When the MCU enters stop mode, the ATD will enter a low-power standby state*.* No conversion operation will occur while in stop. If the MCU is configured to go into stop2 or stop1 mode, the ATD will be reset upon wake-up from stop and must be reinitialized.

KBI — During stop3, the KBI pins that are enabled continue to function as interrupt sources that are capable of waking the MCU from stop3. The KBI is disabled in stop1 and stop2 and must be reinitialized after waking up from either of these modes.

SCI — Take precautions to avoid going into stop mode while SCI communications are in progress. Since clocks are stopped, any serial character that was being received or sent will be stopped, causing the communication to fail. No SCI characters can be received while the MCU is stopped. When the MCU enters stop mode, the clocks to the SCI1 and SCI2 modules stop. The modules halt operation. If the MCU is configured to go into stop2 or stop1 mode, the SCI modules will be reset upon wake-up from stop and must be reinitialized.

SPI — It would be unusual to go into stop mode while SPI communications are in progress. Since clocks are stopped, any serial transfer that was in progress will be stopped. Since the SPI is a synchronous serial communication interface, there is no lower limit on the communication speed. Although it would be unusual, a transfer that was in progress when the MCU went into stop3 can resume after stop. No SPI transfers can be completed while the MCU is stopped. When the MCU enters stop mode, the clocks to the SPI module stop. The module halts operation. If the MCU is configured to go into stop2 or stop1 mode, the SPI module will be reset upon wake-up from stop and must be reinitialized.

IIC — When the MCU enters stop mode, the clocks to the IIC module stop. The module halts operation. If the MCU is configured to go into stop2 or stop1 mode, the IIC module will be reset upon wake-up from stop and must be reinitialized.

Voltage Regulator — The voltage regulator enters a low-power standby state when the MCU enters any of the stop modes unless the LVD is enabled in stop mode or BDM is enabled.

3.6.8 System Options Register (SOPT)

This register may be read at any time. Bits 3 and 2 are unimplemented and always read 0. This is a write-once register so only the first write after reset is honored. Any subsequent attempt to write to SOPT (intentionally or unintentionally) is ignored to avoid accidental changes to these sensitive settings. SOPT should be written during the user's reset initialization program to set the desired controls even if the desired settings are the same as the reset settings.

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	COPE[1]	COPT[1]	STOPE[1]	▨	0	0	BKGDPE	▨
Write:					▨	▨		
Reset:	1	1	0	1	0	0	1	1

▨ = Unimplemented or Reserved

[1] This bit can be written only one time after reset. Additional writes are ignored.

Figure 3-1. System Options Register (SOPT)

COPE — COP Watchdog Enable

This write-once bit defaults to 1 after reset. This bit does not relate directly to modes of operation, but is shown here because some bits in this register can be written only once after reset.

1 = COP watchdog timer enabled (force reset on timeout).
0 = COP watchdog timer disabled.

COPT — COP Watchdog Timeout

This write-once bit defaults to 1 after reset. This bit does not relate directly to modes of operation, but is shown here because some bits in this register can be written only once after reset.

1 = Long timeout period selected (2^{18} cycles of BUSCLK).
0 = Short timeout period selected (2^{13} cycles of BUSCLK).

STOPE — Stop Mode Enable

This write-once bit defaults to 0 after reset, which disables stop mode. If stop mode is disabled and a user program attempts to execute a STOP instruction, an illegal opcode reset is forced.

1 = Stop mode enabled.
0 = Stop mode disabled.

BKGDPE — Background Debug Mode Pin Enable

The BKGDPE bit enables the PTD0/BKGD/MS pin to function as BKGD/MS. When the bit is clear, the pin will function as PTD0, which is an output only general purpose I/O. This pin always defaults to BKGD/MS function after any reset.

1 = BKGD pin enabled.
0 = BKGD pin disabled.

3.6.9 System Power Management Status and Control 1 Register (SPMSC1)

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	LVDF	0	LVDIE	LVDRE[1]	LVDSE[1]	LVDE[1]	0	0
Write:	▨	LVDACK					▨	▨
Reset:	0	0	0	1	1	1	0	0

▨ = Unimplemented or Reserved

[1] This bit can be written only one time after reset. Additional writes are ignored.

Figure 3-2. System Power Management Status and Control 1 Register (SPMSC1)

LVDF — Low-Voltage Detect Flag

Provided LVDE = 1, this read-only status bit indicates a low-voltage detect event. This bit does not relate directly to modes of operation, but is shown here because some bits in this register can be written only once after reset.

LVDACK — Low-Voltage Detect Acknowledge

This write-only bit is used to acknowledge low voltage detection events (write 1 to clear LVDF). Reads always return logic 0. This bit does not relate directly to modes of operation, but is shown here because some bits in this register can be written only once after reset.

LVDIE — Low-Voltage Detect Interrupt Enable

This read/write bit enables hardware interrupt requests for LVDF. This bit does not relate directly to modes of operation, but is shown here because some bits in this register can be written only once after reset.
1 = Request a hardware interrupt when LVDF = 1.
0 = Hardware interrupt disabled (use polling).

LVDRE — Low-Voltage Detect Reset Enable

This read/write bit enables LVDF events to generate a hardware reset (provided LVDE = 1). This bit does not relate directly to modes of operation, but is shown here because some bits in this register can be written only once after reset.
1 = Force an MCU reset when LVDF = 1.
0 = LVDF does not generate hardware resets.

LVDSE — Low-Voltage Detect Stop Enable

Provided LVDE = 1, this read/write bit determines whether the low-voltage detect function operates when the MCU is in stop mode. This bit does not relate directly to modes of operation, but is shown here because some bits in this register can be written only once after reset.
1 = Low-voltage detect enabled during stop mode.
0 = Low-voltage detect disabled during stop mode.

LVDE — Low-Voltage Detect Enable

This read/write bit enables low-voltage detect logic and qualifies the operation of other bits in this register. This bit does not relate directly to modes of operation, but is shown here because some bits in this register can be written only once after reset.

1 = LVD logic enabled.
0 = LVD logic disabled.

3.6.10 System Power Management Status and Control 2 Register (SPMSC2)

This register is used to report the status of the low voltage warning function, and to configure the stop mode behavior of the MCU.

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	LVWF	0	LVDV	LVWV	PPDF	0	PDC	PPDC
Write:	▨	LVWACK			▨	PPDACK		
Power-on reset:	0[1]	0	0	0	0	0	0	0
LVD reset:	0[1]	0	U	U	0	0	0	0
Any other reset:	0[1]	0	U	U	0	0	0	0

▨ = Unimplemented or Reserved U = Unaffected by reset

[1] LVWF will be set in the case when V_{Supply} transitions below the trip point or after reset and V_{Supply} is already below V_{LVW}.

Figure 3-3. System Power Management Status and Control 2 Register (SPMSC2)

LVWF — Low-Voltage Warning Flag

The LVWF bit indicates the low voltage warning status. This bit does not relate directly to modes of operation, but is shown here because some bits in this register can be written only once after reset.

1 = Low voltage warning is present or was present.
0 = Low voltage warning **not** present.

LVWACK — Low-Voltage Warning Acknowledge

The LVWF bit indicates the low voltage warning status. This bit does not relate directly to modes of operation, but is shown here because some bits in this register can be written only once after reset.

Writing a logic 1 to LVWACK clears LVWF to a logic 0 if a low voltage warning is not present.

LVDV — Low-Voltage Detect Voltage Select

The LVDV bit selects the LVD trip point voltage (V_{LVD}). This bit does not relate directly to modes of operation, but is shown here because some bits in this register can be written only once after reset.
1 = High trip point selected (for 3 V system).
0 = Low trip point selected (for 2 V system).

LVWV — Low-Voltage Warning Voltage Select

The LVWV bit selects the LVW trip point voltage (V_{LVW}). This bit does not relate directly to modes of operation, but is shown here because some bits in this register can be written only once after reset.
1 = High trip point selected (for 3 V system).
0 = Low trip point selected (for 2 V system).

PPDF — Partial Power Down Flag

The PPDF bit indicates that the MCU has exited stop2 mode.
1 = Stop2 mode recovery.
0 = Not stop2 mode recovery.

PPDACK — Partial Power Down Acknowledge

Writing a logic 1 to PPDACK clears the PPDF bit.

PDC — Power Down Control

The write-once PDC bit controls entry into the power down (stop2 and stop1) modes.
1 = Power down modes are enabled.
0 = Power down modes are disabled.

PPDC — Partial Power Down Control

The write-once PPDC bit controls which power down mode, stop1 or stop2, is selected.

1 = Stop2, partial power down, mode enabled if PDC set.
0 = Stop1, full power down, mode enabled if PDC set.

Table 3-5. Stop Mode Selection and Source of Exit

Mode	SPMC2 Configuration		Source of Exit	Condition Upon Exit
	PDC	PPDC		
Stop1	1	0	IRQ or reset	POR
Stop2	1	1	IRQ or reset, RTI	POR (PPDF bit set in SPMSCR)
Stop3	0	Don’t care	IRQ or reset, RTI, KBI	If reset is used, then POR; else, normal operation continues from the interrupt vector

Section 4. On-Chip Memory

4.1 Introduction

This section shows the overall 64-Kbyte memory map and then explains each major memory block in greater detail.

- Direct-page registers, high-page registers, and nonvolatile registers are shown in tables which provide the register names, absolute addresses, and the arrangement of control and status bits within the registers.
- The RAM description includes information about initialization of the system stack pointer.
- The FLASH section explains programming and erase operations and block protection.
- The security section explains how internal FLASH and RAM contents can be protected against unauthorized access.
- The register descriptions explain the control and status bits associated with the FLASH memory module.

4.2 HCS08 Core-Defined Memory Map

In the HCS08 architecture, the core defines the address decode for six major blocks within the 64-Kbyte memory space. The on-chip memory modules use these block decode signals as module selects. The base address for each peripheral module is determined by additional decode logic in a system integration module which defines a block of addresses for each peripheral. The peripheral then uses this module select and additional low-order address lines to develop the select signals for each register within the module.

4.2.1 HCS08 Memory Map

The five major memory spaces that are defined by the core are shown in **Table 4-1**. Refer to the data sheet for a particular derivative for exact information about the size and boundaries of each of these blocks. **4.2.2 MC9S08GB60 Memory Map** shows the memory map for the MC9S08GB60 as a representative example of an HCS08 MCU memory map.

Table 4-1. Core-Defined Memory Spaces

Name	Address	Comment
Direct-page registers	$0000–$00xx	Up to 128 bytes
RAM	$00xx–	Includes some direct page locations
High-page registers	$1800–$18yy	System configuration
FLASH Memory	–$FFFF	Up to 60 Kbytes
Vectors	$FFC0–$FFFF	Up to 32 x 2 bytes

Direct-page registers include the I/O port registers and most peripheral control and status registers. Locating these registers in direct address space ($0000–$00xx) allows bit manipulation instructions to be used to set, clear, or test any bit in these registers with the BSET, BCLR, BRSET, and BRCLR instructions. Using the direct addressing mode versions of other instructions to access these registers also saves program space and execution time compared to the more general extended addressing mode instructions.

The RAM memory block starts immediately after the end of the direct-page register block and extends to higher addresses. For example in the MC9S08GB60, the direct-page registers are located at $0000–$007F and the 4096-byte RAM is located at $0080–$107F. This places a portion of the RAM in the direct addressing space so that frequently used program variables can take advantage of code size and execution time savings offered by the direct addressing mode version of many CPU instructions. Also, since the bit manipulation instruction only support direct addressing mode, this allows bit-addressable RAM variables.

High-page registers are located at $1800 to $182B. These are registers that are used less often than the direct-page registers so they are not located in the more valuable direct address space. This space includes a few system configuration registers such as the COP watchdog and low-voltage detect setup controls, the debug module registers, and the FLASH module registers.

A few of the registers in the high-page register area should always be located at the same addresses in all HCS08 derivatives. The SBDFR register at $1801 includes the BDFR control bit which allows a background debug host to reset the MCU by way of a serial command. There is also a device identification number in the SDIDH:SDIDL register pair at $1806 and $1807. These registers allow a host debug system to determine the type of HCS08 and the mask set revision number. This information allows the debug host to be aware of memory types and sizes, register names, bit names, and addresses in the target MCU.

FLASH memory fills the 64-Kbyte memory map to $FFFF. The starting address of this block depends on how much FLASH memory is included in the MCU. For example if there is 16 Kbytes of FLASH, it will be located at $C000–$FFFF. If the FLASH memory block overlaps the high-page register space, the register block has priority so the FLASH locations at the conflicting addresses are not accessible. This only occurs when there is more than 57 Kbytes of FLASH.

The vector space is part of the FLASH memory at $FFC0–$FFFF but it is separately decoded so that other HCS08 modules can recognize when an interrupt vector is being fetched.

Specific HCS08 derivatives have other address areas such as a block of nonvolatile registers and illegal address blocks. These areas are decoded in a system integration module rather than in the core.

4.2.2 MC9S08GB60 Memory Map

This section describes the memory map of the MC9S08GB60. The data sheet for each HCS08 device provides similar information explaining the detailed memory map for that HCS08 derivative.

As shown in **Figure 4-1**, on-chip memory in the MC9S08GB60 consists of RAM, FLASH program memory, plus I/O and control/status registers. The registers are divided into three groups:

- Direct-page registers ($0000 through $007F)
- High-page registers ($1800 through $182B)
- Nonvolatile registers ($FFB0 through $FFBF)

Reset and interrupt vectors are at $FFCC through $FFFF. An illegal address detect feature on some derivatives forces the MCU to reset if the CPU attempts to access data or execute an instruction from any address that is identified as an illegal address in the 64-Kbyte memory map.

Background debug mode (BDM) accesses do not trigger an illegal access error. On the MC9S08GB60, all 64 Kbytes of memory space are used for memory and registers so this device does not have any illegal address locations.

Unused and reserved locations in register areas are not considered designated illegal addresses and do not trigger illegal address resets.

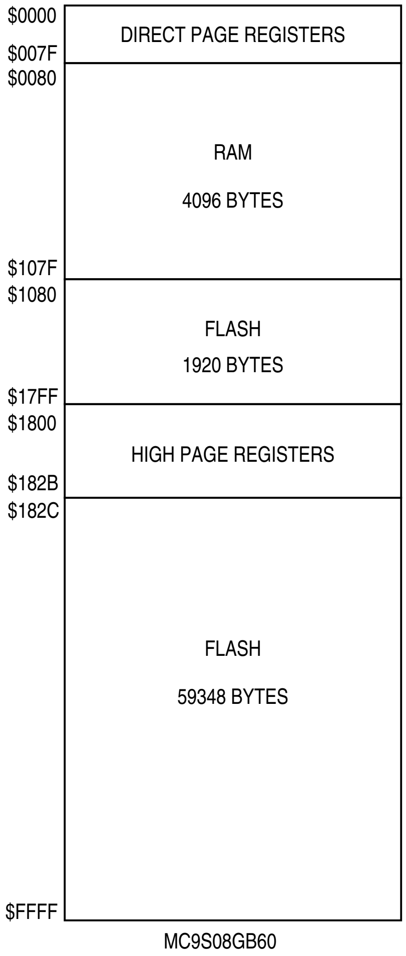


Figure 4-1. MC9S08GB60 Memory Map

4.2.3 Reset and Interrupt Vector Assignments

Table 4-2 shows address assignments for reset and interrupt vectors in the MC9S08GB60. For names and address assignments for vectors in other HCS08 derivatives, always refer to the appropriate data sheet. The vector names shown in this table are the labels used in the equate file provided by Motorola for the MC9S08GB60. For more details about resets, interrupts, interrupt priority, and local interrupt mask controls, refer to **Section 5. Resets and Interrupts**.

Table 4-2. Reset and Interrupt Vectors for the MC9S08GB60

Address (High/Low)	Vector	Vector Name
$FFC0:FFC1 ↕ $FFCA:FFCB	Unused Vector Space (available for user program)	
$FFCC:FFCD	RTI	Vrti
$FFCE:FFCF	IIC	Viic
$FFD0:FFD1	ATD Conversion	Vatd
$FFD2:FFD3	Keyboard	Vkeyboard
$FFD4:FFD5	SCI2 Transmit	Vsci2tx
$FFD6:FFD7	SCI2 Receive	Vsci2rx
$FFD8:FFD9	SCI2 Error	Vsci2err
$FFDA:FFDB	SCI1 Transmit	Vsci1tx
$FFDC:FFDD	SCI1 Receive	Vsci1rx
$FFDE:FFDF	SCI1 Error	Vsci1err
$FFE0:FFE1	SPI	Vspi
$FFE2:FFE3	TPM2 Overflow	Vtpm2ovf
$FFE4:FFE5	TPM2 Channel 4	Vtpm2ch4
$FFE6:FFE7	TPM2 Channel 3	Vtpm2ch3
$FFE8:FFE9	TPM2 Channel 2	Vtpm2ch2
$FFEA:FFEB	TPM2 Channel 1	Vtpm2ch1
$FFEC:FFED	TPM2 Channel 0	Vtpm2ch0
$FFEE:FFEF	TPM1 Overflow	Vtpm1ovf
$FFF0:FFF1	TPM1 Channel 2	Vtpm1ch2
$FFF2:FFF3	TPM1 Channel 1	Vtpm1ch1
$FFF4:FFF5	TPM1 Channel 0	Vtpm1ch0
$FFF6:FFF7	ICG	Vicg
$FFF8:FFF9	Low Voltage Detect	Vlvd
$FFFA:FFFB	IRQ	Virq
$FFFC:FFFD	SWI	Vswi
$FFFE:FFFF	Reset	Vreset

4.3 Register Addresses and Bit Assignments

The registers in the MC9S08GB60 are divided into these three groups:

- Direct-page registers are located in the first 128 locations in the memory map, so they are accessible with efficient direct addressing mode instructions.
- High-page registers are used much less often, so they are located above $1800 in the memory map. This leaves more room in the direct page for more frequently used registers and variables.
- The nonvolatile register area consists of a block of 16 locations in the 60-Kbyte FLASH memory at $FFB0–$FFBF.

  Nonvolatile register locations include:

  – Two values which are loaded into working registers at reset

  – An 8-byte backdoor comparison key which optionally allows a user to gain controlled access to secure memory

  – A reserved location for storage of a trim adjustment value that could be determined during final testing at Motorola

  Since the nonvolatile register locations are FLASH memory, they must be erased and programmed like other FLASH memory locations.

Direct-page registers can be accessed with efficient direct addressing mode instructions. Bit manipulation instructions can be used to access any bit in any direct-page register. **Table 4-3** is a summary of all user-accessible direct-page registers and control bits.

The registers in **Table 4-3** can use the more efficient direct addressing mode so, as a reminder, only the low order half of the addresses in the first column are shown in bold. In **Table 4-4** and **Table 4-5** the whole address in column one is shown in bold. In **Table 4-3**, **Table 4-4**, and **Table 4-5**, the register names in column two are shown in bold to set them apart from the bit names to the right. Cells that are not associated with named bits are shaded. A shaded cell with a 0 indicates this unused bit always reads as a 0. Shaded cells with dashes indicate unused or reserved bit locations that could read as 1s or 0s.

Table 4-3. Direct-Page Register Summary (Sheet 1 of 3)

Address	Register Name	Bit 7	6	5	4	3	2	1	Bit 0
$0000	PTAD	PTAD7	PTAD6	PTAD5	PTAD4	PTAD3	PTAD2	PTAD1	PTAD0
$0001	PTAPE	PTAPE7	PTAPE6	PTAPE5	PTAPE4	PTAPE3	PTAPE2	PTAPE1	PTAPE0
$0002	PTASE	PTASE7	PTASE6	PTASE5	PTASE4	PTASE3	PTASE2	PTASE1	PTASE0
$0003	PTADD	PTADD7	PTADD6	PTADD5	PTADD4	PTADD3	PTADD2	PTADD1	PTADD0
$0004	PTBD	PTBD7	PTBD6	PTBD5	PTBD4	PTBD3	PTBD2	PTBD1	PTBD0
$0005	PTBPE	PTBPE7	PTBPE6	PTBPE5	PTBPE4	PTBPE3	PTBPE2	PTBPE1	PTBPE0
$0006	PTBSE	PTBSE7	PTBSE6	PTBSE5	PTBSE4	PTBSE3	PTBSE2	PTBSE1	PTBSE0
$0007	PTBDD	PTBDD7	PTBDD6	PTBDD5	PTBDD4	PTBDD3	PTBDD2	PTBDD1	PTBDD0
$0008	PTCD	PTCD7	PTCD6	PTCD5	PTCD4	PTCD3	PTCD2	PTCD1	PTCD0
$0009	PTCPE	PTCPE7	PTCPE6	PTCPE5	PTCPE4	PTCPE3	PTCPE2	PTCPE1	PTCPE0
$000A	PTCSE	PTCSE7	PTCSE6	PTCSE5	PTCSE4	PTCSE3	PTCSE2	PTCSE1	PTCSE0
$000B	PTCDD	PTCDD7	PTCDD6	PTCDD5	PTCDD4	PTCDD3	PTCDD2	PTCDD1	PTCDD0
$000C	PTDD	PTDD7	PTDD6	PTDD5	PTDD4	PTDD3	PTDD2	PTDD1	PTDD0
$000D	PTDPE	PTDPE7	PTDPE6	PTDPE5	PTDPE4	PTDPE3	PTDPE2	PTDPE1	PTDPE0
$000E	PTDSE	PTDSE7	PTDSE6	PTDSE5	PTDSE4	PTDSE3	PTDSE2	PTDSE1	PTDSE0
$000F	PTDDD	PTDDD7	PTDDD6	PTDDD5	PTDDD4	PTDDD3	PTDDD2	PTDDD1	PTDDD0
$0010	PTED	PTED7	PTED6	PTED5	PTED4	PTED3	PTED2	PTED1	PTED0
$0011	PTEPE	PTEPE7	PTEPE6	PTEPE5	PTEPE4	PTEPE3	PTEPE2	PTEPE1	PTEPE0
$0012	PTESE	PTESE7	PTESE6	PTESE5	PTESE4	PTESE3	PTESE2	PTESE1	PTESE0
$0013	PTEDD	PTEDD7	PTEDD6	PTEDD5	PTEDD4	PTEDD3	PTEDD2	PTEDD1	PTEDD0
$0014	IRQSC	0	0	IRQEDG	IRQPE	IRQF	IRQACK	IRQIE	IRQMOD
$0015	Reserved	—	—	—	—	—	—	—	—
$0016	KBISC	KBEDG7	KBEDG6	KBEDG5	KBEDG4	KBF	KBACK	KBIE	KBIMOD
$0017	KBIPE	KBIPE7	KBIPE6	KBIPE5	KBIPE4	KBIPE3	KBIPE2	KBIPE1	KBIPE0
$0018	SCI1BDH	0	0	0	SBR12	SBR11	SBR10	SBR9	SBR8
$0019	SCI1BDL	SBR7	SBR6	SBR5	SBR4	SBR3	SBR2	SBR1	SBR0
$001A	SCI1C1	LOOPS	SCISWAI	RSRC	M	WAKE	ILT	PE	PT
$001B	SCI1C2	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK
$001C	SCI1S1	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF
$001D	SCI1S2	0	0	0	0	0	0	0	RAF
$001E	SCI1C3	R8	T8	TXDIR	0	ORIE	NEIE	FEIE	PEIE
$001F	SCI1D	Bit 7	6	5	4	3	2	1	Bit 0
$0020	SCI2BDH	0	0	0	SBR12	SBR11	SBR10	SBR9	SBR8
$0021	SCI2BDL	SBR7	SBR6	SBR5	SBR4	SBR3	SBR2	SBR1	SBR0
$0022	SCI2C1	LOOPS	SCISWAI	RSRC	M	WAKE	ILT	PE	PT
$0023	SCI2C2	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK
$0024	SCI2S1	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF
$0025	SCI2S2	0	0	0	0	0	0	0	RAF
$0026	SCI2C3	R8	T8	TXDIR	0	ORIE	NEIE	FEIE	PEIE
$0027	SCI2D	Bit 7	6	5	4	3	2	1	Bit 0

Table 4-3. Direct-Page Register Summary (Sheet 2 of 3)

Address	Register Name	Bit 7	6	5	4	3	2	1	Bit 0
$0028	SPIC1	SPIE	SPE	SPTIE	MSTR	CPOL	CPHA	SSOE	LSBFE
$0029	SPIC2	0	0	0	MODFEN	BIDIROE	0	SPISWAI	SPC0
$002A	SPIBR	0	SPPR2	SPPR1	SPPR0	0	SPR2	SPR1	SPR0
$002B	SPIS	SPRF	0	SPTEF	MODF	0	0	0	0
$002C	Reserved	0	0	0	0	0	0	0	0
$002D	SPID	Bit 7	6	5	4	3	2	1	Bit 0
$002E	Reserved	0	0	0	0	0	0	0	0
$002F	Reserved	0	0	0	0	0	0	0	0
$0030	TPM1SC	TOF	TOIE	CPWMS	CLKSB	CLKSA	PS2	PS1	PS0
$0031	TPM1CNTH	Bit 15	14	13	12	11	10	9	Bit 8
$0032	TPM1CNTL	Bit 7	6	5	4	3	2	1	Bit 0
$0033	TPM1MODH	Bit 15	14	13	12	11	10	9	Bit 8
$0034	TPM1MODL	Bit 7	6	5	4	3	2	1	Bit 0
$0035	TPM1C0SC	CH0F	CH0IE	MS0B	MS0A	ELS0B	ELS0A	0	0
$0036	TPM1C0VH	Bit 15	14	13	12	11	10	9	Bit 8
$0037	TPM1C0VL	Bit 7	6	5	4	3	2	1	Bit 0
$0038	TPM1C1SC	CH1F	CH1IE	MS1B	MS1A	ELS1B	ELS1A	0	0
$0039	TPM1C1VH	Bit 15	14	13	12	11	10	9	Bit 8
$003A	TPM1C1VL	Bit 7	6	5	4	3	2	1	Bit 0
$003B	TPM1C2SC	CH2F	CH2IE	MS2B	MS2A	ELS2B	ELS2A	0	0
$003C	TPM1C2VH	Bit 15	14	13	12	11	10	9	Bit 8
$003D	TPM1C2VL	Bit 7	6	5	4	3	2	1	Bit 0
$003E– $003F	Reserved	— —	— —	— —	— —	— —	— —	— —	— —
$0040	PTFD	PTFD7	PTFD6	PTFD5	PTFD4	PTFD3	PTFD2	PTFD1	PTFD0
$0041	PTFPE	PTFPE7	PTFPE6	PTFPE5	PTFPE4	PTFPE3	PTFPE2	PTFPE1	PTFPE0
$0042	PTFSE	PTFSE7	PTFSE6	PTFSE5	PTFSE4	PTFSE3	PTFSE2	PTFSE1	PTFSE0
$0043	PTFDD	PTFDD7	PTFDD6	PTFDD5	PTFDD4	PTFDD3	PTFDD2	PTFDD1	PTFDD0
$0044	PTGD	PTGD7	PTGD6	PTGD5	PTGD4	PTGD3	PTGD2	PTGD1	PTGD0
$0045	PTGPE	PTGPE7	PTGPE6	PTGPE5	PTGPE4	PTGPE3	PTGPE2	PTGPE1	PTGPE0
$0046	PTGSE	PTGSE7	PTGSE6	PTGSE5	PTGSE4	PTGSE3	PTGSE2	PTGSE1	PTGSE0
$0047	PTGDD	PTGDD7	PTGDD6	PTGDD5	PTGDD4	PTGDD3	PTGDD2	PTGDD1	PTGDD0
$0048	ICGC1	0	RANGE	REFS	CLKS		OSCSTEN	—*	0
$0049	ICGC2	LOLRE	MFD			LOCRE	RFD		
$004A	ICGS1	CLKST		REFST	LOLS	LOCK	LOCS	ERCS	ICGIF
$004B	ICGS2	0	0	0	0	0	0	0	DCOS
$004C	ICGFLTU	0	0	0	0	FLT			
$004D	ICGFLTL	FLT							
$004E	ICGTRM	TRIM							
$004F	Reserved	0	0	0	0	0	0	0	0

* This bit is reserved for Motorola internal use only. Always write a 0 to this bit.

Table 4-3. Direct-Page Register Summary (Sheet 3 of 3)

Address	Register Name	Bit 7	6	5	4	3	2	1	Bit 0
$0050	ATDC	ATDPU	DJM	RES8	SGN	PRS			
$0051	ATDSC	CCF	ATDIE	ATDCO	ATDCH				
$0052	ATDRH	BIT9	BIT 8	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2
$0053	ATDRL	BIT1	BIT0	0	0	0	0	0	0
$0054	ATDPE	ATDPE7	ATDPE6	ATDPE5	ATDPE4	ATDPE3	ATDPE2	ATDPE1	ATDPE0
$0055– $0057	Reserved	— —	— —	— —	— —	— —	— —	— —	— —
$0058	IICA	ADDR							0
$0059	IICF	MULT		ICR					
$005A	IICC	IICEN	IICIE	MST	TX	TXAK	RSTA	0	0
$005B	IICS	TCF	IAAS	BUSY	ARBL	0	SRW	IICIF	RXAK
$005C	IICD	DATA							
$005D– $005F	Reserved	— —	— —	— —	— —	— —	— —	— —	— —
$0060	TPM2SC	TOF	TOIE	CPWMS	CLKSB	CLKSA	PS2	PS1	PS0
$0061	TPM2CNTH	Bit 15	14	13	12	11	10	9	Bit 8
$0062	TPM2CNTL	Bit 7	6	5	4	3	2	1	Bit 0
$0063	TPM2MODH	Bit 15	14	13	12	11	10	9	Bit 8
$0064	TPM2MODL	Bit 7	6	5	4	3	2	1	Bit 0
$0065	TPM2C0SC	CH0F	CH0IE	MS0B	MS0A	ELS0B	ELS0A	0	0
$0066	TPM2C0VH	Bit 15	14	13	12	11	10	9	Bit 8
$0067	TPM2C0VL	Bit 7	6	5	4	3	2	1	Bit 0
$0068	TPM2C1SC	CH1F	CH1IE	MS1B	MS1A	ELS1B	ELS1A	0	0
$0069	TPM2C1VH	Bit 15	14	13	12	11	10	9	Bit 8
$006A	TPM2C1VL	Bit 7	6	5	4	3	2	1	Bit 0
$006B	TPM2C2SC	CH2F	CH2IE	MS2B	MS2A	ELS2B	ELS2A	0	0
$006C	TPM2C2VH	Bit 15	14	13	12	11	10	9	Bit 8
$006D	TPM2C2VL	Bit 7	6	5	4	3	2	1	Bit 0
$006E	TPM2C3SC	CH3F	CH3IE	MS3B	MS3A	ELS3B	ELS3A	0	0
$006F	TPM2C3VH	Bit 15	14	13	12	11	10	9	Bit 8
$0070	TPM2C3VL	Bit 7	6	5	4	3	2	1	Bit 0
$0071	TPM2C4SC	CH4F	CH4IE	MS4B	MS4A	ELS4B	ELS4A	0	0
$0072	TPM2C4VH	Bit 15	14	13	12	11	10	9	Bit 8
$0073	TPM2C4VL	Bit 7	6	5	4	3	2	1	Bit 0
$0074– $007F	Reserved	— —	— —	— —	— —	— —	— —	— —	— —

High-page registers, shown in **Table 4-4**, are accessed much less often than other I/O and control registers so they have been located outside the direct addressable memory space, starting at $1800.

Table 4-4. High-Page Register Summary

Address	Register Name	Bit 7	6	5	4	3	2	1	Bit 0
$1800	**SRS**	POR	PIN	COP	ILOP	0	ICG	LVD	0
$1801	**SBDFR**	0	0	0	0	0	0	0	BDFR
$1802	**SOPT**	COPE	COPT	STOPE	—	0	0	BKGDPE	—
$1803– $1805	Reserved	— —	— —	— —	— —	— —	— —	— —	— —
$1806	**SDIDH**	REV3	REV2	REV1	REV0	ID11	ID10	ID9	ID8
$1807	**SDIDL**	ID7	ID6	ID5	ID4	ID3	ID2	ID1	ID0
$1808	**SRTISC**	RTIF	RTIACK	RTICLKS	RTIE	0	RTIS2	RTIS1	RTIS0
$1809	**SPMSC1**	LVDF	LVDACK	LVDIE	LVDRE	LVDSE	LVDE	0	0
$180A	**SPMSC2**	LVWF	LVWACK	LVDV	LVWV	PPDF	PPDACK	PDC	PPDC
$180B– $180F	Reserved	— —	— —	— —	— —	— —	— —	— —	— —
$1810	**DBGCAH**	Bit 15	14	13	12	11	10	9	Bit 8
$1811	**DBGCAL**	Bit 7	6	5	4	3	2	1	Bit 0
$1812	**DBGCBH**	Bit 15	14	13	12	11	10	9	Bit 8
$1813	**DBGCBL**	Bit 7	6	5	4	3	2	1	Bit 0
$1814	**DBGFH**	Bit 15	14	13	12	11	10	9	Bit 8
$1815	**DBGFL**	Bit 7	6	5	4	3	2	1	Bit 0
$1816	**DBGC**	DBGEN	ARM	TAG	BRKEN	RWA	RWAEN	RWB	RWBEN
$1817	**DBGT**	TRGSEL	BEGIN	0	0	TRG3	TRG2	TRG1	TRG0
$1818	**DBGS**	AF	BF	ARMF	0	CNT3	CNT2	CNT1	CNT0
$1819– $181F	Reserved	— —	— —	— —	— —	— —	— —	— —	— —
$1820	**FCDIV**	DIVLD	PRDIV8	DIV5	DIV4	DIV3	DIV2	DIV1	DIV0
$1821	**FOPT**	KEYEN	FNORED	0	0	0	0	SEC01	SEC00
$1822	Reserved	—	—	—	—	—	—	—	—
$1823	**FCNFG**	0	0	KEYACC	0	0	0	0	0
$1824	**FPROT**	FPOPEN	FPDIS	FPS2	FPS1	FPS0	0	0	0
$1825	**FSTAT**	FCBEF	FCCF	FPVIOL	FACCERR	0	FBLANK	0	0
$1826	**FCMD**	FCMD7	FCMD6	FCMD5	FCMD4	FCMD3	FCMD2	FCMD1	FCMD0
$1827– $182B	Reserved	— —	— —	— —	— —	— —	— —	— —	— —

Nonvolatile FLASH registers, shown in **Table 4-5**, are located in the FLASH memory and include two nonvolatile setup registers for the FLASH memory module plus an 8-byte backdoor key which optionally can be used to gain access to secure memory resources. During reset events, the contents of the two locations in the nonvolatile register area of the FLASH memory are transferred into corresponding working registers in the high-page registers to control security and block protection options.

Table 4-5. Nonvolatile Register Summary

Address	Register Name	Bit 7	6	5	4	3	2	1	Bit 0
$FFB0– $FFB7	NVBACKKEY	8-Byte Comparison Key							
$FFB8– $FFBC	Reserved	— —	— —	— —	— —	— —	— —	— —	— —
$FFBD	NVPROT	FPOPEN	FPDIS	FPS2	FPS1	FPS0	0	0	0
$FFBE	Reserved[1]	—	—	—	—	—	—	—	—
$FFBF	NVOPT	KEYEN	FNORED	0	0	0	0	SEC01	SEC00

[1] This location can be used to store a trim value for the ICG.

Provided the key enable (KEYEN) bit is 1, the 8-byte comparison key can be used to temporarily disengage memory security. This key mechanism can be accessed only through user code running in secure memory. (A security key cannot be entered directly through background debug commands.) This security key can be disabled completely by programming the KEYEN bit to 0. If the security key is disabled, the only way to disengage security is by mass erasing the FLASH (normally through the background debug interface) and verifying that FLASH is blank. To avoid returning to secure mode after the next reset, program the security bits (SEC01:SEC00) to the unsecured state (1:0). See **4.6 Security (MC9S08GB60)** for more details about secure memory.

4.4 RAM

The MC9S08GB60 includes 4096 bytes of static RAM located from \$0080 to \$107F. The first 128 bytes of RAM (\$0080–\$00FF) can be accessed using the more efficient direct addressing mode, and any single bit in this area can be accessed with the bit manipulation instructions (BCLR, BSET, BRCLR, and BRSET). Locating the most frequently accessed program variables in this area of RAM is preferred.

Provided the V_{DD} supply voltage remains above the minimum RAM retention voltage and stop1 mode is not entered, RAM locations retain their contents. If stop1 mode is selected by setting the PDC bit and clearing the PPDC bit in SPMSC2, when stop1 is entered, the internal voltage regulator is turned off and voltage is disabled to internal circuitry, including the RAM. Upon exit from stop1, RAM contents are uninitialized and all other registers return to their reset state. (See **Section 3. Modes of Operation** for more information about stop modes.)

For compatibility with older M68HC05 MCUs, the HCS08 resets the stack pointer to \$00FF. In the MC9S08GB60, it is usually best to reinitialize the stack pointer to the top of the RAM (\$107F) so the direct page RAM (\$0080–\$00FF) can be used for frequently accessed RAM variables and bit-addressable program variables. Include the following 2-instruction sequence in your reset initialization routine (where RamLast is equated to \$107F in the equate file provided by Motorola).

```
            LDHX     #RamLast+1     ;point one past RAM
            TXS                     ;SP<-(H:X-1)
```

4.5 60-Kbyte FLASH

The 60-Kbyte FLASH memory is intended primarily for program storage. In-circuit programming allows the operating program to be loaded into the FLASH memory after final assembly of the application product. It is possible to program the entire 60-Kbyte array through the single-wire background debug interface in about three seconds. Because no special voltages are needed for FLASH erase and programming operations, in-application programming is also possible through the serial communications interface (SCI) (RS232 interface) or some other software-controlled communication path. For a more detailed discussion of in-circuit and in-application programming, refer to **4.8 FLASH Application Examples**.

4.5.1 Features

Features of the FLASH memory include:

- FLASH — 61268 bytes (120 pages of 512 bytes each)
- Single power supply program and erase
- Command interface for fast program and erase operation
- Fast automated byte program, page or mass erase, and blank check operations (about three seconds to program 60 Kbytes)
- Up to 100,000 program/erase cycles at typical temperature and voltage
- Flexible block protection
- Security feature for FLASH and RAM
- Auto power-down for low-frequency read accesses

This FLASH memory module includes integrated program/erase voltage generators and separate command processor state machines which are capable of performing automated byte programming, page (512 bytes FLASH) or mass erase, and blank check commands. Commands are written to the command interface, and status flags report errors and indicate when commands are complete.

Blocks of 512, 1K, 2K, 4K, 8K, 16K, or 32K bytes at the end of the FLASH memory can be block protected. Another control bit allows for block protection of the whole 60-Kbyte FLASH array (see **4.7.4 FLASH Protection Register (FPROT and NVFPROT)**. Block protect settings are programmed into a nonvolatile setup register (NVFPROT). A security mechanism can be engaged to prevent unauthorized access to the FLASH and RAM memory contents. An optional user-controlled backdoor key mechanism can be used to allow controlled access to secure memory contents for development purposes.

4.5.2 Program, Erase, and Blank Check Commands

Before any program or erase command can be accepted, the FLASH clock divider register (FCDIV) must be written to set the internal clock for the FLASH module to a frequency (f_{FCLK}) between 150 kHz and 200 kHz (see **4.7.1 FLASH Clock Divider Register (FCDIV)**). This register can be written only once, so normally this write is done during reset initialization. One period of the resulting clock ($1/f_{FCLK}$) is used by the command processor to time program and erase pulses. An integer number of these timing pulses are used by the command processor to complete a program or erase command.

Commands are written to the command interfaces of the FLASH to do any of these:

- Program a byte in the FLASH array
- Erase a 512-byte page of FLASH memory
- Mass erase the whole 60-Kbyte FLASH array
- Check all bytes in the FLASH array for the erased state ($FF)

A strictly monitored procedure must be followed or the command will not be accepted. This minimizes the possibility of any unintended change to the FLASH memory contents. The command buffer empty flag (FCBEF) indicates when the command buffer has room to write a new command. The command complete flag (FCCF) indicates when all commands are complete and no new command is waiting in the associated FLASH command buffer. A command sequence must be completed by writing a 1 to FCBEF to register the command before starting any new command for the FLASH memory.

Figure 4-2 demonstrates the procedure for issuing commands. Two types of errors can arise as commands are issued:

- A protection violation error is indicated by the FPVIOL flag in FSTAT if the command tries to erase or write to a FLASH location that is block protected (see **4.7.4 FLASH Protection Register (FPROT and NVFPROT)**).
- Any other violation of the required sequence or other error condition will set the access error (FACCERR) flag bit in the FSTAT register. Refer to **4.5.4 Access Errors** for a detailed list of actions that cause access errors.

Assuming no protection violation or access errors arise, a command sequence can be simplified to three basic steps. They are:

1. Write a data value to an address in the FLASH array. The address and data information from this write is latched into the command buffer and this write is a required first step in any command sequence. For erase and blank check commands, the value of the data is not important. For page erase commands, the address may be any address in the 512-byte page of FLASH to be erased. For mass erase and blank check commands, the address can be any address in the 60-Kbyte FLASH memory.
2. Write the command code for the desired command to FCMD. The five valid commands are blank check ($05), byte program ($20), burst program ($25), page erase ($40), and mass erase ($41). The command code is latched into the command buffer.
3. Write a 1 to the FCBEF bit in FSTAT to clear FCBEF and register the command (including its address and data information).

A partial command sequence can be aborted manually by writing a 0 to FCBEF any time after the write to the memory array and before writing the 1 that clears FCBEF and registers the complete command. Aborting a command in this way sets the FACCERR access error flag which must be cleared before starting a new command.

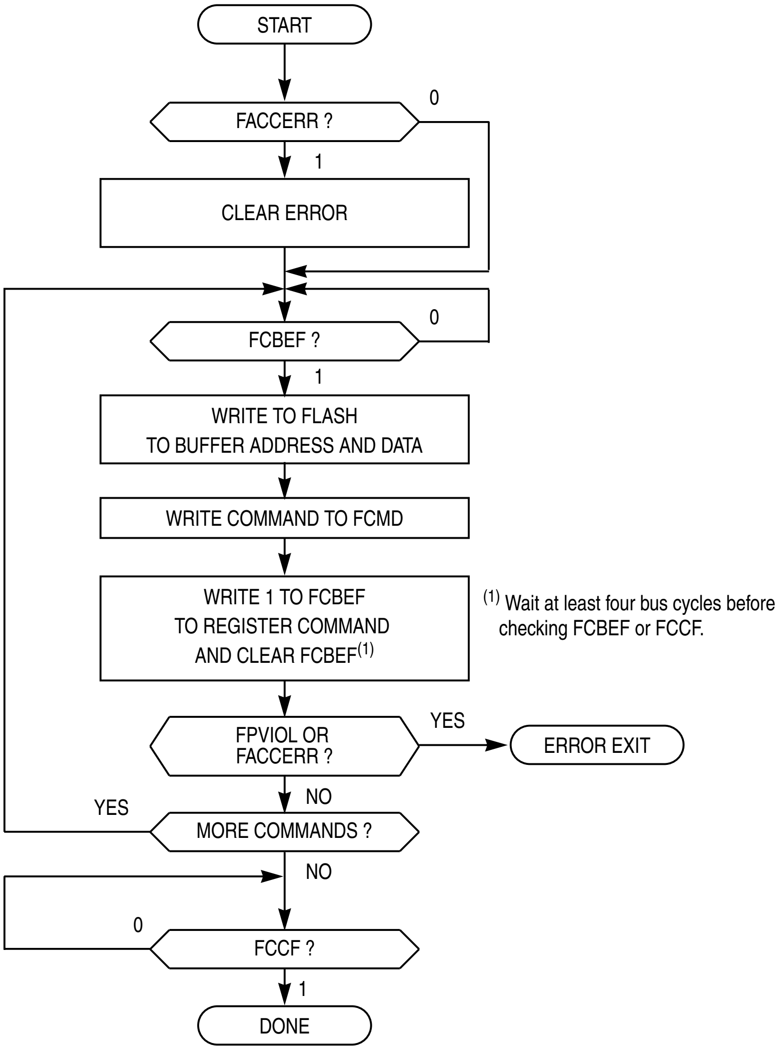


Figure 4-2. FLASH Command Flowchart

4.5.3 Command Timing and Burst Programming

This section explains the sequence and timing of nonvolatile memory commands in greater detail. When more than one byte within a row is programmed one after the other, it is called burst programming. Byte programming takes slightly longer for the first byte in a row compared to queued byte programming commands for subsequent bytes within the same row.

4.5.3.1 Rows and FLASH Organization

The 60-Kbyte FLASH memory array is made up of 120 pages of 512 bytes each. Each page is made up of 8 rows of 64 bytes each, beginning at address $1000. Address lines A5–A0 define an address within a FLASH row, A8–A6 identify the row number, and A15–A9 identify the page number. Whole pages of 512 bytes are the smallest block of FLASH that may be erased. The first 128 bytes ($1000–$107F) of the first FLASH row are hidden behind the higher priority RAM located at these same locations.

Rows are important because a burst program command takes less time when the address is within the same row as the previous byte or burst program command. To benefit from this reduced program time, the burst programming command must be registered in the command buffer before the previous byte programming operation in the same row is completed (otherwise, the small extra overhead for a new byte programming operation applies).

4.5.3.2 Program Command Timing Sequence

For this discussion, we assume the FCDIV setting results in a 5-µs timing pulse to the command state machine. If the FCDIV setting and system clock speed result in a different timing pulse period, all programming time intervals will need to be adjusted accordingly.

A complete program command consists of seven timing intervals. They are:

- Start — 0 to 5 µs, depending on synchronization between the command and the 200-kHz internal nonvolatile memory clock. When the command buffer is kept full, each command ends at an edge of the 200-kHz clock. The new command needs to wait a full period to synchronize to the clock so the start time can normally be taken to be the full 5 µs.
- Nonvolatile setup — 5 µs
- Program setup — 10 µs
- Program byte — 20 µs

- Program hold — 10 ns (negligible)
- Nonvolatile hold — 5 µs (minus the program hold time)
- Memory recover time — 5 µs

Programming more than one location in the same row (and as long as the command buffer remains filled with a burst program command so there is no gap between commands) is called burst programming, and all steps except the byte programming time are skipped.

Table 4-6 shows program and erase times. System clock and control bit settings determine the frequency of FCLK (f_{FCLK}). The time for one cycle of FCLK is $t_{Fcyc} = 1/f_{FCLK}$. The times are shown as a number of cycles of FCLK and as an absolute time for the case where t_{Fcyc} = 5 µs.

Table 4-6. Program and Erase Times

Parameter	Cycles of FCLK	Time if FCLK = 200 kHz
Byte program	9	45 µs
Byte program (burst)	4	20 µs[1]
Page erase	4000	20 ms
Mass erase	40,000	200 ms

[1] Excluding start/end overhead

4.5.4 Access Errors

Any of the following specific actions will cause the access error flag (FACCERR) in FSTAT to be set. In the case of an access error, FACCERR must be cleared by writing a 1 to FACCERR in FSTAT before starting a new command.

- Writing to a FLASH address before the internal FLASH clock frequency has been set by writing to the FCDIV register
- Writing to an unimplemented FLASH location before writing to FCMD (MC9S08GB60 has no unimplemented FLASH locations.)
- Writing to a FLASH address while FCBEF is not set (A new command cannot be started until the command buffer is empty.)

- Writing a second time to a FLASH address before registering the previous command (There is only one write to FLASH for every command.)
- Writing a second time to FCMD before registering the previous command (There is only one write to FCMD for every command.)
- Writing to any FLASH control register other than FCMD after writing to a FLASH address
- Writing any command code other than the five allowed codes ($05, $20, $25, $40, or $41) to FCMD
- Writing to any FLASH control register other than FSTAT (to clear FCBEF and register the command) after writing the command to FCMD
- The MCU enters stop mode while a program or erase command is in progress (The command is aborted.)
- Writing the byte program, burst program, or page erase command code ($20, $25, or $40) with a background debug command while the MCU is secured (The background debug controller can only do blank check and mass erase commands when the MCU is secure.)
- Writing 0 to FCBEF to cancel a partial command

4.5.5 Vector Redirection

Whenever any block protection is enabled, the reset and interrupt vectors will be protected. Vector redirection allows users to modify interrupt vector information without unprotecting bootloader and reset vector space. Vector redirection is enabled by programming the FNORED bit in the NVOPT register located at address $FFBF to zero. For redirection to occur, at least some portion but not all of the FLASH memory must be block protected by programming the NVPROT register located at address $FFBD. All of the interrupt vectors (memory locations $FFC0–$FFFD) are redirected, while the reset vector ($FFFE:FFFF) is not.

For example, if 512 bytes of FLASH are protected, the protected address region is from $FE00 through $FFFF. The interrupt vectors

($FFC0–$FFFD) are redirected to the locations $FDC0–$FDFD. Now, if an SPI interrupt is taken for instance, the values in the locations $FDE0:FDE1 are used for the vector instead of the values in the locations $FFE0:FFE1. This allows the user to reprogram the unprotected portion of the FLASH with new program code including new interrupt vector values while leaving the protected area, which includes the default vector locations, unchanged.

4.5.6 FLASH Block Protection (MC9S08GB60)

Block protection prevents program or erase changes for FLASH memory locations in a designated address range. Mass erase is disabled when any block of FLASH is protected. The MC9S08GB60 allows a block of memory at the end of FLASH and/or the entire 60 Kbytes of FLASH memory to be block protected. A disable control bit and a 3-bit control field allow you to set the size of this block to 512, 1K, 2K, 4K, 8K, 16K, or 32K bytes. A separate control bit allows block protection of the whole 60-Kbyte FLASH memory array. All five of these control bits are located in the FPROT register (see **4.7.4 FLASH Protection Register (FPROT and NVFPROT)**).

At reset, the high-page register (FPROT) is loaded with the contents of the NVFPROT location which is in the nonvolatile register block of the FLASH memory. The value in FPROT cannot be changed directly from application software so a runaway program cannot alter the block protection settings. If the last 512 bytes of FLASH which includes the NVFPROT register is protected, the application program cannot alter the block protection settings (intentionally or unintentionally). The FPROT control bits can be written by background debug commands to allow a way to erase a protected FLASH memory.

One use for block protection is to block protect an area of FLASH memory for a bootloader program. Then this bootloader program can be used to erase the rest of the FLASH memory and reprogram it. Since the bootloader is protected, it remains intact even if MCU power is lost in the middle of an erase and reprogram operation.

4.6 Security (MC9S08GB60)

The MC9S08GB60 includes circuitry to prevent unauthorized access to the contents of FLASH and RAM memory. When security is engaged, FLASH and RAM are considered secure resources. Direct-page registers, high-page registers, and the background debug controller are considered unsecured resources. Programs executing within secure memory have normal access to any MCU memory locations and resources. Attempts to access a secure memory location with a program executing from an unsecured memory space or through the background debug interface are blocked (writes are ignored and reads return all 0s).

Security is engaged or disengaged based on the state of two nonvolatile register bits (SEC01:SEC00) in the FOPT register. During reset, the contents of the nonvolatile location NVFOPT are copied from FLASH into the working FOPT register in high-page register space. A user engages security by programming the NVFOPT location which can be done at the same time the FLASH memory is programmed. The 1:0 state disengages security while the other three combinations engage security. Notice that the erased state (1:1) makes the MCU secure. During development, whenever the FLASH is erased, it is good practice to immediately program the SEC00 bit to 0 in NVFOPT so SEC01:SEC00 = 1:0. This would allow the MCU to remain unsecured after a subsequent reset.

The on-chip debug module cannot be enabled while the MCU is secure. The separate background debug controller can still be used for non-intrusive background memory access commands, but the MCU cannot enter active background mode except by holding BKGD/MS low at the rising edge of reset.

A user can choose to allow or disallow a security unlocking mechanism through an 8-byte backdoor security key. If the nonvolatile KEYEN bit in NVFOPT/FOPT is 0, the backdoor key is disabled and there is no way to disengage security without completely erasing all FLASH locations. If KEYEN is 1, a secure user program can temporarily disengage security by:

1. Writing 1 to KEYACC in the FCNFG register. This makes the FLASH module interpret writes to the backdoor comparison key locations (NVBACKKEY through NVBACKKEY+7) as values to be

compared against the key rather than as the first step in a FLASH program or erase command.

2. Writing the user-entered key values to the NVBACKKEY through NVBACKKEY+7 locations. These writes must be done in order starting with the value for NVBACKKEY and ending with NVBACKKEY+7. Normally, user software would get the key codes from outside the MCU system through a communication interface such as the SCI.
3. Writing 0 to KEYACC in the FCNFG register. If the 8-byte key that was just written matches the key stored in the FLASH locations, SEC01:SEC00 are automatically changed to 1:0 and security will be disengaged until the next reset.

The security key can be written only from a secure memory, so it cannot be entered through background commands without the cooperation of a secure user program.

The backdoor comparison key (NVBACKKEY through NVBACKKEY+7) is located in FLASH memory locations in the nonvolatile register space so users can program these locations just as they would program any other FLASH memory location. The nonvolatile registers are in the same 512-byte block of FLASH as the reset and interrupt vectors, so block protecting that space also block protects the backdoor comparison key. Block protects cannot be changed from user application programs, so if the vector space is block protected, the backdoor security key mechanism cannot permanently change the block protect, security settings, or the backdoor key.

Security can always be disengaged through the background debug interface by following these steps:

1. Disable any block protections by writing FPROT. FPROT can be written only with background debug commands, not from application software.
2. Mass erase FLASH, if necessary.
3. Blank check FLASH. Provided FLASH is completely erased, security is disengaged until the next reset.

   To avoid returning to secure mode after the next reset, program NVFOPT so SEC01:SEC00 = 1:0.

4.7 FLASH Registers and Control Bits (MC9S08GB60)

Although these registers and bits are representative of the FLASH registers and control bits in any HCS08 derivative, always refer to the data sheet for a specific HCS08 derivative when writing application software. The FLASH module in the MC9S08GB60 has six 8-bit registers in the high-page register space, two locations in the nonvolatile register space in FLASH memory which are copied into two corresponding high-page control registers at reset. There is also an 8-byte comparison key in FLASH memory. Refer to **Table 4-4** and **Table 4-5** for the absolute address assignments for all FLASH registers. This section refers to registers and control bits only by their names. Normally, an equate or header file provided by Motorola is used to translate these names into the appropriate absolute addresses.

4.7.1 FLASH Clock Divider Register (FCDIV)

Bit 7 of this register is a read-only status flag. Bits 6 through 0 may be read at any time but can be written only one time. Before any erase or programming operations are possible, write to this register to set the frequency of the clock for the nonvolatile memory system within acceptable limits.

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	DIVLD	PRDIV8	DIV5	DIV4	DIV3	DIV2	DIV1	DIV0
Write:	▨							
Reset:	0	0	0	0	0	0	0	0

▨ = Unimplemented or Reserved

Figure 4-3. FLASH Clock Divider Register (FCDIV)

DIVLD — Divisor Loaded Status Flag

When set, this read-only status flag indicates that the FCDIV register has been written since reset. Reset clears this bit and the first write to this register causes this bit to become set regardless of the data written.

1 = FCDIV has been written since reset; erase and program operations enabled for FLASH

0 = FCDIV has not been written since reset; erase and program operations disabled for FLASH

PRDIV8 — Prescale (Divide) FLASH Clock by 8

1 = Clock input to the FLASH clock divider is the bus rate clock divided by 8

0 = Clock input to the FLASH clock divider is the bus rate clock

[DIV5:DIV0] — Divisor for FLASH Clock Divider

The FLASH clock divider divides the bus rate clock (or the bus rate clock divided by 8 if PRDIV8 = 1) by the value in the 6-bit [DIV5:DIV0] field plus one. The resulting frequency of the internal FLASH clock must fall within the range of 200 kHz to 150 kHz for proper FLASH operation. Program/Erase timing pulses are one cycle of this internal FLASH clock which corresponds to a range of 5 µs to 6.7 µs. The automated programming logic uses an integer number of these pulses to complete an erase or program operation.

Equation 1: if PRDIV8 = 0, then $f_{FCLK} = f_{Bus} \div ([DIV5:DIV0] + 1)$

Equation 2: if PRDIV8 = 1, then $f_{FCLK} = f_{Bus} \div (8 \times ([DIV5:DIV0] + 1))$

Table 4-7 shows the appropriate values for PRDIV8 and [DIV5:DIV0] for selected bus frequencies.

Table 4-7. FLASH Clock Divider Settings

f_{Bus}	PRDIV8 (Binary)	[DIV5:DIV0] (Decimal)	f_{FCLK}	Program/Erase Timing Pulse (5 µs Min, 6.7 µs Max)
20 MHz	1	12	192.3 kHz	5.2 µs
10 MHz	0	49	200 kHz	5 µs
8 MHz	0	39	200 kHz	5 µs
4 MHz	0	19	200 kHz	5 µs
2 MHz	0	9	200 kHz	5 µs
1 MHz	0	4	200 kHz	5 µs
200 kHz	0	0	200 kHz	5 µs
150 kHz	0	0	150 kHz	6.7 µs

4.7.2 FLASH Options Register (FOPT and NVFOPT)

During reset, the contents of the nonvolatile location NVOPT are copied from FLASH into FOPT. Bits 6 through 2 are not used and always read 0. This register may be read at any time, but writes have no meaning or effect. To change the value in this register, erase and reprogram the NVOPT location in FLASH memory as usual and then issue a new MCU reset.

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	KEYEN	FNORED	0	0	0	0	SEC01	SEC00
Write:	▨	▨	▨	▨	▨	▨	▨	▨
Reset:	This register is loaded from nonvolatile location NVOPT during reset.							

▨ = Unimplemented or Reserved

Figure 4-4 FLASH Options Register (FOPT)

KEYEN — Backdoor Key Mechanism Enable

When this bit is 0, the backdoor key mechanism cannot be used to disengage security. The backdoor key mechanism is accessible only from user (secured) firmware. BDM commands cannot be used to write key comparison values that would unlock the backdoor key. For more detailed information about the backdoor key mechanism, refer to **4.6 Security (MC9S08GB60)**.

1 = If user firmware writes an 8-byte value that matches the nonvolatile backdoor key (NVBACKKEY through NVBACKKEY+7 in that order), security is temporarily disengaged until the next MCU reset.

0 = No backdoor key access allowed

FNORED — Vector Redirection Disable

When this bit is 1, then vector redirection is disabled.

1 = Vector redirection disabled.

0 = Vector redirection enabled.

SEC01:SEC00 — Security State Code

This 2-bit field determines the security state of the MCU as shown in **Table 4-8**. When the MCU is secure, the contents of RAM and FLASH memory cannot be accessed by instructions from any unsecured source including the background debug interface. For more detailed information about security, refer to **4.6 Security (MC9S08GB60)**.

Table 4-8. Security States

SEC01:SEC00	Description
0:0	secure
0:1	secure
1:0	unsecured
1:1	secure

SEC01:SEC00 changes to 1:0 after successful backdoor key entry or a successful blank check of the FLASH memory.

4.7.3 FLASH Configuration Register (FCNFG)

Bit 5 may be read or written at any time. The remaining bits always read 0 and cannot be written.

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	0	0	KEYACC	0	0	0	0	0
Write:	▨	▨		▨	▨	▨	▨	▨
Reset:	0	0	0	0	0	0	0	0

▨ = Unimplemented or Reserved

Figure 4-5. FLASH Configuration Register (FCNFG)

KEYACC — Enable Writing of Access Key

This bit enables writing of the backdoor comparison key. For more detailed information about the backdoor key mechanism, refer to **4.6 Security (MC9S08GB60)**.

1 = Writes to NVBACKKEY ($FFB0–$FFB7) are interpreted as comparison key writes.

0 = Writes to $FFB–$FFB7 are interpreted as the start of a FLASH programming or erase command.

4.7.4 FLASH Protection Register (FPROT and NVFPROT)

During reset, the contents of the nonvolatile location NVFPROT is copied from FLASH into FPROT. Bit 6 is not used and always reads 0. This register may be read at any time, but user program writes have no meaning or effect. Background debug commands can write to FPROT at $1824.

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	FPOPEN	FPDIS	FPS2	FPS1	FPS0	0	0	0
Write:	(1)	(1)	(1)	(1)	(1)	▨	▨	▨
Reset:	This register is loaded from nonvolatile location NVPROT during reset.							

▨ = Unimplemented or Reserved

1. Background commands can be used to change the contents of these bits in FPROT.

Figure 4-6. FLASH Protection Register (FPROT)

FPOPEN — Open Unprotected FLASH for Program/Erase

1 = Any FLASH location, not otherwise block protected or secured, may be erased or programmed.

0 = Whole FLASH is block protected (no program or erase allowed).

FPDIS — FLASH Protection Disable

1 = No FLASH block is protected.

0 = FLASH block specified by FPS2:FPS1:FPS0 is block protected (program and erase not allowed).

FPS2:FPS1:FPS0 — FLASH Protect Selects

When FPDIS = 0, this 3-bit field determines the size of a protected block of FLASH locations at the high address end of the FLASH (see **Table 4-9**). Protected FLASH locations cannot be erased or programmed.

Table 4-9. High Address Protected Block

FPS2:FPS1:FPS0	Protected Address Range	Protected Block Size	Redirected Vectors[1]
0:0:0	$FE00–$FFFF	512 bytes	$FDC0–$FDFD[2]
0:0:1	$FC00–$FFFF	1 Kbytes	$FBC0–$FBFD
0:1:0	$F800–$FFFF	2 Kbytes	$F7C0–$F7FD
0:1:1	$F000–$FFFF	4 Kbytes	$EFC0–$EFFD
1:0:0	$E000–$FFFF	8 Kbytes	$DFC0–$DFFD
1:0:1	$C000–$FFFF	16 Kbytes	$BFC0–$BFFD
1:1:0	$8000–$FFFF	32 Kbytes	$7FC0–$7FFD
1:1:1	$8000–$FFFF	32 Kbytes	$7FC0–$7FFD

[1] No redirection if FPOPEN = 0, or FNORED = 1.
[2] Reset vector is not redirected.

4.7.5 FLASH Status Register (FSTAT)

Bits 3, 1, and 0 always read 0 and writes have no meaning or effect. The remaining five bits are status bits that can be read at any time. Writes to these bits have special meanings that are discussed in the bit descriptions.

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	FCBEF	FCCF	FPVIOL	FACCERR	0	FBLANK	0	0
Write:		▨			▨	▨	▨	▨
Reset:	1	1	0	0	0	0	0	0

▨ = Unimplemented or Reserved

Figure 4-7. FLASH Status Register (FSTAT)

FCBEF — FLASH Command Buffer Empty Flag

FLASH commands are buffered so a second command can be written into the buffer while the command processor is executing another command during a burst programming sequence. As soon as a command is finished, the command processor can start on an additional burst programming command if one is present in the buffer.

FCBEF is set automatically when the command buffer can accept a new command. A command is registered, and FCBEF is cleared, by writing a 1 to the FCBEF bit. Writing 0 to FCBEF, after a write to the FLASH but before the FCBEF clear that registers the command, causes the partially entered command to be manually aborted and clears the command buffer.

1 = A new command may be written to the command buffer.
0 = Command buffer is full (not ready for additional commands).

FCCF — FLASH Command Complete Flag

FCCF is set automatically when the command buffer is empty and no command is being processed. FCCF is cleared automatically when a new command is started (by writing 1 to FCBEF to register a command). Writing to FCCF has no meaning or effect.

1 = All commands complete
0 = Command in progress

FPVIOL — Protection Violation Flag

FPVIOL is set automatically when FCBEF is cleared to register a command that attempts to erase or program a location in a protected block (the erroneous command is ignored). FPVIOL is cleared automatically by writing a 1 to FPVIOL.

1 = An attempt was made to erase or program a protected location.
0 = No protection violation

FACCERR — Access Error Flag

FACCERR is set automatically when the proper command sequence is not followed exactly (the erroneous command is ignored), if a program or erase operation is attempted before the FCDIV register has been initialized, or if the MCU enters stop while a command was in progress. For a more detailed discussion of the exact actions that are considered access errors, see **4.5.4 Access Errors**. FACCERR is cleared by writing a 1 to FACCERR. Writing a 0 to FACCERR has no meaning or effect.

1 = An access error has occurred.
0 = No access error

FBLANK — FLASH Verified as All Blank (erased) Flag

FBLANK is set automatically at the conclusion of a blank check command if the entire FLASH array was verified to be erased. FBLANK is cleared by clearing FCBEF to write a new valid command. Writing to FBLANK has no meaning or effect.

1 = After a blank check command is completed and FCCF = 1, FBLANK = 1 indicates the FLASH array is completely erased (all $FF).

0 = After a blank check command is completed and FCCF = 1, FBLANK = 0 indicates the FLASH array is not completely erased.

4.7.6 FLASH Command Register (FCMD)

Bits 7, 4, 3, and 1 always read 0 and cannot be written by user application programs. Only five command codes are recognized in normal user modes as shown in **Table 4-10**. Refer to **4.5.2 Program, Erase, and Blank Check Commands** for a detailed discussion of FLASH programming and erase operations.

	Bit 7	6	5	4	3	2	1	Bit 0
Read: Write:	FCMP7	FCMP6	FCMP5	FCMP4	FCMP3	FCMP2	FCMP1	FCMP0
Reset:	0	0	0	0	0	0	0	0

Figure 4-8. FLASH Command Register (FCMD)

Table 4-10. FLASH Commands

Command	FCMD	Equate File Label
Blank check	$05	mBlank
Byte program	$20	mByteProg
Byte program — burst mode	$25	mBurstProg
Page erase (512 bytes/page)	$40	mPageErase
Mass erase (all FLASH)	$41	mMassErase

All other command codes are illegal and generate an access error.

It is not necessary to perform a blank check command after a mass erase operation. Blank check is required only as part of the security unlocking mechanism.

4.8 FLASH Application Examples

This section discusses several examples to demonstrate how programming and erase operations are performed on the FLASH in an HCS08 MCU. These examples focus on the routines that would be found in typical application systems as opposed to the programs that are used to program the initial application programs into the FLASH the first time. Normally, a third-party development tool would be used to program the first application programs (including programs such as those shown in these examples) into the HCS08 system.

A complete monitor program is presented and discussed in application note AN2140, *Serial Monitor for MC9S08GB60*. This bootloader resides in protected FLASH at the high-address end of the FLASH and works through the asynchronous serial communications interface (SCI1) of the MC9S08GB60 to allow a user to program or erase FLASH, or debug user applications.

A set of primitive binary monitor commands is supported by this monitor so a host debugger running on your PC can read and write memory or registers, set breakpoints, trace instructions, or go to a user program. Refer to AN2140 for more information.

Most third-party debug systems and programmers use the background debug interface for all programming operations. Typically, they would download a small routine into the RAM of the target system and then jump to that routine. This is more efficient than manipulating the FLASH programming controls through serial background debug commands so it is the preferred method when larger blocks of nonvolatile memory need to be programmed. Since the nonvolatile memory modules in HCS08 devices have built-in state machines to process critical timing operations, it is possible to manipulate the programming controls directly through serial background commands. Normally, this would only be done if the development host needed to program a few individual locations.

4.8.1 Initialization of the FLASH Module Clock

The internal state machines that control programming and erase operations on the FLASH use a 150 kHz to 200 kHz clock (FCLK) which is derived by dividing the BUSCLK. The FLASH clock divider register (FCDIV) is used to set the divider. FCDIV can only be written one time after reset and no programming or erase operations are allowed until this register has been written. It is customary to write this register during a reset initialization routine shortly after reset.

The divider must be set so that FCLK is between 150 kHz and 200 kHz. Programming and erase operations use a fixed number of these clock cycles so the closer FCLK is to 200 kHz, the faster commands can be performed. For example if FCLK is 200 kHz, it takes 45 microseconds to program a single random location in FLASH. If FCLK is 150 kHz, the same byte program operation takes 60 microseconds.

Refer to **Figure 4-9** for the following discussion. The first part of this code example shows an application equate which sets up the initialization value for the FCDIV register. The second part shows the two lines of code that would be placed in the reset initialization routine. Notice that we could not use a MOV instruction to set the initial value in FCDIV because it is a high-page register and MOV can only be used for immediate, direct, or indexed operands. The initialization value shown in this example is for a system that has a 32.768 kHz crystal and is using the FLL to multiply this up to BUSCLK = 18.874368 MHz. The value in FCDIV causes this to be divided by 8 × 12, producing FCLK = 196.608 kHz (as close to 200 kHz as possible without going over).

```
initFCDIV:  equ     %01001011     ;FLASH clock divider
;                    ||||||||
;                    |||||||+-DIV0 \
;                    ||||||+--DIV1 |
;                    |||||+---DIV2  >-- divide by (11+1)
;                    ||||+----DIV3 |    BUSCLK/(8*12)~=196,608 Hz
;                    |||+-----DIV4 |
;                    ||+------DIV5 /
;                    |+-------PRDIV8 -- divide (prescale) by 8
;                    +--------DIVLD --- read-only status
```

```
             lda    initFCDIV
             sta    FCDIV            ;set fFCLK = about 200kHz
```

Figure 4-9. FCLK Initialization

The requirement for FCLK to be at least 150 kHz implies that BUSCLK must also be at least 150 kHz (because the smallest divide that can be set by FCDIV is 1). This requirement only applies to programming and erase operations, not to reads. This means lower bus frequencies may be used to reduce power consumption, but the bus frequency must be at least 150 kHz during program and erase operations.

Applications that adjust the bus frequency during normal operations (using post-FLL divider controls), must be aware of the FCLK frequency requirements for programing and erase operations. Since the FCDIV register is write-once, it cannot be adjusted to accommodate dynamic changes in bus frequency. During program and erase operations, the bus clock would need to be changed to make FCLK fall within legal limits. Many applications do not adjust the bus clock frequency dynamically so this issue does not arise.

4.8.2 Erase One 512-Byte Page in FLASH

Program and erase operations for the FLASH memory are a little more complicated compared to many application programs because it is not possible to execute a program out of FLASH during FLASH program and erase operations. This example shows one way to overcome this limitation by placing the routine on the stack so the CPU is executing out of stack RAM while the FLASH is unavailable due to the program or erase operation.

The example shown in **Figure 4-10** is located in the FLASH memory and can be used to erase one 512-byte page of FLASH (that is, any page other than the page where this routine is located). This routine is useful because HCS08 devices have no separate EEPROM. In an HCS08 device, one or more pages of FLASH could be used for storage of nonvolatile configuration values or logged history data. Typically, the main body of the application code, including these routines, would reside in a block protected portion of the FLASH. A BDM interface pod is required to change the block protection settings so protected code cannot be erased accidentally or altered as a result of an application program error.

This FlashErase1 routine calls the DoOnStack subroutine which, in turn, copies a small instruction sequence onto the stack and jumps to that stack routine to complete the requested FLASH program or erase command before returning to the calling program in FLASH. The initial steps in the FLASH program or erase command can be executed from within the FLASH, but the command sequence itself should not be executed from within the FLASH memory.

```
;*********************************************************************
;* FlashErase1 - erases one page of FLASH (512 bytes)
;*
;* On entry... H:X - points at a location in the page to be erased
;*
;* Calling convention:
;*          jsr    FlashErase1
;*
;* Uses: DoOnStack which uses SpSub
;* Returns: H:X unchanged and A = FSTAT shifted left by 2 bits
;*  Z=1 if OK, Z=0 if protect violation or access error
;*  uses 32 bytes of stack space + 2 bytes for BSR/JSR used to call it
;*********************************************************************
FlashErase1: psha                 ;adjust sp for DoOnStack entry
             lda    #(mFPVIOL+mFACCERR) ;mask
             sta    FSTAT         ;abort any command and clear errors
             lda    #mPageErase   ;mask pattern for page erase command
             bsr    DoOnStack     ;finish command from stack-based sub
             ais    #1            ;deallocate data location from stack
             rts                  ;Z = 0 means there was an error
;********************
```

Figure 4-10. Erase One 512-Byte Page in FLASH

The PDHA instruction at the beginning of FlashErase1 places a dummy data value onto the stack so the DoOnStack subroutine can fetch it with an LDA SpSubSize+6,sp instruction later. The AIS #1 instruction just before the RTS instruction at the end of FlashErase1 deallocates this byte before returning.

Just in case there was a pending protection violation or access error (FPVIOL or FACCERR) from some previous operation, the second and third instructions in FlashErase1 will clear these flags so the command processor is ready to receive a new command. Within this example case we do not check these error flags because we are assuming we know what we are doing. However, some applications will include additional checks of FPVIOL and FACCERR to guard against unintended errors such as an attempt to erase a protected location.

4.8.3 DoOnStack Subroutine

This is an unusual subroutine because it moves instructions onto the stack and then jumps there so that the FLASH command subroutine finishes execution from the stack RAM. This solves the problem that you cannot execute instructions out of the FLASH memory while any program or erase operation is in progress. The DoOnStack subroutine is located in FLASH, but during the critical portion of the routine when the program or erase command is actually in progress, the CPU will be executing instructions on the stack (that is, in the on-chip RAM).

```
;*********************************************************************
;* DoOnStack - copy SpSub onto stack and call it (see also SpSub)
;*  Deallocates the stack space used by SpSub after returning from it.
;*  Allows flash prog/erase command to execute out of RAM (on stack)
;*  while flash is out of the memory map.
;*  This routine can be used for flash byte-program or erase commands
;*
;* Calling Convention:
;*          psha                  ;save data to program (or dummy
;*                                ; data for an erase command)
;*          lda    #(mFPVIOL+mFACCERR) ;mask
;*          sta    FSTAT         ;abort any command and clear errors
;*          lda    #mByteProg    ;mask pattern for byte prog command
;*          jsr    DoOnStack     ;execute prog code from stack RAM
;*          ais    #1            ;deallocate data location from stack
;*                               ; without disturbing A or CCR
;*
;*                 or substitute #mPageErase for page erase
```

```
;*
;* Uses 29 bytes on stack + 2 bytes for BSR/JSR used to call it
;* returns H:X unchanged and A=0 and Z=1 if no flash errors
;********************************************************************
DoOnStack:   pshx
             pshh                  ;save pointer to flash
             psha                  ;save command on stack
             ldhx   #SpSubEnd      ;point at last byte to move to stack
SpMoveLoop:  lda    ,x             ;read from flash
             psha                  ;move onto stack
             aix    #-1            ;next byte to move
             cphx   #SpSub-1       ;past end?
             bne    SpMoveLoop     ;loop till whole sub on stack
             tsx                   ;point to sub on stack
             tpa                   ;move CCR to A for testing
             and    #$08           ;check the I mask
             bne    I_set          ;skip if I already set
             sei                   ;block interrupts while FLASH busy
             lda    SpSubSize+6,sp ;preload data for command
             jsr    ,x             ;execute the sub on the stack
             cli                   ;ok to clear I mask now
             bra    I_cont         ;continue to stack de-allocation
I_set:       lda    SpSubSize+6,sp ;preload data for command
             jsr    ,x             ;execute the sub on the stack
I_cont:      ais    #SpSubSize+3   ;deallocate sub body + H:X + command
                                   ;H:X flash pointer OK from SpSub
             lsla                  ;A=00 & Z=1 unless PVIOL or ACCERR
             rts                   ;to flash where DoOnStack was called
;********************
```

Figure 4-11. DoOnStack Subroutine (Complete FLASH Command)

First, DoOnStack pushes the FLASH location pointer (H:X) and the command code (A) onto the stack to free up these CPU registers. H:X is set to point at the last byte of the SpSub subroutine. Next, a 5-instruction loop copies the stack routine from FLASH onto the stack one byte at a time. After moving the last byte onto the stack, SP points at the next lower address. The TSX instruction adds one to SP as the value is copied to the H:X register pair. This leaves H:X pointing at the first byte of the routine that was just moved onto the stack.

The next several instructions are used to determine whether or not interrupts are masked. If interrupts are masked (I set to 1), A is loaded with the data for the FLASH program or erase operation and the copy of SpSub on the stack is called. If interrupts were not masked, an SEI instruction is used to block interrupts, A is loaded, SpSub is called (JSR ,X), and the ACLI re-enables interrupts. The stack subroutine is described in **4.8.4 SpSub Subroutine** immediately below.

After returning from SpSub, the AIS #SpSubSize+3 instruction deallocates the stack space used for SpSub and associated parameters. ASLA moves the PVIOL and ACCERR error flags to the most significant 2 bits of A. A should now be 0 if there were no errors.

4.8.4 SpSub Subroutine

The SpSub subroutine (see **Figure 4-12**) is moved onto the stack by the DoOnStack subroutine (described in **4.8.3 DoOnStack Subroutine** immediately above) and then it is called (from DoOnStack). This subroutine completes the program or erase command and then waits for all FLASH commands to finish before returning. These instructions are located on the stack in on-chip RAM when they are executed. This satisfies the requirement that you cannot execute instructions out of FLASH while a program or erase command is in progress.

```
;*******************************************************************
;* SpSub - This variation of SpSub performs all of the steps for
;* programming or erasing flash from RAM. SpSub is copied onto the
;* stack, SP is copied to H:X, and then the copy of SpSub in RAM is
;* called using a JSR 0,X instruction.
;*
;* At the time SpSub is called, the data to be programmed (dummy data
;* for an erase command), is in A and the flash address is on the
;* stack above SpSub. After return, PVIOL and ACCERR flags are in bits
;* 6 and 5 of A. If A is shifted left by one bit after return, it
;* should be zero unless there was a flash error.
;*
;* Uses 24 bytes on stack + 2 bytes if a BSR/JSR calls it
;*******************************************************************
SpSub:       ldhx   SpSubSize+4,sp ;get flash address from stack
             sta    0,x            ;write to flash; latch addr and data
             lda    SpSubSize+3,sp ;get flash command
             sta    FCMD           ;write the flash command
             lda    #mFCBEF        ;mask to initiate command
             sta    FSTAT          ;[pwpp] register command
             nop                   ;[p] want min 4~ from w cycle to r
ChkDone:     lda    FSTAT          ;[prpp] so FCCF is valid
             lsla                  ;FCCF now in MSB
             bpl    ChkDone        ;loop if FCCF = 0
SpSubEnd:    rts                   ;back into DoOnStack in flash
SpSubSize:   equ    (*-SpSub)

;********************
```

Figure 4-12. SpSub Subroutine (Executes on Stack)

In SpSub, H:X is loaded (using a stack pointer-relative LDHX instruction) with the address for the FLASH program or erase operation. The STA o,x instruction completes the first step of the FLASH program or erase command sequence. Next, another stack pointer-relative LOAD instruction is used to load A with the command code for a PageErase or a ByteProgram command and this code is written to FCMD. The next two instruction write a 1 to the FCBEF bit in FSTAT to register the command and start the program or erase operation.

The cycle-by-cycle activity for the STA FSTAT, NOP, and LDA FSTAT instructions is shown in square brackets in the comment fields of these instructions because there is a requirement that there must be at least four cycles after the FSTAT write that registers the command before the first read to check the FCBEF or FCCF status flags. The p cycles are program fetch cycles, the w cycle is where the FSTAT register was written, and the r cycle is where the FSTAT register is read.

Next, the ASLA instruction moves the FCCF flag to the MSB of the accumulator and sets or clears the N bit in the CCR according to the value of FCCF (now in this MSB). If FCCF was clear, the BPL instruction will cause a branch back to ChkDoneE1 to repeat the status check. When FCCF is set, the branch will fall through indicating the command is finished and no additional commands are pending. At this point, the FLASH reappears in the memory map so it is safe to use the RTS instruction to return to the calling program in FLASH.

4.8.5 Program One Byte of FLASH

This example demonstrates a simple routine to program a single location in FLASH. It assumes the location was previously blank (erased to $FF) and does not perform any error checking. We assume we are following the correct programming procedure so we will not get access errors and we assume the programmer knows that the location is not located in a protected block which would cause a protection violation error. This example uses the DoOnStack and SpSub routines described in **4.8.3 DoOnStack Subroutine** and **4.8.4 SpSub Subroutine** above.

```
;*********************************************************************
;* FlashProg1 - programs one byte of FLASH
;*  This routine waits for the command to complete before returning.
;*  assumes location was blank. This routine can be run from FLASH
;*
;* On entry... H:X - points at the FLASH byte to be programmed
;*             A holds the data for the location to be programmed
;*
;* Calling convention:
;*           jsr    FlashProg1
;*
;* Uses: DoOnStack which uses SpSub
;* Returns: H:X unchanged and A = FSTAT shifted left by 2 bits
;*  Z=1 if OK, Z=0 if protect violation or access error
;*  uses 32 bytes of stack space + 2 bytes for BSR/JSR used to call it
;*********************************************************************
FlashProg1:  psha                 ;temporarily save entry data
             lda    #(mFPVIOL+mFACCERR) ;mask
             sta    FSTAT         ;abort any command and clear errors
             lda    #mByteProg    ;mask pattern for byte prog command
             bsr    DoOnStack     ;execute prog code from stack RAM
             ais    #1            ;deallocate data location from stack
             rts                  ;Z = 0 means there was an error
;********************
```

Figure 4-13. Program One Byte in FLASH

One advantage of the way FlashProg1 and FlashErase1 are written is that this code can reside in FLASH. Only the code for the actual programming or erase operation is copied onto the stack so it can be executed in RAM while the FLASH is out of the memory map.

One drawback to this approach is that each command must be completed before anything else can be done. For applications where only a few locations are programmed at a time, this limitation is not serious. On the other hand, this approach would not be appropriate for programming larger blocks of data into the FLASH. For those cases use an approach where the entire programming algorithm is located in a RAM routine. Burst programming commands can be queued such that there is always another command waiting in a buffer so it can immediately transfer into the on-chip command processor as soon as the previous command finishes. In the case of programming multiple bytes within the same 64-byte FLASH row, this allows burst programming which takes less than half as long as programming a single isolated byte.

Section 5. Resets and Interrupts

5.1 Introduction

This section discusses the basic reset and interrupt mechanisms along with the various sources of reset and interrupts in most HCS08 derivatives. Some interrupt sources from peripheral modules are discussed in greater detail within other sections of this reference manual. This section gathers information about all reset and interrupt sources in one place for easy reference. A few reset and interrupt sources, including the computer operating properly (COP) watchdog and periodic interrupt timer, are not part of on-chip peripheral systems that have their own sections. These functions and their registers are described in this section. For more information about the reset and interrupt sources for a specific derivative, refer to the appropriate data sheet.

5.2 Reset and Interrupt Features for MC9S08GB60

The set of reset and interrupt sources differs for each HCS08 derivative. This section describes the sources for the first HCS08 device (MC9S08GB60). Refer to the data sheet for a specific device for more information.

Reset and interrupt sources include:

- Eight possible sources of reset:
  - Power-on detection (POR)
  - External $\overline{\text{RESET}}$ pin with enable
  - COP watchdog with enable and two timeout choices
  - Illegal address (not applicable on the MC9S08GB60)
  - Illegal opcode detect
  - Clock generator loss-of-lock and loss-of-clocks

  - Low-voltage detect (LVD) with enable
  - Serial command from a background debug host
- Reset status register to indicate cause of most recent reset
- 25 separate interrupt vectors (reduces polling overhead):
  - Software interrupt instruction (SWI)
  - IRQ pin with enable, choice of polarity, level, and/or edge
  - Low-voltage detect with enable
  - Clock generator loss-of-lock or loss-of-clocks
  - Ten timer interrupts; two overflow, eight channels total for two TPMs
  - One SPI interrupt
  - Six SCI interrupts; Rx, Tx, and error for each of two SCIs
  - Keyboard wakeup
  - ATD conversion complete
  - Periodic wakeup from stop with enable and multiple rates based on a separate 1-kHz self-clocked source or an external source

5.3 MCU Reset

Reset provides a way to start processing from a known set of initial conditions. During reset, most control and status registers are forced to initial values and the program counter is loaded from the reset vector ($FFFE:$FFFF). On-chip peripheral modules are disabled and I/O pins are initially configured as general-purpose high-impedance inputs with pullup devices disabled. The I bit in the condition code register (CCR) is set to block maskable interrupts until the user program has a chance to initialize the stack pointer (SP) and system control settings. SP is forced to $00FF at reset, but this is almost never where the stack should be located in an HCS08 system. Normally, SP should be changed during reset initialization.

The MCU defaults to using the self-clocked mode (approximately 4 MHz bus clock) so it doesn't need to wait for the external oscillator to start and stabilize. In most systems, the user's initialization program will configure the clock module to operate at the system's optimal frequency.

5.4 Computer Operating Properly (COP) Watchdog

The COP watchdog is intended to force a system reset when the application software fails to execute as expected. To prevent a system reset from the COP timer (when it is enabled), application software must reset the COP timer periodically. If the application program gets lost and fails to reset the COP before it times out, a system reset is generated to force the system back to a known starting point. The COP watchdog is enabled and controlled by the SOPT register (see **5.8.4 System Options Register (SOPT)** for additional information). The COP timer is reset by writing any value to the address of the reset status register (SRS). This write does not affect the data in the read-only SRS register. Instead, the act of writing to this address is decoded and sends a reset signal to the COP timer.

After any reset, the COP timer is enabled, because depending on any application program instructions to enable the watchdog that is supposed to detect software errors is not reliable. If the COP watchdog is not used in an application, it can be disabled by clearing the COPE bit in the write-once SOPT register. Also, the COPT bit can be used to choose one of two timeout periods (2^{18} or 2^{13} cycles of the bus rate clock). Even if the application will use the reset default settings in COPE and COPT, you should still write to the write-once SOPT register during reset initialization to lock in the settings so they cannot be changed accidentally if the application program gets lost.

The write to SRS that services (clears) the COP timer should not be placed in an interrupt service routine (ISR) because the ISR could continue to be executed periodically even if the main application program fails.

5.5 Interrupts

Interrupts provide a way to save the current CPU status and registers, execute an interrupt service routine (ISR), and then restore the CPU status so that processing resumes where it left off before the interrupt. Other than software interrupt (SWI), which is a program instruction, interrupts are caused by hardware events such as an edge on the IRQ pin or the reception of a serial I/O character. The debug module can also generate SWI interrupts under certain circumstances (see **7.5.9 Hardware Breakpoints and ROM Patching**).

If an event occurs in an enabled interrupt source, an associated read-only status flag will become set, but the CPU will not respond until and unless the local interrupt mask is a logic 1 to enable the interrupt and the I bit in the condition code register (CCR) is logic 0 to allow interrupts. The global interrupt mask (I bit) in the CCR is initially set after reset which masks (prevents) all maskable interrupt sources. This allows the user program to initialize the stack pointer and perform other system setup before clearing the I bit to allow the CPU to respond to interrupts.

When the CPU receives a qualified interrupt request, it completes the current instruction before responding to the interrupt. The interrupt sequence follows the same cycle-by-cycle sequence as the SWI instruction and consists of:

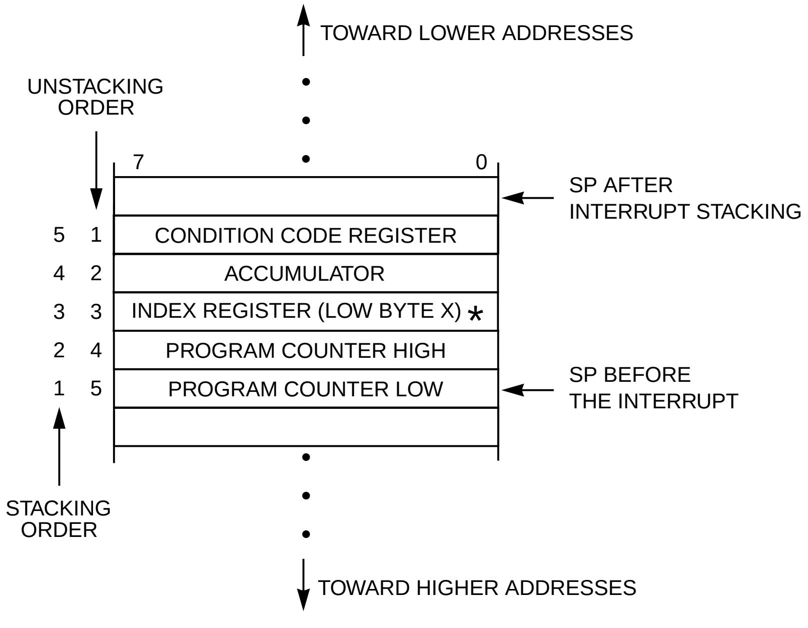
- Saving the CPU registers on the stack
- Setting the I bit in the CCR to mask further interrupts
- Fetching the interrupt vector for the highest priority interrupt that is currently pending
- Filling the instruction queue with the first three bytes of program information starting from the address fetched from the interrupt vector locations

While the CPU is responding to the interrupt, the I bit is automatically set to avoid the possibility of another interrupt interrupting the ISR itself (this is called nesting of interrupts). Normally, the I bit is restored to 0 when the CCR is restored from the value that was stacked on entry to the ISR. In rare cases, the I bit may be cleared inside an ISR (after clearing the status flag that generated the interrupt) so that other interrupts can be serviced without waiting for the first service routine to finish. This practice is not recommended for anyone other than the most experienced programmers because it can lead to subtle program errors that are difficult to debug.

The interrupt service routine ends with a return-from-interrupt (RTI) instruction which restores the CCR, A, X, and PC registers to their pre-interrupt values by reading the previously saved information off the stack. For compatibility with the M68HC08, the H register is not automatically saved and restored. So it is good programming practice to push H onto the stack at the start of the interrupt service routine (ISR) and restore it just before the RTI that is used to return from the ISR.

5.5.1 Interrupt Stack Frame

Figure 5-1 shows the contents and organization of a stack frame. Before the interrupt, the stack pointer (SP) points at the next available byte location on the stack. The current values of CPU registers are stored on the stack starting with the low-order byte of the program counter (PCL) and ending with the condition code register (CCR). After stacking, the SP points at the next available location on the stack which is the address that is one less than the address where the CCR was saved. The PC value that is stacked is the address of the instruction in the main program that would have executed next if the interrupt had not occurred.



* High byte (H) of index register is not stacked.

Figure 5-1. Interrupt Stack Frame

When an RTI instruction is executed, these values are recovered from the stack in reverse order. As part of the RTI sequence, the CPU fills the instruction pipeline by reading three bytes of program information, starting from the PC address that was just recovered from the stack.

The status flag that caused the interrupt must be acknowledged (cleared) before returning from the ISR. Typically, the flag should be cleared at the beginning of the ISR so that if another interrupt is generated by this same source, it will be registered so it can be serviced after completion of the current ISR.

5.5.2 External Interrupt Request (IRQ) Pin

External interrupts are managed by the IRQ status and control register (IRQSC). When the IRQ function is enabled, synchronous logic monitors the pin for edge-only or edge-and-level events. When the MCU is in stop mode and system clocks are shut down, an asynchronous path is used so the IRQ (if enabled) can wake the MCU from stop.

5.5.2.1 Pin Configuration Options

The IRQ pin enable (IRQPE) control bit in the IRQSC register must be 1 in order for the IRQ pin to act as the interrupt request (IRQ) input. As an IRQ input, the user can choose the polarity of edges or levels detected (IRQEDG), whether the pin detects edges-only or edges and levels (IRQMOD), and whether an event causes an interrupt or just sets the IRQF flag which can be polled by software.

When the IRQ pin is configured to detect rising edges, an optional pulldown resistor is available rather than a pullup resistor. BIH and BIL instructions may be used to detect the level on the IRQ pin when the pin is configured to act as the IRQ input.

NOTE: *The voltage measured on the pulled up IRQ pin may be as low as V_{DD} – 0.7 V. The internal gates connected to this pin are pulled all the way to V_{DD}. All other pins with enabled pullup resistors will have an unloaded measurement of V_{DD}.*

5.5.2.2 Edge and Level Sensitivity

Synchronous logic is used to detect edges. Prior to detecting an edge, the IRQ pin must be at its deasserted logic level. A falling edge is detected when the enabled IRQ input signal is seen at logic 1 during one bus cycle and then at logic 0 during the next cycle. A rising edge is detected when the input signal is seen as a logic 0 during one bus cycle and then a logic 1 during the next cycle.

The IRQMOD control bit can be set to reconfigure the detection logic so that it detects edges and levels. In this mode, the IRQF status flag becomes set when an edge is detected (when the IRQ pin changes from the deasserted to the asserted level), but the flag is continuously set (and cannot be cleared) as long as the IRQ pin remains at the asserted level.

5.5.3 Interrupt Vectors, Sources, and Local Masks

Table 5-1 provides a summary of all interrupt sources in the MC9S08GB60. Higher-priority sources are located toward the bottom of the table. The high-order byte of the address for the interrupt service routine is located at the first address in the vector address column, and the low-order byte of the address for the interrupt service routine is located at the next higher address. The vector name is the label used in the equate or header file provided by Motorola.

When an interrupt condition occurs, an associated flag bit becomes set. If the associated local interrupt mask is 1, an interrupt request is sent to the CPU. Within the CPU, if the global interrupt mask (I bit in the CCR) is 0, the CPU will finish the current instruction, stack the PCL, PCH, X, A, and CCR CPU registers, set the I bit, and then fetch the interrupt vector for the highest priority pending interrupt. Processing then continues in the interrupt service routine.

Table 5-1. Interrupt Summary (MC9S08GB60)

Vector Priority	Address (High/Low)	Vector Name	Module	Source	Enable	Description
Lower	$FFC0/FFC1 through $FFCA/FFCB	Unused Vector Space (available for user program)				
	$FFCC/FFCD	Vrti	System control	RTIF	RTIE	Real-time interrupt
	$FFCE/FFCF	Viic	IIC	IICIS	IICIE	IIC control
	$FFD0/FFD1	Vatd	ATD	COCO	AIEN	AD conversion complete
	$FFD2/FFD3	Vkeyboard	KBI	KBF	KBIE	Keyboard pins
	$FFD4/FFD5	Vsci2tx	SCI2	TDRE TC	TIE TCIE	SCI2 transmit
	$FFD6/FFD7	Vsci2rx	SCI2	IDLE RDRF	ILIE RIE	SCI2 receive
	$FFD8/FFD9	Vsci2err	SCI2	OR NF FE PF	ORIE NFIE FEIE PFIE	SCI2 error
	$FFDA/FFDB	Vsci1tx	SCI1	TDRE TC	TIE TCIE	SCI1 transmit
	$FFDC/FFDD	Vsci1rx	SCI1	IDLE RDRF	ILIE RIE	SCI1 receive
	$FFDE/FFDF	Vsci1err	SCI1	OR NF FE PF	ORIE NFIE FEIE PFIE	SCI1 error
	$FFE0/FFE1	Vspi	SPI	SPIF MODF SPTEF	SPIE SPIE SPTIE	SPI
	$FFE2/FFE3	Vtpm2ovf	TPM2	TOF	TOIE	TPM2 overflow
	$FFE4/FFE5	Vtpm2ch4	TPM2	CH4F	CH4IE	TPM2 channel 4
	$FFE6/FFE7	Vtpm2ch3	TPM2	CH3F	CH3IE	TPM2 channel 3
	$FFE8/FFE9	Vtpm2ch2	TPM2	CH2F	CH2IE	TPM2 channel 2
	$FFEA/FFEB	Vtpm2ch1	TPM2	CH1F	CH1IE	TPM2 channel 1
	$FFEC/FFED	Vtpm2ch0	TPM2	CH0F	CH0IE	TPM2 channel 0
	$FFEE/FFEF	Vtpm1ovf	TPM1	TOF	TOIE	TPM1 overflow
	$FFF0/FFF1	Vtpm1ch2	TPM1	CH2F	CH2IE	TPM1 channel 2
	$FFF2/FFF3	Vtpm1ch1	TPM1	CH1F	CH1IE	TPM1 channel 1
	$FFF4/FFF5	Vtpm1ch0	TPM1	CH0F	CH0IE	TPM1 channel 0
	$FFF6/FFF7	Vicg	ICG	ICGIF (LOLS/LOCS)	LOLRE/LOCRE	ICG
	$FFF8/FFF9	Vlvd	System control	LVDF	LVDIE	Low-voltage detect
	$FFFA/FFFB	Virq	IRQ	IRQF	IRQIE	IRQ pin
	$FFFC/FFFD	Vswi	Core	SWI Instruction	—	Software interrupt
Higher	$FFFE/FFFF	Vreset	Systemcontrol	COP LVD $\overline{\text{RESET}}$ pin Illegal opcode	COPE LVDRE — —	Watchdog timer Low-voltage detect External pin Illegal opcode

5.6 Low-Voltage Detect (LVD) System

The 9S08GB/GT includes a system to protect against low voltage conditions in order to protect memory contents and control MCU system states during supply voltage variations. The system is comprised of a power-on reset (POR) circuit and an LVD circuit with a user selectable trip voltage, either high (V_{LVDH}) or low (V_{LVDL}). The LVD circuit is enabled when LVDE in SPMSC1 is high and the trip voltage is selected by LVDV in SPMSC2. The LVD is disabled upon entering any of the stop modes unless the LVDSE bit is set. If LVDSE and LVDE are both set, then the MCU cannot enter stop1 or stop2, and the current consumption in stop3 with the LVD enabled will be greater.

5.6.1 Power-On Reset Operation

When power is initially applied to the MCU, or when the supply voltage drops below the V_{POR} level, the POR circuit will cause a reset condition. As the supply voltage rises, the LVD circuit will hold the chip in reset until the supply has risen above the V_{LVDH} level. Both the POR bit and the LVD bit in SRS are set following a POR.

5.6.2 LVD Reset Operation

The LVD can be configured to generate a reset upon detection of a low voltage condition by setting LVDRE to 1. Once an LVD reset has occurred, the LVD system will hold the MCU in reset until the supply voltage has risen above the level determined by LVDV. LVDV is not altered when an LVD reset occurs. The LVD bit in the SRS register is set following either an LVD reset or POR.

5.6.3 LVD Interrupt Operation

When a low voltage condition is detected and the LVD circuit is configured for interrupt operation (LVDE set, LVDIE set, and LVDRE clear), then LVDF will be set and an LVD interrupt will occur.

5.6.4 Low-Voltage Warning (LVW)

The LVD system has a low voltage warning flag to indicate to the user that the supply voltage is approaching, but is still above, the LVD voltage. The LVW does not have an interrupt associated with it. There are two user selectable trip voltages for the LVW, one high (V_{LVWH}) and one low (V_{LVWL}). The trip voltage is selected by LVWV in SPMSC2.

5.7 Real-Time Interrupt (RTI)

The real-time interrupt function can be used to generate periodic interrupts based on a divide of the external oscillator during run mode. It can also be used to wake the MCU from stop2 using the internal 1-kHz reference, or from stop3 using either the internal reference or the external oscillator if it is enabled in stop modes. The RTICLKS bit in the system real-time interrupt status and control register (SRTISC) is used to select between these two modes of operation.

The SRTISC register includes a read-only status flag, a write-only acknowledge bit, and a 3-bit control value (RTIS2:RTIS1:RTIS0) used to disable the clock source to the real-time interrupt or select one of seven wakeup delays between 8 ms and 1.024 seconds. The 1-kHz clock source and therefore the periodic rates have a tolerance of about 30 percent. The RTI has a local interrupt enable, RTIE, to allow masking of the real-time interrupt. It can be disabled by writing 0:0:0 to RTIS2:RTIS1:RTIS0 so the clock source is disabled and no interrupts will be generated. See **5.8.6 System Real-Time Interrupt Status and Control Register (SRTISC)** for detailed information about this register.

5.8 Reset, Interrupt, and System Control Registers and Control Bits

One 8-bit register in the direct page register space and five 8-bit registers in the high-page register space are related to reset and interrupt systems.

Refer to the direct-page register summary in **Section 4. On-Chip Memory** of this reference manual for the absolute address assignments for all registers. This section refers to registers and control bits only by their names. An equate or header file provided by Motorola is used to translate these names into the appropriate absolute addresses.

Some control bits in the SOPT and SPMSC2 registers are related to modes of operation. Although brief descriptions of these bits are provided here, the related functions are discussed in greater detail in **Section 3. Modes of Operation**.

This section describes register and bit details for the MC9S08GB60. Although these descriptions are representative of HCS08 devices, you should always refer to the data sheet for details about a specific HCS08 device.

5.8.1 Interrupt Request Status and Control Register (IRQSC)

This direct page register includes two unimplemented bits which always read 0, four read/write bits, one read-only status bit, and one write-only bit. These bits are used to configure the IRQ function, report status, and acknowledge IRQ events.

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	0	0	IRQEDG	IRQPE	IRQF	0	IRQIE	IRQMOD
Write:	▨	▨			▨	IRQACK		
Reset:	0	0	0	0	0	0	0	0

▨ = Unimplemented or Reserved

Figure 5-2. Interrupt Request Status and Control Register (IRQSC)

IRQEDG — Interrupt Request (IRQ) Edge Select

This read/write control bit is used to select the polarity of edges or levels on the IRQ pin that cause IRQF to be set. The IRQMOD control bit determines whether the IRQ pin is sensitive to both edges and levels or just edges. When the IRQ pin is enabled as the IRQ input and is configured to detect rising edges, the optional pullup resistor is reconfigured as an optional pulldown resistor.

1 = IRQ is rising edge or rising edge/high-level sensitive.
0 = IRQ is falling edge or falling edge/low-level sensitive.

IRQPE — IRQ Pin Enable

This read/write control bit enables the IRQ pin function. When this bit is set the IRQ pin can be used as an interrupt request. Also, when this bit is set, either an internal pull-up or an internal pull-down resistor is enabled depending on the state of the IRQMOD bit.

1 = IRQ pin function is enabled.
0 = IRQ pin function is disabled.

IRQF — IRQ Flag

This read-only status bit indicates when an interrupt request event has occurred.

1 = IRQ event detected.
0 = No IRQ request.

IRQACK — IRQ Acknowledge

This write-only bit is used to acknowledge interrupt request events (write 1 to clear IRQF). Writing 0 has no meaning or effect. Reads always return logic 0. If edge-and-level detection is selected (IRQMOD = 1), IRQF cannot be cleared while the IRQ pin remains at its asserted level.

IRQIE — IRQ Interrupt Enable

This read/write control bit determines whether IRQ events generate a hardware interrupt request.

1 = Hardware interrupt requested whenever IRQF = 1.
0 = Hardware interrupt requests from IRQF disabled (use polling).

IRQMOD — IRQ Detection Mode

This read/write control bit selects either edge-only detection or edge-and-level detection. The IRQEDG control bit determines the polarity of edges and levels that are detected as interrupt request events. See **5.5.2.2 Edge and Level Sensitivity** for more details.

1 = IRQ event on falling edges and low levels or on rising edges and high levels.
0 = IRQ event on falling edges or rising edges only.

5.8.2 System Reset Status Register (SRS)

This register includes seven read-only status flags to indicate the source of the most recent reset. When a debug host forces reset by writing 1 to BDFR in the FBDFR register, none of the status bits in SRS will be set. Writing any value to this register address clears the COP watchdog timer without affecting the contents of this register. The reset state of these bits depends on what caused the MCU to reset.

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	POR	PIN	COP	ILOP	0	ICG	LVD	0
Write:	Writing any value to SRS address clears COP watchdog timer.							
Power-on reset:	1	0	0	0	0	0	1	0
Low-voltage reset:	0	0	0	0	0	0	1	0
Any other reset:	0	[1]	[1]	[1]	0	[1]	0	0

[1] Any of these reset sources that are active at the time of reset will cause the corresponding bit(s) to be set; bits corresponding to sources that are not active at the time of reset will be cleared.

Figure 5-3. System Reset Status (SRS)

POR — Power-On Reset

Reset was caused by the power-on detection logic. Since the internal supply voltage was ramping up at the time, the low-voltage reset (LVR) status bit is also set to indicate that the reset occurred while the internal supply was below the LVR threshold.
1 = POR caused reset
0 = Reset not caused by POR

PIN — External Reset Pin

Reset was caused by an active-low level on the external reset pin.
1 = Reset came from external reset pin.
0 = Reset not caused by external reset pin

COP — Computer Operating Properly (COP) Watchdog

Reset was caused by the COP watchdog timer timing out. This reset source may be blocked by COPE = 0.
1 = Reset caused by COP timeout
0 = Reset not caused by COP timeout

ILOP — Illegal Opcode

Reset was caused by an attempt to execute an unimplemented or illegal opcode. The STOP instruction is considered illegal if stop is disabled by STOPE = 0 in the SOPT register. The BGND instruction is considered illegal if active background mode is disabled by ENBDM = 0 in the BDCSC register.
1 = Reset caused by an illegal opcode
0 = Reset not caused by an illegal opcode

ICG — Internal Clock Generation Module Reset

Reset was caused by an ICG module reset.
1 = Reset caused by ICG module.
0 = Reset not caused by ICG module.

LVD — Low Voltage Detect

If the LVDRE and LVDSE bits are set and the supply drops below the LVD trip voltage, an LVD reset will occur. This bit is also set by POR.
1 = Reset caused by LVD trip or POR.
0 = Reset not caused by LVD trip or POR.

5.8.3 System Background Debug Force Reset Register (SBDFR)

This register contains a single write-only control bit. A serial background command such as WRITE_BYTE must be used to write to SBDFR. Attempts to write this register from a user program are ignored. Reads always return $00.

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	0	0	0	0	0	0	0	0
Write:	▨	▨	▨	▨	▨	▨	▨	BDFR[1]
Reset:	0	0	0	1	0	0	0	0

▨ = Unimplemented or Reserved

[1] BDFR is writable only through serial background debug commands, not from user programs.

Figure 5-4. System Integration Module Control Register (SBDFR)

BDFR — Background Debug Force Reset

A serial background command such as WRITE_BYTE may be used to allow an external debug host to force a target system reset. Writing logic 1 to this bit forces an MCU reset. This bit cannot be written from a user program.

5.8.4 System Options Register (SOPT)

This register may be read at any time. Bits 3, 2, and 0 are unimplemented and always read 0. This is a write-once register so only the first write after reset is honored. Any subsequent attempt to write to SOPT (intentionally or unintentionally) is ignored to avoid accidental changes to these sensitive settings. SOPT should be written during the user's reset initialization program to set the desired controls even if the desired settings are the same as the reset settings.

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	COPE	COPT	STOPE	▨	0	0	BKGDPE	▨
Write:					▨	▨		
Reset:	1	1	0	1	0	0	1	1

▨ = Unimplemented or Reserved

Figure 5-5. System Options Register (SOPT)

COPE — COP Watchdog Enable

This write-once bit defaults to 1 after reset.
1 = COP watchdog timer enabled (force reset on timeout)
0 = COP watchdog timer disabled

COPT — COP Watchdog Timeout

This write-once bit defaults to 1 after reset.
1 = Long timeout period selected (2^{18} cycles of BUSCLK)
0 = Short timeout period selected (2^{13} cycles of BUSCLK)

STOPE — Stop Mode Enable

This write-once bit defaults to 0 after reset, which disables stop mode. If stop mode is disabled and a user program attempts to execute a STOP instruction, an illegal opcode reset is forced.
1 = Stop mode enabled
0 = Stop mode disabled

BKGDPE — Background Debug Mode Pin Enable

The BKGDPE bit enables the PTD0/BKGD/MS pin to function as BKGD/MS. When the bit is clear, the pin will function as PTD0, which is an output only general purpose I/O. This pin always defaults to BKGD/MS function after any reset.
1 = BKGD pin enabled.
0 = BKGD pin disabled.

5.8.5 System Device Identification Register (SDIDH, SDIDL)

This read-only register is included so host development systems can identify the HCS08 derivative and revision number. This allows the development software to recognize where specific memory blocks, registers, and control bits are located in a target MCU.

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	REV3	REV2	REV1	REV0	ID11	ID10	ID9	ID8
Reset:	0[1]	0[1]	0[1]	0[1]	0	0	0	0
Read:	ID7	ID6	ID5	ID4	ID3	ID2	ID1	ID0
Reset:	0	0	0	0	0	0	1	0

[1] The revision number that is hard coded into these bits reflects the current silicon revision level.

Figure 5-6. System Device Identification Register (SDIDH, SDIDL)

REV[3:0] — Revision Number

The high-order 4 bits of address $1806 are hard coded to reflect the current mask set revision number (0–F).

ID[11:0] — Part Identification Number

Each derivative in the HCS08 Family has a unique identification number. The 9S08GB/GT is hard coded to the value $003.

5.8.6 System Real-Time Interrupt Status and Control Register (SRTISC)

This register contains one read-only status flag, one write-only acknowledge bit, three read/write delay selects, and three unimplemented bits, which always read 0.

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	RTIF	0	RTICLKS	RTIE	0	RTIS2	RTIS1	RTIS0
Write:	▨	RTIACK			▨			
Reset:	0	0	0	0	0	0	0	0

▨ = Unimplemented or Reserved

Figure 5-7. System RTI Status and Control Register (SRTISC)

RTIF — Real-Time Interrupt Flag

This read-only status bit indicates the periodic wakeup timer has timed out.

1 = Periodic wakeup timer timed out.
0 = Periodic wakeup timer not timed out.

RTIACK — Real-Time Interrupt Acknowledge

This write-only bit is used to acknowledge real-time interrupt request (write 1 to clear RTIF). Writing 0 has no meaning or effect. Reads always return logic 0.

RTICLKS — Real-Time Interrupt Clock Select

This read/write bit selects the clock source for the real-time interrupt.

1 = Real-time interrupt request clock source is external clock.
0 = Real-time interrupt request clock source is internal 1-kHz oscillator.

RTIE — Real-Time Interrupt Enable

This read-write bit enables real-time interrupts.

1 = Real-time interrupts enabled.
0 = Real-time interrupts disabled.

RTIS2:RTIS1:RTIS0 — Real-Time Interrupt Delay Selects

These read/write bits select the wakeup delay for the RTI. The clock source for the real-time interrupt is a self-clocked source which oscillates at about 1 kHz, is independent of other MCU clock sources. Using external clock source the delays will be crystal frequency divided by value in RTIS2:RTIS1:RTIS0.

Table 5-2. Real-Time Interrupt Frequency

RTIS2:RTIS1:RTIS0	1-kHz Clock Source Delay[1]	Using External Clock Source Delay (crystal frequency)
0:0:0	Disable periodic wakeup timer	Disable periodic wakeup timer
0:0:1	8 ms	divide by 256
0:1:0	32 ms	divide by 1024
0:1:1	64 ms	divide by 2048
1:0:0	128 ms	divide by 4096
1:0:1	256 ms	divide by 8192
1:1:0	512 ms	divide by 16384
1:1:1	1.024 s	divide by 32768

1. Normal values are shown in this column based on f_{RTI} = 1 kHz. See the appropriate data sheet f_{RTI} for the tolerance on these values.

5.8.7 System Power Management Status and Control 1 Register (SPMSC1)

This register is used to control actions associated with low V_{DD} detection circuitry. If low-voltage detection is enabled, by setting LVDE =1, bits 5-3 control the action associated with the low voltage detection. LVDF is a flag, used to alert the occurrence of low voltage. LVDAC is used to acknowledge and clear LVDF.

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	LVDF	0	LVDIE	LVDRE[1]	LVDSE[1]	LVDE[1]	0	0
Write:	▨	LVDACK					▨	▨
Reset:	0	0	0	1	1	1	0	0

▨ = Unimplemented or Reserved

1. This bit can be written only one time after reset. Additional writes are ignored.

Figure 5-8. System Power Management Status and Control 1 Register (SPMSC1)

LVDF — Low-Voltage Detect Flag

Provided LVDE = 1, this read-only status bit indicates a low-voltage detect event.

LVDACK — Low-Voltage Detect Acknowledge

This write-only bit is used to acknowledge low voltage detection errors (write 1 to clear LVDF). Reads always return logic 0.

LVDIE — Low-Voltage Detect Interrupt Enable

This read/write bit enables hardware interrupt requests for LVDF.
1 = Request a hardware interrupt when LVDF = 1.
0 = Hardware interrupt disabled (use polling).

LVDRE — Low-Voltage Detect Reset Enable

This read/write bit enables LVDF errors to generate a hardware reset (provided LVDE = 1).
1 = Force an MCU reset when LVDF = 1.
0 = LVDF does not generate hardware resets.

LVDSE — Low-Voltage Detect Stop Enable

Provided LVDE = 1, this read/write bit determines whether the low-voltage detect function operates when the MCU is in stop mode.
1 = Low-voltage detect enabled during stop mode.
0 = Low-voltage detect disabled during stop mode.

LVDE — Low-Voltage Detect Enable

This read/write bit enables low-voltage detect logic and qualifies the operation of other bits in this register.
1 = LVD logic enabled.
0 = LVD logic disabled.

5.8.8 System Power Management Status and Control 2 Register (SPMSC2)

This register is used to report the status of the low voltage warning function, and to configure the stop mode behavior of the MCU.

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	LVWF	0	LVDV	LVWV	PPDF	0	PDC	PPDC
Write:	▨	LVWACK			▨	PPDACK		
Power-on reset:	0[1]	0	0	0	0	0	0	0
LVD reset:	0[1]	0	U	U	0	0	0	0
Any other reset:	0[1]	0	U	U	0	0	0	0

▨ = Unimplemented or Reserved   U = Unaffected by reset

1. LVWF will be set not just in the case when V_{Supply} transitions below the trip point but also after reset and V_{Supply} is already below V_{LVW}.

Figure 5-9. System Power Management Status and Control 2 Register (SPMSC2)

LVWF — Low-Voltage Warning Flag

The LVWF bit indicates the low voltage warning status.
1 = Low voltage warning is present or was present.
0 = Low voltage warning **not** present.

LVWACK — Low-Voltage Warning Acknowledge

The LVWF bit indicates the low voltage warning status.

Writing a logic 1 to LVWACK clears LVWF to a logic 0 if a low voltage warning is not present.

LVDV — Low-Voltage Detect Voltage Select

The LVDV bit selects the LVD trip point voltage (V_{LVD}).
1 = High trip point selected ($V_{LVD} = V_{LVDH}$).
0 = Low trip point selected ($V_{LVD} = V_{LVDL}$).

LVWV — Low-Voltage Warning Voltage Select

The LVWV bit selects the LVW trip point voltage (V_{LVW}).
1 = High trip point selected ($V_{LVW} = V_{LVWH}$).
0 = Low trip point selected ($V_{LVW} = V_{LVWL}$).

PPDF — Partial Power Down Flag

The PPDF bit indicates that the MCU has exited the stop2 mode.
1 = Stop2 mode recovery.
0 = Not stop2 mode recovery.

PPDACK — Partial Power Down Acknowledge

Writing a logic 1 to PPDACK clears the PPDF bit.

PDC — Power Down Control

The write-once PDC bit controls entry into the power down (stop2 and stop1) modes.
1 = Power down modes are enabled.
0 = Power down modes are disabled.

PPDC — Partial Power Down Control

The write-once PPDC bit controls which power down mode, stop1 or stop2, is selected.
1 = Stop2, partial power down, mode enabled if PDC set.
0 = Stop1, full power down, mode enabled if PDC set.

Section 6. Central Processor Unit (CPU)

6.1 Introduction

The HCS08 CPU is the latest generation in a series of a CPU family that started in 1979 with the NMOS (N-channel metal-oxide semiconductor) M6805 Family. Next Motorola developed the M146805 Family using metal gate CMOS (complementary MOS). Eventually, this process was replaced by silicon gate CMOS and Motorola developed the M68HC05 CPU. The next major step in this series was the M68HC08 which significantly expanded the instruction set to allow more efficient C compilers. The current HCS08 CPU has been developed using a new process-independent design methodology, allowing it to keep pace with rapid developments in silicon processing technology.

Compared with the M68HC08 CPU, the HCS08 CPU added:

- New addressing modes for LDHX instruction:
  - Extended addressing mode (EXT)
  - Indexed — no offset (IX)
  - Indexed — 8-bit offset (IX1)
  - Indexed — 16-bit offset (IX2)
  - Stack pointer relative — 8-bit offset (SP1)
- New addressing modes for STHX and CPHX instructions:
  - Extended addressing mode (EXT)
  - Stack pointer relative — 8-bit offset (SP1)
- New background (BGND) instruction
- Operating bus frequency increased to 20 MHz on first derivatives
- Instruction queue (or pipeline) to improve instruction throughput

The new addressing modes for instructions involving the 16-bit H:X register pair improve the efficiency of C compilers. The BGND instruction is used only in debug situations to implement software breakpoints.

The instruction queue improves instruction throughput because it makes the opcode and one byte of operand information available to the CPU immediately at the start of an instruction. Without the queue, the CPU would have to spend the first few cycles of an instruction waiting for the program information to be fetched into the CPU. On any change of flow — such as branch, jump, or interrupt — the CPU performs three program fetches to fill this instruction queue. The instruction queue caused some changes in the cycle counts and the order of operations within instructions compared to the M68HC08 CPU, but the benefits from being able to start instructions sooner more than offset the costs for filling the queue on changes of flow.

6.2 Programmer's Model and CPU Registers

Figure 6-1 shows the programmer's model for the HCS08 CPU. These registers are not located in the memory map of the microcontroller. They are built directly inside the CPU logic.

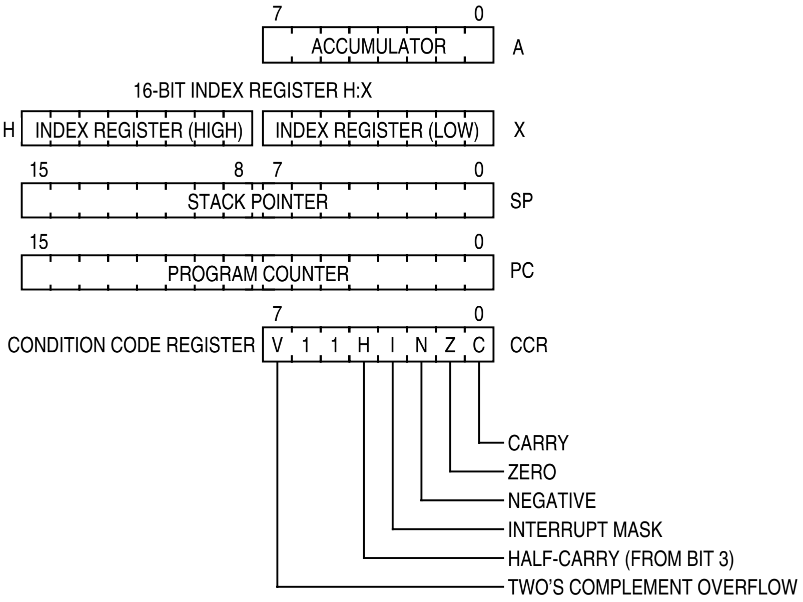


Figure 6-1. CPU Registers

6.2.1 Accumulator (A)

This general-purpose 8-bit register is the primary data register for the HCS08. Data can be read into A from memory with a load accumulator (LDA) instruction or from the stack with a pull (PULA) instruction. The data in A can be written into memory with a store accumulator (STA) or onto the stack with a push (PSHA). Various addressing mode variations allow a great deal of flexibility in specifying the memory location involved in a load or store instruction. Transfer instructions allow values to be transferred from A to X (TAX), from X to A (TXA), from A to the CCR (TAP), or from the CCR to A (TPA). The P in TAP and TPA stands for processor status. The nibble-swap A (NSA) instruction exchanges the high-order four bits of A with the low-order four bits.

You can also perform mathematical, shift, and logical operations on the value in A as in ADD, SUB, ASLA, RORA, INCA, DECA, AND, ORA, EOR, etc. In some of these instructions, such as INCA or ASLA, the value in A is the only input operand and the result replaces the value in A. In other cases, such as ADD or AND, there are two operands: the value in A and a second value from memory. The result of the arithmetic or logical operation replaces the value in A.

Multiply and divide instructions use A as an operand and also store part of the result in A. MUL does an unsigned multiply of A times X and stores the 16-bit result in X:A. DIV does an unsigned 16-bit by 8-bit divide of H:A by X and stores the result in A and the remainder in H.

The decimal adjust A (DAA) instruction is used, after an ADD or ADC instruction involving two BCD numbers, to correct the value in A to a valid 2-digit BCD number with a proper carry indication. For a more detailed discussion of this instruction, refer to **6.5.2.4 BCD Arithmetic**.

It should be apparent that the accumulator is a very busy register, so it would be helpful if some operations could avoid using A. For instance, memory-to-memory move instructions (MOV) are helpful. DBNZ also helps because it allows a loop counter to be implemented in a memory variable rather than the accumulator. The X register can also be used as a second general-purpose 8-bit data register in many cases. Some arithmetic operations such as clear, increment, decrement, complement, negate, and shift can also be used with the X register.

6.2.2 Index Register (H:X)

This 16-bit index register is actually two separate 8-bit registers (H and X). The indexed addressing modes use H:X as a 16-bit base reference pointer and variations of indexed addressing allow an instruction-supplied 16-bit offset, 8-bit offset, or no offset. Other variations of indexed addressing automatically increment the 16-bit index register after the index is used to access a memory operand. Refer to **6.3.6 Indexed Addressing Mode** for a more detailed discussion of the indexed addressing mode.

The 8-bit X register (low-order half of H:X) can also be used as a general purpose data register. The read-modify-write instructions (ASLX, ASRX, CLRX, COMX, DECX, INCX, LSLX, LSRX, NEGX, ROLX, RORX, and TSTX) allow a subset of the ALU operations that can be performed on the accumulator. Be careful not to try to use these instructions when you really want to affect the full 16-bit H:X index register because these instructions only affect X. Consider the following instructions and sequences to get 16-bit versions of 8-bit operations on X.

```
            ldhx   #$0000        ;16-bit version of CLRX
            aix    #1            ;16-bit version of INCX
            aix    #-1           ;16-bit version of DECX
            cphx   #$0000        ;16-bit version of TSTX
```

Load, store, push, and pull instructions are available for X with the same addressing mode variations as the ones used for A. There are also load and store instructions for the 16-bit H:X register pair; however, not as many different addressing modes are offered. There are push (PSHH) and pull (PULH) instructions for H, and simple 2-instruction sequences can be used to push and pull the full 16-bit index register (H:X).

```
            pshx                 ;push low half of H:X
            pshh                 ;push high half of H:X

            pulh                 ;pull high half of H:X
            pulx                 ;pull low half of H:X
```

Sometimes the stack pointer value needs to be transferred to the H:X register pair so H:X can act as a pointer to information on the stack. The stack pointer always points at the next available location on the stack, but normally the index register should point directly at data. Because of this, the 16-bit value in SP is incremented by one as it is transferred to H:X with a TSX instruction. Because of this adjustment, after a TSX instruction H:X points at the last byte of data that was stacked. A complementary adjustment takes place during a TXS instruction. (The value is decremented by one during TXS.) One way to think about this is that the 16-bit address points at the next available stack location when it is in SP and to the last byte of information that was stacked when it is in H:X.

For compatibility with the earlier M68HC05, interrupts do not save the H register on the stack. It is good practice to include a PSHH instruction as the first instruction in interrupt service routines (to save H) and to include a PULH instruction (to restore H) as the last instruction before the RTI that ends the service routine. You may leave these instructions out if you are absolutely sure H is not altered in your interrupt service routine, but be sure there are no AIX instructions or instructions that use the post-increment variation of indexed addressing because these instructions could cause H to change. Unless you really can't tolerate the extra two bytes of program space, one extra temporary byte on the stack, and five bus cycles of execution time, it is much safer to simply include the PSHH and PULH as a matter of habit.

Multiply and divide instructions use X as an operand, and MUL also stores part of the result in X. MUL does an unsigned multiply of A times X and stores the 16-bit result in X:A. DIV does an unsigned 16-bit by 8-bit divide of H:A by X and stores the result in A and the remainder in H.

6.2.3 Stack Pointer (SP)

This 16-bit address pointer register is used by the CPU to automatically maintain a last-in-first-out (LIFO) stack. When the CPU executes a jump- or branch-to-subroutine (JSR or BSR) instruction, it automatically saves the return address on the stack. When the return-from-subroutine (RTS) instruction at the end of the subroutine is executed, this return address is automatically recovered from the stack so execution resumes where it left off when the subroutine was called. Since SP is a full 16-bit register, the stack may be located anywhere in the memory map, and it may be any size up to the size of available RAM on the chip.

The stack pointer always points at the next available location on the stack. When a value is pushed onto the stack, it is written to the address pointed to by the SP and then SP is automatically decremented to point at the next available location. When a value is pulled from the stack, SP is first incremented to point at the most recent data that was pushed on the stack, and then the data is read from the address now pointed to by SP. Notice that the data pointed to by SP is not changed in the process of pulling it from the stack. If you were to look at memory below where SP is currently pointing, you would see old values that were previously stored on the stack. When new values are pushed onto the stack, they over-write whatever is in those memory locations. If RAM in the area of the stack was cleared during reset initialization, the maximum depth that the stack has grown to can be seen by noticing which memory locations are still clear.

For compatibility with the earlier M68HC05, SP is set to $00FF by reset. This is almost never where the top of the stack in new HCS08 applications should be because the RAM in the area from the end of the input/output (I/O) and control registers to $00FF is more valuable for frequently accessed variables. The memory area from $0000 to $00FF can be accessed using the direct addressing mode which saves program space and executes faster than general accesses to other memory locations.

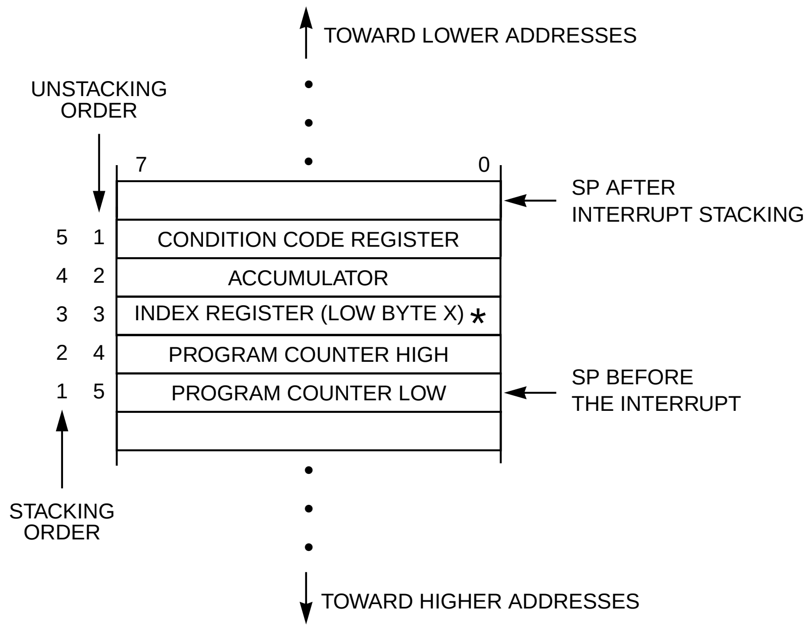
Also for compatibility with the M68HC05, the reset stack pointer (RSP) instruction forces the low-order half of SP to $FF. In the M68HC05, this forced SP to the same value ($00FF) it had after reset. RSP is seldom used in the HCS08 because it doesn't affect the high-order half of SP, and, therefore, it doesn't necessarily restore SP to its reset value.

In new HCS08 programs you would typically initialize SP to point at the highest address in the on-chip RAM. Normally, the following 2-instruction sequence is included within the first few instructions of a reset initialization routine.

```
            ldhx   #RamLast+1    ;point one past RAM
            txs                  ;SP<-(H:X-1)
```

Normally, RamLast is defined in an equate or header file that describes the particular HCS08 device used in your application. RamLast+1 causes H:X to be loaded with the next higher address past the end of RAM because the TXS instruction includes an automatic adjustment (decrement by 1) on the value during the transfer. This adjustment makes SP point at the next available location on the stack. In this case, SP now points at the last (highest address value) location in RAM, and this will be the first location where data will be stacked. The stack will grow toward lower addresses as values are pushed onto the stack.

When an interrupt is requested, the CPU saves the current contents of CPU registers on the stack so, after finishing the interrupt service routine, processing can resume where it left off. Figure 6-2 shows the order that CPU registers are saved on the stack in response to an interrupt. Before the interrupt, SP points to the next available location on the stack. As each value is saved on the stack, the data is stored to the location pointed to by SP and SP automatically is decremented to point at the next available location on the stack. The return-from-interrupt (RTI) instruction that concludes the interrupt service routine restores the CPU registers by pulling them from the stack in the reverse order. Refer to **6.4.2 Interrupts** and **5.5 Interrupts** for more detailed discussions of interrupts.



*High byte (H) of index register is not stacked.

Figure 6-2. Interrupt Stack Frame

For compatibility with the earlier M68HC05 CPU, interrupts do not save the H register on the stack. It is good practice to include a PSHH instruction as the first instruction in your interrupt service routines (to save H) and to include a PULH instruction (to restore H) as the last instruction before the RTI that ends the service routine.

The add immediate value to SP (AIS) instruction may be used to allocate space on the stack for local variables. Although this is most common in C programs, the technique is also useful for assembly language programs. The following code example demonstrates allocation and deallocation of space for local variables on the stack. There is a more detailed discussion of stack techniques in **6.5.6 Stack-Related Instructions**.

```
            ais    #-5          ;allocate 5 bytes for locals
"            "      "            "
            ais    #5           ;deallocate local space
```

SP-relative indexed addressing with 8-bit offset (SP1) or 16-bit offset (SP2) allows many instructions to directly access the information on the stack. This is important for efficient C compilers and the same techniques can be used in assembly language programs.

Push and pull instructions are similar to store and load instructions except they load or store the data relative to the current SP value rather than accessing a specific memory location. The stack must always be “balanced,” meaning that for every operation that places a byte of data on the stack, there must be a corresponding operation that removes a byte of data. For each JSR or BSR, there should be an RTS. For each interrupt or SWI, there should be an RTI. For each push, there should be a pull. If you allocate space for locals with an AIS instruction, you should have a corresponding AIS instruction to deallocate the same amount of space.

Suppose you had a subroutine that included a PSHA instruction, but you forgot to do a corresponding PULA before returning from the subroutine. The return from subroutine (RTS) would not work correctly because SP would not be pointing at the correct return address when RTS was executed.

Another error is a subroutine that calls itself, but doesn’t have a reliable way to limit the number of nesting iterations. This produces a stack that grows beyond the space set aside for the stack. Usually this ends when stack operations start storing things on top of RAM variables or I/O and control registers. This is called stack overflow, and it can also happen when an interrupt service routine clears the I mask inside the service routine which makes nested interrupts possible. Each level of nesting adds at least five more bytes to the stack.

6.2.4 Program Counter (PC)

The program counter is a 16-bit register that contains the address of the next instruction or operand to be fetched.

During normal execution, the program counter automatically increments to the next sequential memory location every time an instruction or operand is fetched. Jump, branch, interrupt, and return operations load the program counter with an address other than that of the next sequential location. This is called a change-of-flow.

During reset, the program counter is loaded with the reset vector which is located at address $FFFE and $FFFF. The vector is the address of the first instruction to be executed after exiting from the reset state.

6.2.5 Condition Code Register

The 8-bit condition code register contains the interrupt mask (I) and five status flags. Bit 6 and bit 5 are permanently set to logic 1. The following paragraphs provide detailed information about the CCR bits and how they are used. **Figure 6-3** identifies the CCR bits and their bit positions.

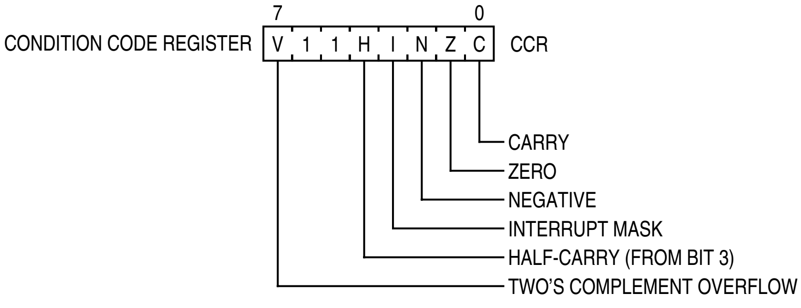


Figure 6-3. Condition Code Register (CCR)

The I bit is an interrupt mask control bit unlike the other bits in the CCR which are processor status bits. The I bit is also the only one of the six implemented bits in the CCR to be initialized by reset. The I bit is forced to 1 at reset so interrupts are blocked until you have initialized the stack pointer. The other five status bits (V, H, N, Z, and C) are unknown after reset and will take on known values only after executing an instruction that affects the bit(s). There is no reason to force these bits to a particular value at reset because it would not make sense to do a conditional branch that used these bits unless you had just executed an instruction that affected them.

The five status bits indicate the results of arithmetic and other instructions. Conditional branch instructions will either branch to a new program location or allow the program to continue to the next instruction after the branch, depending on the values in the CCR status bits. Simple conditional branch instructions (BCC, BCS, BNE, BEQ, BHCC, BHCS, BMC, BMS, BPL, and BMI) cause a branch depending on the state of a single CCR bit. Other branch instructions are controlled by a more complex combination of two or three of the CCR bits. For example branch if less than or equal (BLE) branches if the Boolean expression [(Z) | (N⊕V)] is true. The V bit (which was not present in the older M68HC05 instruction set) allows signed branches because V is the two's complement overflow indicator. Separate unsigned branch instructions are based on the C bit which is effectively an overflow indicator for unsigned operations.

Often, the conditional branch immediately follows the instruction that caused the CCR bit(s) to be updated as in this sequence:

```
             cmp    #5            ;compare accumulator A to 5
             blt    less          ;branch if A<5
more:        deca                 ;do this if A not < 5
less:
```

Other instructions may be executed between the test and the conditional branch as long as only instructions that do not disturb the CCR bits that affect the conditional branch are used. A common example is when a test is performed in a subroutine or function and the conditional branch is not executed until the subroutine has returned to the main program. This is a form of parameter passing (that is, information is returned to the calling program in the condition code bits). Consider the following example which checks a character, received through the SCI, to see if it is the ASCII code for a valid hexadecimal digit 0–9, a–f, or A–F.

```
 "            "      "             "
             lda    SCI1D         ;read character from SCI
             jsr    upcase        ;strip MSB & make upper case
             jsr    ishex         ;see if valid hex digit
             bne    errorHex      ;branch if char wasn't hex
goodHex:     nop                  ;here if it was good hex digit
errorHex:                         ;here if it wasn't
 "            "      "             "
```

```
*********************
* ishex - check character for valid hexadecimal (0-9 or A-F)
*  on entry A contains an unknown upper-case character
*  returns with original character in A and Z set or cleared
*  if A was valid hexadecimal, Z=1, otherwise Z=0
*********************
ishex:       psha                 ;save original character
             cmp    #'0'          ;check for < ASCII zero
             blo    nothex        ;branches if C=0 (Z also 0)
             cmp    #'9'          ;check for 0-9
             bls    okhex         ;branches if ASCII 0-9
             cmp    #'A'          ;check for < ASCII A
             blo    nothex        ;branches if C=0 (Z also 0)
             cmp    #'F'          ;check for A-F
             bhi    nothex        ;branches if > ASCII F
okhex:       clra                 ;forces Z bit to 1
nothex:      pula                 ;restore original character
             rts                  ;return Z=1 if char was hex
*********************
```

Figure 6-4. Parameter Passing in CCR Bits

Three branch instructions could lead to the exit sequence at nothex and in each case the programmer knows that the Z bit in the CCR would have to be 0 if the branch was taken. There are two ways to get to okhex and in each case the Z bit could be either 0 or 1, so the CLRA instruction is used to force the Z bit to be set to 1. The PULA and RTS instructions are executed after the tests that updated the Z bit but before the BNE errorHex instruction that uses the Z value. This works because the programmer checked the instruction set details to be sure PULA and RTS would not disturb the Z bit. This example shows that it is just as important to know which instructions do not change CCR status bits as it is to know which instructions do affect CCR status bits.

I — Interrupt Mask

The interrupt mask bit is a global interrupt mask that blocks all maskable interrupt sources while I = 1. Reset forces the I bit to logic 1 to block interrupts until the application program can initialize the stack pointer. If interrupts were allowed before the stack pointer was initialized, CPU register values could get saved (written) to inappropriate memory locations. The user program can set or clear I using the set interrupt mask (SEI) and clear interrupt mask (CLI) instructions, respectively.

The I bit is set automatically in response to any interrupt (including the SWI instruction) to prevent unwanted nesting of interrupts. It is possible to explicitly allow nesting of interrupts in a controlled manner by including a CLI instruction inside an interrupt service routine; however, this is not usually recommended because it can lead to subtle system errors which are particularly difficult to find and correct.

The WAIT and STOP instructions automatically clear the I bit because interrupts are the normal way to wake up the CPU from stop or wait modes. These instructions could have been designed so a separate CLI instruction was needed before executing WAIT or STOP. However, clearing I within these instructions saves the program space and execution time the separate CLI would have required, and prevents any possibility of an interrupt getting recognized after I is cleared but before the WAIT or STOP instruction.

When an interrupt occurs, The CCR value is saved on the stack before the I bit is automatically set (I would be 0 in the stacked CCR value). When the return-from-interrupt (RTI) instruction is executed to return to the main program, the act of restoring the CCR value from the stack normally clears the I bit.

When the I bit is set, the change takes effect too late in the instruction to prevent an interrupt at the instruction boundary immediately following an SEI or TAP instruction. In the case of setting I with a TAP or SEI instruction, I is actually set at the instruction boundary at the end of the TAP or SEI instruction. In the case of clearing I with a TAP or CLI instruction, I is actually cleared at the instruction boundary at the end of the TAP or SEI instruction. Because of this, the next instruction, after a CLI or TAP that cleared I, will always execute even if an interrupt was already waiting when the CLI or TAP that cleared I was executed. In the case of the RTI instruction, the CCR is restored during the first cycle of the instruction so the 1-cycle delay, associated with clearing I, expires several cycles before the RTI instruction finishes. WAIT and STOP also clear I in the middle of the instruction, so the delay expires before actually entering wait or stop mode.

V — Two's Complement Overflow Flag

This bit is set by the CPU when a two's complement overflow results from an arithmetic operation on signed binary values. For an addition operation, the V bit will be set if the sign (bit 7) is the same for both operands that were being added, but different from the sign of the result. For a subtract or compare operation, the V bit will be set if a positive number (bit 7=0) is subtracted from a negative number (bit 7=1) and the result is positive, or if a negative number is subtracted from a positive number and the result is negative.

The most common use of the V bit is to support the signed conditional branches (BLT, BLE, BGE, and BGT) after executing a CMP, CPHX, CPX, SBC, or SUB instruction. These instructions cause the ALU to subtract the contents of the referenced memory location (m) from the contents of a CPU register (r) and to set or clear V, N, Z, and C according to the results. (C is used for unsigned branches but not for signed conditional branches.) In the case of BLT, for example, the branch will be taken if the CPU register (r) was less than the memory location (m).

Several other instructions affect the V bit, and a clever programmer can sometimes use the condition of the V bit to control program flow. The Boolean formula for each affected CCR bit is given in the instruction details in **Appendix A. Instruction Set Details**.

The ADD and ADC instructions set V if both operands had the same sign and the sign of the result is different. Since no simple branch instructions are based on V alone, a sequence of two instructions is needed to test for two's complement overflow after an add operation. You could say BGE to no_overflow, followed by BMI to no_overflow, and falling through both of these branches implies there was a signed overflow condition. Operations like this are not common, but they can be understood by studying Boolean formulae and the Boolean equations for the branches in the instruction set detail pages in **Appendix A. Instruction Set Details**.

Arithmetic or logical shift left (ASL or LSL) is like multiplying a binary number by two. In this case, the V bit will be set if the sign of the result is different from the original signed value. The meaning of V after a right shift is less useful for signed arithmetic operations but could have some useful logical meaning in some systems.

The DAA instruction can change the V bit, so don't try to do a signed branch after a DAA instruction without executing a new compare or subtract instruction.

H — Half-Carry (Carry from Bit 3 to Bit 4)

The half-carry flag is intended for use with operations involving binary-coded-decimal (BCD) numbers. A BCD number is a decimal number from 0 through 9 which is coded into a single 4-bit binary value. This allows a single 8-bit value to hold exactly two BCD digits. The hexadecimal values $A through $F are considered illegal BCD values. The ALU's normal binary addition function can be used to add BCD numbers, but the results need to be checked and corrected so the result is still a valid BCD value. In the earlier M68HC08, the programmer had to do this checking and correction in a small program using the BHCC and BHCS conditional branch instructions. The HCS08 includes the decimal adjust accumulator (DAA) instruction to simplify the checking and correction operation into a single instruction.

The H bit is affected only by a few instructions. RTI restores the H bit to the value it had before servicing an interrupt. TAP allows the programmer to directly load all CCR bits with the contents of the accumulator. The multiply instruction (MUL) clears H as a side effect of its operation so avoid using a MUL instruction between an add operation and the DAA, BHCC, or BHCS instruction that needs the H bit value.

The add instructions (ADD and ADC) are the only instructions that affect the H bit in a meaningful way. These instructions set the H bit if there was a carry out of bit 3 into bit 4 of the result (from one BCD digit to the next). Although BHCC and BHCS instructions could be used to build a program that restores the result of an addition with BCD operands into a valid BCD result, it is more likely that you would use the DAA instruction because it performs the whole checking and correction operation in a single instruction. Refer to **6.5.2.4 BCD Arithmetic** for a more detailed explanation of BCD arithmetic.

N — Negative Flag

This flag indicates that the most significant bit of the result was set (1). It is called the negative flag because in two's complement notation a number is said to be negative if its most significant bit is a logic 1. If an operation involves 16-bit numbers (such as LDHX or CPHX), the N bit will be set if bit 15 of the result is set. In practice, this flag has many uses that are not related to signed arithmetic.

Branch if plus (BPL) and branch if minus (BMI) are simple branches which branch based solely on the value in the N bit. The N bit is also used by the signed branches BLT, BLE, BGE, and BGT since it indicates the sign of the result. All load, store, move, arithmetic, logical, shift, and rotate instructions cause the N bit to be updated. TAP allows N to be set directly from the value in bit 2 of A, and RTI restores N to the value that was saved on the stack when the interrupt service routine started.

The most significant bit of an I/O port, a control register, or a memory variable can be tested efficiently because just loading data from or storing data to a location automatically updates the N bit. In the following code fragment, a port is read where a switch is connected to bit 7. The N bit indicates whether the switch was on or off without any further test.

```
             lda    PTAD          ;read I/O port A
             bmi    swOff         ;branches if PTA7 was high
swOn:        nop                  ;here if MSB=0
swOff:                            ;here if MSB=1 (sw off)
```

Z — Zero Flag

The Z bit is set to indicate the result of an operation was $00 (or $0000 if it was a 16-bit operation). The related branch instructions are branch if equal (BEQ) and branch if not equal (BNE) because compare instructions perform an internal subtraction of a memory operand from the contents of a CPU register. If the original operands were equal, the result of this internal subtraction would be 0 and Z would be set to 1.

Branch if equal (BEQ) and branch if not equal (BNE) are simple branches which branch based solely on the value in the Z bit. The Z bit is also used by the signed branches BLE and BGT and the unsigned branches BLS and BHI. All load, store, move, arithmetic, logical, shift, and rotate instructions cause the Z bit to be updated. TAP allows Z to be set directly from the value in bit 1 of A, and RTI restores Z to the value that was saved on the stack when the interrupt service routine started.

Figure 6-4. Parameter Passing in CCR Bits shows an example where the Z bit is used to pass information back to a main program from a subroutine. To understand this example, study how compare instructions affect CCR bits and the Boolean formulae that are used by the branch instructions. This information is found in **Appendix A. Instruction Set Details**.

C — Carry (Out of Bit 7)

After an addition operation, the C bit is set if the source operands were both greater than or equal to $80 or if one of the operands was greater than or equal to $80 and the result was less than $80. This is equivalent to an unsigned overflow. A subtract or compare performs a subtraction of a memory operand from the contents of a CPU register so after a subtract operation, the C bit is set if the unsigned value of the memory operand was greater than the unsigned value of the CPU register. This is equivalent to an unsigned borrow or underflow.

Branch if carry clear (BCC) and branch if carry set (BCS) are simple branches which branch based solely on the value in the C bit. The C bit is also used by the unsigned branches BLO, BLS, BHS, and BHI. Add, subtract, shift, and rotate instructions cause the C bit to be updated. After a divide instruction, C is set if there was an attempt to perform an illegal divide-by-zero operation. TAP allows C to be set directly from the value in bit 0 of A, and RTI restores C to the value that was saved on the stack when the interrupt service routine started. The branch if bit set (BRSET) and branch if bit clear (BRCLR) instructions copy the tested bit into the C bit to facilitate efficient serial-to-parallel conversion algorithms. Set carry (SEC) and clear carry (CLC) allow the carry bit to be set or cleared directly. This is useful in combination with the shift and rotate instructions and for routines that pass status information back to a main program, from a subroutine, in the C bit.

The C bit is included in shift and rotate operations so those operations can easily be extended to multibyte operands. The shift and rotate operations can be considered 9-bit shifts which include an 8-bit operand or CPU register and the carry bit of the CCR. After a logical shift, C holds the bit that was shifted out of the 8-bit operand. If a rotate instruction is used next, this C bit is shifted into the operand for the rotate, and the bit that gets shifted out the other end of the operand replaces the value in C so it can be used in subsequent rotate instructions. Refer to **6.5.4 Shift and Rotate Instructions** to see a more detailed demonstration of this technique.

6.3 Addressing Modes

Whenever the MCU reads information from memory or writes information into memory, an addressing mode is used to determine the exact address where the information is read from or written to. This section explains several different ways to address memory, and each is useful in varying programming situations. For instance, in some addressing modes, the address is determined by the assembler when the program is written. Other addressing modes allow the address to be influenced by the contents of CPU registers. This is important because it allows the address to be computed during execution of the program.

Every opcode tells the CPU to perform a certain operation in a certain way. Many instructions such as load accumulator (LDA) allow several different ways to specify the memory location to be operated on, and each addressing mode variation requires a separate opcode. All of these variations use the same instruction mnemonic, and the assembler knows which opcode to use based on the syntax of the operand field. In some cases, special characters are used to indicate a specific addressing mode (such as the # [pound] symbol which indicates immediate addressing mode). In other cases, the value of the operand tells the assembler which addressing mode to use. For example, the assembler chooses direct addressing mode instead of extended addressing mode if the operand address is between $0000 and $00FF.

Some instructions use more than one addressing mode. For example, the move instructions use one addressing mode to access the source value from memory and a second addressing mode to access the destination memory location. For these move instructions, both addressing modes are listed in the documentation. All branch instructions use relative (REL) addressing mode to determine the destination for the branch, but BRCLR, BRSET, CBEQ, and DBNZ also need to access a memory operand. These instructions are classified by the addressing mode used for the memory operand, and the relative addressing mode for the branch offset is just assumed.

In the following paragraphs, the discussion includes how each addressing mode works and the syntax clues the assembler uses to know that the programmer wants a specific addressing mode.

6.3.1 Inherent Addressing Mode (INH)

This addressing mode is used when the CPU inherently knows everything it needs to complete the instruction, and no addressing information is supplied in the source code. Usually, the operands that the CPU needs are located in the CPU's internal registers, as in ASLA, CLRX, DAA, DIV, RSP, and others. Instructions like clear carry bit (CLC) and set interrupt mask (SEI) affect a single bit within the CCR. A few inherent instructions, including no operation (NOP) and background (BGND), have no operands.

Another group of instructions listed as inherent (INH) actually access memory based on the value of the stack pointer. Instructions of this type include PSHx, PULx, RTI, RTS, and SWI. A purist could argue that SWI uses a form of indexed addressing to push CPU register values onto the stack and extended addressing to fetch the SWI vector, but since there is no program-supplied addressing information, it is considered an inherent instruction.

6.3.2 Relative Addressing Mode (REL)

Relative addressing mode is used to specify the destination address for branch instructions. Typically, the programmer specifies the destination with a program label or an expression in the operand field of the branch instruction. The assembler calculates the difference between the location counter (which points at the next address after the branch instruction at the time) and the address represented by the label or expression in the operand field. This difference is called the offset and is an 8-bit two's complement number. The assembler stores this offset in the object code for the branch instruction.

During execution, the CPU evaluates the condition that controls the branch. If the branch condition is true, the CPU sign-extends the offset to a 16-bit value, adds the offset to the current PC, and uses this as the address where it will fetch the next instruction and continue execution rather than continuing execution with the next instruction after the branch. Since the offset is an 8-bit two's complement value, the destination must be within the range –128 to +127 locations from the address that follows the last byte of object code for the branch instruction.

A common method to create a simple infinite loop is to use a branch instruction that branches to itself. This is sometimes used to end short code segments during debug. Typically, to get out of this infinite loop, use the debug host (through background commands) to stop the program, examine registers and memory or to start execution from a new location. This construct is not used in normal application programs except in the case where the program has detected an error and wants to force the COP watchdog timer to timeout. (The branch in the infinite loop executes repeatedly until the watchdog timer eventually causes a reset.)

6.3.3 Immediate Addressing Mode (IMM)

In this addressing mode, the operand is located immediately after the opcode in the instruction stream. This addressing mode is used when the programmer wants to use an explicit value that is known at the time the program is written. A # (pound) symbol is used to tell the assembler to use the operand as a data value rather than an address where the desired value should be accessed.

The size (8 bits or 16 bits) of the immediate operand is assumed based on the size of the CPU register indicated in the instruction. For example, a load A or add A instruction implies an 8-bit operand while a load H:X or compare H:X instruction implies a 16-bit operand to match the width of the H:X register pair. The assembler automatically will truncate or extend the operand as needed to match the size needed for the instruction. Most assemblers generate a warning if a 16-bit operand is provided where an 8-bit operand was expected.

A common programming error is to accidentally forget the # symbol before an immediate operand. In the following example, the first instruction tells the assembler to compare the contents of the H:X register pair to the address of tableEnd. Leaving the # symbol off in the second instruction tells the assembler to compare the contents of the H:X register pair to the 16-bit value stored at tableEnd and tableEnd+1 (using extended addressing mode).

```
C04A 65 00BF                cphx   #tableEnd     ;H:X points at end of table?
C04D 75 BF                  cphx   tableEnd      ;compare to value at tableEnd
C04F A6 55                  lda    #$55          ;load pattern $55 into A
C051 B6 55                  lda    $55           ;load A from address $0055
```

It is the programmer's responsibility to use the # symbol to tell the assembler when immediate addressing should be used. The assembler does not consider it an error to leave off the # symbol because the resulting statement is still a valid instruction (although it may mean something different than the programmer intended).

6.3.4 Direct Addressing Mode (DIR)

This addressing mode is used to access operands located in direct address space ($0000 through $00FF). This is a more efficient addressing mode than extended addressing because the upper 8 bits of the address are implied rather than being explicitly provided in the instruction. This saves a byte of program space and the bus cycle that would have been needed to fetch this byte.

The programmer does not use any special syntax to choose this mode. Rather, the assembler evaluates the label or expression in the operand field and automatically chooses direct addressing mode if the resulting address is in the range $0000 through $00FF. During execution, the CPU gets the low byte of the direct address from the operand byte that follows the opcode, appends a high byte of $00, and uses this 16-bit address ($00xx) to access the intended operand.

Most of the I/O and control registers are located in the first 64 or 128 bytes of memory (a few rarely used registers are located in high memory at $18xx). Some of the on-chip RAM is also located in the direct page to allow frequently accessed variables to be located there so direct addressing can be used. After reset, the stack pointer points at $00FF and it is recommended that you change SP to point at the top of RAM instead, to make the RAM below $00FF available for direct addressed variables.

6.3.5 Extended Addressing Mode (EXT)

In the extended addressing mode, the full 16-bit address of the operand is included in the object code in the next two bytes after the opcode. This addressing mode can be used to access any location in the 64-Kbyte memory map. Normally, the programmer uses a program label to specify the address and the assembler substitutes the equivalent 16-bit address as the program is assembled.

6.3.6 Indexed Addressing Mode

Indexed addressing mode is sometimes called indirect addressing mode because a CPU index register is used as a reference, an offset is optionally added to the index reference, and the resulting address is then used to access the intended operand. In some cases the value in the index register is incremented automatically after the operand has been accessed. This can save programming steps by making the index register point at the next operand in a list or by incrementing a loop count.

An important feature of indexed addressing mode is that the operand address is computed during execution based on the then-current contents of a CPU index register rather than being a constant address location that was determined during program assembly. This allows the programmer to write a compact program loop that accesses successive values in a list or table on each pass through the loop. It also allows a program to be written that accesses different operand locations depending on the results of earlier program instructions (rather than accessing a location that was determined when the program was written).

6.3.6.1 Indexed, No Offset (IX)

In this variation of indexed addressing, the content of the H:X index register pair is used as the address of the operand to be accessed.

6.3.6.2 Indexed, No Offset with Post Increment (IX+)

In this variation of indexed addressing, the content of the H:X index register pair is used to access the intended operand, and then the H:X register pair is incremented by one. CBEQ and MOV instructions are the only instructions which use this addressing mode.

```
             ldhx   #stringBytes  ;point at top of block
             lda    #' '          ;pattern to search for
findSP:      cbeq   x+,foundSP    ;found ASCII space ($20) ?
; H:X pointing at location after space
             bra    findSP        ;keep looking
foundSP:     aix    #-1           ;back up to the space
```

6.3.6.3 Indexed, 8-Bit Offset (IX1)

In this variation of indexed addressing, an instruction-supplied unsigned 8-bit offset is added to the H:X register pair to form the address of the operand to be accessed. The addition of the offset is an internal calculation that does not affect the contents of H:X.

6.3.6.4 Indexed, 8-Bit Offset with Post Increment (IX1+)

In this variation of indexed addressing, an instruction-supplied unsigned 8-bit offset is added to the H:X register pair to form the address of the operand to be accessed. The addition of the offset is an internal calculation that does not affect the contents of H:X. After the operand has been accessed, the H:X register pair is incremented by one. CBEQ is the only instruction which uses this addressing mode.

6.3.6.5 Indexed, 16-Bit Offset (IX2)

In this variation of indexed addressing, an instruction-supplied unsigned 16-bit offset is added to the H:X register pair to form the address of the operand to be accessed. The addition of the offset is an internal calculation that does not affect the contents of H:X.

This addressing mode is particularly useful for addressing two data structures in different areas of memory from a single index reference value in H:X. The following example demonstrates this technique.

```
                ; string compare with one string in flash, the other in RAM (IX2)
C064 45 0088                 ldhx   #moveBlk1     ;point at string 1 in RAM
C067 65 0092   chkLoop:      cphx   #moveBlk1+10  ;see if past end
C06A 27 06                   beq    stringOK      ;if so, you are done
C06C D6 BF7F                 lda    (stringBytes-moveBlk1),x ;load from flash
C06F 71 F6                   cbeq   x+,chkLoop    ;compare to byte in flash
C071 9D        stringBad:    nop                  ;here if string didn't match
               stringOK:                          ;here if it did
```

In the example, the two data structures have similar structures. One is in RAM and holds current data values. The second data structure is a set of constant values in FLASH memory. The assembler computes the expression (stringBytes–moveBlk1) to get the 16-bit offset from moveBlk1 in RAM to stringBytes in flash. As the index is incremented (in the CBEQ instruction), the LDA (stringBytes-moveBlk1),X accesses the next byte from stringBytes and CBEQ 0,X+,chkLoop accesses the next byte from moveBlk1 in RAM.

6.3.6.6 SP-Relative, 8-Bit Offset (SP1)

In this variation of indexed addressing, an instruction-supplied unsigned 8-bit offset is added to the stack pointer (SP) to form the address of the operand to be accessed. The addition of the offset is an internal calculation that does not affect the contents of SP. Note that the SP points at the next available location on the stack rather than the last value that was pushed onto the stack, so read operations with an offset of zero are normally not useful.

Stack pointer relative addressing is most commonly used to access parameters and local variables on the stack. This is a common practice for compiled C code. Depending on the number of stack relative accesses and what the H:X register pair is being used for, the compiler will sometimes temporarily save the current H:X value and move SP into H:X to allow indexed addressing from H:X rather than SP because SP-relative addressing typically takes an extra cycle and byte of program space compared to H:X-relative addressing.

```
C072 A7 FD                  ais    #-3           ;space for 3 bytes of locals
               ; sp+1 is a byte sized local
               ; sp+2:sp+3 is a 16-bit local (an integer variable)
C074 9E6F 02                clr    2,sp          ;clear high byte of local int
C077 9E6F 03                clr    3,sp          ;clear low byte of local int
C07A A6 04                  lda    #4
C07C 9EE7 01                sta    1,sp          ;set local byte to 4
```

```
               ; tsx & index based on H:X to save code size comapred to previous
               ; tsx cost 1 byte but saved 4 (overall savings equal 3 bytes)
C07F A7 FD                  ais    #-3           ;space for 3 bytes of locals
C081 95                     tsx                  ;H:X <- SP+1
C082 6F 01                  clr    1,x           ;clear high byte of local int
C084 6F 02                  clr    2,x           ;clear low byte of local int
C086 A6 04                  lda    #4
C088 F7                     sta    ,x            ;set local byte to 4
```

6.3.6.7 SP-Relative, 16-Bit Offset (SP2)

In this variation of indexed addressing, an instruction-supplied unsigned 16-bit offset is added to the stack pointer (SP) to form the address of the operand to be accessed. The addition of the offset is an internal calculation that does not affect the contents of SP. Note that the SP points at the next available location on the stack rather than the last value that was pushed onto the stack.

This addressing mode is used to access data that is more than 255 locations deep in the stack. If the offset is 255 or less, the assembler will automatically use the more efficient SP1 addressing mode.

6.4 Special Operations

Most of what the CPU does is described by the instruction set, but a few special operations need to be considered, such as how the CPU gets started at the beginning of an application program after power is first applied. Once the program is running, the current instruction normally determines what the CPU will do next. A few exceptional events can cause the CPU to temporarily suspend normal program execution. Reset events force the CPU to start over at the beginning of the application program as directed by the contents of the reset vector. Hardware interrupts can come from external pins or from internal peripheral modules. These interrupts cause the CPU to complete the current instruction and then respond to the interrupt rather than continuing to the next instruction in the application program. Finally, a host development system can cause the CPU to go to active background mode rather than continuing to the next instruction in the application program.

Wait and stop modes are activated as the result of the WAIT and STOP instructions, respectively; however, these special instructions also affect other systems in the microcontroller (MCU). While these modes are active, CPU activity is suspended indefinitely until some hardware event occurs to wake up the MCU.

6.4.1 Reset Sequence

Processing begins at the trailing edge of a reset event. The number of things that can cause reset events can vary slightly from one HCS08 derivative to another; however, the most common sources are power-on reset, the external $\overline{\text{RESET}}$ pin, low-voltage reset, COP watchdog timeout, illegal opcode detect, and illegal address access. For more information about how the MCU recognizes reset events and determines the differences between internal and external causes, refer to **Section 5. Resets and Interrupts**. For detailed information about all of the possible causes of reset in a particular HCS08 derivative, refer to the appropriate technical data sheet.

Reset events force the MCU to immediately stop what it is doing and begin responding to reset. Any instruction that was in process will be aborted immediately without completing any remaining clock cycles. A short sequence of activities is completed to decide whether the source of reset was internal or external and to record the cause of reset. For the remainder of the time the reset source remains active, the internal clocks are stopped to save power. At the trailing edge of the reset event, the clocks resume and the CPU exits from the reset condition.

The CPU performs a 6-cycle sequence to exit reset before starting the first program instruction. The high-order byte of the reset vector is fetched from $FFFE and stored in the high-order byte of the program counter. The low-order byte of the reset vector is fetched from $FFFF and stored in the low-order byte of the program counter. The next bus cycle is a free cycle where the CPU does not access memory because the low-order half of the vector is not yet available to the CPU. Whenever the CPU performs a memory read operation, there is a 1 cycle delay before the data has time to propagate into the CPU where it can be used in any subsequent operation. Next, the CPU places the program counter address on the bus to fetch the first byte of program information and then increments the program counter. (The program counter contained the reset vector that was just fetched from $FFFE, FFFF.) The next cycle (fifth in the reset sequence) fetches the second byte of program information into the instruction queue, and the next cycle (last in the reset sequence) accesses the third byte of program information so it is on its way into the instruction queue.

After the 6-cycle reset sequence, two bytes of program information are available to the CPU in the instruction queue and a third byte is on its way. Notice that MCU operations form a continuous stream of activity and different parts of the system see different events within this stream at any particular instant in time. To avoid confusion, the user's perception of a bus cycle is used as the single point of reference for all further discussions. See **6.4.6 User's View of a Bus Cycle**.

6.4.2 Interrupts

As the name implies, interrupts interrupt the normal flow of instructions. Except for the SWI instruction, interrupts are caused by hardware events and are generally asynchronous to the operating program. The software interrupt instruction (SWI) behaves like other interrupts except that it is not maskable (cannot be inhibited by the I bit in the CCR being 1).

When an interrupt is requested, the CPU completes the current instruction before responding to the interrupt. The interrupt sequence follows the same cycle-by-cycle sequence as the SWI instruction and consists of saving the CPU registers on the stack, setting the I bit in the CCR to mask further interrupts, fetching the interrupt vector for the highest priority interrupt that is currently pending, and filling the instruction queue with the first three bytes of program information for the interrupt service routine. For more information about how the MCU recognizes and processes interrupts, refer to **Section 5. Resets and Interrupts**.

The interrupt mask bit (I bit) in the CCR acts as a global interrupt mask. When I is 1, interrupt requests are ignored by the CPU. Immediately after reset, the I bit is 1 so that interrupts are disabled. Before clearing the I bit to enable interrupts, initialize the stack pointer.

For compatibility with the earlier M68HC05 Family, the stack pointer is automatically initialized to $00FF at reset. This is rarely where you want the stack to be located in an HCS08 system because this would cause the stack to use valuable direct address space (the space from $0000 through $00FF). Usually, the stack pointer should be set to point at the highest address in the on-chip RAM. Since there isn't a load stack pointer instruction, load H:X with the address of the last RAM location plus one, and then transfer this value to SP. There is an automatic adjustment of the 16-bit value as it is transferred from H:X to SP so the stack pointer will point at the next available location on the stack (in this case, so H:X points at the last location in the on-chip RAM). Refer to **6.2.3 Stack Pointer (SP)** for a more detailed explanation of the stack pointer.

Again for compatibility with the earlier M68HC05, the HCS08 does not stack the high-order half of the index register (H) in response to an interrupt. In rare cases, you can choose not to stack H inside the interrupt service routine if you are absolutely sure the service routine will never alter H. Many instructions, including AIX and post-increment indexed addressing versions of instructions, can cause H to change. Therefore, it is generally safer to include a PSHH instruction as the first instruction in the interrupt service routine and a PULH instruction as the last instruction before the RTI that ends the service routine.

6.4.3 Wait Mode

Wait mode is entered by executing a WAIT instruction. This instruction clears the I bit in the CCR (so interrupts can wake up the MCU from wait mode), and then shuts down the clocks in the CPU to save power. The CPU remains in this low-power state until an interrupt or reset event wakes it up. For more detailed information about wait mode, refer to **3.5 Wait Mode**.

6.4.4 Stop Mode

Stop mode is entered by executing a STOP instruction. This instruction clears the I bit in the CCR (so interrupts can wake up the MCU from stop mode), and then shuts down the clocks in the CPU to save power. Depending on other control settings in the MCU, the system oscillator may be completely disabled to reduce power consumption even further. The CPU remains in this low-power state until an interrupt or reset event wakes it up. The wakeup sequence depends on whether the oscillator was completely stopped, and what type of clock generation system is controlling the particular derivative. For more detailed information about stop mode, refer to **3.6 Stop Modes**.

6.4.5 Active Background Mode

Active background mode refers to the condition where the CPU has stopped executing user program instructions and is waiting for serial commands from the background debug system. The CPU cannot enter active background mode unless it has been enabled by a serial WRITE_CONTROL command which has set the ENDBM bit in the BDCSCR. (BDCSCR is a status and control register within the background debug controller (BDC) and is not accessible from the user program.) The usual way the CPU gets into active background mode is in response to a BACKGROUND command through the serial background communication interface (BKGD pin). The CPU can also enter active background mode due to a reset event where the BKGD pin is held low at the trailing edge of reset, due to a BGND instruction, or in response to a hardware breakpoint event.

Reset with BKGD low provides a way for a development system to gain control of a target MCU immediately after reset before any user reset vector is fetched and before any user instructions are executed. This is important in systems where the program memory and vectors are not yet programmed.

BGND instructions are used only by development systems to set software breakpoints and should never be used in normal application programs. If a program runaway condition causes the CPU to encounter a BGND instruction when no development system is connected to the BKGD pin, ENBDM would be 0 and the BGND instruction would be treated as an illegal opcode.

The hardware breakpoint that is built into the BDC system is only accessible by serial commands through BKGD, so this breakpoint would only occur if a development system is connected to BKGD and ENBDM = 1.

Some HCS08 MCUs can have additional hardware breakpoints built outside the CPU and BDC systems. These hardware breakpoints can be controlled by user programs as well as development systems. If this type of hardware breakpoint is encountered while ENBDM = 1, the CPU completes the current instruction and then enters active background mode. If this type of hardware breakpoint is encountered while ENBDM = 0, the CPU will execute an SWI instruction rather than trying to execute an illegal BGND instruction. With proper planning, this mechanism can be used to allow a form of ROM patching. Refer to **7.5.9 Hardware Breakpoints and ROM Patching**.

The CPU can remain in the active background mode indefinitely until a serial GO, TRACE1, or TAGGO command causes it to return to the user's application program. In a 3-cycle sequence on exit from active background mode, the CPU does three program fetches to fill the instruction queue. There is no way for the CPU to know whether the development system has altered program memory, so the CPU always refills the instruction queue upon exit from active background mode.

6.4.6 User's View of a Bus Cycle

In modern microcontrollers, operations are pipelined such that different parts of the circuit can be working on different information at any particular instant in time. To avoid confusion, it is important to have a single consistent point of reference so other system timing can be related to this common reference. This common reference point for the HCS08 is a bus cycle. A read bus cycle is considered to begin with the CPU internally generating an address which is then presented to the internal address bus. The addressed memory location then places the requested data on the internal data bus after a memory access time. A write cycle begins like a read cycle, with the CPU internally generating an address which is then presented to the internal address bus. Next the data to be written is presented to the internal data bus and remains valid long enough for the memory access to be completed.

Since these internal activities are not directly visible from the outside of the chip, we must relate this to an external event such as the trailing edge of a reset event. The cycle-by-cycle sequence for the reset operation is vvfppp where the first two v cycles are the bus cycles where the upper and lower bytes of the reset vector are fetched from $FFFE and $FFFF, respectively. The f cycle is a free cycle where the CPU does not use the internal buses. The three p cycles are used to fill the instruction queue with the first three bytes of object code for the user program beginning at the address just fetched from the reset vector. From this point, a user can tell exactly what should be on the internal buses for every bus cycle of a program because every CPU instruction and exception event has a known sequence of bus cycles.

The exit from reset is synchronized to an internal bus clock so there is an uncertainty of up to one bus cycle from the actual release of the active low at the reset pin and when the first v cycle starts. There is a propagation delay from the external oscillator input (if present) to the internal bus clock which is not specified because the user cannot access the internal bus clock to make a measurement.

6.5 Instruction Set Description by Instruction Types

In this section, the instruction is listed by types of instructions. Explanations of how these instructions can be used within the context of an application program are provided. Example code segments are used to show practical examples of common programming constructs.

6.5.1 Data Movement Instructions

This group of instructions is used to move data between memory and CPU registers, between memory locations, or between CPU registers. Load, store, and move instructions automatically update the condition codes based on the value of the data. This allows conditional branching with BEQ, BNE, BPL, and BMI immediately after a load, store, or move instruction without having to do a separate test or compare instruction.

6.5.1.1 Loads and Stores

Table 6-1. Load and Store Instructions

Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
LDA *#opr8i* LDA *opr8a* LDA *opr16a* LDA *oprx16*,X LDA *oprx8*,X LDA ,X LDA *oprx16*,SP LDA *oprx8*,SP	Load Accumulator from Memory	A ← (M)	0	–	–	↕	↕	–	IMM DIR EXT IX2 IX1 IX SP2 SP1	A6 B6 C6 D6 E6 F6 9ED6 9EE6	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
LDHX *#opr16i* LDHX *opr8a* LDHX *opr16a* LDHX ,X LDHX *oprx16*,X LDHX *oprx8*,X LDHX *oprx8*,SP	Load Index Register (H:X) from Memory	H:X ← (M:M + $0001)	0	–	–	↕	↕	–	IMM DIR EXT IX IX2 IX1 SP1	45 55 32 9EAE 9EBE 9ECE 9EFE	jj kk dd hh ll ee ff ff ff	3 4 5 5 6 5 5
LDX *#opr8i* LDX *opr8a* LDX *opr16a* LDX *oprx16*,X LDX *oprx8*,X LDX ,X LDX *oprx16*,SP LDX *oprx8*,SP	Load X (Index Register Low) from Memory	X ← (M)	0	–	–	↕	↕	–	IMM DIR EXT IX2 IX1 IX SP2 SP1	AE BE CE DE EE FE 9EDE 9EEE	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
STA *opr8a* STA *opr16a* STA *oprx16*,X STA *oprx8*,X STA ,X STA *oprx16*,SP STA *oprx8*,SP	Store Accumulator in Memory	M ← (A)	0	–	–	↕	↕	–	DIR EXT IX2 IX1 IX SP2 SP1	B7 C7 D7 E7 F7 9ED7 9EE7	dd hh ll ee ff ff ee ff ff	3 4 4 3 2 5 4
STHX *opr8a* STHX *opr16a* STHX *oprx8*,SP	Store H:X (Index Reg.)	(M:M + $0001) ← (H:X)	0	–	–	↕	↕	–	DIR EXT SP1	35 96 9EFF	dd hh ll ff	4 5 5
STX *opr8a* STX *opr16a* STX *oprx16*,X STX *oprx8*,X STX ,X STX *oprx16*,SP STX *oprx8*,SP	Store X (Low 8 Bits of Index Register) in Memory	M ← (X)	0	–	–	↕	↕	–	DIR EXT IX2 IX1 IX SP2 SP1	BF CF DF EF FF 9EDF 9EEF	dd hh ll ee ff ff ee ff ff	3 4 4 3 2 5 4

Load A and load X cause an 8-bit value to be read from memory into accumulator A or into the X register. Load H:X causes one 8-bit value to be read from memory into the H register and a second 8-bit value to be read from the next sequential memory location into the X register. Load A and load X each allow eight different addressing modes for maximum flexibility in accessing memory. LDHX allows seven different addressing modes to specify the memory locations of the values being read.

The following instructions demonstrate some of the uses for load instructions. This collection of instructions is not intended to be a meaningful program. Rather, they are unrelated load instructions to demonstrate the many possible addressing modes that allow access to memory in different ways.

```
               ; load A - various addressing modes
               ; immediate (IMM) addressing mode examples
C089 A6 55                  lda    #$55          ;IMM - $ means hexadecimal
C08B A6 64                  lda    #100          ;decimal 100 (hexadecimal $64)
C08D A6 3F                  lda    #%00111111    ;% means binary
C08F A6 41                  lda    #'A'          ;single quotes around ASCII
C091 A6 8D                  lda    #illegalOp    ;label used as immediate value
               ; direct (DIR) addressing mode examples
C093 B6 55                  lda    $55           ;load from address $0055
C095 B6 9D                  lda    directByte    ;label as a direct address
               ; extended (EXT) addressing mode
C097 C6 FFFE                lda    $FFFE         ;high byte of reset vector
C09A C6 0101                lda    extByte       ;label used as an address
C09D C6 C09D                lda    *             ;* means "here", loads opcode
C0A0 C6 009D                lda    fwdRef        ;forces ext addressing mode
               ; not all assemblers treat forward references the same way
     0000 009D fwdRef:      equ    directByte    ;forward referenced direct

C0A3 45 C007                ldhx   #stringBytes  ;point at string in flash
               ; indexed addressing mode (relative to H:X index register pair)
C0A6 D6 4081                lda    (moveBlk1-stringBytes),x  ;IX2 mode
C0A9 E6 01                  lda    1,x           ;IX1 - 8-bit offset
C0AB F6                     lda    ,x            ;IX - no offset

               ; indexed addressing mode (relative to SP stack pointer)
C0AC 45 0001                ldhx   #1
C0AF 94                     txs                  ;temp move SP for 16-bit offset ex.
C0B0 9ED6 012C              lda    300,sp        ;SP2 - 16-bit offset
C0B4 9EE6 01                lda    1,sp          ;SP1 - 8-bit offset
```

Since one operand input to the arithmetic logic unit (ALU) is connected to the A accumulator, you typically need to use an LDA instruction to read one value into A before performing mathematical or logical operations involving a second operand.

```
; add A + B (assumes sum is < or = 255)
             lda    oprA          ;oprA -> accumulator
             add    oprB          ;oprA + oprB -> accumulator
```

In some cases, you can plan your program so that the results that were stored in accumulator A as the result of one operation can be used as an operand in a subsequent operation. This can save the need to store one result and reload the accumulator with the next operand.

```
; add A + B + C (assumes sum is < or = 255)
             lda    oprA          ;oprA -> accumulator
             add    oprB          ;oprA + oprB -> accumulator
             add    oprC          ;accum. + oprC -> accum.
```

The next example shows an intermediate value being saved on the stack. This is sometimes faster than storing temporary results in memory. The amount of savings depends on what addressing mode would be needed to store the temporary value in memory and whether the X register was needed for something else at the time.

```
; compute (A + B) - (C + D) (assumes no carry or borrow)
             lda    oprC          ;oprC -> accumulator
             add    oprD          ;oprC + oprD -> accumulator
             psha                 ;intermediate result to SP+1
             lda    oprA          ;oprA -> accumulator
             add    oprB          ;oprA + oprB -> accumulator
             sub    1,sp          ;(A+B)-(C+D) to accumulator
             ais    #1            ;deallocate local space
```

Table 6-2. BSET, BCLR, Move, and Transfer Instructions

Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
BSET *n*,*opr8a*	Set Bit *n* in Memory	Mn ← 1	–	–	–	–	–	–	DIR (b0) DIR (b1) DIR (b2) DIR (b3) DIR (b4) DIR (b5) DIR (b6) DIR (b7)	10 12 14 16 18 1A 1C 1E	dd dd dd dd dd dd dd dd	5 5 5 5 5 5 5 5
BCLR *n*,*opr8a*	Clear Bit n in Memory	Mn ← 0	–	–	–	–	–	–	DIR (b0) DIR (b1) DIR (b2) DIR (b3) DIR (b4) DIR (b5) DIR (b6) DIR (b7)	11 13 15 17 19 1B 1D 1F	dd dd dd dd dd dd dd dd	5 5 5 5 5 5 5 5
MOV *opr8a*,*opr8a* MOV *opr8a*,X+ MOV #*opr8i*,*opr8a* MOV ,X+,*opr8a*	Move	$(M)_{destination} \leftarrow (M)_{source}$ H:X ← (H:X) + $0001 in IX+/DIR and DIR/IX+ Modes	0	–	–	↕	↕	–	DIR/DIR DIR/IX+ IMM/DIR IX+/DIR	4E 5E 6E 7E	dd dd dd ii dd dd	5 5 4 5
TAX	Transfer Accumulator to X (Index Register Low)	X ← (A)	–	–	–	–	–	–	INH	97		1
TXA	Transfer X (Index Reg. Low) to Accumulator	A ← (X)	–	–	–	–	–	–	INH	9F		1
TAP	Transfer Accumulator to CCR	CCR ← (A)	↕	↕	↕	↕	↕	↕	INH	84		1
TPA	Transfer CCR to Accumulator	A ← (CCR)	–	–	–	–	–	–	INH	85		1
NSA	Nibble Swap Accumulator	A ← (A[3:0]:A[7:4])	–	–	–	–	–	–	INH	62		1

6.5.1.2 Bit Set and Bit Clear

Bit set (BSET) and bit clear (BCLR) instructions can be thought of as bit-sized store instructions, but these instructions actually read a full 8-bit location, modify the specified bit, and then re-write the whole 8-bit location. In certain cases, such as when the target location is something other than a RAM variable, this subtle behavior can lead to unexpected results. If a BSET or BCLR instruction attempts to change a bit in a nonvolatile memory location, naturally, the bit will not change because nonvolatile memories require a more complex sequence of operations to make changes.

Some status bits are cleared by a sequence involving a read of the status bit followed by a write to another register in the peripheral module. Some users are surprised to find that a BSET or BCLR instruction has satisfied the requirement to read the status register. To avoid such problems, just remember that the BSET and BCLR instructions are read-modify-write instructions that access a full 8-bit location in parallel.

Some control or I/O registers do not access the same physical logic states for reads and writes. In general, do not use read-modify-write instructions on these locations because they may produce unexpected results.

```
                    ; BSET example - turns on TE without changing RE
    C0D3 16 1B                   bset   TE,SCI1C2     ;enable SCI transmitter
                    ; functionally equivalent to...
    C0D5 B6 1B                   lda    SCI1C2        ;read current SCCR2 value
    C0D7 AA 08                   ora    #mTE           ;OR in TE bit (mask)
    C0D9 B7 1B                   sta    SCI1C2        ;upate value in SCCR2
```

6.5.1.3 Memory-to-Memory Moves

Move instructions can be helpful in an accumulator architecture like the HCS08 where the number of registers is limited. MOV performs a read of an 8-bit value from one memory location and stores the value in a different location. Like the load and store instructions, MOV causes the N and Z bits in the CCR to be updated according to the value of the data being moved.

Although load and store instructions could be used to do the same thing as a MOV instruction, MOV does not require the accumulator to be saved so that A can be used as the transport means for the move operation. In many cases, the MOV approach is faster and smaller (object code size) than the load-store combination. MOV allows four different address mode combinations to specify the source and destination locations for the move.

The following example shows how move instructions can be used to initialize several register values.

```
    C0DB 6E 03 00                mov    #$03,PTAD     ;0011 to 4 LS bits
    C0DE 6E 0F 03                mov    #$0F,PTADD    ;make 4 LS bits outputs
    C0E1 6E F0 01                mov    #$F0,PTAPE    ;pullups on 4 MS bits
```

The next example shows a string move operation using load and store instructions rather than move instructions.

```
               ; block move example to move a string to a RAM block
C0E4 45 0088                ldhx   #moveBlk1     ;point at destination block
C0E7 D6 BF7F   movLoop1:    lda    (stringBytes-moveBlk1),x  ;get source byte
C0EA 27 04                  beq    dunLoop1      ;null terminator ends loop
C0EC F7                     sta    ,x            ;save to destination block
C0ED 5C                     incx                 ;next location (assumes DIR)
C0EE 20 F7                  bra    movLoop1      ;continue loop
               dunLoop1:
```

6.5.1.4 Register Transfers and Nibble Swap

TAX and TXA offer an efficient way to transfer a value from A to X or from X to A. Depending on whether the X register is already being used, this can be an efficient way to temporarily save the accumulator value so A can be used for some other operation.

TAP and TPA provide a means for moving the value from A into the CCR (processor status byte) or from the CCR into A. This is used more in development tools like debug monitors than in normal user programs.

The nibble swap A (NSA) instruction exchanges the upper and lower nibbles of the accumulator (A). An 8-bit value is called a byte and a nibble is the upper- or lower-order four bits of a byte. Each nibble corresponds to exactly one hexadecimal digit. This instruction is useful for conversions between binary or hexadecimal and ASCII, and for operations on binary-coded-decimal (BCD) numbers.

```
*********************
* chexl - convert upper nibble of A to ASCII
* chexr - convert lower nibble of A to ASCII
*  on entry A contains any binary (hexadecimal) number
*  returns with resulting ASCII character in A
*********************
chexl:       nsa                  ;swap nibble into low half
chexr:       and    #$0F          ;strip off upper nibble
             add    #$30          ;now $30 - $3F
             cmp    #$39          ;check for < or = '9'
             bls    dunChex       ;if so, just return
             add    #7            ;adjust to $41-$46
dunChex:     rts                  ;return with ASCII in A
*********************
```

6.5.2 Math Instructions

Math instructions include the traditional add, subtract, multiply, and divide operations, a collection of utility instructions including increment, decrement, clear, negate (two's complement), compare, and test, and a decimal adjust instruction for computations involving BCD numbers. The compare instructions are actually subtract operations where the CCR bits are affected but the result is not written back to a CPU register. The test instructions affect the N and Z condition code bits, but do not affect the tested value.

6.5.2.1 Add, Subtract, Multiply, and Divide

Table 6-3. Add, Subtract, Multiply, and Divide Instructions

Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
ADC *#opr8i* ADC *opr8a* ADC *opr16a* ADC *oprx16*,X ADC *oprx8*,X ADC ,X ADC *oprx16*,SP ADC *oprx8*,SP	Add with Carry	A ← (A) + (M) + (C)	↕	↕	–	↕	↕	↕	IMM DIR EXT IX2 IX1 IX SP2 SP1	A9 B9 C9 D9 E9 F9 9ED9 9EE9	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
ADD *#opr8i* ADD *opr8a* ADD *opr16a* ADD *oprx16*,X ADD *oprx8*,X ADD ,X ADD *oprx16*,SP ADD *oprx8*,SP	Add without Carry	A ← (A) + (M)	↕	↕	–	↕	↕	↕	IMM DIR EXT IX2 IX1 IX SP2 SP1	AB BB CB DB EB FB 9EDB 9EEB	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
AIX *#opr8i*	Add Immediate Value (Signed) to Index Register (H:X)	H:X ← (H:X) + (M) M is sign extended to a 16-bit value	–	–	–	–	–	–	IMM	AF	ii	2
SUB *#opr8i* SUB *opr8a* SUB *opr16a* SUB *oprx16*,X SUB *oprx8*,X SUB ,X SUB *oprx16*,SP SUB *oprx8*,SP	Subtract	A ← (A) – (M)	↕	–	–	↕	↕	↕	IMM DIR EXT IX2 IX1 IX SP2 SP1	A0 B0 C0 D0 E0 F0 9ED0 9EE0	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
SBC *#opr8i* SBC *opr8a* SBC *opr16a* SBC *oprx16*,X SBC *oprx8*,X SBC ,X SBC *oprx16*,SP SBC *oprx8*,SP	Subtract with Carry	A ← (A) – (M) – (C)	↕	–	–	↕	↕	↕	IMM DIR EXT IX2 IX1 IX SP2 SP1	A2 B2 C2 D2 E2 F2 9ED2 9EE2	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
MUL	Unsigned multiply	X:A ← (X) × (A)	–	0	–	–	–	0	INH	42		5
DIV	Divide	A ← (H:A)÷(X) H ← Remainder	–	–	–	–	↕	↕	INH	52		6

The ADD instructions add the value in A to a memory operand and store the result in A. ADC adds the value in A, plus the carry bit from a previous operation, to a memory operand and stores the result in A. This operation allows performance of multibyte additions as demonstrated by the following example.

```
; add 8-bit operand to 24-bit sum
             lda    oprA          ;8-bit operand to A
             add    sum24+2       ;LS byte of 24-bit sum
             sta    sum24+2       ;update LS byte
             lda    sum24+1       ;middle byte of 24-bit sum
             adc    #0            ;propigate any carry
             sta    sum24+1       ;update middle byte
             lda    sum24         ;get MS byte of 24-bit sum
             adc    #0            ;propigate carry into MS byte
             sta    sum24         ;update MS byte
```

The AIX instruction adds a signed 8-bit value to the 16-bit H:X index register pair and stores the result back into H:X. Unlike other arithmetic instructions, AIX does not affect the CCR bits.

```
             ldhx   #tblOfStruct  ;H:X pointing at first struct
; aix to update pointer into table of 5-byte structures
             aix    #5            ;point to next 5-byte struct
```

The SUB instructions subtract a memory operand from the value in A and store the result in A. The carry status bit acts as a borrow indicator for this subtraction. SBC subtracts a memory operand and the carry bit from a previous operation from the value in A and stores the result back in A. This operation allows performance of multibyte subtractions as demonstrated by the following example.

```
; 16-bit subtract... result16 = oprE - oprF
             lda    oprE+1        ;low half of oprE
             sub    oprF+1        ;oprE(lo) - oprF(lo)
             sta    result16+1    ;low half of result
             lda    oprE          ;high half of oprE
             sbc    oprF          ;oprE(hi) - oprF(hi) - borrow
             sta    result16      ;high half of result
```

MUL multiplies the unsigned 8-bit value in X by the unsigned 8-bit value in A and stores the 16-bit result in X:A where the upper eight bits of the result are stored in X and the lower eight bits of the result are in A. There is no possibility of a carry (or overflow) since the result will always fit into X:A, so C is cleared after this operation.

DIV divides the 16-bit unsigned value in H:A by the 8-bit unsigned value in X and stores the 8-bit result in A and the 8-bit remainder in H. The divisor in X is left unchanged so it could be used in later calculations. Z indicates whether the result was zero, and C indicates whether there was an attempt to divide by zero or if there was an overflow. An overflow will occur if the result was greater than 255.

This first divide example shows a simple 8-bit by 8-bit integer divide to get an 8-bit result.

```
; divide examples
; 8/8 integer divide... A = A/X
             clrh                 ;clear MS byte of dividend
             lda    divid8        ;load 8-bit dividend
             ldx    divisor       ;load divisor
             div                  ;H:A/X -> A, remainder -> H
             sta    quotient8     ;save result
```

The second divide example demonstrates how to use DIV to perform an 8-bit by 8-bit divide and another DIV to resolve the remainder into a fractional result (eight more places to the right of the radix point).

```
; 8/8 integer divide, resolve remainder to 8 fractional bits...
; r8.f8 = A/X, remainder resolved into 8-bit binary fraction
; 16-bit result -> (8-bit integer result).(8-bit fraction)
             clrh                 ;clear MS byte of dividend
             lda    divid8        ;load 8-bit dividend
             ldx    divisor       ;load divisor
             div                  ;H:A/X -> A, remainder -> H
             sta    quotient16    ;upper integer part of result
             clra                 ;H:A = remainder:0
             div                  ;H:A/X -> A
             sta    quotient16+1  ;lower fractional part
```

In the third divide example, we divide an 8-bit dividend by a larger 8-bit divisor to get a 16-bit fractional result where the radix point is just left of the MSB of the result. In a binary fraction, the MSB has a weight of one-half, the next bit to the right has a weight of one-fourth, and so on.

```
; 8/8 fractional divide, 16-bit fractional result
; .r16 = H/X, result is a 16-bit binary fraction
; radix assumed to be in same position for H and X
; 16-bit result -> .(16-bit fraction)
; divid8 and divisor defined so H & X both loaded with one ldhx
             clra                 ;clear LS byte of dividend
             ldhx   divid8        ;H:X = dividend:divisor
             div                  ;H:A/X -> A, remainder -> H
             sta    quotient16    ;upper byte of result
             clra                 ;H:A = remainder:0
             div                  ;H:A/X -> A
             sta    quotient16+1  ;next 8 bits of result
```

The fourth divide example uses a technique like long division to do an unbounded 16-bit by 8-bit integer divide.

```
; unbounded 16/8 integer divide (equivalent to long division)
; r16.f8 = H:A/X, result is 16-bit int.8-bit binary fraction
             clrh                 ;clear MS byte of dividend
             lda    divid16       ;upper byte of dividend
             ldx    divisor       ;load divisor
             div                  ;H:A/X -> A, remainder -> H
             sta    quotient24    ;upper byte of result
             lda    divid16+1     ;H:A = remainder:dividend(lo)
             div                  ;H:A/X -> A, remainder -> H
             sta    quotient24+1  ;next byte of result
             clra                 ;H:A = remainder:0
             div                  ;H:A/X -> A
             sta    quotient24+2  ;fractional bits of result
```

The fifth divide example demonstrates a 16-bit by 8-bit divide with overflow checking.

```
; bounded 16/8 integer divide (with overflow checking)
; r8 = H:A/X, result is 8-bit integer
             ldhx   divid16       ;H:X = 16-bit dividend
             txa                  ;H:A = 16-bit dividend
             ldx    divisor       ;X = 8-bit divisor
             div                  ;H:A/X -> A, remainder -> H
             bcs    divOvrflow    ;Overflow?
             sta    quotient8     ;upper byte of result

divOvrflow:                       ;here on overflow
```

Table 6-4. Other Arithmetic Instructions

Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
INC *opr8a* INCA INCX INC *oprx8*,X INC ,X INC *oprx8*,SP	Increment	M ← (M) + $01 A ← (A) + $01 X ← (X) + $01 M ← (M) + $01 M ← (M) + $01 M ← (M) + $01	↕	–	–	↕	↕	–	DIR INH INH IX1 IX SP1	3C 4C 5C 6C 7C 9E6C	dd ff ff	5 1 1 5 4 6
DEC *opr8a* DECA DECX DEC *oprx8*,X DEC ,X DEC *oprx8*,SP	Decrement	M ← (M) – $01 A ← (A) – $01 X ← (X) – $01 M ← (M) – $01 M ← (M) – $01 M ← (M) – $01	↕	–	–	↕	↕	–	DIR INH INH IX1 IX SP1	3A 4A 5A 6A 7A 9E6A	dd ff ff	5 1 1 5 4 6
CLR *opr8a* CLRA CLRX CLRH CLR *oprx8*,X CLR ,X CLR *oprx8*,SP	Clear	M ← $00 A ← $00 X ← $00 H ← $00 M ← $00 M ← $00 M ← $00	0	–	–	0	1	–	DIR INH INH INH IX1 IX SP1	3F 4F 5F 8C 6F 7F 9E6F	dd ff ff	5 1 1 1 5 4 6
NEG *opr8a* NEGA NEGX NEG *oprx8*,X NEG ,X NEG *oprx8*,SP	Negate (Two's Complement)	M ← – (M) = $00 – (M) A ← – (A) = $00 – (A) X ← – (X) = $00 – (X) M ← – (M) = $00 – (M) M ← – (M) = $00 – (M) M ← – (M) = $00 – (M)	↕	–	–	↕	↕	↕	DIR INH INH IX1 IX SP1	30 40 50 60 70 9E60	dd ff ff	5 1 1 5 4 6
CMP *#opr8i* CMP *opr8a* CMP *opr16a* CMP *oprx16*,X CMP *oprx8*,X CMP ,X CMP *oprx16*,SP CMP *oprx8*,SP	Compare Accumulator with Memory	(A) – (M) (CCR Updated But Operands Not Changed)	↕	–	–	↕	↕	↕	IMM DIR EXT IX2 IX1 IX SP2 SP1	A1 B1 C1 D1 E1 F1 9ED1 9EE1	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
CPHX *opr16a* CPHX *#opr16i* CPHX *opr8a* CPHX *oprx8*,SP	Compare Index Register (H:X) with Memory	(H:X) – (M:M + $0001) (CCR Updated But Operands Not Changed)	↕	–	–	↕	↕	↕	EXT IMM DIR SP1	3E 65 75 9EF3	hh ll jj kk dd ff	6 3 5 6
CPX *#opr8i* CPX *opr8a* CPX *opr16a* CPX *oprx16*,X CPX *oprx8*,X CPX ,X CPX *oprx16*,SP CPX *oprx8*,SP	Compare X (Index Register Low) with Memory	(X) – (M) (CCR Updated But Operands Not Changed)	↕	–	–	↕	↕	↕	IMM DIR EXT IX2 IX1 IX SP2 SP1	A3 B3 C3 D3 E3 F3 9ED3 9EE3	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
TST *opr8a* TSTA TSTX TST *oprx8*,X TST ,X TST *oprx8*,SP	Test for Negative or Zero	(M) – $00 (A) – $00 (X) – $00 (M) – $00 (M) – $00 (M) – $00	0	–	–	↕	↕	–	DIR INH INH IX1 IX SP1	3D 4D 5D 6D 7D 9E6D	dd ff ff	4 1 1 4 3 5
DAA	Decimal Adjust Accumulator After ADD or ADC of BCD Values	$(A)_{10}$	U	–	–	↕	↕	↕	INH	72		1

6.5.2.2 Increment, Decrement, Clear, and Negate

Increment and decrement instructions let you adjust the value in A, X, or a memory location by one. Clear instructions let you force an 8-bit value in A, X, H, or a memory location to zero.

Negate instructions perform a two's complement operation that is equivalent to multiplying a signed 8-bit value by negative one. Functionally, this instruction inverts all the bits in A, X, or the memory location and then adds one. The value $80 represents the signed number –128. The negative of this value would be +128, but the largest positive number that can be represented with a two's complement, 8-bit number is +127. If A was $80 and you execute a NEGA instruction, the CPU first inverts all the bits to get $7F and then adds one to get $80. Since this causes the sign to change from positive to negative, the V bit in the CCR is set to indicate the error.

6.5.2.3 Compare and Test

CMP instructions affect CCR bits exactly like the corresponding SUB instruction, but the result is not stored back into the accumulator so A and the memory operand are left unchanged. Compare instructions compare the contents of A, X, or the H:X register pair to a memory operand. In the case of CPHX, M is the address of the referenced memory location, H corresponds to memory location M, and X corresponds to memory location M+1. CPHX performs a 16-bit subtraction (without storing the result back to H:X).

The test instructions are equivalent to subtracting zero from A, X, or a memory operand. This operation clears V and sets or clears N and Z according to what was in the tested value. The tested value is not changed.

6.5.2.4 BCD Arithmetic

In a binary coded decimal (BCD) number, one hexadecimal digit (4 binary bits) represents a single decimal number from 0 to 9. When two 8-bit BDC numbers are added, the CPU actually does a normal binary addition. Depending on the BCD values involved, this could result in a value that is no longer a valid 2-digit BCD number. Based on the H and

C condition code bits that resulted from an ADD or ADC instruction involving two legal BCD numbers, the decimal adjust A (DAA) instruction “corrects” the result to the proper BCD result and sets or clears the C bit as needed to reflect the result of the BCD addition. In the past, this was done with a relatively complex set of instructions that tested the values of each BCD digit of the result and the H and C bits. The DAA instruction greatly simplifies this operation.

The following examples demonstrate two of the possible cases that can result from adding 8-bit BDC numbers and the actions taken by a DAA instruction to correct the results to the appropriate BCD result and carry flag. The first example shows a BCD addition that does not require adjustment. The second example shows a case where the result was not a legal BCD value and the carry did not reflect the correct BCD result. In this second example, the DAA instruction adds a correction factor and adjusts the carry flag to reflect the correct BCD result.

```
             lda    #$11          ;BCD 11
             add    #$22          ;11 + 22 = 33
             daa                  ;no adjustment in this case
```

```
             LDA      #$59          ;BCD 59
             ADD      #$57          ;59 + 57 = $B0
; C=0, H=1, A=$B0 - wanted 59 + 57 = 116 or A=$16 with carry set
             DAA                    ;adds $66 and sets carry
; $B0 + $66 = $16 with carry bit set
```

6.5.3 Logical Operation Instructions

These instructions perform eight bitwise Boolean operations in parallel. For the complement instruction, each bit of the register or memory operand is inverted. The other logical instructions involve two operands, one in the accumulator (A) and the other in memory. Immediate, direct, extended, or indexed (relative to H:X or SP) addressing modes may be used to access the memory operand. Each bit of the accumulator is ANDed, ORed, or exclusive-ORed with the corresponding bit of the memory operand. The result of the logical operation is stored into the accumulator, overwriting the original operand.

Table 6-5. Logical Operation Instructions

Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
AND *#opr8i* AND *opr8a* AND *opr16a* AND *oprx16*,X AND *oprx8*,X AND ,X AND *oprx16*,SP AND *oprx8*,SP	Logical AND	A ← (A) & (M)	0	–	–	↕	↕	–	IMM DIR EXT IX2 IX1 IX SP2 SP1	A4 B4 C4 D4 E4 F4 9ED4 9EE4	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
ORA *#opr8i* ORA *opr8a* ORA *opr16a* ORA *oprx16*,X ORA *oprx8*,X ORA ,X ORA *oprx16*,SP ORA *oprx8*,SP	Inclusive OR Accumulator and Memory	A ← (A) | (M)	0	–	–	↕	↕	–	IMM DIR EXT IX2 IX1 IX SP2 SP1	AA BA CA DA EA FA 9EDA 9EEA	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
EOR *#opr8i* EOR *opr8a* EOR *opr16a* EOR *oprx16*,X EOR *oprx8*,X EOR ,X EOR *oprx16*,SP EOR *oprx8*,SP	Exclusive OR Memory with Accumulator	A ← (A ⊕ M)	0	–	–	↕	↕	–	IMM DIR EXT IX2 IX1 IX SP2 SP1	A8 B8 C8 D8 E8 F8 9ED8 9EE8	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
COM *opr8a* COMA COMX COM *oprx8*,X COM ,X COM *oprx8*,SP	Complement (One's Complement)	M ← ($\overline{M}$)= $FF – (M) A ← ($\overline{A}$) = $FF – (A) X ← ($\overline{X}$) = $FF – (X) M ← ($\overline{M}$) = $FF – (M) M ← ($\overline{M}$) = $FF – (M) M ← ($\overline{M}$) = $FF – (M)	0	–	–	↕	↕	1	DIR INH INH IX1 IX SP1	33 43 53 63 73 9E63	dd ff ff	5 1 1 5 4 6
BIT *#opr8i* BIT *opr8a* BIT *opr16a* BIT *oprx16*,X BIT *oprx8*,X BIT ,X BIT *oprx16*,SP BIT *oprx8*,SP	Bit Test	(A) & (M) (CCR Updated but Operands Not Changed)	0	–	–	↕	↕	–	IMM DIR EXT IX2 IX1 IX SP2 SP1	A5 B5 C5 D5 E5 F5 9ED5 9EE5	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4

6.5.3.1 AND, OR, Exclusive-OR, and Complement

These instructions provide the basic AND, OR, exclusive-OR, and invert functions needed to perform Boolean logical functions.

```
             lda    #$0C          ;bit pattern 00001100
             and    #$0A          ;bit pattern 00001010
; result is..........$08.......................00001000

             lda    #$35          ;bit pattern 00110101
             and    #$0F          ;bit pattern 00001111
; result is..........$05.......................00000101
```

You may notice some similarity between the AND operation and the BCLR instruction. However, BCLR can be used only on memory locations $0000–$00FF and can clear only one bit at a time while AND can clear any combination of bits and may be used with several different addressing modes to identify the memory operand to be ANDed with A.

```
             lda    #$0C          ;bit pattern 00001100
             ora    #$0A          ;bit pattern 00001010
; result is..........$0E.......................00001110
```

You may notice some similarity between the ORA operation and the BSET instruction; however, BSET can be used only on memory locations $0000–$00FF and can set only one bit at a time while ORA can set any combination of bits and may be used with several different addressing modes to identify the memory operand to be ORed with A.

Exclusive-OR can be used to toggle bits in an operand. One operand is considered a mask where each bit that is set in the mask corresponds to a bit value in the other operand that will be toggled (inverted). The next example reads an I/O port, exclusive-ORs it with an immediate mask value of $03 to toggle the two least significant bits, and then writes the updated result to the I/O port.

```
C162 A6 0C                  lda    #$0C          ;bit pattern 00001100
C164 A8 0A                  eor    #$0A          ;bit pattern 00001010
                 ; result is..........$06.......................00000110

C166 B6 00                  lda    PTAD          ;read I/O port A
C168 A8 03                  eor    #$03          ;inverts 2 LSBs
C16A B7 00                  sta    PTAD          ;update I/O port A
```

Complement instructions simply invert each bit of the operand. Don't confuse this with the negate instruction which performs the arithmetic operation equivalent to multiplication by minus one.

```
             lda    #$C5          ;bit pattern 11000101
             coma                 ;result is   00111010
```

6.5.3.2 BIT Instruction

The BIT instruction ANDs each bit of A with the corresponding bit of the addressed memory operand (just like AND), but the result is not stored to the accumulator. The N and Z condition codes are set or cleared according to the results of the AND operation to allow conditional branches after the BIT instruction. If you load A with a mask value where each bit that is set in the mask corresponds to a bit in the memory operand to be tested, then execute a BIT instruction, the Z bit will be set if none of the tested bits were 1s.

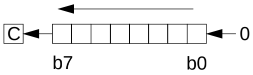
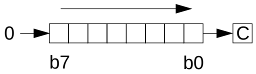
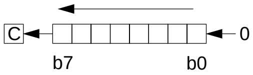
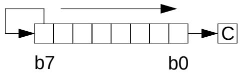
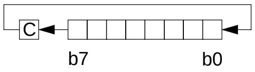
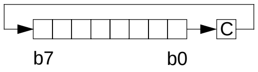
```
             lda    SCI1S1      ;read SCI status register
             bit    #(mOR+mNF+mFE+mPF) ;mask of all error flags
             bne    sciError    ;branch if any flags set
; A still contains undisturbed status register

sciError:                         ;here if any error flags
```

6.5.4 Shift and Rotate Instructions

All of the shift and rotate instructions operate on a 9-bit field consisting of an 8-bit value in A, X, or a memory location and the C bit in the CCR. Drawings are provided in the instruction descriptions to show where the C bit fits into the shift or rotate operation. The logical shift instructions are simple shifts which shift a zero into the first bit of the value and shift the last bit into the carry bit. The arithmetic shifts treat the value to be shifted as a signed two’s complement number. An arithmetic shift left is like multiplying a value by 2 and an arithmetic shift right is like dividing the number by 2. The arithmetic shift right (ASR) instruction copies the original most significant bit (MSB) back into the MSB to preserve the sign of the operand. ASL and LSL are just two different mnemonics for the same instruction because there is no functional difference between the logical and arithmetic shifts to the left.

Table 6-6. Shift and Rotate Instructions

Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
LSL *opr8a* LSLA LSLX LSL *oprx8*,X LSL ,X LSL *oprx8*,SP	Logical Shift Left (Same as ASL)		↕	–	–	↕	↕	↕	DIR INH INH IX1 IX SP1	38 48 58 68 78 9E68	dd ff ff	5 1 1 5 4 6
LSR *opr8a* LSRA LSR*X* LSR *oprx8*,X LSR ,X LSR *oprx8*,SP	Logical Shift Right		↕	–	–	0	↕	↕	DIR INH INH IX1 IX SP1	34 44 54 64 74 9E64	dd ff ff	5 1 1 5 4 6
ASL *opr8a* ASLA ASLX ASL *oprx8*,X ASL ,X ASL *oprx8*,SP	Arithmetic Shift Left (Same as LSL)		↕	–	–	↕	↕	↕	DIR INH INH IX1 IX SP1	38 48 58 68 78 9E68	dd ff ff	5 1 1 5 4 6
ASR *opr8a* ASRA ASRX ASR *oprx8*,X ASR ,X ASR *oprx8*,SP	Arithmetic Shift Right		↕	–	–	↕	↕	↕	DIR INH INH IX1 IX SP1	37 47 57 67 77 9E67	dd ff ff	5 1 1 5 4 6
ROL *opr8a* ROLA ROLX ROL *oprx8*,X ROL ,X ROL *oprx8*,SP	Rotate Left through Carry		↕	–	–	↕	↕	↕	DIR INH INH IX1 IX SP1	39 49 59 69 79 9E69	dd ff ff	5 1 1 5 4 6
ROR *opr8a* RORA RORX ROR *oprx8*,X ROR ,X ROR *oprx8*,SP	Rotate Right through Carry		↕	–	–	↕	↕	↕	DIR INH INH IX1 IX SP1	36 46 56 66 76 9E66	dd ff ff	5 1 1 5 4 6

Including the carry bit in the shifts and rotates allows extension of these operations to multibyte values. The following examples show a 24-bit value being shifted either right or left.

```
; 24-bit left shift
             clc                  ;clear C bit
; initial condition sum24 = hhhh hhhh : mmmm mmmm : llll llll : 0
             lsl    sum24+2       ;C to LSB of low byte
; now sum24 = hhhh hhhh : mmmm mmmm : C=l(7) : llll lll0
             rol    sum24+1       ;rotate middle byte
; now sum24 = hhhh hhhh : C=m(7) : mmmm mmml : llll lll0
             rol    sum24         ;rotate high byte
; now sum24 = C=h(7) : hhhh hhhm : mmmm mmml : llll lll0
```

```
; 24-bit right shift
             clc                  ;clear C bit
; initial condition sum24 = 0 : hhhh hhhh : mmmm mmmm : llll llll
             lsr    sum24         ;C to MSB of high byte
; now sum24 = 0hhh hhhh : C=h(0) : mmmm mmmm : llll llll
             rol    sum24+1       ;rotate middle byte
; now sum24 = 0hhh hhhh : hmmm mmmm : C=m(0) : llll lll0
             rol    sum24+2       ;rotate low byte
; now sum24 = 0hhh hhhm : hmmm mmmm : mlll llll : C=l(0)
```

Figure 6-5. Multibyte Shifts

6.5.5 Jump, Branch, and Loop Control Instructions

The instructions in this group cause a change of flow which means that the CPU loads a new address into the program counter so program execution continues at a location other than the next memory location after the current instruction.

Jump instructions cause an unconditional change in the execution sequence to a new location in a program. Branch and loop control instructions cause a conditional change in the execution sequence. Branch and loop control instructions use relative addressing mode to conditionally branch to a location that is relative to the location of the branch. Processor status indicators in the CCR control whether a conditional branch or loop control instruction will branch to a new address or simply continue to the next instruction in the program. BRA is a special case because the branch *always* occurs and BRN is special because the branch is *never* taken (this is functionally equivalent to a 2-byte, 3-cycle NOP). BIL and BIH are special because they use the state of the IRQ pin rather than the condition of a bit(s) in the CCR to decide whether to branch.

Table 6-7. Jump and Branch Instructions

Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
JMP *opr8a* JMP *opr16a* JMP *oprx16*,X JMP *oprx8*,X JMP ,X	Jump	PC ← Jump Address	–	–	–	–	–	–	DIR EXT IX2 IX1 IX	BC CC DC EC FC	dd hh ll ee ff ff	3 4 4 3 3
BRA *rel*	Branch Always	No Test	–	–	–	–	–	–	REL	20	rr	3
BRN *rel*	Branch Never	Uses 3 Bus Cycles	–	–	–	–	–	–	REL	21	rr	3
BEQ *rel*	Branch if Equal	Branch if (Z) = 1	–	–	–	–	–	–	REL	27	rr	3
BNE *rel*	Branch if Not Equal	Branch if (Z) = 0	–	–	–	–	–	–	REL	26	rr	3
BCC *rel*	Branch if Carry Bit Clear	Branch if (C) = 0	–	–	–	–	–	–	REL	24	rr	3
BCS *rel*	Branch if Carry Bit Set (Same as BLO)	Branch if (C) = 1	–	–	–	–	–	–	REL	25	rr	3
BPL *rel*	Branch if Plus	Branch if (N) = 0	–	–	–	–	–	–	REL	2A	rr	3
BMI *rel*	Branch if Minus	Branch if (N) = 1	–	–	–	–	–	–	REL	2B	rr	3
BIL *rel*	Branch if IRQ Pin Low	Branch if IRQ pin = 0	–	–	–	–	–	–	REL	2E	rr	3
BIH *rel*	Branch if IRQ Pin High	Branch if IRQ pin = 1	–	–	–	–	–	–	REL	2F	rr	3
BMC *rel*	Branch if Interrupt Mask Clear	Branch if (I) = 0	–	–	–	–	–	–	REL	2C	rr	3
BMS *rel*	Branch if Interrupt Mask Set	Branch if (I) = 1	–	–	–	–	–	–	REL	2D	rr	3
BHCC *rel*	Branch if Half Carry Bit Clear	Branch if (H) = 0	–	–	–	–	–	–	REL	28	rr	3
BHCS *rel*	Branch if Half Carry Bit Set	Branch if (H) = 1	–	–	–	–	–	–	REL	29	rr	3
BLT *rel*	Branch if Less Than (Signed Operands)	Branch if (N ⊕ V) = 1	–	–	–	–	–	–	REL	91	rr	3
BLE *rel*	Branch if Less Than or Equal To (Signed Operands)	Branch if (Z) | (N ⊕ V) = 1	–	–	–	–	–	–	REL	93	rr	3
BGE *rel*	Branch if Greater Than or Equal To (Signed Operands)	Branch if (N ⊕ V) = 0	–	–	–	–	–	–	REL	90	rr	3
BGT *rel*	Branch if Greater Than (Signed Operands)	Branch if (Z) | (N ⊕ V) = 0	–	–	–	–	–	–	REL	92	rr	3
BLO *rel*	Branch if Lower (Same as BCS)	Branch if (C) = 1	–	–	–	–	–	–	REL	25	rr	3
BLS *rel*	Branch if Lower or Same	Branch if (C) | (Z) = 1	–	–	–	–	–	–	REL	23	rr	3
BHS *rel*	Branch if Higher or Same (Same as BCC)	Branch if (C) = 0	–	–	–	–	–	–	REL	24	rr	3
BHI *rel*	Branch if Higher	Branch if (C) | (Z) = 0	–	–	–	–	–	–	REL	22	rr	3

6.5.5.1 Unconditional Jump and Branch

Jump (JMP), branch always (BRA), and branch never (BRN) are unconditional and do not depend on the state of any CCR bits. Jump may be used to go to any memory location in the 64-Kbyte address space while branch instructions are limited to destinations within –128 to +127 locations from the address immediately after the branch offset byte.

The following example illustrates the use of a JMP instruction to extend the range of a conditional branch. For every conditional branch instruction there is another branch that uses the opposite condition. For example the opposite of a branch if equal (BEQ) instruction is the branch if not equal (BNE) instruction. Suppose you wrote the instruction:

```
;            beq    farAway       ;more than 128 locs away
```

and the assembler flagged an error because farAway was more than 128 locations away. You can replace the BEQ with a BNE that branches around a jump instruction like this:

```
             bne    aroundJ       ;skip if NOT equal
             jmp    farAway       ;jump if equal
aroundJ:                          ;here if not equal
```

6.5.5.2 Simple Branches

The simple branches only depend on the state of a single condition (a CCR bit or the IRQ pin state).

Table 6-8. Simple Branch Summary

Branch Condition	Branch if True	Branch if False
Z	BEQ	BNE
C	BCS	BCC
N	BMI	BPL
IRQ pin	BIH	BIL
I	BMS	BMC
H	BHCS	BHCC

6.5.5.3 Signed Branches

Branch if less than (BLT), branch if less than or equal (BLE), branch if greater than or equal (BGE), and branch if greater than (BGT) are used after operations involving signed numbers. The simple branches, branch if equal (BEQ), and branch if not equal (BNE) can also be used after operations involving signed numbers.

The M68HC05 Family did not implement the V bit in the CCR, so it could not do signed branches. The difference between signed and unsigned branches is that the signed branches use the exclusive-OR of N and V in place of the C bit which is used in the Boolean equations that control the unsigned branches. The exclusive-OR of N and V provides an indication of overflow above +127 (+32,767) or borrow below –128 (–32,768). The C bit indicates overflow beyond +255 (+65,535).

6.5.5.4 Unsigned Branches

Branch if lower (BLO), branch if lower or same (BLS), branch if higher or same (BHS), and branch if higher (BHI) are used after operations involving unsigned numbers. The simple branches, branch if equal (BEQ) and branch if not equal (BNE), can also be used after operations involving unsigned numbers.

Table 6-9. Bit Branches and Loop Control

Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
BRCLR *n*,*opr8a*,*rel*	Branch if Bit *n* in Memory Clear	Branch if (Mn) = 0	–	–	–	–	–	↕	DIR (b0) DIR (b1) DIR (b2) DIR (b3) DIR (b4) DIR (b5) DIR (b6) DIR (b7)	01 03 05 07 09 0B 0D 0F	dd rr dd rr dd rr dd rr dd rr dd rr dd rr dd rr	5 5 5 5 5 5 5 5
BRSET *n*,*opr8a*,*rel*	Branch if Bit *n* in Memory Set	Branch if (Mn) = 1	–	–	–	–	–	↕	DIR (b0) DIR (b1) DIR (b2) DIR (b3) DIR (b4) DIR (b5) DIR (b6) DIR (b7)	00 02 04 06 08 0A 0C 0E	dd rr dd rr dd rr dd rr dd rr dd rr dd rr dd rr	5 5 5 5 5 5 5 5
CBEQ *opr8a*,*rel* CBEQA #*opr8i*,*rel* CBEQX #*opr8i*,*rel* CBEQ *oprx8*,X+,*rel* CBEQ ,X+,*rel* CBEQ *oprx8*,SP,*rel*	Compare and Branch if Equal	Branch if (A) = (M) Branch if (A) = (M) Branch if (X) = (M) Branch if (A) = (M) Branch if (A) = (M) Branch if (A) = (M)	–	–	–	–	–	–	DIR IMM IMM IX1+ IX+ SP1	31 41 51 61 71 9E61	dd rr ii rr ii rr ff rr rr ff rr	5 4 4 5 5 6
DBNZ *opr8a*,*rel* DBNZA *rel* DBNZX *rel* DBNZ *oprx8*,X,*rel* DBNZ ,X,*rel* DBNZ *oprx8*,SP,*rel*	Decrement and Branch if Not Zero	Decrement A, X, or M Branch if (result) ≠ 0 DBNZX Affects X Not H	–	–	–	–	–	–	DIR INH INH IX1 IX SP1	3B 4B 5B 6B 7B 9E6B	dd rr rr rr ff rr rr ff rr	7 4 4 7 6 8

6.5.5.5 Bit Condition Branches

These branch instructions test a single bit in a memory operand in direct addressing space ($0000–$00FF) and BRSET branches if the tested bit is set while BRCLR branches if the bit was clear. Although this seems like a limited number of locations, it includes all of the I/O and control register space and a significant portion of the RAM where program variables may be located. By having separate opcodes for each bit position, these instructions are particularly efficient, requiring only three bytes of object code and five bus cycles.

```
waitRDRF:    brclr  RDRF,SCI1S1,waitRDRF ;loop till RDRF set

             brclr  OneSecond,flags,skipUpdate
updateTime:  bclr   OneSecond,flags ;acknowledge one sec flag

skipUpdate:
```

6.5.5.6 Loop Control

The CBEQ instructions compare the contents of the accumulator to a memory location and branch if they are equal to each other. CBEQA and CBEQX allow A or X to be compared against an immediate operand. The H:X-relative indexed versions of CBEQ automatically increment H:X after comparing A to the indexed memory location. These variations can be used to check through a list of values in memory looking for a particular value such as a null at the end of a string, a carriage return, or an end-of-file mark. The other variations of CBEQ allow a memory location to be used as a loop counter. (The incrementing or decrementing of this loop count would be performed by other instructions in the loop.)

```
             lda    #$0D          ;ASCII <cr>
             cbeq   oprA,gotCR    ;skip if oprA=$0D
; here if oprA is anything but <cr>

gotCR:                            ;here if oprA was <cr>

; similar but IMM addr mode instead of DIR
             lda    SCI1DRL       ;read SCI character
             cbeqa  #$0D,gotCR    ;branch if it was <cr>
```

Other examples showing the CBEQ instruction can be found in **6.3.6.2 Indexed, No Offset with Post Increment (IX+)** and **6.3.6.5 Indexed, 16-Bit Offset (IX2)**.

The DBNZ instructions decrement A, X, or a memory location and then branch if the decremented value is still not zero. This provides an efficient way to implement a loop counter.

```
             lda    #4            ;loop count
             sta    directByte    ;save in RAM

loopTop:     nop                  ;start of program loop

             dbnz   directByte,loopTop ;loop directByte times
```

```
; use local on stack for loop count
             lda    #4            ;loop count
             psha                 ;put loop count on stack

loopTop1:    nop                  ;start of program loop

             dbnz   1,sp,loopTop1 ;loop directByte times
```

6.5.6 Stack-Related Instructions

Table 6-10. Stack-Related Instructions

Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
RSP	Reset Stack Pointer	SP ← $FF (High Byte Not Affected)	–	–	–	–	–	–	INH	9C		1
TXS	Transfer Index Reg. to SP	SP ← (H:X) – $0001	–	–	–	–	–	–	INH	94		2
TSX	Transfer SP to Index Reg.	H:X ← (SP) + $0001	–	–	–	–	–	–	INH	95		2
JSR *opr8a* JSR *opr16a* JSR *oprx16*,X JSR *oprx8*,X JSR ,X	Jump to Subroutine	PC ← (PC) + *n* (*n* = 1, 2, or 3) Push (PCL); SP ← (SP) – $0001 Push (PCH); SP ← (SP) – $0001 PC ← Unconditional Address	–	–	–	–	–	–	DIR EXT IX2 IX1 IX	BD CD DD ED FD	dd hh ll ee ff ff	5 6 6 5 5
BSR *rel*	Branch to Subroutine	PC ← (PC) + $0002 push (PCL); SP ← (SP) – $0001 push (PCH); SP ← (SP) – $0001 PC ← (PC) + *rel*	–	–	–	–	–	–	REL	AD	rr	5
RTS	Return from Subroutine	SP ← SP + $0001; Pull (PCH) SP ← SP + $0001; Pull (PCL)	–	–	–	–	–	–	INH	81		6
SWI	Software Interrupt	PC ← (PC) + $0001 Push (PCL); SP ← (SP) – $0001 Push (PCH); SP ← (SP) – $0001 Push (X); SP ← (SP) – $0001 Push (A); SP ← (SP) – $0001 Push (CCR); SP ← (SP) – $0001 I ← 1; PCH ← Interrupt Vector High Byte PCL ← Interrupt Vector Low Byte	–	–	1	–	–	–	INH	83		11
RTI	Return from Interrupt	SP ← (SP) + $0001; Pull (CCR) SP ← (SP) + $0001; Pull (A) SP ← (SP) + $0001; Pull (X) SP ← (SP) + $0001; Pull (PCH) SP ← (SP) + $0001; Pull (PCL)	↕	↕	↕	↕	↕	↕	INH	80		9
PSHA	Push Accumulator onto Stack	Push (A); SP ← (SP) – $0001	–	–	–	–	–	–	INH	87		2
PSHH	Push H (Index Register High) onto Stack	Push (H); SP ← (SP) – $0001	–	–	–	–	–	–	INH	8B		2
PSHX	Push X (Index Register Low) onto Stack	Push (X); SP ← (SP) – $0001	–	–	–	–	–	–	INH	89		2
PULA	Pull Accumulator from Stack	SP ← (SP + $0001); Pull (A)	–	–	–	–	–	–	INH	86		3
PULH	Pull H (Index Register High) from Stack	SP ← (SP + $0001); Pull (H)	–	–	–	–	–	–	INH	8A		3
PULX	Pull X (Index Register Low) from Stack	SP ← (SP + $0001); Pull (X)	–	–	–	–	–	–	INH	88		3
AIS *#opr8i*	Add Immediate Value (Signed) to Stack Pointer	SP ← (SP) + (M) M is sign extended to a 16-bit value	–	–	–	–	–	–	IMM	A7	ii	2

The reset stack pointer (RSP) instruction was included for compatibility with the earlier M6805. This instruction loads the low-order half of SP with $FF and does not affect the high-order half of SP. In the older architectures, the high half of SP was hard-wired to $00 so RSP would force SP to its reset state ($00FF). In HCS08 systems, $00FF would rarely be used as the starting point of the stack. Also, you cannot be sure the upper half would remain $00, so RSP is not usually useful in new HCS08 programs.

Transfer H:X to SP (TXS) is most commonly used to set up the initial SP value during reset initialization. Since SP points one location below where the last actual value is located on the stack, the value in H:X is decremented by one during the TXS transfer from H:X to SP. The following two instructions may be used to set SP to point to the last location in RAM which is the normal location for the stack in HCS08 systems.

```
             ldhx   #RamLast+1    ;point one past RAM
             txs                  ;SP<-(H:X-1)
```

Transfer SP to H:X (TSX) is typically used to copy the SP value into H:X so subsequent instructions can access variables on the stack with H:X-relative indexed addressing instructions which are slightly more efficient than SP-relative indexed instructions. Because SP points at the next available location on the stack, the value is automatically incremented by one during the transfer so H:X points at the most recently stacked byte of information on the stack after the TSX transfer.

Jump-to-subroutine (JSR) and branch-to-subroutine (BSR) instructions are used to go to a sequence of instructions (a subroutine) somewhere else in a program. Normally, at the end of the subroutine, a return-from-subroutine (RTS) instruction causes the CPU to return to the next instruction after the JSR or BSR that called the subroutine.

The software interrupt (SWI) instruction is similar to a JSR except that the X, A, and CCR registers are saved on the stack in addition to the return PC address, and, rather than specifying a subroutine address as part of the instruction, the interrupt service routine address is fetched from an interrupt vector near the end of memory. In the case of SWI, the vector is located at $FFFC and $FFFD.

The more detailed sequence of events for the SWI is:

1. PC is advanced to the next location after the SWI opcode (this is the return address.)
2. Push PCL — Store PC (low byte) at location pointed to by SP and then decrement SP.
3. Push PCH.
4. Push X, A, and CCR in that order — At the end of this sequence the SP points at the next address below where the CCR was pushed.
5. Set I bit in CCR so interrupts are disabled while executing the interrupt service routine.
6. Load PCH from $FFFC — Fetch high byte of the address for the interrupt service routine.
7. Load PCL from $FFFD.
8. Go to the address that was fetched from $FFFC:FFFD.

For compatibility with the earlier M68HC05, the H register is not automatically stacked. It is good practice to manually push H at the beginning of the interrupt service routine and to pull H just before returning from the interrupt service routine.

Other hardware interrupts cause the CPU to execute the same sequence of micro-instructions as the SWI except that each hardware interrupt source has a different interrupt vector which holds the address of the interrupt service routine.

Normally, the last instruction in an interrupt service routine is a return from interrupt (RTI). RTI restores the CCR, A, X, PCH, and PCL in the opposite order that they were saved on the stack. As each byte is pulled from the stack, SP is incremented by one to point at the data to be pulled and the appropriate register is loaded from the address pointed to by SP. After executing RTI, the program resumes at the return address that was just pulled off the stack during the RTI.

The interrupt mask (I bit in the CCR) is set during entry to the interrupt just after the CCR is stacked. During the RTI, the pre-interrupt value of the CCR is restored which typically restores the I bit to 0 to allow new interrupts.

Push A (PSHA), push X (PSHX), and push H (PSHH) allow individual CPU registers to be saved on the stack. The push operation stores the selected register in memory where SP is pointing and then decrements SP so it points at the next available location on the stack. Pull A, X, and H (PULA, PULX, and PULH) allow A, X, or H to be loaded with data from the stack. The pull operation first increments SP and then loads the selected register with the contents of the memory location pointed to by SP.

The following example shows one use of pushes and pulls. Some C compilers use X:A to pass a 16-bit parameter to a function. This code segment shows how this integer value is saved on the stack (lines 604 and 605) and then later gets loaded into H:X (line 620) where it can be used as an index pointer. Notice that you can push one register (line 605) and then pull that value into a different register. (Nothing about the value on the stack associates it with a particular CPU register.)

```
               *********************
               * multAcc - 4 iteration mutiply-accumulate example
               *********************
               ; 9 stack bytes used for this routine including return addr
               ; a pointer is passed in X:A, 3 bytes are used for stack locals,
               ; and two bytes are used for temporary storage on stack
               ; pntr points at list of 4 constant multipliers k(0) - k(3)
               ; VarY is a 16-bit integer, VarN is an 8-bit loop count
               ; VarY = sum( k(0)*oprA + k(1)*oprB + k(2)*oprC + k(3)*oprD)
               ; return result (VarY) in X:A
C1F2 87        multAcc:     psha                 ;save pntr LS byte
C1F3 89                     pshx                 ;save pntr MS byte
C1F4 A7 FD                  ais    #-3           ;allocate for 3 local bytes
               ; at this point VarN @ 1,sp; VarY(hi) @ 2,sp; VarY(lo) @ 3,sp;
               ; pntr(hi) @ 4,sp; pntr(lo) @ 5,sp
C1F6 9E6F 02                clr    2,sp          ;VarY MS byte on stack
C1F9 9E6F 03                clr    3,sp          ;VarY LS byte on stack
C1FC A6 04                  lda    #4            ;loop count
C1FE 9EE7 01                sta    1,sp          ;VarN = 4
C201 45 00A0                ldhx   #oprA         ;operands oprA-oprD
C204 F6        iteration:   lda    ,x            ;get operand(n)
C205 AF 01                  aix    #1            ;point to next operand
C207 89                     pshx                 ;MS byte of oprX pointer
C208 8B                     pshh                 ;LS byte of oprX pointer
               ; at this point VarN @ 3,sp; VarY(hi) @ 4,sp; VarY(lo) @ 5,sp;
               ; pntr(hi) @ 6,sp; pntr(lo) @ 7,sp
C209 9EFE 06                ldhx   6,sp          ;load pntr from stack (6,sp)
C20C 9E6C 07                inc    7,sp          ;pntr(lo)=pntr(lo)+1
C20F 26 03                  bne    skip          ;skip if no carry
C211 9E6C 06                inc    6,sp          ;add carry into pntr(hi)
C214 FE        skip:        ldx    ,x            ;load k(n)
C215 42                     mul                  ;A*X -> X:A
C216 9EEB 05                add    5,sp          ;add to VarY(lo)
C219 9EE7 05                sta    5,sp          ;update VarY(lo)
C21C 9F                     txa                  ;MS byte to A
C21D 8A                     pulh                 ;restore oprX pointer (hi)
C21E 88                     pulx                 ;restore oprX pointer (lo)
               ; at this point VarN @ 1,sp; VarY(hi) @ 2,sp; VarY(lo) @ 3,sp;
               ; pntr(hi) @ 4,sp; pntr(lo) @ 5,sp
C21F 9EE9 02                adc    2,sp          ;add with carry to VarY(hi)
C222 9EE7 02                sta    2,sp          ;update VarY(hi)
C225 9E6B 01 DB             dbnz   1,sp,iteration ;dec VarN and loop if not 0
C229 9EEE 02                ldx    2,sp          ;VarY(hi)
C22C 9EE6 03                lda    3,sp          ;VarY(lo)
C22F A7 05                  ais    #5            ;deallocate all locals
C231 81                     rts                  ;return VarY in X:A
               *********************
```

The add immediate to stack pointer (AIS) instruction allows an 8-bit signed immediate value to be added to SP. This is most commonly used to allocate and deallocate space on the stack for local variables. Adding a negative number to SP allocates space on the stack and adding a positive number to SP deallocates space.

```
            ais    #-5          ;allocate 5 bytes for locals
            ais    #5           ;deallocate local space
```

6.5.7 Miscellaneous Instructions

Table 6-11. Miscellaneous Instructions

Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
NOP	No Operation	Uses 1 Bus Cycle	–	–	–	–	–	–	INH	9D		1
SEC	Set Carry Bit	C ← 1	–	–	–	–	–	1	INH	99		1
CLC	Clear Carry Bit	C ← 0	–	–	–	–	–	0	INH	98		1
SEI	Set Interrupt Mask Bit	I ← 1	–	–	1	–	–	–	INH	9B		1
CLI	Clear Interrupt Mask Bit	I ← 0	–	–	0	–	–	–	INH	9A		1
BGND	Enter Active Background if ENBDM = 1	Waits For and Processes BDM Commands Until GO, TRACE1, or TAGGO	–	–	–	–	–	–	INH	82		5+
WAIT	Enable Interrupts; Wait for Interrupt	I bit ← 0; Halt CPU	–	–	0	–	–	–	INH	8F		2+
STOP	Enable Interrupts: Stop Processing Refer to MCU Documentation	I bit ← 0; Stop Processing	–	–	0	–	–	–	INH	8E		2+

The no-operation (NOP) instruction is typically used in software generated delay programs. It consumes execution time but does not cause any changes to condition code bits or other CPU registers. This example uses a software loop including a NOP to generate a 1 ms delay.

```
               *********************
               * dly1ms - delay 1ms at bus frequency = 20MHz
               *********************
               ; 1 bus cycle = 50 nanoseconds so 20,000 cycles = 1ms
               ; JSR (EXT) takes [5 or 6] cycles. Total overhead is 24-25 cycles
               ; total delay 20000 = 8n+24; so n = 19976/8 = 2497
C232 8B        dly1ms:      pshh                 ;[2] save H
C233 89                     pshx                 ;[2] save X
C234 9D                     nop                  ;[1] makes n even
C235 45 09C0                ldhx   #2496         ;[3] loop count
C238 AF FF     loop1ms:     aix    #-1           ;[2] H:X = H:X - 1
C23A 65 0000                cphx   #$0000        ;[3] check for zero
C23D 26 F9                  bne    loop1ms       ;[3] loop till H:X = $0000
C23F 88                     pulx                 ;[3] restore X
C240 8A                     pulh                 ;[3] restore H
C241 81                     rts                  ;[6] return
               *********************
```

One way the set and clear carry (SEC and CLC) instructions can be used is to force the value of the carry bit before doing a shift or rotate instruction. See **Figure 6-5** for more information.

Set interrupt mask (SEI) and clear interrupt mask (CLI) instructions are used to disable or enable interrupts, respectively. After reset, the I bit is set to prevent interrupts before the stack pointer and other system conditions have been initialized. After enough system initialization has been completed, use a CLI instruction to enable interrupts. In some programs, it is necessary to prevent interrupts during some sensitive code sequence. SEI is used before the sequence and CLI is used after the sequence to prevent interrupts during the sensitive code sequence.

The background (BGND), WAIT, and STOP instructions are unusual in that they cause the CPU to stop executing new instructions for an indefinite period of time. A hardware event, such as an interrupt or a serial background debug command, is needed to tell the CPU when it is time to resume processing normal instructions. In the instruction detail tables, these instructions are listed with a minimum number of bus cycles, followed by a + (plus) to indicate that this is the minimum number of cycles needed to complete these instructions.

BGND instructions can be used by a development system to set software breakpoints in a user program that is being debugged. Normal user programs never use the BGND instruction. When the CPU encounters a BGND instruction, it checks the ENBDM control bit in the background debug controller module. This control bit is not accessible to a user program; it can be changed only by reset or a serial background command. If ENBDM = 0 (its default state), BGND opcodes are treated as illegal instructions which cause an MCU reset. For more information about background debug mode, see **7.3 Background Debug Controller (BDC)**.

WAIT causes the CPU to shut down its clocks to save power. Other peripheral systems continue to run normally. An interrupt or reset event is needed to wake up the CPU from wait mode. The interrupt can come from the external IRQ pin or from an internal peripheral system. See **3.5 Wait Mode** for a detailed discussion of the wait mode.

STOP forces the MCU to turn off all system clocks to reduce system power to an absolute minimum. In previous M68HC05 and M68HC08 systems, all clocks including the oscillator were disabled in stop mode. Depending on the version of the clock generation circuitry in an HCS08 system, you can set control bits so the oscillator and the timebase module continue to operate in stop mode. This provides a means of waking the MCU from stop mode periodically without any external components. All clocks other than the oscillator and a small number of flip-flops in the timebase module are stopped in this mode, so system power is reduced to a bare minimum.

The HCS08 always starts out using a self-clocked clock source after reset or stop to avoid delays associated with crystal startup. After stop, the CPU starts execution by responding to the interrupt or reset event that woke it up. For more detailed information, refer to **3.6 Stop Modes**.

6.6 Summary Instruction Table

Instruction Set Summary Nomenclature

The nomenclature listed here is used in the instruction descriptions in **Table 6-1** through **Table 6-12**.

Operators

( )	=	Contents of register or memory location shown inside parentheses
←	=	Is loaded with (read: “gets”)
&	=	Boolean AND
|	=	Boolean OR
⊕	=	Boolean exclusive-OR
×	=	Multiply
÷	=	Divide
:	=	Concatenate
+	=	Add
–	=	Negate (two’s complement)

CPU registers

A	=	Accumulator
CCR	=	Condition code register
H	=	Index register, higher order (most significant) 8 bits
X	=	Index register, lower order (least significant) 8 bits
PC	=	Program counter
PCH	=	Program counter, higher order (most significant) 8 bits
PCL	=	Program counter, lower order (least significant) 8 bits
SP	=	Stack pointer

Memory and addressing

M	=	A memory location or absolute data, depending on addressing mode
M:M + $0001	=	A 16-bit value in two consecutive memory locations. The higher-order (most significant) 8 bits are located at the address of M, and the lower-order (least significant) 8 bits are located at the next higher sequential address.

Condition code register (CCR) bits

V	=	Two’s complement overflow indicator, bit 7
H	=	Half carry, bit 4
I	=	Interrupt mask, bit 3
N	=	Negative indicator, bit 2
Z	=	Zero indicator, bit 1
C	=	Carry/borrow, bit 0 (carry out of bit 7)

CCR activity notation

–	=	Bit not affected
0	=	Bit forced to 0
1	=	Bit forced to 1
↕	=	Bit set or cleared according to results of operation
U	=	Undefined after the operation

Machine coding notation

dd	=	Low-order 8 bits of a direct address $0000–$00FF (high byte assumed to be $00)
ee	=	Upper 8 bits of 16-bit offset
ff	=	Lower 8 bits of 16-bit offset or 8-bit offset
ii	=	One byte of immediate data
jj	=	High-order byte of a 16-bit immediate data value
kk	=	Low-order byte of a 16-bit immediate data value
hh	=	High-order byte of 16-bit extended address
ll	=	Low-order byte of 16-bit extended address
rr	=	Relative offset

Source form

Everything in the source forms columns, *except expressions in italic characters,* is literal information which must appear in the assembly source file exactly as shown. The initial 3- to 5-letter mnemonic is always a literal expression. All commas, pound signs (#), parentheses, and plus signs (+) are literal characters.

n — Any label or expression that evaluates to a single integer in the range 0–7

opr8i — Any label or expression that evaluates to an 8-bit immediate value

opr16i — Any label or expression that evaluates to a 16-bit immediate value

opr8a — Any label or expression that evaluates to an 8-bit value. The instruction treats this 8-bit value as the low order 8 bits of an address in the direct page of the 64-Kbyte address space ($00xx).

opr16a — Any label or expression that evaluates to a 16-bit value. The instruction treats this value as an address in the 64-Kbyte address space.

oprx8 — Any label or expression that evaluates to an unsigned 8-bit value, used for indexed addressing

oprx16 — Any label or expression that evaluates to a 16-bit value. Since the HCS08 has a 16-bit address bus, this can be either a signed or an unsigned value.

rel — Any label or expression that refers to an address that is within –128 to +127 locations from the next address after the last byte of object code for the current instruction. The assembler will calculate the 8-bit signed offset and include it in the object code for this instruction.

Address modes

INH	=	Inherent (no operands)
IMM	=	8-bit or 16-bit immediate
DIR	=	8-bit direct
EXT	=	16-bit extended
IX	=	16-bit indexed no offset
IX+	=	16-bit indexed no offset, post increment (CBEQ and MOV only)
IX1	=	16-bit indexed with 8-bit offset from H:X
IX1+	=	16-bit indexed with 8-bit offset, post increment (CBEQ only)
IX2	=	16-bit indexed with 16-bit offset from H:X
REL	=	8-bit relative offset
SP1	=	Stack pointer with 8-bit offset
SP2	=	Stack pointer with 16-bit offset

Table 6-12. Instruction Set Summary (Sheet 1 of 6)

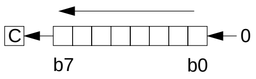
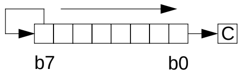
Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
ADC *#opr8i* ADC *opr8a* ADC *opr16a* ADC *oprx16*,X ADC *oprx8*,X ADC ,X ADC *oprx16*,SP ADC *oprx8*,SP	Add with Carry	A ← (A) + (M) + (C)	↕	↕	–	↕	↕	↕	IMM DIR EXT IX2 IX1 IX SP2 SP1	A9 B9 C9 D9 E9 F9 9ED9 9EE9	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
ADD *#opr8i* ADD *opr8a* ADD *opr16a* ADD *oprx16*,X ADD *oprx8*,X ADD ,X ADD *oprx16*,SP ADD *oprx8*,SP	Add without Carry	A ← (A) + (M)	↕	↕	–	↕	↕	↕	IMM DIR EXT IX2 IX1 IX SP2 SP1	AB BB CB DB EB FB 9EDB 9EEB	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
AIS *#opr8i*	Add Immediate Value (Signed) to Stack Pointer	SP ← (SP) + (M) M is sign extended to a 16-bit value	–	–	–	–	–	–	IMM	A7	ii	2
AIX *#opr8i*	Add Immediate Value (Signed) to Index Register (H:X)	H:X ← (H:X) + (M) M is sign extended to a 16-bit value	–	–	–	–	–	–	IMM	AF	ii	2
AND *#opr8i* AND *opr8a* AND *opr16a* AND *oprx16*,X AND *oprx8*,X AND ,X AND *oprx16*,SP AND *oprx8*,SP	Logical AND	A ← (A) & (M)	0	–	–	↕	↕	–	IMM DIR EXT IX2 IX1 IX SP2 SP1	A4 B4 C4 D4 E4 F4 9ED4 9EE4	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
ASL *opr8a* ASLA ASLX ASL *oprx8*,X ASL ,X ASL *oprx8*,SP	Arithmetic Shift Left (Same as LSL)		↕	–	–	↕	↕	↕	DIR INH INH IX1 IX SP1	38 48 58 68 78 9E68	dd ff ff	5 1 1 5 4 6
ASR *opr8a* ASRA ASRX ASR *oprx8*,X ASR ,X ASR *oprx8*,SP	Arithmetic Shift Right		↕	–	–	↕	↕	↕	DIR INH INH IX1 IX SP1	37 47 57 67 77 9E67	dd ff ff	5 1 1 5 4 6
BCC *rel*	Branch if Carry Bit Clear	Branch if (C) = 0	–	–	–	–	–	–	REL	24	rr	3
BCLR *n*,*opr8a*	Clear Bit n in Memory	Mn ← 0	–	–	–	–	–	–	DIR (b0) DIR (b1) DIR (b2) DIR (b3) DIR (b4) DIR (b5) DIR (b6) DIR (b7)	11 13 15 17 19 1B 1D 1F	dd dd dd dd dd dd dd dd	5 5 5 5 5 5 5 5
BCS *rel*	Branch if Carry Bit Set (Same as BLO)	Branch if (C) = 1	–	–	–	–	–	–	REL	25	rr	3
BEQ *rel*	Branch if Equal	Branch if (Z) = 1	–	–	–	–	–	–	REL	27	rr	3
BGE *rel*	Branch if Greater Than or Equal To (Signed Operands)	Branch if (N ⊕ V) = 0	–	–	–	–	–	–	REL	90	rr	3
BGND	Enter Active Background if ENBDM = 1	Waits For and Processes BDM Commands Until GO, TRACE1, or TAGGO	–	–	–	–	–	–	INH	82		5+
BGT *rel*	Branch if Greater Than (Signed Operands)	Branch if (Z) | (N ⊕ V) = 0	–	–	–	–	–	–	REL	92	rr	3

Table 6-12. Instruction Set Summary (Sheet 2 of 6)

Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
BHCC *rel*	Branch if Half Carry Bit Clear	Branch if (H) = 0	–	–	–	–	–	–	REL	28	rr	3
BHCS *rel*	Branch if Half Carry Bit Set	Branch if (H) = 1	–	–	–	–	–	–	REL	29	rr	3
BHI *rel*	Branch if Higher	Branch if (C) | (Z) = 0	–	–	–	–	–	–	REL	22	rr	3
BHS *rel*	Branch if Higher or Same (Same as BCC)	Branch if (C) = 0	–	–	–	–	–	–	REL	24	rr	3
BIH *rel*	Branch if IRQ Pin High	Branch if IRQ pin = 1	–	–	–	–	–	–	REL	2F	rr	3
BIL *rel*	Branch if IRQ Pin Low	Branch if IRQ pin = 0	–	–	–	–	–	–	REL	2E	rr	3
BIT *#opr8i* BIT *opr8a* BIT *opr16a* BIT *oprx16*,X BIT *oprx8*,X BIT ,X BIT *oprx16*,SP BIT *oprx8*,SP	Bit Test	(A) & (M) (CCR Updated but Operands Not Changed)	0	–	–	↕	↕	–	IMM DIR EXT IX2 IX1 IX SP2 SP1	A5 B5 C5 D5 E5 F5 9ED5 9EE5	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
BLE *rel*	Branch if Less Than or Equal To (Signed Operands)	Branch if (Z) | (N ⊕ V) = 1	–	–	–	–	–	–	REL	93	rr	3
BLO *rel*	Branch if Lower (Same as BCS)	Branch if (C) = 1	–	–	–	–	–	–	REL	25	rr	3
BLS *rel*	Branch if Lower or Same	Branch if (C) | (Z) = 1	–	–	–	–	–	–	REL	23	rr	3
BLT *rel*	Branch if Less Than (Signed Operands)	Branch if (N ⊕ V) = 1	–	–	–	–	–	–	REL	91	rr	3
BMC *rel*	Branch if Interrupt Mask Clear	Branch if (I) = 0	–	–	–	–	–	–	REL	2C	rr	3
BMI *rel*	Branch if Minus	Branch if (N) = 1	–	–	–	–	–	–	REL	2B	rr	3
BMS *rel*	Branch if Interrupt Mask Set	Branch if (I) = 1	–	–	–	–	–	–	REL	2D	rr	3
BNE *rel*	Branch if Not Equal	Branch if (Z) = 0	–	–	–	–	–	–	REL	26	rr	3
BPL *rel*	Branch if Plus	Branch if (N) = 0	–	–	–	–	–	–	REL	2A	rr	3
BRA *rel*	Branch Always	No Test	–	–	–	–	–	–	REL	20	rr	3
BRCLR *n*,*opr8a*,*rel*	Branch if Bit *n* in Memory Clear	Branch if (Mn) = 0	–	–	–	–	–	↕	DIR (b0) DIR (b1) DIR (b2) DIR (b3) DIR (b4) DIR (b5) DIR (b6) DIR (b7)	01 03 05 07 09 0B 0D 0F	dd rr dd rr dd rr dd rr dd rr dd rr dd rr dd rr	5 5 5 5 5 5 5 5
BRN *rel*	Branch Never	Uses 3 Bus Cycles	–	–	–	–	–	–	REL	21	rr	3
BRSET *n*,*opr8a*,*rel*	Branch if Bit *n* in Memory Set	Branch if (Mn) = 1	–	–	–	–	–	↕	DIR (b0) DIR (b1) DIR (b2) DIR (b3) DIR (b4) DIR (b5) DIR (b6) DIR (b7)	00 02 04 06 08 0A 0C 0E	dd rr dd rr dd rr dd rr dd rr dd rr dd rr dd rr	5 5 5 5 5 5 5 5
BSET *n*,*opr8a*	Set Bit *n* in Memory	Mn ← 1	–	–	–	–	–	–	DIR (b0) DIR (b1) DIR (b2) DIR (b3) DIR (b4) DIR (b5) DIR (b6) DIR (b7)	10 12 14 16 18 1A 1C 1E	dd dd dd dd dd dd dd dd	5 5 5 5 5 5 5 5

Table 6-12. Instruction Set Summary (Sheet 3 of 6)

Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
BSR *rel*	Branch to Subroutine	PC ← (PC) + $0002 push (PCL); SP ← (SP) – $0001 push (PCH); SP ← (SP) – $0001 PC ← (PC) + *rel*	–	–	–	–	–	–	REL	AD	rr	5
CBEQ *opr8a*,*rel* CBEQA #*opr8i*,*rel* CBEQX #*opr8i*,*rel* CBEQ *oprx8*,X+,*rel* CBEQ ,X+,*rel* CBEQ *oprx8*,SP,*rel*	Compare and Branch if Equal	Branch if (A) = (M) Branch if (A) = (M) Branch if (X) = (M) Branch if (A) = (M) Branch if (A) = (M) Branch if (A) = (M)	–	–	–	–	–	–	DIR IMM IMM IX1+ IX+ SP1	31 41 51 61 71 9E61	dd rr ii rr ii rr ff rr rr ff rr	5 4 4 5 5 6
CLC	Clear Carry Bit	C ← 0	–	–	–	–	–	0	INH	98		1
CLI	Clear Interrupt Mask Bit	I ← 0	–	–	0	–	–	–	INH	9A		1
CLR *opr8a* CLRA CLRX CLRH CLR *oprx8*,X CLR ,X CLR *oprx8*,SP	Clear	M ← $00 A ← $00 X ← $00 H ← $00 M ← $00 M ← $00 M ← $00	0	–	–	0	1	–	DIR INH INH INH IX1 IX SP1	3F 4F 5F 8C 6F 7F 9E6F	dd ff ff	5 1 1 1 5 4 6
CMP #*opr8i* CMP *opr8a* CMP *opr16a* CMP *oprx16*,X CMP *oprx8*,X CMP ,X CMP *oprx16*,SP CMP *oprx8*,SP	Compare Accumulator with Memory	(A) – (M) (CCR Updated But Operands Not Changed)	↕	–	–	↕	↕	↕	IMM DIR EXT IX2 IX1 IX SP2 SP1	A1 B1 C1 D1 E1 F1 9ED1 9EE1	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
COM *opr8a* COMA COMX COM *oprx8*,X COM ,X COM *oprx8*,SP	Complement (One's Complement)	$M \leftarrow (\overline{M}) = \$FF - (M)$ $A \leftarrow (\overline{A}) = \$FF - (A)$ $X \leftarrow (\overline{X}) = \$FF - (X)$ $M \leftarrow (\overline{M}) = \$FF - (M)$ $M \leftarrow (\overline{M}) = \$FF - (M)$ $M \leftarrow (\overline{M}) = \$FF - (M)$	0	–	–	↕	↕	1	DIR INH INH IX1 IX SP1	33 43 53 63 73 9E63	dd ff ff	5 1 1 5 4 6
CPHX *opr16a* CPHX #*opr16i* CPHX *opr8a* CPHX *oprx8*,SP	Compare Index Register (H:X) with Memory	(H:X) – (M:M + $0001) (CCR Updated But Operands Not Changed)	↕	–	–	↕	↕	↕	EXT IMM DIR SP1	3E 65 75 9EF3	hh ll jj kk dd ff	6 3 5 6
CPX #*opr8i* CPX *opr8a* CPX *opr16a* CPX *oprx16*,X CPX *oprx8*,X CPX ,X CPX *oprx16*,SP CPX *oprx8*,SP	Compare X (Index Register Low) with Memory	(X) – (M) (CCR Updated But Operands Not Changed)	↕	–	–	↕	↕	↕	IMM DIR EXT IX2 IX1 IX SP2 SP1	A3 B3 C3 D3 E3 F3 9ED3 9EE3	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
DAA	Decimal Adjust Accumulator After ADD or ADC of BCD Values	$(A)_{10}$	U	–	–	↕	↕	↕	INH	72		1
DBNZ *opr8a*,*rel* DBNZA *rel* DBNZX *rel* DBNZ *oprx8*,X,*rel* DBNZ ,X,*rel* DBNZ *oprx8*,SP,*rel*	Decrement and Branch if Not Zero	Decrement A, X, or M Branch if (result) ≠ 0 DBNZX Affects X Not H	–	–	–	–	–	–	DIR INH INH IX1 IX SP1	3B 4B 5B 6B 7B 9E6B	dd rr rr rr ff rr rr ff rr	7 4 4 7 6 8
DEC *opr8a* DECA DECX DEC *oprx8*,X DEC ,X DEC *oprx8*,SP	Decrement	M ← (M) – $01 A ← (A) – $01 X ← (X) – $01 M ← (M) – $01 M ← (M) – $01 M ← (M) – $01	↕	–	–	↕	↕	–	DIR INH INH IX1 IX SP1	3A 4A 5A 6A 7A 9E6A	dd ff ff	5 1 1 5 4 6
DIV	Divide	A ← (H:A)÷(X) H ← Remainder	–	–	–	–	↕	↕	INH	52		6

Table 6-12. Instruction Set Summary (Sheet 4 of 6)

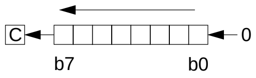
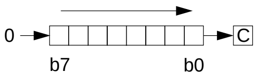
Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
EOR *#opr8i* EOR *opr8a* EOR *opr16a* EOR *oprx16*,X EOR *oprx8*,X EOR ,X EOR *oprx16*,SP EOR *oprx8*,SP	Exclusive OR Memory with Accumulator	A ← (A ⊕ M)	0	–	–	↕	↕	–	IMM DIR EXT IX2 IX1 IX SP2 SP1	A8 B8 C8 D8 E8 F8 9ED8 9EE8	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
INC *opr8a* INCA INCX INC *oprx8*,X INC ,X INC *oprx8*,SP	Increment	M ← (M) + $01 A ← (A) + $01 X ← (X) + $01 M ← (M) + $01 M ← (M) + $01 M ← (M) + $01	↕	–	–	↕	↕	–	DIR INH INH IX1 IX SP1	3C 4C 5C 6C 7C 9E6C	dd ff ff	5 1 1 5 4 6
JMP *opr8a* JMP *opr16a* JMP *oprx16*,X JMP *oprx8*,X JMP ,X	Jump	PC ← Jump Address	–	–	–	–	–	–	DIR EXT IX2 IX1 IX	BC CC DC EC FC	dd hh ll ee ff ff	3 4 4 3 3
JSR *opr8a* JSR *opr16a* JSR *oprx16*,X JSR *oprx8*,X JSR ,X	Jump to Subroutine	PC ← (PC) + *n* (*n* = 1, 2, or 3) Push (PCL); SP ← (SP) – $0001 Push (PCH); SP ← (SP) – $0001 PC ← Unconditional Address	–	–	–	–	–	–	DIR EXT IX2 IX1 IX	BD CD DD ED FD	dd hh ll ee ff ff	5 6 6 5 5
LDA *#opr8i* LDA *opr8a* LDA *opr16a* LDA *oprx16*,X LDA *oprx8*,X LDA ,X LDA *oprx16*,SP LDA *oprx8*,SP	Load Accumulator from Memory	A ← (M)	0	–	–	↕	↕	–	IMM DIR EXT IX2 IX1 IX SP2 SP1	A6 B6 C6 D6 E6 F6 9ED6 9EE6	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
LDHX *#opr16i* LDHX *opr8a* LDHX *opr16a* LDHX ,X LDHX *oprx16*,X LDHX *oprx8*,X LDHX *oprx8*,SP	Load Index Register (H:X) from Memory	H:X ← (M:M + $0001)	0	–	–	↕	↕	–	IMM DIR EXT IX IX2 IX1 SP1	45 55 32 9EAE 9EBE 9ECE 9EFE	jj kk dd hh ll ee ff ff ff	3 4 5 5 6 5 5
LDX *#opr8i* LDX *opr8a* LDX *opr16a* LDX *oprx16*,X LDX *oprx8*,X LDX ,X LDX *oprx16*,SP LDX *oprx8*,SP	Load X (Index Register Low) from Memory	X ← (M)	0	–	–	↕	↕	–	IMM DIR EXT IX2 IX1 IX SP2 SP1	AE BE CE DE EE FE 9EDE 9EEE	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
LSL *opr8a* LSLA LSLX LSL *oprx8*,X LSL ,X LSL *oprx8*,SP	Logical Shift Left (Same as ASL)		↕	–	–	↕	↕	↕	DIR INH INH IX1 IX SP1	38 48 58 68 78 9E68	dd ff ff	5 1 1 5 4 6
LSR *opr8a* LSRA LSR*X* LSR *oprx8*,X LSR ,X LSR *oprx8*,SP	Logical Shift Right		↕	–	–	0	↕	↕	DIR INH INH IX1 IX SP1	34 44 54 64 74 9E64	dd ff ff	5 1 1 5 4 6
MOV *opr8a*,*opr8a* MOV *opr8a*,X+ MOV *#opr8i*,*opr8a* MOV ,X+,*opr8a*	Move	$(M)_{destination}$ ← $(M)_{source}$ H:X ← (H:X) + $0001 in IX+/DIR and DIR/IX+ Modes	0	–	–	↕	↕	–	DIR/DIR DIR/IX+ IMM/DIR IX+/DIR	4E 5E 6E 7E	dd dd dd ii dd dd	5 5 4 5

Table 6-12. Instruction Set Summary (Sheet 5 of 6)

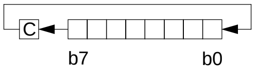
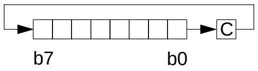
Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
MUL	Unsigned multiply	X:A ← (X) × (A)	–	0	–	–	–	0	INH	42		5
NEG *opr8a* NEGA NEGX NEG *oprx8*,X NEG ,X NEG *oprx8*,SP	Negate (Two's Complement)	M ← – (M) = $00 – (M) A ← – (A) = $00 – (A) X ← – (X) = $00 – (X) M ← – (M) = $00 – (M) M ← – (M) = $00 – (M) M ← – (M) = $00 – (M)	↕	–	–	↕	↕	↕	DIR INH INH IX1 IX SP1	30 40 50 60 70 9E60	dd ff ff	5 1 1 5 4 6
NOP	No Operation	Uses 1 Bus Cycle	–	–	–	–	–	–	INH	9D		1
NSA	Nibble Swap Accumulator	A ← (A[3:0]:A[7:4])	–	–	–	–	–	–	INH	62		1
ORA *#opr8i* ORA *opr8a* ORA *opr16a* ORA *oprx16*,X ORA *oprx8*,X ORA ,X ORA *oprx16*,SP ORA *oprx8*,SP	Inclusive OR Accumulator and Memory	A ← (A) | (M)	0	–	–	↕	↕	–	IMM DIR EXT IX2 IX1 IX SP2 SP1	AA BA CA DA EA FA 9EDA 9EEA	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
PSHA	Push Accumulator onto Stack	Push (A); SP ← (SP) – $0001	–	–	–	–	–	–	INH	87		2
PSHH	Push H (Index Register High) onto Stack	Push (H); SP ← (SP) – $0001	–	–	–	–	–	–	INH	8B		2
PSHX	Push X (Index Register Low) onto Stack	Push (X); SP ← (SP) – $0001	–	–	–	–	–	–	INH	89		2
PULA	Pull Accumulator from Stack	SP ← (SP + $0001); Pull (A)	–	–	–	–	–	–	INH	86		3
PULH	Pull H (Index Register High) from Stack	SP ← (SP + $0001); Pull (H)	–	–	–	–	–	–	INH	8A		3
PULX	Pull X (Index Register Low) from Stack	SP ← (SP + $0001); Pull (X)	–	–	–	–	–	–	INH	88		3
ROL *opr8a* ROLA ROLX ROL *oprx8*,X ROL ,X ROL *oprx8*,SP	Rotate Left through Carry		↕	–	–	↕	↕	↕	DIR INH INH IX1 IX SP1	39 49 59 69 79 9E69	dd ff ff	5 1 1 5 4 6
ROR *opr8a* RORA RORX ROR *oprx8*,X ROR ,X ROR *oprx8*,SP	Rotate Right through Carry		↕	–	–	↕	↕	↕	DIR INH INH IX1 IX SP1	36 46 56 66 76 9E66	dd ff ff	5 1 1 5 4 6
RSP	Reset Stack Pointer	SP ← $FF (High Byte Not Affected)	–	–	–	–	–	–	INH	9C		1
RTI	Return from Interrupt	SP ← (SP) + $0001; Pull (CCR) SP ← (SP) + $0001; Pull (A) SP ← (SP) + $0001; Pull (X) SP ← (SP) + $0001; Pull (PCH) SP ← (SP) + $0001; Pull (PCL)	↕	↕	↕	↕	↕	↕	INH	80		9
RTS	Return from Subroutine	SP ← SP + $0001; Pull (PCH) SP ← SP + $0001; Pull (PCL)	–	–	–	–	–	–	INH	81		6
SBC *#opr8i* SBC *opr8a* SBC *opr16a* SBC *oprx16*,X SBC *oprx8*,X SBC ,X SBC *oprx16*,SP SBC *oprx8*,SP	Subtract with Carry	A ← (A) – (M) – (C)	↕	–	–	↕	↕	↕	IMM DIR EXT IX2 IX1 IX SP2 SP1	A2 B2 C2 D2 E2 F2 9ED2 9EE2	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4

Table 6-12. Instruction Set Summary (Sheet 6 of 6)

Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
SEC	Set Carry Bit	C ← 1	–	–	–	–	–	1	INH	99		1
SEI	Set Interrupt Mask Bit	I ← 1	–	–	1	–	–	–	INH	9B		1
STA *opr8a* STA *opr16a* STA *oprx16*,X STA *oprx8*,X STA ,X STA *oprx16*,SP STA *oprx8*,SP	Store Accumulator in Memory	M ← (A)	0	–	–	↕	↕	–	DIR EXT IX2 IX1 IX SP2 SP1	B7 C7 D7 E7 F7 9ED7 9EE7	dd hh ll ee ff ff ee ff ff	3 4 4 3 2 5 4
STHX *opr8a* STHX *opr16a* STHX *oprx8*,SP	Store H:X (Index Reg.)	(M:M + $0001) ← (H:X)	0	–	–	↕	↕	–	DIR EXT SP1	35 96 9EFF	dd hh ll ff	4 5 5
STOP	Enable Interrupts: Stop Processing Refer to MCU Documentation	I bit ← 0; Stop Processing	–	–	0	–	–	–	INH	8E		2+
STX *opr8a* STX *opr16a* STX *oprx16*,X STX *oprx8*,X STX ,X STX *oprx16*,SP STX *oprx8*,SP	Store X (Low 8 Bits of Index Register) in Memory	M ← (X)	0	–	–	↕	↕	–	DIR EXT IX2 IX1 IX SP2 SP1	BF CF DF EF FF 9EDF 9EEF	dd hh ll ee ff ff ee ff ff	3 4 4 3 2 5 4
SUB *#opr8i* SUB *opr8a* SUB *opr16a* SUB *oprx16*,X SUB *oprx8*,X SUB ,X SUB *oprx16*,SP SUB *oprx8*,SP	Subtract	A ← (A) – (M)	↕	–	–	↕	↕	↕	IMM DIR EXT IX2 IX1 IX SP2 SP1	A0 B0 C0 D0 E0 F0 9ED0 9EE0	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
SWI	Software Interrupt	PC ← (PC) + $0001 Push (PCL); SP ← (SP) – $0001 Push (PCH); SP ← (SP) – $0001 Push (X); SP ← (SP) – $0001 Push (A); SP ← (SP) – $0001 Push (CCR); SP ← (SP) – $0001 I ← 1; PCH ← Interrupt Vector High Byte PCL ← Interrupt Vector Low Byte	–	–	1	–	–	–	INH	83		11
TAP	Transfer Accumulator to CCR	CCR ← (A)	↕	↕	↕	↕	↕	↕	INH	84		1
TAX	Transfer Accumulator to X (Index Register Low)	X ← (A)	–	–	–	–	–	–	INH	97		1
TPA	Transfer CCR to Accumulator	A ← (CCR)	–	–	–	–	–	–	INH	85		1
TST *opr8a* TSTA TSTX TST *oprx8*,X TST ,X TST *oprx8*,SP	Test for Negative or Zero	(M) – $00 (A) – $00 (X) – $00 (M) – $00 (M) – $00 (M) – $00	0	–	–	↕	↕	–	DIR INH INH IX1 IX SP1	3D 4D 5D 6D 7D 9E6D	dd ff ff	4 1 1 4 3 5
TSX	Transfer SP to Index Reg.	H:X ← (SP) + $0001	–	–	–	–	–	–	INH	95		2
TXA	Transfer X (Index Reg. Low) to Accumulator	A ← (X)	–	–	–	–	–	–	INH	9F		1
TXS	Transfer Index Reg. to SP	SP ← (H:X) – $0001	–	–	–	–	–	–	INH	94		2
WAIT	Enable Interrupts; Wait for Interrupt	I bit ← 0; Halt CPU	–	–	0	–	–	–	INH	8F		2+

Table 6-13. Opcode Map (Sheet 1 of 2)

Bit-Manipulation		Branch	Read-Modify-Write					Control		Register/Memory					
00 5 BRSET0 3 DIR	10 5 BSET0 2 DIR	20 3 BRA 2 REL	30 5 NEG 2 DIR	40 1 NEGA 1 INH	50 1 NEGX 1 INH	60 5 NEG 2 IX1	70 4 NEG 1 IX	80 9 RTI 1 INH	90 3 BGE 2 REL	A0 2 SUB 2 IMM	B0 3 SUB 2 DIR	C0 4 SUB 3 EXT	D0 4 SUB 3 IX2	E0 3 SUB 2 IX1	F0 3 SUB 1 IX
01 5 BRCLR0 3 DIR	11 5 BCLR0 2 DIR	21 3 BRN 2 REL	31 5 CBEQ 3 DIR	41 4 CBEQA 3 IMM	51 4 CBEQX 3 IMM	61 5 CBEQ 3 IX1+	71 5 CBEQ 2 IX+	81 6 RTS 1 INH	91 3 BLT 2 REL	A1 2 CMP 2 IMM	B1 3 CMP 2 DIR	C1 4 CMP 3 EXT	D1 4 CMP 3 IX2	E1 3 CMP 2 IX1	F1 3 CMP 1 IX
02 5 BRSET1 3 DIR	12 5 BSET1 2 DIR	22 3 BHI 2 REL	32 5 LDHX 3 EXT	42 5 MUL 1 INH	52 6 DIV 1 INH	62 1 NSA 1 INH	72 1 DAA 1 INH	82 5+ BGND 1 INH	92 3 BGT 2 REL	A2 2 SBC 2 IMM	B2 3 SBC 2 DIR	C2 4 SBC 3 EXT	D2 4 SBC 3 IX2	E2 3 SBC 2 IX1	F2 3 SBC 1 IX
03 5 BRCLR1 3 DIR	13 5 BCLR1 2 DIR	23 3 BLS 2 REL	33 5 COM 2 DIR	43 1 COMA 1 INH	53 1 COMX 1 INH	63 5 COM 2 IX1	73 4 COM 1 IX	83 11 SWI 1 INH	93 3 BLE 2 REL	A3 2 CPX 2 IMM	B3 3 CPX 2 DIR	C3 4 CPX 3 EXT	D3 4 CPX 3 IX2	E3 3 CPX 2 IX1	F3 3 CPX 1 IX
04 5 BRSET2 3 DIR	14 5 BSET2 2 DIR	24 3 BCC 2 REL	34 5 LSR 2 DIR	44 1 LSRA 1 INH	54 1 LSRX 1 INH	64 5 LSR 2 IX1	74 4 LSR 1 IX	84 1 TAP 1 INH	94 2 TXS 1 INH	A4 2 AND 2 IMM	B4 3 AND 2 DIR	C4 4 AND 3 EXT	D4 4 AND 3 IX2	E4 3 AND 2 IX1	F4 3 AND 1 IX
05 5 BRCLR2 3 DIR	15 5 BCLR2 2 DIR	25 3 BCS 2 REL	35 4 STHX 2 DIR	45 3 LDHX 3 IMM	55 4 LDHX 2 DIR	65 3 CPHX 3 IMM	75 5 CPHX 2 DIR	85 1 TPA 1 INH	95 2 TSX 1 INH	A5 2 BIT 2 IMM	B5 3 BIT 2 DIR	C5 4 BIT 3 EXT	D5 4 BIT 3 IX2	E5 3 BIT 2 IX1	F5 3 BIT 1 IX
06 5 BRSET3 3 DIR	16 5 BSET3 2 DIR	26 3 BNE 2 REL	36 5 ROR 2 DIR	46 1 RORA 1 INH	56 1 RORX 1 INH	66 5 ROR 2 IX1	76 4 ROR 1 IX	86 3 PULA 1 INH	96 5 STHX 3 EXT	A6 2 LDA 2 IMM	B6 3 LDA 2 DIR	C6 4 LDA 3 EXT	D6 4 LDA 3 IX2	E6 3 LDA 2 IX1	F6 3 LDA 1 IX
07 5 BRCLR3 3 DIR	17 5 BCLR3 2 DIR	27 3 BEQ 2 REL	37 5 ASR 2 DIR	47 1 ASRA 1 INH	57 1 ASRX 1 INH	67 5 ASR 2 IX1	77 4 ASR 1 IX	87 2 PSHA 1 INH	97 1 TAX 1 INH	A7 2 AIS 2 IMM	B7 3 STA 2 DIR	C7 4 STA 3 EXT	D7 4 STA 3 IX2	E7 3 STA 2 IX1	F7 2 STA 1 IX
08 5 BRSET4 3 DIR	18 5 BSET4 2 DIR	28 3 BHCC 2 REL	38 5 LSL 2 DIR	48 1 LSLA 1 INH	58 1 LSLX 1 INH	68 5 LSL 2 IX1	78 4 LSL 1 IX	88 3 PULX 1 INH	98 1 CLC 1 INH	A8 2 EOR 2 IMM	B8 3 EOR 2 DIR	C8 4 EOR 3 EXT	D8 4 EOR 3 IX2	E8 3 EOR 2 IX1	F8 3 EOR 1 IX
09 5 BRCLR4 3 DIR	19 5 BCLR4 2 DIR	29 3 BHCS 2 REL	39 5 ROL 2 DIR	49 1 ROLA 1 INH	59 1 ROLX 1 INH	69 5 ROL 2 IX1	79 4 ROL 1 IX	89 2 PSHX 1 INH	99 1 SEC 1 INH	A9 2 ADC 2 IMM	B9 3 ADC 2 DIR	C9 4 ADC 3 EXT	D9 4 ADC 3 IX2	E9 3 ADC 2 IX1	F9 3 ADC 1 IX
0A 5 BRSET5 3 DIR	1A 5 BSET5 2 DIR	2A 3 BPL 2 REL	3A 5 DEC 2 DIR	4A 1 DECA 1 INH	5A 1 DECX 1 INH	6A 5 DEC 2 IX1	7A 4 DEC 1 IX	8A 3 PULH 1 INH	9A 1 CLI 1 INH	AA 2 ORA 2 IMM	BA 3 ORA 2 DIR	CA 4 ORA 3 EXT	DA 4 ORA 3 IX2	EA 3 ORA 2 IX1	FA 3 ORA 1 IX
0B 5 BRCLR5 3 DIR	1B 5 BCLR5 2 DIR	2B 3 BMI 2 REL	3B 7 DBNZ 3 DIR	4B 4 DBNZA 2 INH	5B 4 DBNZX 2 INH	6B 7 DBNZ 3 IX1	7B 6 DBNZ 2 IX	8B 2 PSHH 1 INH	9B 1 SEI 1 INH	AB 2 ADD 2 IMM	BB 3 ADD 2 DIR	CB 4 ADD 3 EXT	DB 4 ADD 3 IX2	EB 3 ADD 2 IX1	FB 3 ADD 1 IX
0C 5 BRSET6 3 DIR	1C 5 BSET6 2 DIR	2C 3 BMC 2 REL	3C 5 INC 2 DIR	4C 1 INCA 1 INH	5C 1 INCX 1 INH	6C 5 INC 2 IX1	7C 4 INC 1 IX	8C 1 CLRH 1 INH	9C 1 RSP 1 INH	▨	BC 3 JMP 2 DIR	CC 4 JMP 3 EXT	DC 4 JMP 3 IX2	EC 3 JMP 2 IX1	FC 3 JMP 1 IX
0D 5 BRCLR6 3 DIR	1D 5 BCLR6 2 DIR	2D 3 BMS 2 REL	3D 4 TST 2 DIR	4D 1 TSTA 1 INH	5D 1 TSTX 1 INH	6D 4 TST 2 IX1	7D 3 TST 1 IX	▨	9D 1 NOP 1 INH	AD 5 BSR 2 REL	BD 5 JSR 2 DIR	CD 6 JSR 3 EXT	DD 6 JSR 3 IX2	ED 5 JSR 2 IX1	FD 5 JSR 1 IX
0E 5 BRSET7 3 DIR	1E 5 BSET7 2 DIR	2E 3 BIL 2 REL	3E 6 CPHX 3 EXT	4E 6 MOV 3 DD	5E 5 MOV 2 DIX+	6E 4 MOV 3 IMD	7E 5 MOV 2 IX+D	8E 2+ STOP 1 INH	9E Page 2	AE 2 LDX 2 IMM	BE 3 LDX 2 DIR	CE 4 LDX 3 EXT	DE 4 LDX 3 IX2	EE 3 LDX 2 IX1	FE 3 LDX 1 IX
0F 5 BRCLR7 3 DIR	1F 5 BCLR7 2 DIR	2F 3 BIH 2 REL	3F 3 CLR 2 DIR	4F 1 CLRA 1 INH	5F 1 CLRX 1 INH	6F 5 CLR 2 IX1	7F 4 CLR 1 IX	8F 2+ WAIT 1 INH	9F 1 TXA 1 INH	AF 2 AIX 2 IMM	BF 3 STX 2 DIR	CF 4 STX 3 EXT	DF 4 STX 3 IX2	EF 3 STX 2 IX1	FF 2 STX 1 IX

INH	**Inherent**	**REL**	**Relative**	**SP1**	**Stack Pointer, 8-Bit Offset**
IMM	**Immediate**	**IX**	**Indexed, No Offset**	**SP2**	**Stack Pointer, 16-Bit Offset**
DIR	**Direct**	**IX1**	**Indexed, 8-Bit Offset**	**IX+**	**Indexed, No Offset with Post Increment**
EXT	**Extended**	**IX2**	**Indexed, 16-Bit Offset**		
DD	**DIR to DIR**	**IMD**	**IMM to DIR**	**IX1+**	**Indexed, 1-Byte Offset with Post Increment**
IX+D	**IX+ to DIR**	**DIX+**	**DIR to IX+**		

Opcode in Hexadecimal	F0 3	**HCS08 Cycles**
	SUB	**Instruction Mnemonic**
Number of Bytes	1 IX	**Addressing Mode**

Table 6-13. Opcode Map (Sheet 2 of 2)

Bit-Manipulation		Branch	Read-Modify-Write					Control		Register/Memory					
						9E60 6 NEG 3 SP1							9ED0 5 SUB 4 SP2	9EE0 4 SUB 3 SP1	
						9E61 6 CBEQ 4 SP1							9ED1 5 CMP 4 SP2	9EE1 4 CMP 3 SP1	
													9ED2 5 SBC 4 SP2	9EE2 4 SBC 3 SP1	
						9E63 6 COM 3 SP1							9ED3 5 CPX 4 SP2	9EE3 4 CPX 3 SP1	9EF3 6 CPHX 3 SP1
						9E64 6 LSR 3 SP1							9ED4 5 AND 4 SP2	9EE4 4 AND 3 SP1	
													9ED5 5 BIT 4 SP2	9EE5 4 BIT 3 SP1	
						9E66 6 ROR 3 SP1							9ED6 5 LDA 4 SP2	9EE6 4 LDA 3 SP1	
						9E67 6 ASR 3 SP1							9ED7 5 STA 4 SP2	9EE7 4 STA 3 SP1	
						9E68 6 LSL 3 SP1							9ED8 5 EOR 4 SP2	9EE8 4 EOR 3 SP1	
						9E69 6 ROL 3 SP1							9ED9 5 ADC 4 SP2	9EE9 4 ADC 3 SP1	
						9E6A 6 DEC 3 SP1							9EDA 5 ORA 4 SP2	9EEA 4 ORA 3 SP1	
						9E6B 8 DBNZ 4 SP1							9EDB 5 ADD 4 SP2	9EEB 4 ADD 3 SP1	
						9E6C 6 INC 3 SP1									
						9E6D 5 TST 3 SP1									
										9EAE 5 LDHX 2 IX	9EBE 6 LDHX 4 IX2	9ECE 5 LDHX 3 IX1	9EDE 5 LDX 4 SP2	9EEE 4 LDX 3 SP1	9EFE 5 LDHX 3 SP1
						9E6F 6 CLR 3 SP1							9EDF 5 STX 4 SP2	9EEF 4 STX 3 SP1	9EFF 5 STHX 3 SP1

INH	Inherent	REL	Relative	SP1	Stack Pointer, 8-Bit Offset
IMM	Immediate	IX	Indexed, No Offset	SP2	Stack Pointer, 16-Bit Offset
DIR	Direct	IX1	Indexed, 8-Bit Offset	IX+	Indexed, No Offset with Post Increment
EXT	Extended	IX2	Indexed, 16-Bit Offset		
DD	DIR to DIR	IMD	IMM to DIR	IX1+	Indexed, 1-Byte Offset with Post Increment
IX+D	IX+ to DIR	DIX+	DIR to IX+		

Note: All Sheet 2 Opcodes are Preceded by the Page 2 Prebyte (9E)

Prebyte (9E) and Opcode in Hexadecimal	9E60 6	HCS08 Cycles
	NEG	Instruction Mnemonic
Number of Bytes	3 SP1	Addressing Mode

6.7 Assembly Language Tutorial

While most readers of this book already have a basic understanding of assembly language programming, assemblers written by different third-party development tool vendors often have subtle differences in syntax rules. This section describes the directives, conventions, and syntax rules that apply to the code examples used in this book. If a novice user uses the same Metrowerks assembler that we used, this section provides enough basic information to start writing simple programs. In all cases, the user should refer to the documentation that came with their particular assembler for more detailed information.

Code examples in this book conform to the source forms shown in the tables at the bottom of each instruction page in **Appendix A. Instruction Set Details**. For readability and consistency with the instruction documentation, all instruction mnemonics use uppercase. Most assemblers ignore case for mnemonics, and many programmers prefer to use lowercase to simplify the process of typing long source files.

6.7.1 Parts of a Listing Line

The fields of the following example line from a Metrowerks CodeWarrior code listing are numbered and explained in the text that follows. This explanation is provided as a reference for the code examples used throughout this manual.

```
   34 C000 A4 7F     upcase:      and    #$7F          ;forces MSB to 0
----- ---- --------- ------------ -----  ---------     -----------------------
 1    2    3         4            5      6             7
```

This second code listing is from the P&E Microcomputer Systems CASMS08Z assembler. P&E includes the same fields 1–7 as the previous figure, but they are in slightly different order and there is an optional field #8 that shows the number of CPU bus cycles for each instruction.

```
 C000 [02] A47F        34  upcase:      and    #$7F          ;forces MSB to 0
----- ---- --------- ----  ------------ ------ ------------  ------------------
 2    8    3         1     4            5      6             7
```

Fields 1, 2, 3, and 8 are generated by the assembler while fields 4, 5, 6, and 7 are part of the source file provided by the user:

- Field 1 (491) is a line number which the assembler added as a reference. This line number is not used by the MCU, but it is a useful reference when people are discussing the program listing.
- Field 2 (C000) is the address where this instruction starts in memory.

- Field 3 (A4 7F) is the object code for the instruction on this listing line. $A4 is the opcode for the AND instruction, and $7F is the immediate data value that will be compared to the accumulator (A).
- Field 4 (upcase:) is a label which the assembler equates to the address shown in field 1. Most assemblers require the colon at the end of a label (where it is defined but not where it is used as an operand in an instruction). This colon is not considered part of the label. Some programmers prefer to put labels on a separate line by themselves so they can use longer, more descriptive names while keeping the instruction mnemonics in field 6 lined up along a vertical line that isn't too far to the right in the listing.
- Field 5 (and) is the instruction mnemonic. Most assemblers ignore the case of the mnemonic, but labels are usually case sensitive.
- Field 6 (#$7F) is the operand field. In this case, the immediate value 7F is hexadecimal as indicated by the $ (dollar) symbol. The # (pound) symbol tells the assembler to use immediate addressing mode.
- Field 7 is a comment. Comments should start with a semicolon character. Everything else to the end of the line is a comment that is not used by the assembler or the MCU. It is just for the benefit of the programmer and others who need to understand the program.
- Field 8 ([02]) is an optional field which tells how many bus cycles this instruction takes. Not all assemblers provide this field. The P&E assembler can provide this field. This field is usually left out of listings, but it is included here because it can be helpful while a programmer is learning the instruction set.

6.7.2 Assembler Directives

This section describes a minimum set of assembler directives to allow a novice user to start writing basic assembly language programs. These basic directives should be supported by any HCS08 assembler. Typical assemblers also include other directives, some of which may be specific to a particular vendor's assembler (especially in the areas of macros and conditional assembly). Always refer to the documentation that came with the assembler you are using for complete information.

P&E Microcomputer Systems makes a distinction between directives and pseudo-ops, while some other vendors use the term directives to describe all of these special operators. Pseudo-ops are reserved command words which go in the instruction mnemonic field. Pseudo-ops are used to set the starting location of a program, to equate a label to a value, to define the location of program variables in memory, or to reserve space for RAM variables. Directives are more general commands to control printing and configuration options for the assembler. In most assemblers, directives are placed in the same field as the instruction mnemonics.

6.7.2.1 BASE — Set Default Number Base for Assembler

Most assemblers use decimal as the default base but P&E assemblers default to treating operands with no prefix as hexadecimal numbers. For all of the examples in this book, we want the default number base to be decimal, so it is good practice to use the following directive at the beginning of all of our source files.

```
          base   10t          ;change default to decimal
```

6.7.2.2 INCLUDE — Specify Additional Source Files

It is often inconvenient to place all source code for a project into a single file. This directive allows you to split the project into two or more separate files. The main file would use INCLUDE directives in the main source file to indicate where the other files should be incorporated into the project. When the assembler encounters an INCLUDE directive, it switches its input stream to the included file until an end-of-file is detected. This effectively replaces the include directive line with the referenced file.

A common use for this directive is to include a chip definition file (sometimes called an equate file). Motorola provides free equate files for its MCUs, so you can use register and bit names in your programs rather than addresses and bit numbers which are not as readable.

This example just uses the file name but you can specify an explicit path for the file if it isn’t located in the main project directory.

```
            include "9S08GB60_v1.equ"
```

6.7.2.3 NOLIST/LIST — Turn Off or Turn On Listing

The assembler reads a source file and generates a composite listing file while it assembles the source file into an object code file for a program. The listing file is a plain text file which includes the object code and generated line numbers in addition to the information from the original source file. The NOLIST and LIST directives allow the programmer to control the production of the listing file.

The most common use of these directives is to suppress the listing while the assembler processes the MCU equate file. This is common because the contents of the equate file are well understood and suppressing this listing can easily save 15 to 20 pages of listing. The programmer may list the equate file separately and keep it on hand for reference.

```
            nolist               ;turn off listing
            include "9S08GB60_v1.equ"
            list                 ;turn listing back on
```

6.7.2.4 ORG — Set Program Starting Location

During assembly the assembler maintains a “location counter” which keeps track of the next available memory location where code or variables could be stored. The ORG directive sets this location counter to a specific address value. This does not produce any actual code in the object file. Rather, it tells the assembler where the next byte of code or data should be located in memory.

Every program needs at least one ORG directive, and programs often include several ORG directives. A typical program includes one ORG directive to set the starting location for variables in RAM. After declaring all RAM variables, a second ORG directive is used to establish the starting location for the application program in ROM or FLASH memory.

A third ORG directive is often used to set the location counter to the start of the interrupt vector space.

```
             org    RamStart      ;start of RAM variables

; ds.b directive doesn't produce any object code.
; Just reserves uninitialized named locations for future use.
resrvBytes:  ds.b   8             ;reserve space for 8 vars
;for move examples setup 2 10-byte blocks that overlap
moveBlk1:    ds.b   10            ;reserve 10 bytes for block 1
blk1end:     equ    *             ;* means 'here'
```

```
             org    RomStart      ;set program starting point

Startup:                          ;ex. label on separate line

; Setup options for COP and STOP in SIMOPT
             lda    #initSIMOPT   ;settings for COP & STOP
             sta    SOPT          ;SIM options (write once)
; Set stack pointer to last (highest) RAM location
             ldhx   #RamLast+1    ;point one past RAM
             txs                  ;SP<-(H:X-1)
```

```
             org    Vrti-2        ;2 before first vector
; leave room for resetISR and defaultISR
resetISR:    dc.b   illegalOp     ;force ilop reset
defaultISR:  rti                  ;just return
; even unused vectors should point at some handler

vecRti:      dc.w   defaultISR    ;handle unused interrupts
vecIic:      dc.w   defaultISR    ;handle unused interrupts
vecAtd:      dc.w   defaultISR    ;handle unused interrupts
vecKeyboard: dc.w   defaultISR    ;handle unused interrupts
vecSci2tx:   dc.w   defaultISR    ;handle unused interrupts
vecSci2rx:   dc.w   defaultISR    ;handle unused interrupts
vecSci2err:  dc.w   defaultISR    ;handle unused interrupts
vecSci1tx:   dc.w   defaultISR    ;handle unused interrupts
vecSci1rx:   dc.w   defaultISR    ;handle unused interrupts
vecSci1err:  dc.w   defaultISR    ;handle unused interrupts
vecSpi:      dc.w   defaultISR    ;handle unused interrupts
vecTpm2ovf:  dc.w   defaultISR    ;handle unused interrupts
vecTpm2ch4:  dc.w   defaultISR    ;handle unused interrupts
vecTpm2ch3:  dc.w   defaultISR    ;handle unused interrupts
vecTpm2ch2:  dc.w   defaultISR    ;handle unused interrupts
vecTpm2ch1:  dc.w   defaultISR    ;handle unused interrupts
vecTpm2ch0:  dc.w   defaultISR    ;handle unused interrupts
vecTpm1ovf:  dc.w   defaultISR    ;handle unused interrupts
vecTpm1ch2:  dc.w   defaultISR    ;handle unused interrupts
vecTpm1ch1:  dc.w   defaultISR    ;handle unused interrupts
vecTpm1ch0:  dc.w   defaultISR    ;handle unused interrupts
vecIcg:      dc.w   defaultISR    ;handle unused interrupts
vecLvd:      dc.w   resetISR      ;force an ilop reset
vecIrq:      dc.w   defaultISR    ;handle unused interrupts
vecSwi:      dc.w   defaultISR    ;handle unused interrupts
vecReset:    dc.w   Startup       ;reset starting point
```

6.7.2.5 EQU — Equate a Label to a Value

This directive tells the assembler what value or address should be associated with a particular label. For example:

```
illegalOp:   equu    $8D           ;$8D is an unused opcode
```

tells the assembler that the label illegalOp is equivalent to the hexadecimal value $8D. The next example illustrates the more interesting case where an asterisk (*) in the operand field is interpreted by the assembler to mean “the current location counter value.”

```
  52                           org    RamStart      ;start of RAM variables
  53
  54                ; ds.b directive doesn't produce any object code.
  55                ; Just reserves uninitialized named locations for future use.
  56 0080           resrvBytes:  ds.b   8             ;reserve space for 8 vars
  57                ;for move examples setup 2 10-byte blocks that overlap
  58 0088           moveBlk1:    ds.b   10            ;reserve 10 bytes for block 1
  59      0000 0092 blk1end:     equ    *             ;* means 'here'
```

In this example, the ds.b directive in line 58 set aside 10 (decimal) locations from address $0088–$0091 so at the time the assembler read the “blk1end: EQU *...” line, the location counter was equal to $0092.

6.7.2.6 dc.b — Define Byte-Sized Constants in Memory

dc.b is used to define 8-bit constant values in memory. This directive is similar to the FCB directive used by some assemblers. In its simplest form, the dc.b directive sets a single memory location equal to a specified 8-bit value. The directive can (and usually does) have a label which associates the address, where the constant is stored, to the label.

```
 108                ******************************************************************
 109                * Define ROM (flash) constants for use in examples
 110                ******************************************************************
 111
 112                           org    RomStart      ;set program starting point
 113
 114 1080 55        hexByte:     dc.b   $55           ;$ prefix means hexadecimal
```

In this example, the dc.b directive defined a constant with the value $55 at location $1080. The ORG directive set the location counter to $1080, so this is the address that was used for the dc.b directive. Since the dc.b used one byte of memory, the location counter is automatically advanced by one, so it points at $1081 after the dc.b directive. The label hexByte is set equal to the address $1080 which is the address where the constant ($55) is located in memory.

```
115 1081 0C        decimalByte: dc.b   12            ;no prefix means decimal
116 1082 5A        binaryByte:  dc.b   %01011010     ;% prefix means binary
117 1083 35        asciiByte:   dc.b   '5'           ;' prefix means ASCII
118 1084 1122 33   multiBytes:  dc.b   $11,$22,$33   ;commas separate operands
119      0000 1087 moveBlk3:    equ    *             ;3rd block for move examples
120 1087 4164 616D stringBytes: dc.b   'Adam apple'  ;string makes ASCII bytes
    108B 2061 7070
    108F 6C65
121 1091 00                     dc.b   0             ;null terminator
```

This example demonstrates various forms of the operand field in dc.b directives.

- Line 115 shows a decimal constant (12) and the assembler stores this in memory as $0C which is the hexadecimal equivalent of decimal 12.
- Line 116 shows the % prefix which indicates a binary value.
- In line 117, the character 5 is surrounded by single quotes to indicate an ASCII value. The assembler stores $35 which is the hexadecimal equivalent of the ASCII character for the number 5.
- Line 118 shows that the operand field can consist of a list of separate constants separated by commas. Notice three bytes were stored in memory.
- Line 120 shows an ASCII string may be enclosed in single quotes. The assembler will store the hexadecimal equivalent of each ASCII character in successive memory locations (one byte per character in the string). The quotes are not included in the constants that are stored in memory. In the case of a string or when more than four bytes of constants are defined on one source code line, the listing will have multiple lines to allow the object

code field to line-wrap to list all of the constant values stored in memory. (See the two extra lines between listing lines 140 and 141 which are considered part of line 140.)

6.7.2.7 dc.w — Define 16-Bit (Word) Constants in Memory

dc.w is used to define 16-bit constant values in memory. This directive is similar to the FDB directive used by some assemblers. In its simplest form, the dc.w directive sets a pair of memory locations to a specified 16-bit value (with the first high-order 8 bits going to the current address pointed to by the location counter and the second low-order value going to the next higher memory address location). The directive can (and usually does) have a label which associates the address, where the upper 8-bit half of the constant is stored, to the label.

```
123 1092 1234      hexWord:     dc.w   $1234         ;takes up two bytes
124 1094 1092      addrWord:    dc.w   hexWord       ;label used as 16-bit addr
125 1096 5678 9ABC multiWord:   dc.w   $5678,$9ABC   ;dc.w with multiple operands
```

Line 123 is a simple case where the hexadecimal constant $1234 is stored in memory, $12 at address $1092 and $34 at $1093. The label hexWord is set equal to $1092 by the assembler because this is the memory address where this constant is stored in memory. Line 124 uses the label hexWord in the operand field of a dc.w directive and the assembler stores the equivalent hexadecimal value $1092, $10 at address $1094 and $92 at address $1095. Line 125 demonstrates that the operand field of an dc.w directive can consist of a list of constants separated by commas. The constants $5678 and $9ABC are shown in the object code field of the listing line.

6.7.2.8 ds.b — Define Storage (Reserve) Memory Bytes

ds.b is used to set aside a specified number of 8-bit memory locations for use as program variables. This directive is similar to the RMB directive in some older assemblers. There is also a ds.w directive that is used to set aside a specified number of 16-bit memory locations for use as program variables. Unlike the dc.b and dc.w directives discussed in the previous two sections, the ds.b and ds.w directives do not produce any object code. ds.b tells the assembler to associate a label to the current address pointed to by the location counter and then to adjust the location counter by the number of bytes set aside by the ds.b directive so the location counter points at the next available memory location. The ds.b directive can be used without a label to just move the location counter, but this is rarely done. It is most often used to set aside memory space for a single named program variable, but it can also be used to set aside space for a larger data structure or table.

```
48                ******************************************************************
49                * Define RAM variables for use in examples
50                ******************************************************************
51
52                             org    RamStart      ;start of RAM variables
53
54                ; ds.b directive doesn't produce any object code.
55                ; Just reserves uninitialized named locations for future use.
56 0080           resrvBytes:  ds.b   8             ;reserve space for 8 vars
57                ;for move examples setup 2 10-byte blocks that overlap
58 0088           moveBlk1:    ds.b   10            ;reserve 10 bytes for block 1
59      0000 0092 blk1end:     equ    *             ;* means 'here'
60                ; another way to define a RAM block
61      0000 000A blk2size:    equ    10            ;size in bytes
62 0092           moveBlk2:    ds.b   blk2size      ;reserve bytes for block 2
63      0000 009C blk2end:     equ    (moveBlk2+blk2size)  ;end tracks size
64
65                ; Setup a flag byte with multiple 1-bit flags
66                ;  name prefixed by m is used to define a mask for logical
67                ;  instructions like AND or ORA; the bit name without the m prefix
68                ;  defines a bit number for BCLR, BSET, BRCLR, and BRSET
69 009C           flags:       ds.b   1             ;reserves a byte
70      0000 0007 SCIready:    equ    7             ;bit number
71      0000 0080 mSCIready:   equ    %10000000     ;bit 7 mask
72      0000 0006 OneSecond:   equ    6             ;bit number
73      0000 0040 mOneSecond:  equ    %01000000     ;bit 6 mask
74
75 009D           directByte:  ds.b   1             ;a variable in direct space
```

In this example, the ORG directive is used to establish the location counter value for the assembler. Line 56 sets aside eight bytes of memory (locations $0080 through $0087). The label resrvBytes is set equal to the starting address for the block or $0080. Line 75 is a much simpler and more common use of ds.b where memory location $009D is set aside for a program variable named directByte. Lines 69 through 73 show an ds.b directive used to set aside an 8-bit location for the program variable named “flags” and then the next four EQU directives are used to identify specific bits within this flag byte.

In the HCS08 architecture, the BCLR, BSET, BRCLR, and BRSET instructions use the bit number (0–7) to choose a specific opcode that is defined to work with the selected bit within a memory location. Other instructions such as AND and ORA use bit masks to identify one or more bit locations to be operated on.

For this reason, bits are defined in two slightly different ways:

- By convention, we use a normal label such as SCIready to define the bit number
- We use the same label preceded by a lowercase m to define the bit mask

In a program, we would then use the plain bit name form whenever we use it in a BCLR, BSET, BRCLR, or BRSET instruction. We use the bit name with a prefix of lowercase m everywhere else. Following a convention such as this helps the programmer avoid confusion and errors. This convention is used in equate files provided by Motorola so it is suggested that the same convention be followed in defining other bit labels.

6.7.3 Labels

User-defined labels are used by the assembler to make the code more readable and to simplify the task of writing programs. For example, it is easier for a programmer to remember a text label like “Start” than a 4-digit hexadecimal address which may change as instructions are added or removed from the program. These labels are significant to the assembler, but not to the actual MCU. The source forms shown on the instruction pages in **Appendix A. Instruction Set Details** never include any labels. In fact, the source forms only show the instruction mnemonic and a representative operand field. A real source program should usually also include a comment field and sometimes a label field.

Some assemblers ignore case in labels so something like “RAM” would be indistinguishable from “ram” or “Ram.” Other assemblers let the programmer set a control flag to decide whether case matters. Always check the documentation for the assembler you are using to be sure you understand how it treats uppercase and lowercase letters.

Older assemblers limited the size of labels to six or eight characters, but modern assemblers allow much longer labels. A few assemblers allow very long names but only consider the first several characters as significant. For example, an assembler that only considered the first 10 characters would not see any difference between the labels LongLabel37 and LongLabel38 although it might consider VeryVeryLongLabel to be acceptable. Again, you should consult the documentation for your assembler. In most assemblers, labels may contain any letters, numbers, or the symbols, underscore (_), or period (.), but the label must start with a letter or underscore (_). Some assemblers allow other characters, but it is safer to limit yourself to these choices to assure easy portability to other assemblers. Notice that labels must NOT contain any space characters because the assembler would not be able to tell this from two separate labels. In this book, underscore characters are not used because some people think they make programs less readable. (This is a subjective opinion and other users think underscore characters improve readability.) Instead, a combination of uppercase and lowercase is used here to make multiword labels, for example, RamLast where an underscore proponent might use ram_last.

A label can be defined only once, but it may be used any number of times within a program. Where a label is defined, the label name must start in the first column of the source line, and most assemblers require a colon after the label where it is defined as in:

```
waitRDRF:    brclr  RDRF,SCI1S1,waitRDRF ;loop till RDRF set
```

Notice that where the label is used in the operand field, there is no colon.

Where longer labels are used, some programmers prefer to place the label on a separate line above the line to which the label refers.

```
131                Startup:                         ;ex. label on separate line
132
133                ; Setup options for COP and STOP in SIMOPT
134 109A A6 00                  lda    #initSIMOPT   ;settings for COP & STOP
```

The label is defined on line 131 and in this case there is an optional comment on the same line. Line 132 is a blank line which produces no object code and is simply used to create a visual separation. Line 133 is a whole-line comment which also does not produce any object code. Line 134 is the first line after the label in line 131 that has any object code, so this is the address assigned to the label by the assembler.

6.7.4 Expressions

The operand field of an instruction or directive can contain an explicit value (using various number bases or conversions), an expression, or a label. Trivial expressions such as RamStart+1 do not require parentheses or brackets. In the P&E assembler, complex expressions must be enclosed in curly braces as in {moveBlk1–RamStart+3}. Most assemblers use parentheses to enclose complex expressions.

Most assemblers allow complex mathematical and logical expressions in any operand field, but practical application programs rarely use complex nested expressions. The most common expressions are small constant offsets to identify a location within a multibyte variable or data structure or to identify the next location past some label (label+1).

```
137 109F 45 1080                ldhx   #RamLast+1    ;point one past RAM
138 10A2 94                     txs                  ;SP<-(H:X-1)
```

In this example, RamLast was equated to the address $107F. We know the TXS instruction is going to automatically subtract one from the address in H:X, so we compensate for this by loading H:X with the address after RamLast (that is RamLast+1). This is an example of a trivial expression that does not need to be enclosed in parentheses.

```
297                  ; add 8-bit operand to 24-bit sum
298 1172 B6 A0                  lda    oprA          ;8-bit operand to A
299 1174 BB A8                  add    sum24+2       ;LS byte of 24-bit sum
300 1176 B7 A8                  sta    sum24+2       ;update LS byte
301 1178 B6 A7                  lda    sum24+1       ;middle byte of 24-bit sum
302 117A A9 00                  adc    #0            ;propigate any carry
303 117C B7 A7                  sta    sum24+1       ;update middle byte
304 117E B6 A6                  lda    sum24         ;get MS byte of 24-bit sum
305 1180 A9 00                  adc    #0            ;propigate carry into MS byte
306 1182 B7 A6                  sta    sum24         ;update MS byte
```

In this example, the label sum24 identifies a 24-bit variable located in three successive bytes of memory. The most significant byte is located at address sum24, the middle byte is at sum24+1 and the least significant byte is located at sum24+2. This is another example of trivial expressions not requiring enclosure in parentheses.

```
 58 0088           moveBlk1:    ds.b   10            ;reserve 10 bytes for block 1
 59      0000 0092 blk1end:     equ    *             ;* means 'here'
 "   "    "    "    "            "      "             "
120 1087 4164 616D stringBytes: dc.b   'Adam apple'  ;string makes ASCII bytes
    108B 2061 7070
    108F 6C65
121 1091 00                     dc.b   0             ;null terminator
 "   "    "    "    "            "      "             "
288                ; block move example to move a string to a RAM block
289 1165 45 0088                ldhx   #moveBlk1     ;point at destination block
290 1168 D6 0FFF   movLoop1:    lda    (stringBytes-moveBlk1),x  ;get source byte
291 116B 27 05                  beq    dunLoop1      ;null terminator ends loop
292 116D E7 00                  sta    0,x           ;save to destination block
293 116F 5C                     incx                 ;next location (assumes DIR)
294 1170 20 F6                  bra    movLoop1      ;continue loop
295                dunLoop1:
```

In line 290 the expression (stringBytes-moveBlk1) is enclosed in parentheses because it involves two labels and the assembler considers this a “complex” expression. The assembler computes the difference of the two 16-bit addresses represented by stringBytes = $1087 and moveBlk1 = $0088 ($1087 – $0088 = $0FFF). The result of the assembler’s computation can be seen after the opcode (D6) in the object code field of the listing in line 290.

```
                   mOR:         equ    %00001000     ;receiver over run
                   mNF:         equ    %00000100     ;receiver noise flag
                   mFE:         equ    %00000010     ;receiver framing error
                   mPF:         equ    %00000001     ;received parity failed
                    "            "      "             "
415 11F3 A5 0F                  bit    #(mOR+mNF+mFE+mPF) ;mask of all error flags
```

In this example, we added the separate bit masks with the arithmetic addition operator. Because each of the four bit masks is an 8-bit value with a different single bit set to 1, this is equivalent to combining the masks with logical OR operators, but the + (plus) is more universal among different assemblers than the OR operator.

6.7.5 Equate File Conventions

Most code for this book was assembled along with an included equate file which defines all MCU registers and control bits by the names used in the data sheet for a specific HCS08 derivative. In that equate file, which is described in greater detail in **Appendix B. Equate File Conventions**, register names use all uppercase letters to match the data sheets. Program labels use a combination of uppercase and lowercase letters. This is not a requirement of the assembler, but rather a convention chosen to make these code listings more consistent with chip documentation.

Bit names are defined in two ways:

- The bit name with no prefix is equated to the bit number (0–7).
- The name preceded by a lower-case m is equated to a bit position mask.

This excerpt from the equate file for the MC9S08GB60 shows the SCI status register with its bits defined according to this convention.

```
SCI1S1:      equ    $1C          ;SCI1 status register 1
SCI2S1:      equ    $24          ;SCI2 status register 1
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
TDRE:        equ    7            ;(bit #7) Tx data register empty
TC:          equ    6            ;(bit #6) transmit complete
RDRF:        equ    5            ;(bit #5) Rx data register full
IDLE:        equ    4            ;(bit #4) idle line detected
OR:          equ    3            ;(bit #3) Rx over run
NF:          equ    2            ;(bit #2) Rx noise flag
FE:          equ    1            ;(bit #1) Rx framing error
PF:          equ    0            ;(bit #0) Rx parity failed
; bit position masks
mTDRE:       equ    %10000000    ;transmit data register empty
mTC:         equ    %01000000    ;transmit complete
mRDRF:       equ    %00100000    ;receive data register full
mIDLE:       equ    %00010000    ;idle line detected
mOR:         equ    %00001000    ;receiver over run
mNF:         equ    %00000100    ;receiver noise flag
mFE:         equ    %00000010    ;receiver framing error
mPF:         equ    %00000001    ;received parity failed
```

The next example shows the use of the bit number variation of a bit definition. The operand field of the BRCLR instruction includes three items separated by commas. RDRF is converted to the number 5 which tells the assembler to use the bit-5 variation of the BRCLR instruction (opcode = $0B). The next item, SCI1S1 tells the assembler the operand to be tested is located at the direct addressing mode address $001C (just 1C in the object code). The last item, waitRDRF, tells the assembler to branch back to the same BRCLR instruction if the RDRF status bit is found to be still clear (0).

```
450 120A 0B 1C FD  waitRDRF:    brclr  RDRF,SCI1S1,waitRDRF ;loop till RDRF set
```

The next example shows an expression combining the bit masks for the OR, NF, FE, and PF status bits. In this example, we used the bit names with a preceding m to get the bit position mask rather than the bit number. We used a simple addition operator (+) to combine the bit masks. Although a logical OR might have been more correct in this case, not all assemblers use the same character to indicate the logical OR operation, so the + is more portable among assemblers. We can use the + because we know the individual bit masks do not overlap.

```
413                ; BIT example to check several error flags in SCI status reg
414 11F1 B6 1C                  lda    SCI1S1      ;read SCI status register
415 11F3 A5 0F                  bit    #(mOR+mNF+mFE+mPF) ;mask of all error flags
416 11F5 26 00                  bne    sciError    ;branch if any flags set
417                ; A still contains undisturbed status register
```

6.7.6 Object Code (S19) Files

The ultimate goal of an assembler is to convert a source code file into the object code that the MCU needs to execute a program. The assembler optionally produces a listing file which acts as a form of primary documentation for the program. In this section we briefly describe the source and listing files and provide a more detailed description of the object code file, which is sometimes called a “dot S 1 9 file.” This name comes from the .s19 filename extension and the internal format of the file.

The whole programming process starts with a planning effort which may involve flowcharts or other forms of documentation which describe what is to be done and roughly how the programmer plans to do it. The first item directly related to the final program is the source file which the programmer types into a text file. The source file uses instruction mnemonics and special syntax rules that are understood by the assembler.

The source file should also include generous comments to help humans who must understand and maintain the program. The following is an example of a short source program.

```
******************************************************************
* Title:  s19example.asm               Copyright (c) Motorola 2003
******************************************************************
* Author: Jim Sibigtroth - Motorola
*
* Description: This is not a complete program, rather it is just
* enough code to demonstrate the relationship between the various
* files in a typical MCU programming project (especially .s19
* files).
*
******************************************************************
             org    $C000
******************************************************************
* upcase - convert ASCII character in A to upper case
*  on entry A contains an unknown character
*  first strip MSB (AND with $7F) to get 7-bit ASCII
*  if A > or = "a" and < or = "z", subtract $20 (A=$41, a=$61)
*  other values unchanged except MSB stripped off (forced to 0)
******************************************************************
upcase:      and    #$7F          ;forces MSB to 0
             cmp    #'a'          ;check for < "a"
             blt    xupcase       ;done if too small
             cmp    #'z'          ;check for > "z"
             bgt    xupcase       ;done if too big
             sub    #$20          ;convert a-z to A-Z
xupcase:     rts                  ;done
*********************

******************************************************************
* ishex - check character for valid hexadecimal (0-9 or A-F)
*  on entry A contains an unknown upper-case character
*  returns with original character in A and Z set or cleared
*  if A was valid hexadecimal then Z=1, otherwise Z=0
******************************************************************
ishex:       psha                 ;save original character
             cmp    #'0'          ;check for < ASCII zero
             blo    nothex        ;branches if C=0 (Z also 0)
             cmp    #'9'          ;check for 0-9
             bls    okhex         ;branches if ASCII 0-9
             cmp    #'A'          ;check for < ASCII A
             blo    nothex        ;branches if C=0 (Z also 0)
             cmp    #'F'          ;check for A-F
             bhi    nothex        ;branches if > ASCII F
okhex:       clra                 ;forces Z bit to 1
nothex:      pula                 ;restore original character
             rts                  ;return Z=1 if char was hex
*********************
```

Figure 6-6. Demonstration Code

The assembler is a third-party development tool which is a computer program that runs on a personal computer or workstation and translates source code files into the hexadecimal numbers to be stored into the memory of the target MCU. The assembler can be requested to produce a listing file which includes both the original source program and a representation of the machine code meaning of each source line. This listing file is intended to act as documentation for the application program. The listing includes more information than the source file, such as the addresses of labels and the opcodes that each instruction mnemonic translates to.

The following code example is the listing file generated by assembling the source file shown in the previous example.

```
 1                ******************************************************************
 2                * Title:  s19example.asm               Copyright (c) Motorola 2003
 3                ******************************************************************
 4                * Author: Jim Sibigtroth - Motorola
 5                *
 6                * Description: This is not a complete program, rather it is just
 7                * enough code to demonstrate the relationship between the various
 8                * files in a typical MCU programming project (especially .s19
 9                * files).
10                *
25                ******************************************************************
26                             org    $C000
27                ******************************************************************
28                * upcase - convert ASCII character in A to upper case
29                *  on entry A contains an unknown character
30                *  first strip MSB (AND with $7F) to get 7-bit ASCII
31                *  if A > or = "a" and < or = "z", subtract $20 (A=$41, a=$61)
32                *  other values unchanged except MSB stripped off (forced to 0)
33                ******************************************************************
34 C000 A4 7F     upcase:      and    #$7F          ;forces MSB to 0
35 C002 A1 61                  cmp    #'a'          ;check for < "a"
36 C004 91 06                  blt    xupcase       ;done if too small
37 C006 A1 7A                  cmp    #'z'          ;check for > "z"
38 C008 92 02                  bgt    xupcase       ;done if too big
39 C00A A0 20                  sub    #$20          ;convert a-z to A-Z
40 C00C 81        xupcase:     rts                  ;done
41                *********************
42
43                ******************************************************************
44                * ishex - check character for valid hexadecimal (0-9 or A-F)
45                *  on entry A contains an unknown upper-case character
46                *  returns with original character in A and Z set or cleared
47                *  if A was valid hexadecimal then Z=1, otherwise Z=0
48                ******************************************************************
49 C00D 87        ishex:       psha                 ;save original character
50 C00E A1 30                  cmp    #'0'          ;check for < ASCII zero
51 C010 25 0D                  blo    nothex        ;branches if C=0 (Z also 0)
52 C012 A1 39                  cmp    #'9'          ;check for 0-9
53 C014 23 08                  bls    okhex         ;branches if ASCII 0-9
54 C016 A1 41                  cmp    #'A'          ;check for < ASCII A
55 C018 25 05                  blo    nothex        ;branches if C=0 (Z also 0)
56 C01A A1 46                  cmp    #'F'          ;check for A-F
57 C01C 22 01                  bhi    nothex        ;branches if > ASCII F
58 C01E 4F        okhex:       clra                 ;forces Z bit to 1
59 C01F 86        nothex:      pula                 ;restore original character
60 C020 81                     rts                  ;return Z=1 if char was hex
61                *********************
```

Figure 6-7. Listing File

The fields of this listing are explained in 6.7.1 Parts of a Listing Line.

The MCU expects the program to be a series of 8-bit values in memory. So far, our program still looks as if it was written for people. The version the computer needs to load into its memory is called an object code file. For Motorola microcontrollers, the most common form of object code file is the .s19 or S-record file. The assembler can be directed to optionally produce a listing file and/or an object code file.

An S-record file is an ASCII text file that can be viewed by a text editor or word processor. Do not edit these files because the structure and content of the files are critical to their proper operation.

Each line of an S-record file is a record. Each record begins with a capital letter S followed by a code number from 0 to 9. The only code numbers that are important in this application are S0, S1, and S9 because other S-number codes apply only to larger systems.

- S0 is an optional header record that may contain the name of the file for the benefit of humans that need to maintain these files.
- S1 records are the main data records.
- An S9 record is used to mark the end of the S-record file.

For the work we are doing with 8-bit microcontrollers, the information in the S9 record is not important, but an S9 record is required at the end of the S-record file. Figure 6-8 shows the syntax of an S1 record.

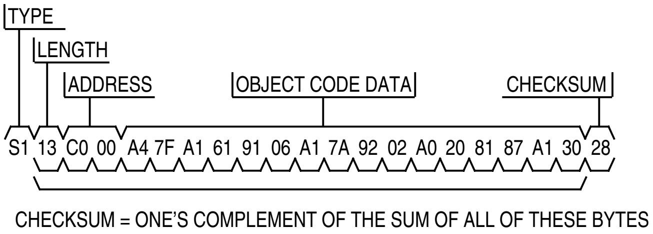

TYPE
LENGTH
ADDRESS
OBJECT CODE DATA
CHECKSUM
S1 13 C0 00 A4 7F A1 61 91 06 A1 7A 92 02 A0 20 81 87 A1 30 28
CHECKSUM = ONE'S COMPLEMENT OF THE SUM OF ALL OF THESE BYTES

Figure 6-8. Syntax of an S1 Record

All of the numbers in an S-record file are in hexadecimal. The type field is S0, S1, or S9 for the S-record files used here. The length field is the number of pairs of hexadecimal digits in the record excluding the type and length fields. The address field is the 16-bit address where the first data byte will be stored in memory. Each pair of hexadecimal digits in the machine code data field represents an 8-bit data value to be stored in successive locations in memory. The checksum field is an 8-bit value that represents the one's complement of the sum of all bytes in the S-record except the type and checksum fields. This checksum is used during loading of the S-record file to verify that the data is complete and correct for each record.

```
S123C000A47FA1619106A17A9202A0208187A130250DA1392308A1412505A14622014F
86F6
S104C020819A
S9030000FC
```

Figure 6-9. S19 Example

You can compare the values in the S-record file with those in the object code field of the listing in **Figure 6-9**. The ORG directive in line 49 of **Figure 6-7** established the starting address at $C000.

Section 7. Development Support

7.1 Introduction

Development support systems in the HCS08 Family include the background debug controller (BDC) and the on-chip debug module (DBG). This architecture marks a major change in the way MCU systems are developed due to advances in the processing technology used to make these devices.

In the past, most development was based on an external tool having access to the address and data buses of the target MCU. This allowed the external tool to monitor cycle-by-cycle activity and intervene at critical points to stop normal execution of the application program. This style of debug has become increasingly difficult to support due to the higher speeds and smaller packages of more modern MCUs. At the same time, the cost of logic circuitry within the MCU has decreased as process improvements and shrinks have allowed more circuitry per unit of die area. Due to mechanical constraints, pads for wire-bond connections have not shrunk as quickly as other circuitry. In today’s technology, a few extra pins cost more than a few thousand logic transistors worth of internal circuitry. Moving the development circuitry inside the MCU to avoid the need for external pins for the address and data buses is now the most cost-effective method.

The BDC provides a single-wire debug interface to the target MCU. This interface provides a convenient means for programming the on-chip FLASH and other non-volatile memories. Also, the BDC is the primary debug interface for development and allows non-intrusive access to memory data and traditional debug features such as CPU register modify, breakpoints, and single instruction trace commands.

In the HCS08 Family, address and data bus signals are not available on external pins (not even in test modes). Debug is done through commands fed into the target MCU via the single-wire background debug interface. The debug module provides a means to selectively trigger and capture bus information so an external development system can reconstruct what happened inside the MCU on a cycle-by-cycle basis without having external access to the address and data signals.

Most HCS08 devices provide two other features related to development. The BDFR control bit in the SBDFR register (usually located at $1801) is a write-only bit that allows a host development system to reset the target MCU with a serial memory modify command through the background debug interface. BDFR cannot be written by user software, so the target MCU cannot be reset accidentally even if user code runs away due to some programming bug. The second development feature is a part identification number in the SDIDH:SDIDL register pair (usually located at $1806, $1807). The upper four bits of SDIDH hold the silicon mask set revision number (0–F), and the remaining 12 bits of the SDIDH:SDIDL register pair hold a 12-bit code number that identifies the device derivative. For example, the first revision of the MC9S08GB60 version of the HSC08 Family has a code number of SDIDH:SDIDL = $0 002). This identification code allows an external development host to associate a register definition file to a particular target MCU so the debugger understands where registers and control bits are located in the target MCU.

7.2 Features

Features of the background debug controller (BDC) include:

- Single dedicated pin for mode selection and background communications
- BDC registers not located in memory map
- SYNC command to determine target communications rate
- Non-intrusive commands for memory access
- Active background mode commands for CPU register access
- GO and TRACE1 commands

- BACKGROUND command can wake CPU from stop or wait modes
- One hardware address breakpoint built into BDC
- Oscillator runs in stop mode, if BDM enabled

Features of the debug module (DBG) include:

- Two trigger comparators:
  - Two address + read/write (R/W) or
  - One full address + data + R/W
- Flexible 8-word by 16-bit FIFO (first-in, first-out) for capture information:
  - Change-of-flow addresses or
  - Event-only data
- Two types of breakpoints:
  - Tag breakpoints for instruction opcodes
  - Force breakpoints for any address access
- Nine trigger modes:
  - A-only
  - A OR B
  - A then B
  - A AND B data (full mode)
  - A AND NOT B data (full mode)
  - Event-only B (store data)
  - A then event-only B (store data)
  - Inside range (A ≤ address ≤ B)
  - Outside range (address < A or address > B)

7.3 Background Debug Controller (BDC)

All MCUs in the HCS08 Family contain a single-wire background debug interface which supports in-circuit programming of on-chip non-volatile memory and sophisticated non-intrusive debug capabilities. Unlike debug interfaces on earlier 8-bit MCUs, this system does not interfere with normal application resources. It does not use any user memory or locations in the memory map and does not share any on-chip peripherals. The single BKGD interface pin is a separate dedicated pin which is not accessible to user programs.

BDM commands are divided into two groups:

- Active background mode commands require that the target MCU is in active background mode (the user program is not running). The BACKGROUND command causes the target MCU to enter active background mode. Active background mode commands allow the CPU registers to be read or written and allow the user to trace one user instruction at a time or GO to the user program from active background mode.
- Non-intrusive commands can be executed at any time even while the user's program is running. Non-intrusive commands allow a user to read or write MCU memory locations or access status and control registers within the background debug controller (BDC).

Typically, a relatively simple interface pod is used to translate commands from a host computer into commands for the custom serial interface to the single-wire background debug system. Depending on the development tool vendor, this interface pod may use a standard RS232 serial port, a parallel printer port, or some other type of communications such as Ethernet or a universal serial bus (USB) to communicate between the host PC and the pod. The pod typically connects to the target system with ground, the BKGD pin, $\overline{\text{RESET}}$ (if there is a reset pin), and sometimes a V_{DD} signal. An open-drain connection to reset allows the host to force a target system reset which is useful to regain control of a lost target system or to control startup of a target system before the on-chip non-volatile memory has been programmed. V_{DD} can sometimes be used to allow the pod to take power from the target system to avoid the need for a separate power supply.

7.3.1 BKGD Pin Description

All commands and bidirectional data for the background debug system are communicated through the BKGD pin.

BKGD is the single-wire background debug interface pin. The primary function of this pin is for bidirectional serial communication of background debug commands and data. During reset, this pin selects between starting in active background mode and starting the user's application program. This pin is also used to request a timed sync response pulse to allow a host development tool to determine the correct clock frequency for background debug serial communications.

Figure 7-1 shows the standard header for connection of a BDM pod. A pod is a small interface device that connects a host computer such as a personal computer to a target HCS08 system. BKGD and GND are the minimum connections required to communicate with a target MCU. The open-drain $\overline{\text{RESET}}$ signal is included in the connector to allow a direct way for the host to force a target system reset. By controlling both BKGD and $\overline{\text{RESET}}$, the host also can force the target system to reset into active background mode rather than start the user application program. (This is useful to gain control of a target MCU whose FLASH program memory has not been programmed yet with a user application program.) The V_{DD} connection can sometimes allow a host debugger pod to take power from the target system rather than using a separate power source for the pod. However, if the pod is powered separately, it can be connected to a running target system without forcing a target system reset or otherwise disturbing the running application program.

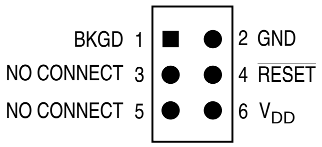


Figure 7-1. Standard BDM Tool Connector

In cases where there is no $\overline{\text{RESET}}$ pin on the MCU or no $\overline{\text{RESET}}$ connection from the debug pod to the target MCU, there are other ways to force a target system reset:

- Write a logic 1 to the BDM force reset (BDFR) bit in the SBDFR register. This bit can only be written using a serial WRITE_BYTE or WRITE_BYTE_WS command.
- Turn power off and back on to force a power-on reset.
- Point the PC at an illegal opcode and use GO or TRACE1 to force an illegal opcode reset.

BKGD is a pseudo-open-drain pin with an on-chip pullup so no external pullup resistor is required (although some users still use an external pullup resistor to improve noise immunity). Unlike typical open-drain pins, the external resistor capacitor (RC) time constant on this pin, which is influenced by external capacitance, plays almost no role in signal rise time. The custom protocol provides for brief, actively driven speedup pulses to force rapid rise times on this pin without risking harmful drive level conflicts. Refer to **7.3.2 Communication Details** for more detail.

When no debugger pod is connected to the 6-pin BDM interface connector, the internal pullup on BKGD chooses the normal operating mode. When a pod is connected, it can pull both BKGD and $\overline{\text{RESET}}$ low, release $\overline{\text{RESET}}$ to select active background mode rather than normal operating mode, then release BKGD. Of course, it is not necessary to force a reset to communicate with the target MCU through the background debug interface. In fact, you can even connect a powered debug pod onto a running target system without disturbing the running application program.

Background debug controller (BDC) serial communications use a custom serial protocol that was first introduced on the M68HC12 Family of microcontrollers. This protocol assumes that the host knows the communication clock rate which is determined by the target BDC clock rate. The BDC clock rate may be the system bus clock frequency or an alternate frequency source depending on the state of the CLKSW control bit in the BDCSCR register. On the MC9S08GB60, the alternate frequency source is a self-clocked local oscillator (ICGLCLK) in the BDC that runs about 8 MHz independent of the bus frequency. On some other

HCS08 derivatives, the alternate frequency source could be the undivided crystal frequency. All communication is initiated and controlled by the host which drives a high-to-low edge to signal the beginning of each bit time. Commands and data are sent most significant bit first (MSB-first).

If a host is attempting to communicate with a target MCU which has an unknown BDC clock rate, a SYNC command may be sent to the target MCU to request a timed sync response signal from which the host can determine the correct communication speed. After establishing communications, the host can read the BDC status and control register and write to the clock switch (CLKSW) control bit to change the source of the BDC clock for further BDC communications if necessary.

7.3.2 Communication Details

The BDC serial interface requires the external controller to generate a falling edge on the BKGD pin to indicate the start of each bit time. The external controller provides this falling edge whether data is transmitted or received.

BKGD is a pseudo-open-drain pin that can be driven either by an external controller or by the MCU. Data is transferred MSB first at 16 BDC clock cycles per bit (nominal speed). The interface times out if 512 BDC clock cycles occur between falling edges from the host. Any BDC command that was in progress when this timeout occurs is aborted without affecting the memory or operating mode of the target MCU system. Refer to **7.3.2.1 BDC Communication Speed Considerations** for more detailed information about the source of the BDC communications clock.

7.3.2.1 BDC Communication Speed Considerations

The custom serial protocol requires the debug pod to know the target BDC communication clock speed. There are two possible sources for this clock frequency (as selected by the CLKSW bit in the BDCSCR register), the bus rate clock or a fixed-frequency alternate clock source that may be different for different HCS08 derivatives. In an MC9S08GB60, this alternate clock source is a self-clocked local oscillator in the BDC module that runs about 8 MHz (independent of the CPU bus frequency). In other HCS08 devices this alternate clock source is the undivided crystal frequency. Future derivatives may use some other source for this alternate clock. Refer to the data sheet for each HCS08 derivative for information about the alternate clock source in the device you are using.

When the MCU is reset in normal user mode, CLKSW is reset to 0 which selects the alternate clock source. This clock source is a fixed frequency that is independent of the bus frequency so it will not change if a user program modifies clock generator settings. This is the preferred clock source for general debugging.

When the MCU is reset in active background mode, CLKSW is reset to 1 which selects the bus clock as the source of the BDC clock. This CLKSW setting is most commonly used during FLASH memory programming because the bus clock can usually be configured to operate at the highest allowed bus frequency which will ensure the fastest possible FLASH programming times. Since the host system is in control of changes to clock generator settings, it can know when a different BDC communication speed should be used. The host programmer also knows that no unexpected change in bus frequency could occur to disrupt BDC communications.

Normally, the CLKSW = 1 option should not be used for general debugging because there is no way to be sure the user's application program with not change the clock generator settings. This is especially true in the case of application programs that are not yet fully debugged.

After any reset (or at any other time), the host system can issue a SYNC command to determine the speed of the BDC clock. CLKSW may be written using a serial WRITE_CONTROL command through the BDC interface. CLKSW is located in the BDCSCR register in the BDC module and it is not accessible in the normal memory map of the MCU. This means that no user program can modify this register (intentionally or unintentionally).

7.3.2.2 Bit Timing Details

The BKGD pin can receive a high or low level or transmit a high or low logic level. The following diagrams show timing for each of these cases. Interface timing is synchronous to clocks in the target BDC, but asynchronous to the external host. The internal BDC clock signal is shown for reference in counting cycles.

Figure 7-2 shows an external host transmitting a logic 1 or 0 to the BKGD pin of a target HCS08 MCU. The host is asynchronous to the target so there is a 0-to-1 cycle delay from the host-generated falling edge to where the target perceives the beginning of the bit time. Ten target BDC clock cycles later, the target senses the bit level on the BKGD pin. Typically, the host actively drives the pseudo-open-drain BKGD pin during host-to-target transmissions to speed up rising edges. Since the target does not drive the BKGD pin during this period, there is no need to treat the line as an open-drain signal during host-to-target transmissions.

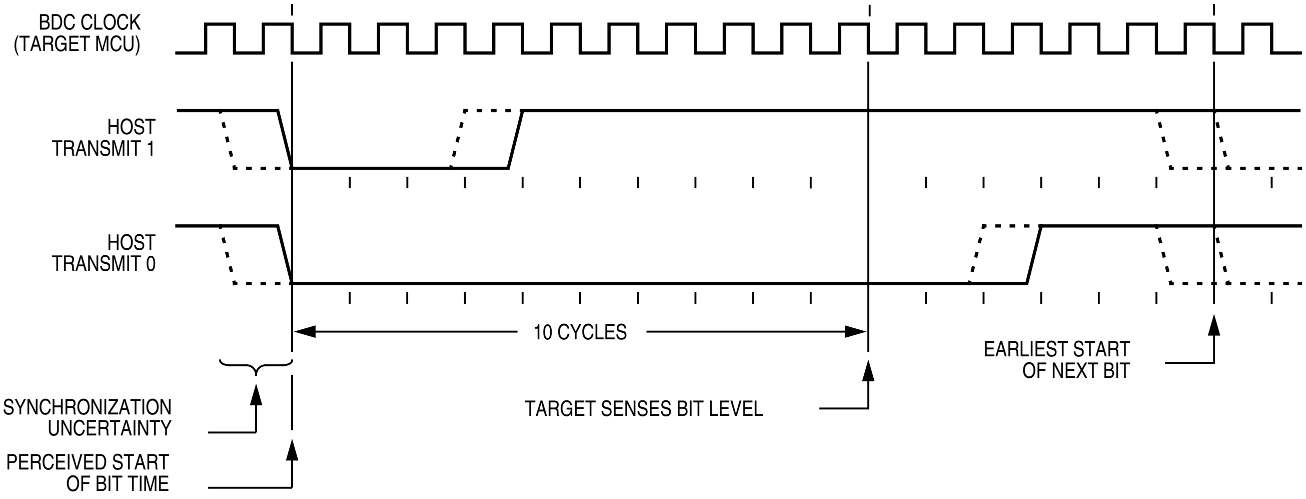


Figure 7-2. BDC Host-to-Target Serial Bit Timing

Figure 7-3 shows the host receiving a logic 1 from the target MCU. Since the host is asynchronous to the target MCU, there is a 0-to-1 cycle delay from the host-generated falling edge on BKGD to the perceived start of the bit time in the target MCU. The host holds the BKGD pin low long enough for the target to recognize it (at least two target BDC cycles). The host must release the low drive before the target MCU drives a brief active-high speedup pulse seven cycles after the perceived start of the bit time. The host should sample the bit level about 10 cycles after it started the bit time.

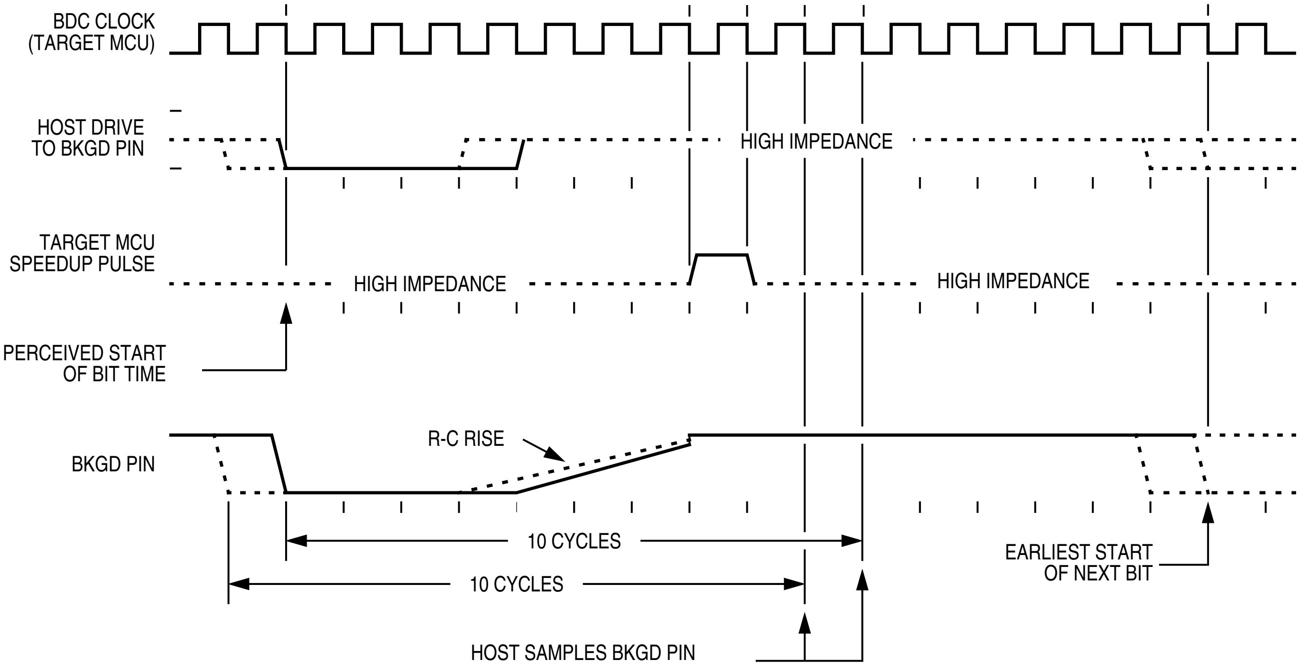


Figure 7-3. BDC Target-to-Host Serial Bit Timing (Logic 1)

Figure 7-4 shows the host receiving a logic 0 from the target MCU. Since the host is asynchronous to the target MCU, there is a 0-to-1 cycle delay from the host-generated falling edge on BKGD to the start of the bit time as perceived by the target MCU. The host initiates the bit time but the target HCS08 finishes it. Since the target wants the host to receive a logic 0, it drives the BKGD pin low for 13 BDC clock cycles, then briefly drives it high to speed up the rising edge. The host samples the bit level about 10 cycles after starting the bit time.

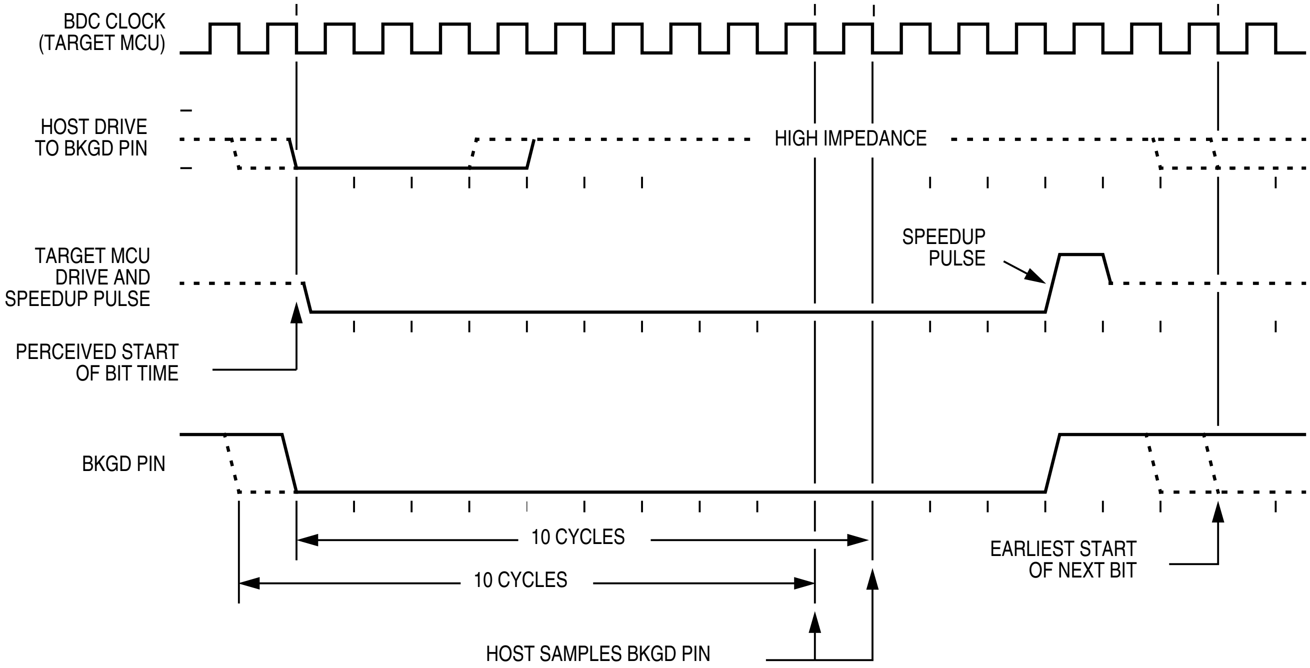


Figure 7-4. BDM Target-to-Host Serial Bit Timing (Logic 0)

7.3.3 BDC Registers and Control Bits

The BDC has two registers:

- The BDC status and control register (BDCSCR) is an 8-bit register containing control and status bits for the background debug controller.
- The BDC breakpoint register (BDCBKPT) holds a 16-bit breakpoint match address.

These registers are accessed with dedicated serial BDC commands and are not located in the memory space of the target MCU (so they do not have addresses and cannot be accessed by user programs).

Some of the bits in the BDCSCR have write limitations; otherwise, these registers may be read or written at any time. For example, the ENBDM control bit may not be written while the MCU is in active background mode. (This prevents the ambiguous condition of the control bit forbidding active background mode while the MCU is already in active background mode.) Also, the four status bits (BDMACT, WS, WSF, and DVF) are read-only status indicators and can never be written by the WRITE_CONTROL serial BDC command. The clock switch (CLKSW) control bit may be read or written at any time; however, this bit should not be written to 1 if the target MCU has an FLL or PLL and user software might change the FLL/PLL settings while debugging is taking place. Changing FLL/PLL settings while CLKSW = 1 causes BDC communications to fail because the host cannot predict the correct communications speed.

7.3.3.1 BDC Status and Control Register

This register can be read or written by serial BDC commands but is not accessible to user programs because it is not located in the normal memory map of the MCU.

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	ENBDM	BDMACT	BKPTEN	FTS	CLKSW	WS	WSF	DVF
Write:		▨				▨	▨	▨
Normal Reset:	0	0	0	0	0	0	0	0
Reset in active background mode:	1	0	0	0	1	0	0	0

Figure 7-5. BDC Status and Control Register (BDCSCR)

ENBDM — Enable BDM (permit active background debug mode)

Typically, this bit is written to 1 by the debug host shortly after the beginning of a debug session or whenever the debug host resets the target and remains 1 until a normal reset clears it.

1 = BDM can be made active to allow active background mode commands.

0 = BDM cannot be made active (non-intrusive commands still allowed).

BDMACT — Background Mode Active Status

This is a read-only status bit.

1 = BDM active and waiting for serial commands

0 = BDM not active

BKPTEN — BDC Breakpoint Enable

If this bit is clear, the BDC breakpoint is disabled and the FTS control bit and BDCBKPT match register are ignored.

1 = BDC breakpoint enabled

0 = BDC breakpoint disabled

FTS — Force/Tag Select

When FTS = 1, a breakpoint is requested whenever the CPU address bus matches the BDCBKPT match register. When FTS = 0, a match between the CPU address bus and the BDCBKPT register causes the fetched opcode to be tagged. If this tagged opcode ever reaches the end of the instruction queue, the CPU enters active background mode rather than executing the tagged opcode.

1 = Breakpoint match forces active background mode at the next instruction boundary (address need not be an opcode).

0 = Tag opcode at breakpoint address and enter active background mode if CPU attempts to execute that instruction.

CLKSW — Select Source for BDC Communications Clock

When the MCU is reset in normal user mode, CLKSW is forced to 0 which selects the fixed alternate frequency source as the BDC clock. In the MC9S08GB60, the alternate frequency source is a local oscillator in the BDC module that runs about 8 MHz. When the MCU is reset in active background mode, CLKSW is forced to 1 which selects the bus clock at the BDC clock. You should avoid using the CLKSW = 1 option while running a user program that might change the bus frequency unexpectedly because this could result in loss of BDC communications.

1 = CPU bus clock

0 = Derivative-specific fixed alternate frequency source

WS — Wait or Stop Status

When the target CPU is in wait or stop mode, most BDC commands cannot function. However, the BACKGROUND command can be used to force the target CPU out of wait or stop mode and into active background mode where all BDC commands work. Whenever the host forces the target MCU into active background mode, the host should issue a READ_STATUS command to check that BDMACT = 1 before attempting other BDC commands.

1 = Target CPU is in wait or stop mode, or a BACKGROUND command was used to change from wait or stop mode to active background mode.

0 = Target CPU is running user application code or is in active background mode (was not in wait or stop mode when background became active).

WSF — Wait or Stop Failure Status

This status bit is set if a memory access command failed due to the target CPU executing a WAIT or STOP instruction at or about the same time. The usual recovery strategy is to issue a BACKGROUND command to get out of wait or stop mode and into active background mode, repeat the command that failed, then return to the user program. (If desired, the host can restore CPU registers and stack values and re-execute the WAIT or STOP instruction.)

1 = Memory access command failed because the CPU entered wait or stop mode.

0 = Memory access did not conflict with a WAIT or STOP instruction.

DVF — Data Valid Failure Status

This status bit is set if a memory access command failed due to the target CPU executing a slow memory access at or about the same time. The usual recovery strategy is to issue READ_LAST commands until the returned status information indicated the original access completed successfully. Since no current HCS08 devices have memory modules that support slow accesses, this bit should always read 0. Consult the data sheet for a specific HCS08 device if you are uncertain about whether it includes any slow memory modules.

1 = Memory access command failed because the CPU was not finished with a slow memory access.
0 = Memory access did not conflict with a slow memory access.

7.3.3.2 BDC Breakpoint Match Register

This 16-bit register holds the address for the hardware breakpoint in the BDC. The BKPTEN and FTS control bits in BDCSCR are used to enable and configure the breakpoint logic. Dedicated serial BDC commands (READ_BKPT and WRITE_BKPT) are used to read and write the BDCBKPT register. Breakpoints are normally set while the target MCU is in active background mode before running the user application program. However, since READ_BKPT and WRITE_BKPT are non-intrusive commands, they could be executed even while the user program is running. For additional information about setup and use of the hardware breakpoint logic in the BDC, refer to **7.3.7 BDC Hardware Breakpoint**.

7.3.4 BDC Commands

BDC commands are sent serially from a host computer to the BKGD pin of the target HCS08 MCU. All commands and data are sent MSB-first using a custom BDC communications protocol. Active background mode commands require that the target MCU is currently in the active background mode while non-intrusive commands may be issued at any time whether the target MCU is in active background mode or running a user application program.

Table 7-1 shows all HCS08 BDC commands, a shorthand description of their coding structure, and the meaning of each command. Subsequent paragraphs describe each command in greater detail.

Table 7-1. BDC Command Summary

Command Mnemonic	Active Background Mode/ Non-Intrusive	Coding Structure[1]	Description
SYNC	Non-intrusive	n/a[2]	Request a timed reference pulse to determine target BDC communication speed
ACK_ENABLE	Non-intrusive	D5/d	Enable handshake. Issues an ACK pulse after the command is executed.
ACK_DISABLE	Non-intrusive	D6/d	Disable handshake. This command does not issue an ACK pulse.
BACKGROUND	Non-intrusive	90/d	Enter active background mode if enabled (ignore if ENBDM bit equals 0)
READ_STATUS	Non-intrusive	E4/SS	Read BDC status from BDCSCR
WRITE_CONTROL	Non-intrusive	C4/CC	Write BDC controls in BDCSCR
READ_BYTE	Non-intrusive	E0/AAAA/d/RD	Read a byte from target memory
READ_BYTE_WS	Non-intrusive	E1/AAAA/d/SS/RD	Read a byte and report status
READ_LAST	Non-intrusive	E8/SS/RD	Re-read byte from address just read and report status
WRITE_BYTE	Non-intrusive	C0/AAAA/WD/d	Write a byte to target memory
WRITE_BYTE_WS	Non-intrusive	C1/AAAA/WD/d/SS	Write a byte and report status
READ_BKPT	Non-intrusive	E2/RBKP	Read BDCBKPT breakpoint register
WRITE_BKPT	Non-intrusive	C2/WBKP	Write BDCBKPT breakpoint register
GO	Active Background Mode	08/d	Go to execute the user application program starting at the address currently in the PC
TRACE1	Active Background Mode	10/d	Trace 1 user instruction at the address in the PC, then return to active background mode
TAGGO	Active Background Mode	18/d	Same as GO but enable external tagging (HCS08 devices have no external tagging pin, so TAGGO is just like GO in an HCS08)
READ_A	Active Background Mode	68/d/RD	Read accumulator (A)

Table 7-1. BDC Command Summary (Continued)

Command Mnemonic	Active Background Mode/ Non-Intrusive	Coding Structure[1]	Description
READ_CCR	Active Background Mode	69/d/RD	Read condition code register (CCR)
READ_PC	Active Background Mode	6B/d/RD16	Read program counter (PC)
READ_HX	Active Background Mode	6C/d/RD16	Read H and X register pair (H:X)
READ_SP	Active Background Mode	6F/d/RD16	Read stack pointer (SP)
READ_NEXT	Active Background Mode	70/d/RD	Increment H:X by one, then read memory byte located at H:X
READ_NEXT_WS	Active Background Mode	71/d/SS/RD	Increment H:X by one, then read memory byte located at H:X. Report status and data.
WRITE_A	Active Background Mode	48/WD/d	Write accumulator (A)
WRITE_CCR	Active Background Mode	49/WD/d	Write condition code register (CCR)
WRITE_PC	Active Background Mode	4B/WD16/d	Write program counter (PC)
WRITE_HX	Active Background Mode	4C/WD16/d	Write H and X register pair (H:X)
WRITE_SP	Active Background Mode	4F/WD16/d	Write stack pointer (SP)
WRITE_NEXT	Active Background Mode	50/WD/d	Increment H:X by one, then write memory byte located at H:X
WRITE_NEXT_WS	Active Background Mode	51/WD/d/SS	Increment H:X by one, then write memory byte located at H:X. Also report status.

1. Key:
   Commands begin with an 8-bit hexadecimal command code in the host-to-target direction (MSB first)
   / — separates parts of the command
   d — delay 16 target BDC clock cycles (the CLKSW bit in BDCSCR controls the source of the BDC clock)
   AAAA — a 16-bit address in the host-to-target direction
   RD — 8 bits of read data in the target-to-host direction
   WD — 8 bits of write data in the host-to-target direction
   RD16 — 16 bits of read data in the target-to-host direction
   WD16 — 16 bits of write data in the host-to-target direction
   SS — contents of BDCSCR in the target-to-host direction (STATUS)
   CC — 8 bits of write data for BDCSCR in the host-to-target direction (CONTROL)
   RBKP — 16 bits of read data in the target-to-host direction (from BDCBKPT breakpoint register)
   WBKP — 16 bits of write data in the host-to-target direction (for BDCBKPT breakpoint register)
2. The SYNC command is a special operation which does not have a command code.

7.3.4.1 SYNC — Request Timed Reference Pulse

The SYNC command is unlike other BDC commands because the host does not necessarily know the correct communications speed to use for BDC communications until after it has analyzed the response to the SYNC command.

To issue a SYNC command, the host:

- Drives the BKGD pin low for at least 128 cycles of the slowest possible BDC clock (Bus rate clock or derivative-specific alternate clock source)
- Drives BKGD high for a brief speedup pulse to get a fast rise time (This speedup pulse is typically one cycle of the host clock which is as fast as the fastest possible target BDC clock.)
- Removes all drive to the BKGD pin so it reverts to high impedance
- Listens to the BKGD pin for the sync response pulse

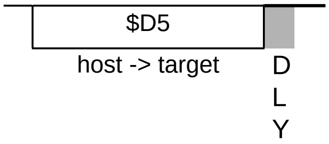
Upon detecting the sync request from the host (which is a much longer low time than would ever occur during normal BDC communications), the target:

- Waits for BKGD to return to a logic high
- Delays 16 cycles to allow the host to stop driving the high speedup pulse
- Drives BKGD low for 128 BDC clock cycles
- Drives a 1-cycle high speedup pulse to force a fast rise time on BKGD
- Removes all drive to the BKGD pin so it reverts to high impedance

The host measures the low time of this 128-cycle sync response pulse and determines the correct speed for subsequent BDC communications. Typically, the host can determine the correct communication speed within a few percent of the actual target speed and the communication protocol can easily tolerate speed errors of several percent.

7.3.4.2 ACK_ENABLE

Enable Host/Target handshake protocol **Non-intrusive**

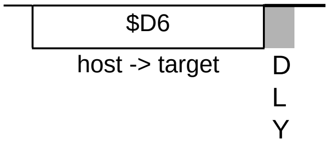


Enables the hardware handshake protocol in the serial communication. The hardware handshake is implemented by an acknowledge (ACK) pulse issued by the target MCU in response to a host command. The ACK_ENABLE command is interpreted and executed in the BDC block without the need to interface with the CPU. However, an acknowledge (ACK) pulse will be issued by the target device after this command is executed. This feature could be used by the host in order to evaluate if the target supports the hardware handshake protocol. If the target supports the hardware handshake protocol the subsequent commands are enabled to execute the hardware handshake protocol, otherwise this command is ignored by the target.

For additional information about the hardware handshake protocol, refer to **7.3.5 Serial Interface Hardware Handshake Protocol** and **7.3.6 Hardware Handshake Abort Procedure**.

7.3.4.3 ACK_DISABLE

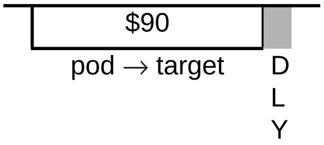
Disable Host/Target handshake protocol **Non-intrusive**



Disables the serial communication handshake protocol. The subsequent commands, issued after the ACK_DISABLE command, will not execute the hardware handshake protocol. This command will not be followed by an ACK pulse.

7.3.4.4 BACKGROUND

Enter Active Background Mode (if Enabled) **Non-intrusive**



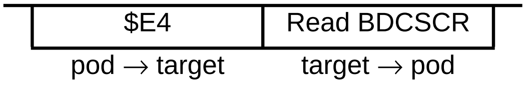
Provided the ENBDM control bit in the BDCSCR is 1 (BDM enabled), the BACKGROUND command causes the target MCU to enter active background mode as soon as the current CPU instruction finishes. If ENBDM is 0 (its default value), the BACKGROUND command is ignored.

A delay of 16 BDC clock cycles is required after the BACKGROUND command to allow the target MCU to finish its current CPU instruction and enter active background mode before a new BDC command can be accepted.

After the target MCU is reset into a normal operating mode, the host debugger would send a WRITE_CONTROL command to enable the active background mode before attempting to send the BACKGROUND command the first time. Normally, the development host would set ENBDM once at the beginning of a debug session or after a target system reset, and then leave the ENBDM bit set during debugging operations. During debugging, the host would use GO and TRACE1 commands to move from active background mode to normal user program execution and would use BACKGROUND commands or breakpoints to return to active background mode.

7.3.4.5 READ_STATUS

Read Status from BDCSCR **Non-intrusive**



This command allows a host to read the contents of the BDC status and control register (BDCSCR). This register is not in the memory map of the target MCU, rather it is built into the BDC logic and is accessible only through READ_STATUS and WRITE_CONTROL serial BDC commands.

The most common use for this command is to allow the host to determine whether the target MCU is executing normal user program instructions or if it is in active background mode. For example, during a typical debug session, the host might set breakpoints in the user's program and then use a GO command to begin normal user program execution. The host would then periodically execute READ_STATUS commands to tell when a breakpoint has been encountered and the target processor has gone into active background mode. Once the target has entered active background mode, the host would read the contents of target CPU registers.

READ_STATUS is also used to tell when a BDC memory write command completes after a DVF failure due to a slow memory access. If a WRITE_BYTE_WS or WRITE_NEXT_WS command indicates a failure due to a slow memory access (DVF = 1), the host should execute READ_STATUS commands until the status response indicates the write access has completed. The write request is latched during the WRITE_BYTE_WS or WRITE_NEXT_WS so there is no need to repeat the write command; just wait for status to indicate the latched request has completed.

READ_STATUS might also be used to check whether the target MCU has gone into wait or stop mode. During a debug session, the host or user may decide it has taken too long to reach a breakpoint in the user program. The host could then issue a READ_STATUS command and check the WS status bit to see if the target MCU is still running user code or if it has entered wait or stop mode. If WS = 0 and BDMACT = 0, meaning it is running user code and is not in wait or stop, the host might choose to issue a BACKGROUND command to stop the user program and enter active background mode where the host can check the CPU registers and find out what the target program is doing.

7.3.4.6 WRITE_CONTROL

Write Control Bits in BDCSCR **Non-intrusive**

$C4	Write BDCSCR
pod → target	pod → target

This command is used to enable active background mode, choose the clock source for BDC communications, and control the hardware breakpoint logic in the BDC by writing to control bits in the BDC status and control register (BDCSCR). This register is not in the memory map of the target MCU, rather it is built into the BDC logic and is only accessible through READ_STATUS and WRITE_CONTROL serial BDC commands. Some bits in BDCSCR have write restrictions such as the status bits BDMACT, WS, WSF, and DVF which are read-only status indicators, and ENBDM which cannot be cleared while BDM is active.

The ENBDM control bit defaults to 0 (active background mode not allowed) when the target MCU is reset in normal operating mode. WRITE_CONTROL is used to enable the active background mode. This is normally done once and ENBDM is left on throughout the debug session. However, the debug system may want to change ENBDM to 0 measure true stop current in the target system (because ENBDM = 1 prevents the clock generation circuitry from disabling the internal clock oscillator or crystal oscillator when the CPU executes a STOP instruction).

The breakpoint enable (BKPTEN) and force/tag select (FTS) control bits are used to control the hardware breakpoint logic in the BDC. This is a single breakpoint that compares the current 16-bit CPU address against the value in the BDCBKPT register. A WRITE_CONTROL command is used to change BKPTEN and FTS, and a WRITE_BKPT command is used to write the 16-bit BDCBKPT address match register.

The CLKSW bit in BDCSCR determines the source of the clock used for BDC communications. If CLKSW = 0 (user mode default), the clock that drives the BDC is the alternate fixed-frequency source. The details of the exact clock source for the BDC in these cases depends on what clock generation circuitry is present in the particular HCS08 derivative MCU. For the MC9S08GB60, when CLKSW = 0, the BDC clock source is a local oscillator in the BDC module (about 8 MHz).

When CLKSW = 1, the CPU bus frequency is used as the clock source to drive BDC communications logic. The CPU bus frequency may be a crystal or an FLL or derived from a PLL. CLKSW should not be set to 1 if the application is using an FLL or PLL and is changing the bus frequency in user programs, because BDC communications require that the host knows the target BDC communications speed and the host has no way to know if/when a user program might change the clock generator settings.

7.3.4.7 READ_BYTE

Read Data from Target Memory Location **Non-intrusive**

$E0	ADDRESS(16)		Read DATA(8)
pod → target	pod → target	D L Y	target → pod

This command is used to read the contents of a memory location in the target MCU without checking the BDC status to be sure the data is valid. In systems which have no slow memory accesses, and the target is currently in active background mode or is known to be executing a program which has no STOP or WAIT instructions, READ_BYTE is faster than the more general READ_BYTE_WS which reports status in addition to returning the requested read data. The most significant use of the READ_BYTE command is during in-circuit FLASH programming where the host downloads data to be programmed at the same time the target CPU is executing the code that actually programs the FLASH memory. Since the host provides the FLASH programming code, it can guarantee that there are no STOP or WAIT instructions.

In general-purpose user programs and especially in programs that have not been debugged, STOP or WAIT instructions and slow memory accesses can occur at any time. To avoid the possibility of invalid read operations, the host should use the READ_BYTE_WS command instead of READ_BYTE to check the status to be sure the read has returned valid data. If the status indicates the read was not valid, the host can execute READ_LAST commands until the status indicates the returned data is valid.

7.3.4.8 READ_BYTE_WS

Read Data from Target and Report Status **Non-intrusive**

$E1	ADDRESS(16)		Read BDCSCR	Read DATA(8)
pod → target	pod → target	DLY	target → pod	target → pod

This is the command normally used by a host debug system to perform general-purpose memory read operations. In addition to returning the data from the requested target memory location, this command returns the contents of the BDC status and control register. The status information can be used to determine whether the data that was returned is valid or not. If a slow memory access was in progress at the time of the read, the data valid failure (DVF) status bit will be 1. If the target MCU was just entering wait or stop mode at the time of the read, the wait/stop failure (WSF) status bit will be 1. If DVF and WSF are both 0, the data that was returned is valid.

In the case of a DVF error, execute READ_LAST commands until the status response indicates the data is correct. In the case of a WSF error, first issue a BACKGROUND command to wake the target CPU from wait or stop and enter active background mode. From there, issue a new READ_BYTE or READ_BYTE_WS command, and if desired adjust the program counter (PC) and stack and re-execute the WAIT or STOP instruction to return the target to wait or stop mode.

If you are sure that the target system has no slow accesses and will not execute a WAIT or STOP instruction during the memory access, use the faster READ_BYTE command instead of READ_BYTE_WS. In user programs that have not been debugged, there is no guarantee that the CPU will not run away and execute an unintended WAIT or STOP instruction.

7.3.4.9 READ_LAST

Re-Read from Last Address with Status **Non-intrusive**

$E8	Read BDCSCR	Read DATA(8)
pod → target	target → pod	target → pod

This command is used only after a READ_BYTE_WS command where the DVF status bit indicated an error. In that case, issue READ_LAST commands until the status bits indicate a valid response. The READ_LAST command uses the memory address from the previous READ_BYTE_WS command so the command is shorter and faster than other read commands.

7.3.4.10 WRITE_BYTE

Write Data to Target Memory Location **Non-intrusive**

$C0	ADDRESS(16)	Write DATA(8)	▨
pod → target	pod → target	pod → target	D L Y

This command is used to write the contents of a memory location in the target MCU without checking the BDC status to be sure the write was completed successfully. In systems which have no slow memory accesses, and the target is currently in active background mode or is known to be executing a program which has no STOP or WAIT instructions, WRITE_BYTE is faster than the more general WRITE_BYTE_WS which reports status in addition to performing the requested write operation. The most significant use of the WRITE_BYTE command is during in-circuit FLASH programming where the host downloads data to be programmed at the same time the target CPU is executing the code that actually programs the FLASH memory. Since the host provides the FLASH programming code, it can guarantee that there are no STOP or WAIT instructions.

In general-purpose user programs and especially in programs that have not been debugged, STOP or WAIT instructions and slow memory accesses can occur at any time. To avoid the possibility of invalid write operations, the host should use the WRITE_BYTE_WS command instead of WRITE_BYTE to check the status to be sure the write was completed successfully.

7.3.4.11 WRITE_BYTE_WS

Write Data to Target and Report Status **Non-intrusive**

$C1	ADDRESS(16)	Write DATA(8)	▨	Read BDCSCR
pod → target	pod → target	pod → target	D L Y	target → pod

This is the command normally used by a host debug system to perform general-purpose memory write operations. In addition to performing the requested write to a target memory location, this command returns the contents of the BDC status and control register. The status information can be used to tell if the write operation was completed successfully. If a slow memory access was in progress at the time of the write, the data valid failure (DVF) status bit will be 1. If the target MCU was just entering wait or stop mode at the time of the read, the wait/stop failure (WSF) status bit will be 1 and the write command is cancelled. If DVF and WSF are both 0, the write operation was completed successfully.

If DVF is set in the returned status value, the write was not completed (although the address and data for the operation are latched). Do READ_STATUS commands until DVF is returned as a 1 to indicate that the write operation was completed successfully. If the WSF bit indicated a WAIT or STOP instruction caused the write operation to fail, do a BACKGROUND command to force the target system out of wait or stop mode and into active background mode. From there, repeat the failed write operation, and if desired adjust the PC and stack and re-execute the WAIT or STOP instruction to return the target to wait or stop mode.

If you are sure that the target system has no slow accesses and will not execute a WAIT or STOP instruction during the memory access, you can use the faster WRITE_BYTE command instead of WRITE_BYTE_WS. In user programs that have not been debugged, there is no guarantee that the CPU will not run away and execute an unintended WAIT or STOP instruction.

7.3.4.12 READ_BKPT

Read 16-Bit BDC Breakpoint Register (BDCBKPT) **Non-intrusive**

$E2	Read data from BDCBKPT register
pod → target	target → pod

This command is used to read the 16-bit BDCBKPT address match register in the hardware breakpoint logic in the BDC.

7.3.4.13 WRITE_BKPT

Write 16-Bit BDC Breakpoint Register (BDCBKPT) **Non-intrusive**

$C2	Write data to BDCBKPT register
pod → target	pod → target

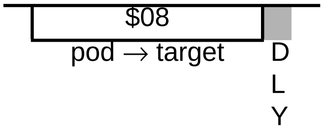
This command is used to write a 16-bit address value into the BDCBKPT register in the BDC. This establishes the address of a breakpoint. The BKPTEN bit in the BDCSCR determines whether the breakpoint is enabled. If BKPTEN = 1 and the FTS control bit in the BDCSCR is set (force), a successful match between the CPU address and the value in the BDCBKPT register will force a transition to active background mode at the next instruction boundary. If BKPTEN = 1 and FTS = 0, the opcode at the address specified in the BDCBKPT register will be tagged as it is fetched into the instruction queue. If and when a tagged opcode reaches the top of the instruction queue and is about to be executed, the MCU will enter active background mode rather than execute the tagged instruction.

In normal debugging environments, breakpoints are established while the target MCU is in active background mode before going to the user's program. However, since this is a non-intrusive command, it could be executed even when the MCU is running a user application program. BDC serial communications are essentially asynchronous to a running user program, so it is impractical to predict the exact time of a BDCBKPT register value change relative to a particular bus cycle of the user's program when the WRITE_BKPT instruction is executed while the user application program is running.

7.3.4.14 GO

Start Execution of User Program Starting at Current PC

Active Background Mode

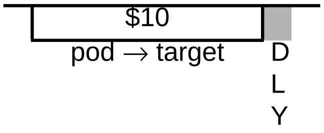


This command is used to exit the active background mode and begin execution of user program instructions starting at the address in the PC. Typically, the host debug monitor program modifies the PC value (using a WRITE_PC command) before issuing a GO command to go to an arbitrary point in the user program. This WRITE_PC command is not needed if the host simply wants to continue the user program where it left off when it entered active background mode.

7.3.4.15 TRACE1

Run One User Instruction Starting at the Current PC

Active Background Mode

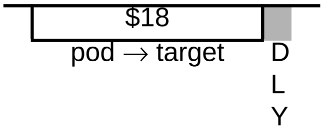


This command is used to run one user instruction and return to active background mode. The address in the PC determines what user instruction will be executed, and the PC value after TRACE1 is completed will reflect the results of the executed instruction.

7.3.4.16 TAGGO

Enable External Tagging and Start Execution of User Program

Active Background Mode

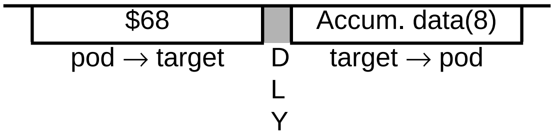


This instruction enables the external tagging function and goes to the user program starting at the address currently in the PC. However, since HCS08 devices do not have an external pin connected to the tagging input of the BDC module, this command is essentially the same as the GO command, so there is no need to use TAGGO commands in an HCS08 system.

7.3.4.17 READ_A

Read Accumulator A of the Target CPU

Active Background Mode

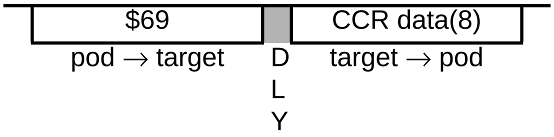


Read the contents of the accumulator (A) of the target CPU. Since the CPU in the target MCU is effectively halted while the target is in active background mode, there is no need to save the target CPU registers on entry into active background mode and no need to restore them on exit from active background to a user program.

7.3.4.18 READ_CCR

Read the Condition Code Register of the Target CPU

Active Background Mode

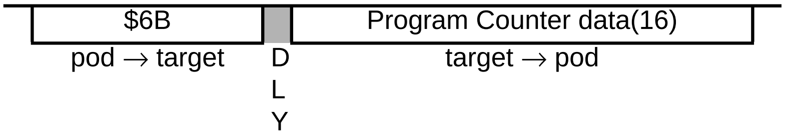


Read the contents of the condition code register (CCR) of the target CPU. Since the CPU in the target MCU is effectively halted while the target is in active background mode, there is no need to save the target CPU registers on entry into active background mode and no need to restore them on exit from active background mode to a user program. The CCR value is not affected by BDC commands (except, of course, the WRITE_CCR command).

7.3.4.19 READ_PC

Read the Program Counter of the Target CPU

Active Background Mode



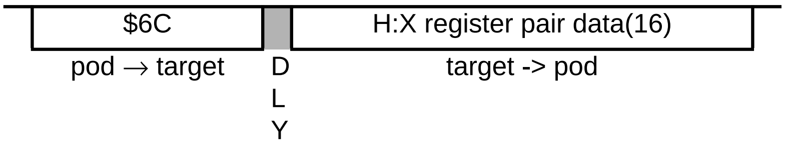
Read the contents of the program counter (PC) of the target CPU. Since the CPU in the target MCU is effectively halted while the target is in active background mode, there is no need to save the target CPU registers on entry into active background mode and no need to restore them on exit from active background mode to a user program.

The value in the PC when the target MCU enters active background mode is the address of the instruction that would have executed next if the MCU had not entered active background mode. If the target CPU was in wait or stop mode when a BACKGROUND command caused it to go to active background mode, the PC will hold the address of the instruction after the WAIT or STOP instruction that was responsible for the target CPU being in wait or stop, and the WS bit will be set. In the boundary case (where an interrupt and a BACKGROUND command arrived at about the same time and the interrupt was responsible for the target CPU leaving wait or stop and then the BACKGROUND command took effect), the WS bit will be clear and the PC will be pointing at the first instruction in the interrupt service routine. In the case of a software breakpoint (where the host placed a BGND opcode at the desired breakpoint address), the PC will be pointing at the address immediately following the inserted BGND opcode, and the host monitor will adjust the PC backward by one after removing the software breakpoint.

7.3.4.20 READ_HX

Read the H:X Register Pair of the Target CPU

Active Background Mode

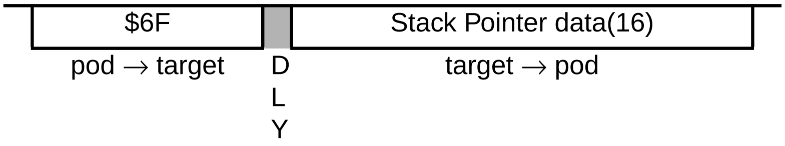


Read the contents of the H:X register pair (H:X) of the target CPU. Since the CPU in the target MCU is effectively halted while the target is in active background mode, there is no need to save the target CPU registers on entry into active background mode and no need to restore them on exit from active background mode to a user program. H and X can be read only as a 16-bit register pair. (There are no BDC commands to read H and X separately.)

7.3.4.21 READ_SP

Read the Stack Pointer of the Target CPU

Active Background Mode

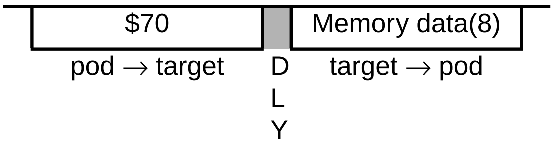


Read the contents of the stack pointer (SP) of the target CPU. Since the CPU in the target MCU is effectively halted while the target is in active background mode, there is no need to save the target CPU registers on entry into active background mode and no need to restore them on exit from active background mode to a user program.

7.3.4.22 READ_NEXT

Increment H:X, Then Read Memory Pointed to by H:X

Active Background Mode



READ_NEXT increments the H:X register pair by one, then reads the memory location pointed to by the incremented 16-bit H:X register pair. This command is similar to the READ_BYTE command except that it uses the value in the H:X index register pair as the address for the operation. There is no address included in this command, so it is more efficient than the READ_BYTE command. Since READ_NEXT uses the H:X register pair of the CPU, it is an active background mode command while READ_BYTE is a non-intrusive command.

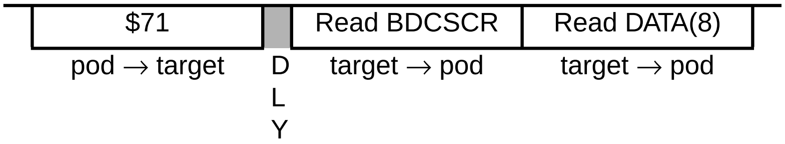
Typically, the host debug system would save the contents of H:X, set H:X to one less than the address of the first byte of a block to be read, execute READ_NEXT commands to read a block of memory, then restore the original contents of H:X (if necessary).

Since READ_NEXT is an active background mode command, there is no concern about errors due to WAIT or STOP instructions and no concern about unexpected slow memory accesses from user code. There could still be slow memory accesses due to the READ_NEXT command itself attempting to access a slow memory location; however, this is completely predictable by the host debug system. In the unusual case of a system that has slow memory and the READ_NEXT operation needs to access memory locations that are slow, use the READ_NEXT_WS command rather than READ_NEXT.

7.3.4.23 READ_NEXT_WS

Increment H:X, Then Read Memory @ H:X and Report Status

Active Background Mode

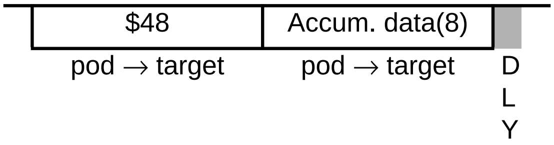


READ_NEXT_WS increments the H:X register pair by one, reads the memory location pointed to by the incremented 16-bit H:X register pair, and returns both the contents of the BDC status and control register (BDCSCR) and the 8-bit data. This command is similar to the READ_NEXT command except that it returns the status from BDCSCR in addition to performing the requested read operation. This status information can be used to tell if the requested read operation returned valid data (DVF = 0). If the status indicates an access failed because it is a slow memory location, execute READ_LAST_WS commands until the status indicates the read data is valid. (Normally, this would require only one READ_LAST_WS command since the BDC serial commands are much slower than the target bus speed.)

7.3.4.24 WRITE_A

Write Accumulator A of the Target CPU

Active Background Mode

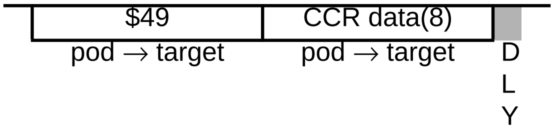


Write new data to the accumulator (A) of the target CPU. This command can be used to change the value in the accumulator before returning to the user application program via a GO or TRACE1 command.

7.3.4.25 WRITE_CCR

Write the Condition Code Register of the Target CPU

Active Background Mode

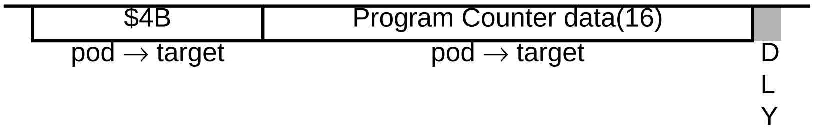


Write new data to the condition code register (CCR) of the target CPU. This command can be used to change the condition codes before returning to the user application program via a GO or TRACE1 command. Other BDC commands do not alter the states of any condition code bits.

7.3.4.26 WRITE_PC

Write the Program Counter of the Target CPU

Active Background Mode

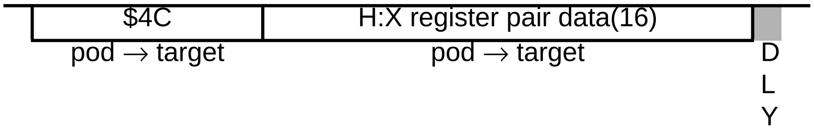


This command is used to change the contents of the program counter (PC) of the target CPU before returning to the user application program via a GO or TRACE1 command.

7.3.4.27 WRITE_HX

Write the H:X Register Pair of the Target CPU

Active Background Mode



Write new data to the H:X index register pair (H:X) of the target CPU. This command can be used to change the value in the 16-bit index register pair (H:X) before returning to the user application program via a GO or TRACE1 command.

7.3.4.28 WRITE_SP

Write the Stack Pointer of the Target CPU **Active Background Mode**

$4F	Stack Pointer data(16)	
pod → target	pod → target	D L Y

Write new data to the stack pointer (SP) of the target CPU. This command can be used to change the value in the stack pointer before returning to the user application program via a GO or TRACE1 command.

7.3.4.29 WRITE_NEXT

Increment H:X, Then Write Memory Pointed to by H:X **Active Background Mode**

$50	Memory data(8)	
pod → target	pod → target	D L Y

WRITE_NEXT increments the H:X register pair by one, then writes to the memory location pointed to by the incremented 16-bit H:X register pair. This command is similar to the WRITE_BYTE command except that it uses the value in the H:X index register pair as the address for the operation. Because no address is included in this command, it is more efficient than the WRITE_BYTE command. Since WRITE_NEXT uses the H:X register pair of the CPU, it is an active background mode command while WRITE_BYTE is a non-intrusive command.

Typically, the host debug system would save the contents of H:X, set H:X to one less than the address of the first byte of a block to be written, execute WRITE_NEXT commands to write a block of memory, then restore the original contents of H:X, if necessary.

Since WRITE_NEXT is an active background mode command, there is no concern about errors due to WAIT or STOP instructions and no concern about unexpected slow memory accesses from user code.

There could still be slow memory accesses due to the WRITE_NEXT command itself attempting to access a slow memory location; however, this is completely predictable by the host debug system. In the unusual case of a system that has slow memory and the WRITE_NEXT operation needs to access memory locations that are slow, use the WRITE_NEXT_WS command rather than WRITE_NEXT.

7.3.4.30 WRITE_NEXT_WS

Increment H:X, Then Write Memory @ H:X and Report Status

Active Background Mode

$51	Memory data(8)		Read BDCSCR
pod → target	pod → target	DLY	target → pod

WRITE_NEXT_WS increments the H:X register pair by one, writes to the memory location pointed to by the incremented 16-bit H:X register pair, attempts to perform the requested write operation, and returns the contents of the BDC status and control register (BDCSCR). This command is similar to the WRITE_NEXT command except that it returns the status from BDCSCR in addition to performing the requested write operation. This status information can be used to tell if the requested write operation was completed successfully (DVF=0). If the status indicates an access failed because it is a slow memory location, the address and data for the operation are latched and you should execute READ_STATUS commands until the status indicates the write was completed successfully. (This would normally only require one READ_STATUS command since the BDC serial commands are much slower than the target bus speed.)

7.3.5 Serial Interface Hardware Handshake Protocol

BDC commands that require CPU execution are ultimately treated at the MCU bus rate. Since the BDC clock source can be asynchronous relative to the bus frequency, when CLKSW = 0, it is necessary to provide a handshake protocol in which the host could determine when an issued command is executed by the CPU. This sub-section will describe the hardware handshake protocol.

The hardware handshake protocol signals to the host controller when an issued command was successfully executed by the target. This protocol is implemented by a low pulse (16 BDC clock cycles) followed by a brief speedup pulse on the BKGD pin, generated by the target MCU when a command, issued by the host, has been successfully executed. See **Figure 7-6**. This pulse is referred to as the ACK pulse. After the ACK pulse is finished, the host can start the data-read portion of the command if the last issued command was a read command, or start a new command if the last command was a write command or a control command (BACKGROUND, GO, GO_UNTIL or TRACE1). The ACK pulse is not issued earlier than 32 BDC clock cycles after the BDC command was issued. The end of the BDC command is assumed to be the 16th BDC clock cycle of the last bit. This minimum delay assures enough time for the host to recognize the ACK pulse. Note also that there is no upper limit for the delay between the command and the related ACK pulse, since the command execution depends on the CPU bus frequency, which in some cases could be very slow compared to the serial communication rate. This protocol allows great flexibility for pod designers, since it does not rely on any accurate time measurement or short response time to any event in the serial communication.

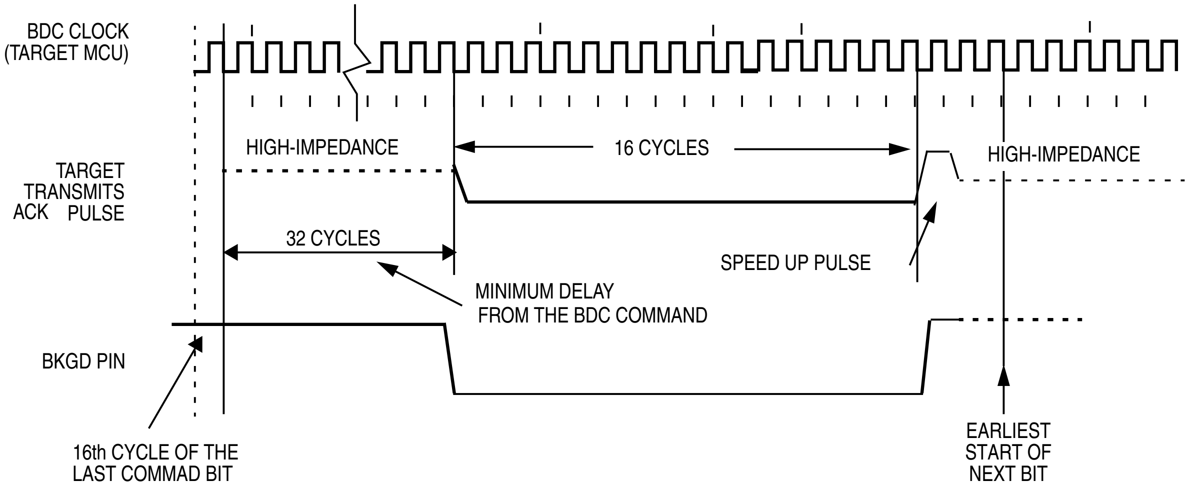


Figure 7-6. Target Acknowledge Pulse (ACK)

NOTE: *If the ACK pulse was issued by the target, the host assumes the previous command was executed. If the CPU enters WAIT or STOP prior to executing a non-intrusive command, the ACK pulse will not be issued, meaning that the BDC command was not executed. After entering WAIT or STOP mode, the BDC command is no longer pending and the DVF status bit is kept one until the next command is successfully executed.*

Figure 7-7 shows the ACK handshake protocol in a command level timing diagram. The READ_BYTE command is used as an example. First, the 8-bit command code is sent by the host, followed by the address of the memory location to be read. The target BDC decodes the command and sends it to the CPU. Upon receiving the BDC command request, the CPU completes the current instruction being executed, the CPU is temporarily halted, the BDC executes the READ_BYTE command and then the CPU continues. This process is referred to as cycle stealing. The READ_BYTE command takes two bus cycles in order to be executed by the CPU. After that, the CPU notifies to the BDC that the requested command was done and then resumes the normal flow of the application program. After detecting the READ_BYTE command is done, the BDC issues an ACK pulse to the host controller, indicating that the addressed byte is ready to be retrieved. After detecting the ACK pulse, the host initiates the data-read portion of the command.

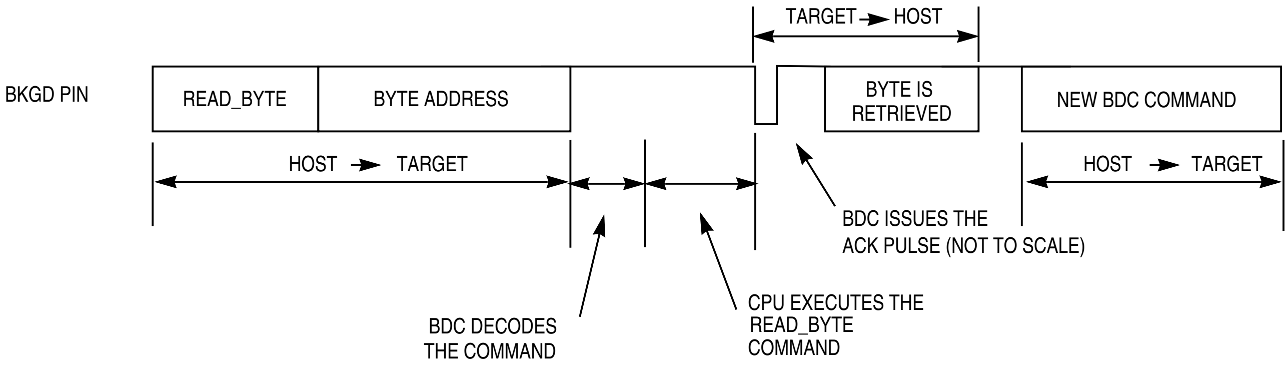


Figure 7-7. Handshake Protocol at Command Level

Unlike a normal bit transfer, where the host initiates the transmission by issuing a negative edge in the BKGD pin, the serial interface ACK handshake pulse is initiated by the target MCU. The hardware handshake protocol in **Figure 7-6** specifies the timing when the BKGD pin is being driven, so the host should follow these timing constraints in order to avoid the risks of an electrical conflict at the BKGD pin.

The ACK handshake protocol does not support nested ACK pulses. If a BDC command is not acknowledged by an ACK pulse, the host first needs to abort the pending command before issuing a new BDC command. When the CPU enters WAIT or STOP mode at about the same time the host issues a command (such as WRITE_BYTE) that requires CPU execution, the target discards the incoming command. Therefore, the command is not acknowledged by the target, which means that the ACK pulse will not be issued in this case. After a certain time the host could decide to abort the ACK protocol in order allow a new command. Therefore, the protocol provides a mechanism in which a command, and therefore a pending ACK, could be aborted. Note that, unlike a regular BDC command, the ACK pulse does not provide a timeout. In the case of a WAIT or STOP instruction where the ACK is prevented from being issued, the ACK would remain pending indefinitely if not aborted. See the handshake abort procedure described in section **7.3.6 Hardware Handshake Abort Procedure** below.

7.3.6 Hardware Handshake Abort Procedure

The abort procedure is based on the SYNC command. In order to abort a command that has not responded with an ACK pulse, the host controller should generate a sync request (by driving BKGD low for at least 128 serial clock cycles and then driving it high for one serial clock cycle as a speedup pulse). By detecting this long low pulse on the BKGD pin, the target executes the sync protocol (see **7.3.4.1 SYNC — Request Timed Reference Pulse**), and assumes that the pending command and therefore the related ACK pulse, are being aborted. Therefore, after the sync protocol completes, the host is free to issue new BDC commands.

Although it is not recommended, the host could abort a pending BDC command by issuing a low pulse on the BKGD pin that is shorter than 128 BDC clock cycles, which will not be interpreted as the SYNC command. The ACK is actually aborted when a negative edge is perceived by the target on the BKGD pin. The short abort pulse should be at least four BDC clock cycles long to allow the negative edge to be detected by the target. In this case the target will not execute the sync protocol but the pending command will be aborted along with the ACK pulse. The potential problem with this abort procedure is when there is a conflict between the ACK pulse and the short abort pulse. In this case the target would not recognize the abort pulse. The worst case is when the pending command is a read command, as for instance the READ_BYTE. If the abort pulse is not perceived by the target, the host will attempt to send a new command after the abort pulse was issued, while the target expects the host to retrieve the accessed memory byte. Host and target will run out of synchronization in this case. However, if the command to be aborted is not a read command, the short abort pulse could be used. After a command is aborted, the target assumes that the next negative edge, after the abort pulse, is the first bit of a new BDC command.

NOTE: *The details about the short abort pulse are being provided only as a reference for the reader to better understand the BDC internal behavior. It is not recommended that this procedure be used in a real application.*

Note that, since the host knows the target BDC clock frequency, the SYNC command does not need to consider the lowest possible target frequency. In this case, the host could issue a SYNC very close to the 128 serial clock cycles length, just providing a small overhead on the pulse length in order to assure the sync pulse will not be misinterpreted by the target. See **7.3.4.1 SYNC — Request Timed Reference Pulse**.

It is important to notice that any issued BDC command that requires CPU execution will be executed at the next instruction boundary, provided the CPU does not enter WAIT or STOP modes. If the host aborts a command by sending the sync pulse, it should then read the BDCSCR after the sync response is issued by the target, checking for DVF = 0, before attempting to send any new command that requires CPU execution. This prevents the new command from being discarded at the BDC-CPU interface, due to the pending command being executed by the CPU. Any new command should be issued only after DVF = 0.

There are two reasons that could cause a command to take too long to be executed, measured in terms of the serial communication rate. Either the BDC clock frequency is much faster than the CPU bus clock frequency, or the CPU is accessing a slow memory, which would cause suspend cycles to occur. The hardware handshake protocol is appropriate for both situations, but the host could also decide to use the software handshake protocol instead. In this case, if the DVF bit is at logic 1, there is a BDC command pending at the BDC-CPU interface. The host controller should monitor the DVF bit and wait until it is at logic 0 in order to be able to issue a new command that requires CPU execution. Note that the WSF bit in the BDCSCR register should be at logic 0 in this case. However, if the WSF bit was at logic 1, the host should assume the last command failed due to a WAIT or STOP instruction being executed by the CPU. In this case, the host controller should enable background mode, using a WRITE_CONTROL command, and then issue a BACKGROUND command in order to put the CPU into active background mode. After that, new commands could be issued, including those that require CPU execution.

Figure 7-8 shows a SYNC command aborting a READ_BYTE. Note that after the command is aborted, a new command could be issued by the host computer.

NOTE: *Figure 7-8 signal timing is not drawn to scale.*

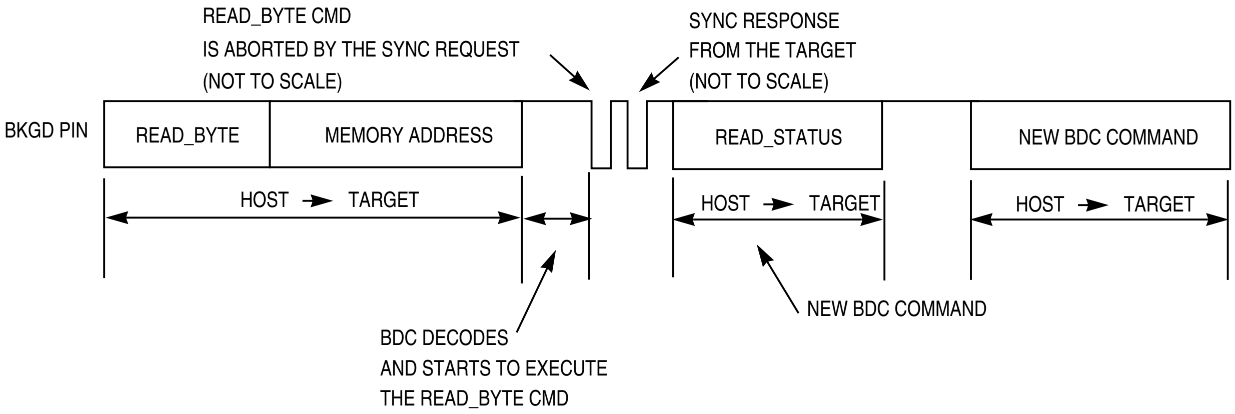


Figure 7-8. ACK Abort Procedure at the Command Level

Figure 7-9 shows a conflict between the ACK pulse and the sync request pulse. This conflict could occur if a pod device is connected to the target BKGD pin and the target is already executing a BDC command. Consider that the target CPU is executing a pending BDC command at the exact moment the pod is being connected to the BKGD pin. In this case an ACK pulse is issued at the same time as the SYNC command. In this case there is an electrical conflict between the ACK speedup pulse and the sync pulse. Since this is not a probable situation, the protocol does not prevent this conflict from happening.

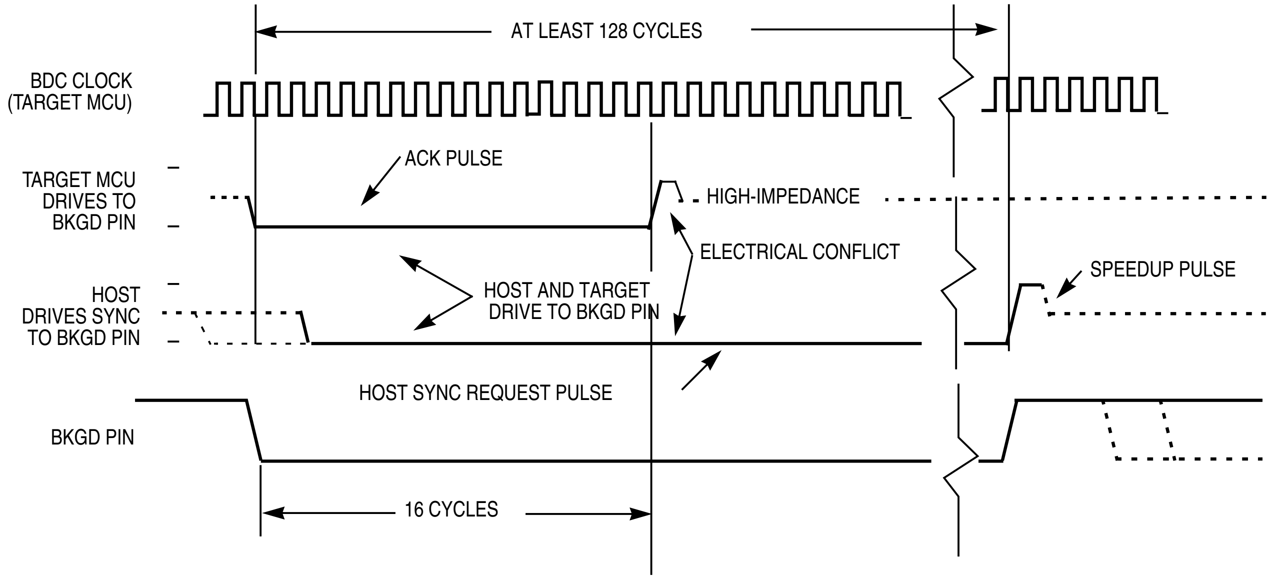


Figure 7-9. ACK Pulse and SYNC Request Conflict

The hardware handshake protocol is enabled by the ACK_ENABLE command and disabled by the ACK_DISABLE command. It also allows for pod devices to choose between the hardware handshake protocol or the software protocol that monitors the BDC status register. The ACK_ENABLE and ACK_DISABLE commands are:

- ACK_ENABLE — Enables the hardware handshake protocol. The target will issue the ACK pulse when a CPU command is executed by the CPU. The ACK_ENABLE command itself also has the ACK pulse as a response.
- ACK_DISABLE — Disables the ACK pulse protocol. In this case the host should verify the state of the DVF bit in the BDC Status and Control register in order to evaluate if there are pending commands and to check if the CPU changed to or from active background mode.

The default state of the protocol, after reset, is hardware handshake protocol disabled.

The commands that do not require CPU execution, or that have the status register included in the retrieved bit stream, do not perform the hardware handshake protocol. Therefore, the target will not respond with an ACK pulse for those commands even if the hardware protocol is enabled. The commands are: READ_STATUS, WRITE_CONTROL, WRITE_BYTE_WS, READ_BYTE_WS, READ_NEXT_WS, WRITE_NEXT_WS, WRITE_BKPT, READ_BKPT, READ_LAST and ACK_DISABLE. See **7.3.4 BDC Commands** for more information on the BDC commands.

NOTE: *The TAGGO command does not have the ACK pulse as a response. Except for no ACK pulse, this command is equivalent to the GO command. It was implemented for compatibility with previous BDC versions. The HCS08 core does not provide support for external tag using the BKGD pin.*

Only commands that require CPU execution perform the hardware handshake protocol. These commands are: WRITE_BYTE, READ_BYTE, WRITE_NEXT, READ_NEXT, WRITE_A, READ_A, WRITE_CCR, READ_CCR, WRITE_SP, READ_SP, WRITE_HX, READ_HX, WRITE_PC, READ_PC. An exception is the ACK_ENABLE command, which does not require CPU execution but responds with the ACK pulse. This feature could be used by the host to evaluate if the target supports the hardware handshake protocol. If an ACK pulse is issued in response to this command, the host knows that the target supports the hardware handshake protocol. If the target does not support the hardware handshake protocol the ACK pulse is not issued. In this case the ACK_ENABLE command is ignored by the target, since it is not recognized as a valid command.

The BACKGROUND command will issue an ACK pulse when the CPU changes from running user code to active background mode. The ACK pulse related to this command could be aborted using the SYNC command.

The GO command will issue an ACK pulse when the CPU exits from active background mode. The ACK pulse related to this command could be aborted using the SYNC command.

The TRACE1 command has the related ACK pulse issued when the CPU enters active background mode after one instruction of the application program is executed. The ACK pulse related to this command could be aborted using the SYNC command.

The GO_UNTIL command is equivalent to a GO command with exception that the ACK pulse, in this case, is issued when the CPU enters into active background mode. This command is an alternative to the GO command and should be used if the host wants to trace if a breakpoint match had occurred which caused the CPU to enter active background mode. Note that the ACK is issued whenever the CPU enters BDM, which could be caused by a BDC breakpoint match, or an external force/tag, or by a BGND instruction being executed. The ACK pulse related to this command could be aborted using the SYNC command.

The TAGGO command is equivalent to the GO command, but will not have an ACK pulse as a response. This command is being kept for backwards compatibility reasons. The GO command should be used instead.

7.3.7 BDC Hardware Breakpoint

The BDC includes one relatively simple hardware breakpoint which compares the CPU address bus to a 16-bit match value in the BDCBKPT register. This breakpoint can generate a forced breakpoint or a tagged breakpoint. A forced breakpoint causes the CPU to enter active background mode at the first instruction boundary following any access to the breakpoint address. The tagged breakpoint causes the instruction opcode at the breakpoint address to be tagged so that the CPU will enter active background mode rather than executing that instruction if and when it reaches the end of the instruction queue. This implies that tagged breakpoints can be placed only at the address of an instruction opcode while forced breakpoints can be set at any address.

The breakpoint enable (BKPTEN) control bit in the BDC status and control register (BDCSCR) is used to enable the breakpoint logic (BKPTEN = 1). When BKPTEN = 0, its default value after reset, the breakpoint logic is disabled and no BDC breakpoints are requested regardless of the values in other BDC breakpoint registers and control bits. The force/tag select (FTS) control bit in BDCSCR is used to select forced (FTS = 1) or tagged (FTS = 0) type breakpoints.

The 8-bit BDCSCR and the 16-bit BDCBKPT address match register are built directly into the BDC and are not accessible in the normal MCU memory map. This means that the user application program cannot access these registers. Dedicated BDC serial commands are the only way to access these registers. READ_STATUS and WRITE_CONTROL are used to read or write BDCSCR, respectively. READ_BKPT and WRITE_BKPT are used to read or write the 16-bit BDCBKPT address match register. A host debug pod can read or write these registers at any time even while a user application program is running. However, it is more common to adjust breakpoint settings while the MCU is in active background mode.

The BDC provides access to control and status signals, which allows more complex breakpoints to be built outside the BDC logic but still on the MCU chip. Some HCS08 derivatives may have additional, more complex, hardware breakpoints. These additional breakpoints need any associated registers and control bits to be accessible through reads and writes to addresses in the normal MCU memory map.

7.3.8 Differences from M68HC12 BDM

Although the bit-level communication protocol is the same as the background debug mode (BDM) interface in the M68HC12 Family, the HCS08 has implemented the background debug controller (BDC) differently than the M68HC12 to reduce the silicon area and to provide new capabilities.

In the M68HC12, the BDM is implemented separately from the CPU and uses a small firmware ROM to control active background mode operations. The HCS08, on the other hand, incorporates background functions directly into the logic of the core CPU, thus eliminating the firmware ROM.

In the HCS08, BDC registers are never in the memory map of the target MCU, so there is no need for the READ_BD_BYTE, READ_BD_WORD, WRITE_BD_BYTE, and WRITE_BD_WORD commands of the M68HC12.

Since the HCS08 CPU has a different CPU register model, the BDC commands that read and write CPU registers are different than those for the M68HC12. In the M68HC12 BDM, the condition codes were stored in a register in the BDM memory map so reading and writing the CCR were done with READ_BD_BYTE and WRITE_BD_BYTE commands. READ_BD_BYTE and WRITE_BD_BYTE were also used to read and write the BDM status register. In the HCS08, however, there are separate commands for reading and writing the status/control register which is not in the memory map of the MCU.

7.3.8.1 8-Bit Architecture

Unlike the 16-bit M68HC12, the HCS08 is an 8-bit architecture. Because of this, the HCS08 BDC does not have word-sized read and write commands. Also, the READ_NEXT and WRITE_NEXT commands operate on byte-sized data rather than word-sized data.

7.3.8.2 Command Formats

All data fields in the M68HC12 BDM are 16 bits even if the command only requires eight bits of data. In contrast, in the HCS08, data fields match the size of the data needed so a command like READ_BYTE will have an 8-bit data field while RD_BYTE_WS has a 16-bit data field to hold the BDC system STATUS byte followed by the data byte.

In the M68HC12 Family, the BDM can wait up to 128 cycles for a free bus cycle to appear to allow the BDM access without disturbing the running user application program. If no free cycle is found, the BDM temporarily freezes the CPU to allow the BDM to complete the requested operation. In the HCS08, this has been simplified such that the BDC always steals a cycle as soon as it can. This has little impact on real-time operation of the user's code because a memory access command takes 8 bits for the command, 16 bits for the address, at least eight bits for the data, and a 16-cycle delay within the command. Each bit time is at least

16 BDC clock cycles so (32 x 16) +16 = 528 cycles, thus the worst case impact is no more than 1/528 cycles, even if there are continuous back-to-back memory access commands through the BDM (which would be very unlikely).

Since the HCS08 BDC doesn’t wait for free cycles, the delays between address and data in read commands and the delay after the data portion of a write command can be much shorter than the 150 cycles recommended for the M68HC12 BDM. In the HCS08, the delay within a memory access command is 16 target bus cycles. For accesses to registers within the BDC (STATUS, and BDCBKPT address match registers), no delay is needed.

7.3.8.3 Read and Write with Status

Because the memory access commands in the HCS08 BDC are actually performed by the CPU circuitry, it is possible for a memory access to fail to complete within the BDC command. The two cases where this can occur are: When the memory access command coincides with the CPU entering stop or wait, or if the CPU was performing a slow memory access when the BDC command arrived. (In HCS08 versions that do not include slow memory devices, this case cannot occur.)

Since there is normally no way to predict when the target CPU might perform a slow access or a STOP or WAIT instruction, the DVF status bit was added to indicate an access error due to a slow access, and the WSF status bit was added to indicate an access failed because the CPU was just entering wait or stop mode. Alternate variations of the READ_BYTE, WRITE_BYTE, READ_NEXT, and WRITE_NEXT commands have been added which automatically return the contents of the BDC status register along with the data portion of the command. In the case of the READ_BYTE and READ_NEXT commands, the READ_BYTE_WS and READ_NEXT_WS commands can be thought of as returning 16 bits of data. In the case of the WRITE_BYTE and WRITE_NEXT commands, the WRITE_BYTE_WS and WRITE_NEXT_WS commands include the byte of status information in the target-to-host direction after the write data byte (which is in the host-to-target direction).

7.3.8.4 BDM Versus Stop and Wait Modes

In the M68HC12 Family, the BDM system is implemented independently from the CPU so memory access commands can still be performed while the target MCU is in wait mode. Stop mode in the M68HC12 causes the oscillator, from which all system clocks are derived, to be stopped. The BDM ceases to function because it has no clocks.

However, the clock architecture of the HCS08 permits the BDC to prevent the oscillator from stopping during stop mode if the ENBDM control bit is set. In such a system, the debug host can use READ_STATUS commands to tell if the target is in wait or stop mode. If the target is in wait or stop (WS bit equals 1), the BACKGROUND command may be used to awaken the target and place it in active background mode.

From active background mode, the debug host can read or write memory or registers. The debug host can then choose to adjust the stack and PC such that a GO command will return the target MCU to wait or stop mode.

7.3.8.5 SYNC Command

The HCS08 has added a SYNC command to allow the host interface pod to determine the correct speed for optimum communications with the target MCU. This is especially useful when the BDC clock in the target MCU is operating from an internal self-clocked local oscillator rather than the CPU bus clock.

To use the SYNC command, the host drives the BKGD pin low for at least 128 target BDC clock cycles then releases the low drive and drives a brief speedup pulse to snap the BKGD pin back to a good logic high level before reverting to high impedance. After a delay to allow the BKGD pin to reach a good high level and to avoid possible interference with the high-driven speedup pulse from the host, the target will drive the BKGD pin low for 128 target BDC clock cycles followed by a 1-cycle driven-high speedup pulse and then reverts to high impedance. The host can measure the duration of this sync pulse to accurately determine the speed of the target’s BDC clock.

7.3.8.6 Hardware Breakpoint

The BDC in the HCS08 includes one 16-bit hardware breakpoint which triggers on a match against the 16-bit address bus. Specific HCS08 derivatives may include additional on-chip hardware breakpoints outside the BDC. The READ_BKPT and WRITE_BKPT commands allow reading or writing the BDCBKPT (address match) register which is built into the BDC logic.

There are also two control bits for the breakpoint in the BDCSCR:

- The BKPTEN bit enables the breakpoint to generate a trigger event in response to a match between the BDCBKPT register and the CPU address bus.
- The force/tag select (FTS) bit determines what a breakpoint trigger event does.

If FTS = 1 (force), the trigger event causes the target MCU to enter active background mode at the next instruction boundary. If FTS = 0 (tag), the trigger event causes the fetched data value to be tagged as it enters the instruction queue. If and when this tagged opcode reaches the top of the queue, the target MCU enters active background mode rather than executing the tagged instruction. The address in the BDCBKPT register must point to an instruction opcode for the tag type breakpoints, but it can be set to any address for a force type breakpoint.

7.4 Part Identification and BDC Force Reset

HCS08 devices include two additional development support features that are not part of the background debug controller (BDC) or debug (DBG) modules. These registers are described in this section.

A 16-bit register pair in the system integration module (SDIDH:SDIDL) provides a way for a development host to determine the derivative type and mask set revision of a target MCU. This allows the development system to associate a register definition file with the target MCU so debug software in the host can know where various memory blocks start and end in the target and the locations for registers and control bits.

An 8-bit control register includes a BDM force reset (BDFR) control bit that allows a host development system to reset the target MCU via a serial command through the background debug communication interface. The BDFR bit is not accessible by user application programs in the target MCU so there is no possibility that a runaway program could accidentally trigger this reset function.

7.4.1 System Device Identification Registers (SDIDH:SDIDL)

This 16-bit read-only register pair is hard-coded with the mask set revision number and derivative identification code.

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	REV3	REV2	REV1	REV0	ID11	ID10	ID9	ID8
Reset:	The value of these bits depends on the device type and mask set revision.							
Read:	ID7	ID6	ID5	ID4	ID3	ID2	ID1	ID0
Reset:	The value of these bits depends on the device type and mask set revision.							

Figure 7-10. System Device Identification Register

REV[3:0] — Mask set Revision Number

This 4-bit field is hard coded to reflect the mask set revision number (0–F) for the MCU die. The initial release of a part is revision number 0:0:0:0.

ID[11:0] — Part Identification Code

This 12-bit field is hard coded with an identification number that identifies the HSC08 derivative type. For example the code for the MC9S08GB60 is $002. Refer to the technical data sheet for other derivatives to find their codes.

7.4.2 System Background Debug Force Reset Register

This register is located in the system integration module, not in the BDC. The system background debug force reset register (SBDFR) is an 8-bit register containing a single control bit which is accessible only from the background debug controller. A serial background command such as WRITE_BYTE must be used to write to SBDFR and attempts to write this register from a user program are ignored. Unlike the other registers in the BDC, SBDFR is located in the normal address space of the MCU (normally located at $1801).

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	0	0	0	0	0	0	0	0
Write:	▨	▨	▨	▨	▨	▨	▨	BDFR
Reset:	0	0	0	0	0	0	0	0

▨ = Unimplemented or Reserved

Figure 7-11. System Background Debug Force Reset Register (SBDFR)

BDFR — Background Debug Force Reset

This write-only control bit provides a means for the background debug host to reset the target MCU without having access to a reset pin.

1 = Force a target system reset.
0 = Writing 0 has no meaning or effect.

7.5 On-Chip Debug System (DBG)

Since HCS08 devices do not have external address and data buses, the most important functions of an in-circuit emulator have been built onto the chip with the MCU. The debug system consists of an 8-stage FIFO which can store address or data bus information, and a flexible trigger system to decide when to capture bus information and what information to capture. This is a little like having a logic analyzer or bus state analyzer built inside the MCU. The system does not use any MCU pins. Rather, it relies on the background debug system (or the CPU) to access debug control registers and to read results out of the 8-stage FIFO.

Unlike the background debug controller, the debug module does include control and status registers that are accessible in the user's memory map. These registers are located in the high register space to avoid using valuable direct page memory space.

Most of the debug module's functions are used during development, and user programs rarely access any of the control and status registers for the debug module. The two exceptions are a ROM-based debug monitor program and ROM patching, a serial monitor program is discussed in application note AN2140/D. ROM patching is discussed in greater detail in **7.5.9 Hardware Breakpoints and ROM Patching**.

7.5.1 Comparators A and B

Two 16-bit comparators (A and B) can optionally be qualified with the R/W signal and an opcode tracking circuit. R/W can be used to detect matches on only read cycles or only write cycles. Separate control bits allow R/W to be ignored for each comparator. The opcode tracking circuitry optionally allows you to specify that a trigger will occur only if the opcode at the specified address is actually executed as opposed to just being read from memory into the instruction queue. This feature allows you to ignore fetches of instructions where a change of flow from a jump, branch, or interrupt causes the CPU to re-fill the instruction queue rather than execute the unused instructions in the queue. The comparators also are capable of magnitude comparisons to support the inside range and outside range trigger modes. Comparators are disabled temporarily during all BDC accesses.

The A comparator is always associated with the 16-bit CPU address. The B comparator compares to the 16-bit CPU address or the 8-bit CPU data bus, depending on the trigger mode selected. Since the CPU data bus is separated into a read data bus and a write data bus, the RWAEN and RWA control bits are used to decide which of these buses to use in comparisons. If RWAEN = 1 (enabled) and RWA = 0 (write), the CPU's write data bus is used. Otherwise, the CPU's read data bus is used.

The currently selected trigger mode determines what the debugger logic does when a comparator detects a qualified match condition. A match can cause:

- Generation of a breakpoint to the CPU
- Storage of data bus values into the FIFO
- Starting to store change-of-flow addresses into the FIFO (begin type trace)
- Stopping the storage of change-of-flow addresses into the FIFO (end type trace)

7.5.2 Bus Capture Information and FIFO Operation

Although processing technology has made on-chip logic less expensive, it still isn't free. Because of this, the number of words of bus capture information that can be stored at a time is limited (eight words in the first HCS08 devices).

To compensate for this limitation, the debugger uses two strategies:

- For tracking the sequence of program instructions, the FIFO only captures addresses related to changes of flow. This allows an external host development tool to reconstruct the flow through dozens or even hundreds of instructions from the eight change-of-flow events before or after a selected trigger point.
- The second strategy is to selectively capture event information. This technique is used to capture only the data associated with read and/or write accesses to a specific address or register.

The usual way to use the FIFO is to set up the trigger mode and other control options, then arm the debugger. When the FIFO has filled or the debugger has stopped storing data into the FIFO, read the information out of it in the order it was stored into the FIFO. Status bits indicate the number of words of valid information that are in the FIFO as data is stored into it.

In most trigger modes, the information stored in the FIFO consists of change-of-flow addresses (16-bit values). In these cases, read DBGFH then DBGFL to get one word of information out of the FIFO. Reading

DBGFL (the low-order half of the FIFO data port) causes the FIFO to shift so the next word of information is available at the FIFO data port. In the event-only trigger modes, 8-bit data information is stored into the FIFO. In these cases, the high-order half of the FIFO (DBGFH) is not used (always stores and reads 0s) and data is read out of the FIFO by simply reading DBGFL. Each time DBGFL is read, the FIFO is shifted so the next data value is available through the FIFO data port at DBGFL.

In trigger modes where the FIFO is storing change-of-flow addresses, there is a delay between CPU addresses and the input side of the FIFO. One consequence of this delay is that if the trigger event itself is a change-of-flow address or if a change-of-flow address appears during the next two bus cycles after a trigger event starts the FIFO, it will not be saved into the FIFO. In the case of an end-trace, if the trigger event is a change-of-flow, it will be saved as the last change-of-flow entry for that debug run.

In event-only trigger modes where the FIFO is storing data, the BEGIN control bit is ignored and all event-only trigger modes are begin-type traces. The event which triggers the start of FIFO data storage is captured as the first data word in the FIFO.

The FIFO can also be used to generate a profile of executed instruction addresses when the debugger is not armed. When ARM = 0, reading DBGFL causes the address of the currently executing instruction to be saved in the FIFO. To use the profiling feature, a host debugger would read addresses out of the FIFO by reading DBGFH then DBGFL at regular periodic intervals. The first eight values would be discarded because they correspond to the eight DBGFL reads needed to initially fill the FIFO. Additional periodic reads of DBGFH and DBGFL return delayed information about executed instructions so the host debugger can develop a profile of executed instruction addresses.

7.5.3 Change-of-Flow information

To minimize the amount of information stored in the FIFO, only information related to instructions that cause a change to the normal sequential execution of instructions is stored. With knowledge of the source and object code program stored in the target system, an external debugger system can reconstruct the path of execution through many instructions from the change-of-flow information stored in the FIFO.

For conditional branch instructions where the branch is taken (branch condition was true), the source address is stored (the address of the conditional branch instruction). If the external debugger finds such an address in the FIFO, it may assume that the branch was taken. Because BRA and BRN instructions are predictable, these events do not cause change-of-flow information to be stored in the FIFO.

Indirect JMP and JSR instructions use the current contents of the H:X index register pair to determine the destination address, so the external debugger cannot predict the destination address from only information in the source and object code. For this reason, the debug system stores the run-time destination address for any indirect JMP or JSR. However, for other JMP and JSR instructions, the external debugger can determine the destination from known source and object code, so no information is stored in the debug FIFO.

For interrupts, return from interrupt (RTI), or return from subroutine (RTS), the destination address is stored in the FIFO as change-of-flow information. In the case of interrupts, the external debugger could tell where the interrupt vector would take program execution, but the debug module needs to store this destination address (address of the interrupt service routine) so the external debugger knows that an interrupt has taken place and execution continued at this address. The destination of an RTI tells the external debugger where the interrupt was recognized in the normal program sequence. RTI and RTS get their destination address from the current values on the stack. The external debugger cannot reliably predict this return address from only the information in the source and object code. Program errors that cause stack problems can be detected by analysis of the change-of-flow information.

Since the FIFO in this debug module is only eight words deep, some care is required when setting up debug runs. For example, if the FIFO is set up to start capturing change-of-flow addresses just before a small loop or a DBNZ instruction that branches to itself, the FIFO will fill very quickly and the information captured will be of little help in debugging a program. Instead, a debug run could be set for an end-trace to show the execution leading to the first iteration of the loop. Another end-trace could be set up to stop at an instruction just after the loop to monitor the behavior of the program for the last iteration of the loop.

7.5.4 Tag vs. Force Breakpoints and Triggers

Tagging is a term that refers to identifying an instruction opcode as it is fetched into the instruction queue, but not taking any other action until and unless that instruction is actually executed by the CPU. This distinction is important because any change-of-flow from a jump, branch, subroutine call, or interrupt causes some instructions that have been fetched into the instruction queue to be thrown away without being executed. Usually, you are only interested in instructions if they are actually executed so the tag mechanism allows you to selectively ignore fetches that do not lead to execution.

A force-type breakpoint waits for the current instruction to finish and then acts upon the breakpoint request. The usual action in response to a breakpoint is to go to active background mode rather than continuing to the next instruction in the user application program.

The tag vs. force terminology is used in two contexts within the debug module. The first context refers to breakpoint requests from the debug module to the CPU. The second refers to match signals from the comparators to the debugger control logic. When a tag-type break request is sent to the CPU, a signal is entered into the instruction queue along with the opcode so that if/when this opcode ever executes, the CPU will effectively replace the tagged opcode with a BGND opcode so the CPU goes to active background mode rather than executing the tagged instruction (or SWI if background mode is disabled (ENBDM = 0)).

The second context is when the TRGSEL control bit in the DBGT register is set to select tag-type operation. In this case, the output from comparator A or B is qualified by a block of logic in the debug module that tracks opcodes and the debugger only produces a trigger if the opcode at the compare address is actually executed. There is separate opcode tracking logic for each comparator so more than one compare event can be tracked through the rebuilt instruction queue at a time. TRGSEL has no effect on breakpoint requests to the CPU.

7.5.5 CPU Breakpoint Requests

In end-trace debug runs (BEGIN = 0), for all trigger modes except event-only modes, CPU breakpoint requests are generated when the trigger event occurs. In begin-trace debug runs (BEGIN = 1), CPU breakpoint requests are generated when the FIFO has been filled. Event-only trigger modes are always begin trace debug runs, so CPU breakpoint requests are generated when the FIFO has been filled.

BRKEN = TAG = 1 while TRGSEL = BEGIN = 0 is a special case that should be avoided because the results could be confusing. When the address match occurs, a tag-type breakpoint request is issued to the CPU. If an exception occurs before this tag reaches the end of the pipe, the intended opcode will be flushed from the pipe, but the tag request from the DBG module remains active waiting for the CPU to acknowledge that it has entered active background mode. The first opcode for the interrupt service routine will end up getting tagged and this is where the CPU will stop rather than at the intended opcode at the match address. To avoid this case, TRGSEL should have been set to 1.

7.5.6 Trigger Modes

The trigger mode controls the overall behavior of a debug run. The 4-bit TRG field in the DBGT register selects one of nine trigger modes. The TRGSEL control bit in the DBGT register modifies the chosen mode by setting whether comparator signals are qualified by opcode tracking logic. The BEGIN bit in DBGT chooses whether the FIFO begins storing data when the qualified trigger is detected (begin trace) or the FIFO stores data in a circular fashion until the qualified trigger is detected (end trigger).

In all trigger modes except the two event-only modes, the FIFO stores change-of-flow addresses. In event-only trigger modes, the FIFO stores 8-bit data values.

In all trigger modes, a match condition for comparator A and/or B is optionally qualified by read/write (R/W) and pipe rebuild logic. R/W comparison is enabled by the associated RWxEN control bit and can be considered an additional input to the associated comparator. In full trigger modes, RWAEN and RWA can be used to enable comparison of R/W and to control whether data comparisons use the CPU read or write data bus and RWBEN and RWB are ignored. When TRGSEL = 1, the R/W qualified match condition is entered into instruction pipe rebuild logic so the trigger is not produced until/unless the tagged opcode reaches the end of the pipe rebuild logic. In event-only trigger modes, TRGSEL is ignored and match signals are never qualified through the pipe rebuild logic.

Begin-trace debug runs start filling the FIFO when the trigger conditions are met and end when the FIFO becomes full (CNT[3:0] = 1:0:0:0). End-trace debug runs start filling the FIFO in circular fashion when the ARM bit is set to 1, and end when the trigger conditions are met. End-trace debug runs can end before the FIFO is full. If more than eight entries are stored into the FIFO during an end-trace debug run, new entries overwrite the oldest entry in the FIFO so that when the debug run ends, the information in the FIFO will be the eight most recent change-of-flow addresses.

A debug run is started by setting up the DBGT register and then writing a 1 to the ARM bit in the DBGC register which sets the ARMF flag and clears the A and B flags and the CNT bits in DBGS. A begin-trace debug run ends when the FIFO gets full. An end-trace run ends when the selected trigger event occurs. Any debug run can be stopped manually by writing a 0 to the ARM bit or the DBGEN bit in DBGC.

7.5.6.1 A-Only Trigger

In the A-only trigger mode, a qualified match on comparator A sets the AF status flag and generates a trigger event. DBGCAH:DBGCAL is compared against the 16-bit CPU address and triggers may be qualified with R/W (by setting RWAEN = 1) and/or by pipe rebuild logic (by setting TRGSEL = 1).

7.5.6.2 A OR B Trigger

In the A OR B trigger mode, a qualified match on comparator A or on comparator B sets the corresponding AF or BF status flag and generates a trigger event. DBGCAH:DBGCAL and DBGCBH:DBGCBL are compared against the 16-bit CPU address and triggers may be qualified with R/W (by setting RWAEN and/or RWBEN to 1) and/or by pipe rebuild logic (by setting TRGSEL=1).

7.5.6.3 A Then B Trigger

In the A Then B trigger mode, a qualified match on comparator A followed by a qualified match on comparator B generates a trigger event. The AF status flag gets set when a qualified match occurs on comparator A. After AF is set, a qualified match on comparator B sets the BF status flag and generates the trigger. DBGCAH:DBGCAL and DBGCBH:DBGCBL are compared against the 16-bit CPU address and triggers may be qualified with R/W (by setting RWAEN and/or RWBEN to 1) and/or by pipe rebuild logic (by setting TRGSEL = 1).

7.5.6.4 Event-Only B Trigger (Store Data)

In event-only trigger modes, data values are stored in the FIFO rather than change-of-flow addresses. In the event-only B trigger mode, a qualified match on comparator B sets the BF status flag and generates a trigger event. DBGCBH:DBGCBL is compared to the 16-bit CPU address. Triggers may be qualified with R/W by setting RWBEN to 1. Do not use TRGSEL = 1 in an event-only trigger mode. DBGCAH:DBGCAL, RWAEN, and RWA are not used in this mode.

7.5.6.5 A Then Event-Only B Trigger (Store Data)

In event-only trigger modes, data values are stored in the FIFO rather than change-of-flow addresses. In the A then event-only B trigger mode, a qualified match on comparator A sets the AF status flag. After AF is set, a qualified match on comparator B sets the BF status flag and generates a trigger event. DBGCAH:DBGCAL and DBGCBH:DBGCBL are compared to the 16-bit CPU address. Triggers may be qualified with R/W by setting RWAEN and/or RWBEN to 1. Do not use TRGSEL = 1 in an event-only trigger mode.

7.5.6.6 A AND B Data Trigger (Full Mode)

This is called a full mode because address, data, and optionally R/W must all match within the same bus cycle to cause a trigger. In the A AND B data trigger mode, a qualified match on comparator A and on comparator B within the same bus cycle generates a trigger event. The AF and BF status flags get set when a qualified match occurs on comparator A and on comparator B in the same bus cycle. DBGCAH:DBGCAL is compared to the 16-bit CPU address and DBGCBL is compared against the 8-bit CPU data bus. If RWAEN = 1 and RWA = 0, DBGCBL is compared to the CPU write data bus; otherwise, DBGCBL is compared to the CPU read data bus. Triggers may be qualified with R/W (by setting RWAEN to 1) and/or by pipe rebuild logic (by setting TRGSEL = 1). DBGCBH, RWBEN, and RWB are not used in this mode.

7.5.6.7 A AND NOT B Data Trigger (Full Mode)

This is called a full mode because address, data, and optionally R/W are all tested within the same bus cycle to cause a trigger. In the A AND NOT B data trigger mode, a qualified match on comparator A, within a bus cycle where data does not match comparator B, generates a trigger event. The AF and BF status flags get set when a qualified match occurs on comparator A and not on comparator B in the same bus cycle. DBGCAH:DBGCAL is compared to the 16-bit CPU address and DBGCBL is compared against the 8-bit CPU data bus. If RWAEN=1 and RWA=0, DBGCBL is compared to the CPU write data bus, otherwise DBGCBL is compared to the CPU read data bus. Triggers may be qualified with R/W (by setting RWAEN to 1) and/or by pipe rebuild logic (by setting TRGSEL=1). DBGCBH, RWBEN, and RWB are not used in this mode.

7.5.6.8 Inside Range Trigger: A ≤ Address ≤ B

In this trigger mode, the comparators are used in a magnitude comparator mode. If the address is greater than or equal to the address in comparator A in the same cycle when the address is less than or equal to the address in comparator B, the AF and BF status flags are set and a trigger event is generated. DBGCAH:DBGCAL and DBGCBH:DBGCBL are compared against the 16-bit CPU address and triggers may be qualified with R/W (by setting RWAEN and/or RWBEN to 1) and/or by pipe rebuild logic (by setting TRGSEL = 1). Obviously, the address in DBGCAH:DBGCAL should be less than the address in DBGCBH:DBGCBL and if RWAEN = RWBEN = 1, RWA should be the same as RWB.

7.5.6.9 Outside Range Trigger: Address < A or Address > B

In this trigger mode, the comparators are used in a magnitude comparator mode. If the address is less than the address in comparator A or greater than the address in comparator B, a trigger event is generated. The AF status flag is set if the address is less than the address in comparator A and the BF status flag is set if the address is greater than the address in comparator B. DBGCAH:DBGCAL and DBGCBH:DBGCBL are compared against the 16-bit CPU address and triggers may be qualified with R/W (by setting RWAEN and/or RWBEN to 1) and/or by pipe rebuild logic (by setting TRGSEL = 1). Obviously, the address in DBGCAH:DBGCAL should be less than the address in DBGCBH:DBGCBL.

7.5.7 DBG Registers and Control Bits

The debug module includes nine bytes of register space for three 16-bit registers and three 8-bit control and status registers. These registers are located in the high register space of the normal memory map so they are accessible to normal application programs. These registers are rarely, if ever, accessed by normal user application programs with the possible exception of a ROM-based debug monitor or a ROM patching mechanism that uses the breakpoint logic.

The modular methodology that is used for HCS08 MCUs implements the fine address decode within each module, but decode logic at the chip level is used to determine the base location for each module. For this reason, always check the documentation for each derivative to determine absolute address locations for registers. Generally, the user will access registers by name and an equate or header file provided by Motorola will translate the register name into the appropriate absolute address for the specific HCS08 derivative. Since registers may not be located at the same address for every derivative MCU, this book only refers to registers and control bits by their names.

7.5.7.1 Debug Comparator A High Register (DBGCAH)

Compare value bits for the high-order eight bits of comparator A. This register is forced to $00 at reset and can be read any time and written only when the ARM bit in the DBGC register is not set.

7.5.7.2 Debug Comparator A Low Register (DBGCAL)

Compare value bits for the low-order eight bits of comparator A. This register is forced to $00 at reset and can be read any time and written only when the ARM bit in the DBGC register is not set.

7.5.7.3 Debug Comparator B High Register (DBGCBH)

Compare value bits for the high-order eight bits of comparator B. This register is forced to $00 at reset and can be read any time and written only when the ARM bit in the DBGC register is not set.

7.5.7.4 Debug Comparator B Low Register (DBGCBL)

Compare value bits for the low-order eight bits of comparator B. This register is forced to $00 at reset and can be read any time and written only when the ARM bit in the DBGC register is not set.

7.5.7.5 Debug FIFO High Register (DBGFH)

This register provides read-only access to the high-order eight bits of the FIFO. Writes to this register have no meaning or effect. In the event-only modes of operation, the FIFO only stores information into the low-order half of each FIFO word, so this register is not used and will read $00.

Reading DBGFH does not cause the FIFO to shift to the next word. When reading 16-bit words out of the FIFO, read DBGFH before reading DBGFL because reading DBGFL causes the FIFO to advance to the next word of information.

7.5.7.6 Debug FIFO Low Register (DBGFL)

This register provides read-only access to the low-order eight bits of the FIFO. Writes to this register have no meaning or effect.

Reading DBGFL causes the FIFO to shift to the next available word of information. When the debug module is operating in an event-only mode, only 8-bit data is stored into the FIFO (high-order half of each FIFO word is unused). When reading 8-bit words out of the FIFO, simply read DBGFL repeatedly to get successive bytes of data from the FIFO. It isn’t necessary to read DBGFH in this case.

Do not attempt to read data from the FIFO while it is still armed (after arming but before the FIFO is filled or ARMF is cleared) because the FIFO is prevented from advancing during reads of DBGFL. This can result in improper sequencing of information in the FIFO.

Reading DBGFL while the FIFO is not armed causes the current opcode address to be stored to the last location in the FIFO. By reading DBGFH then DBGFL periodically, external host software can develop a profile of program execution. After eight reads from the FIFO, the ninth read will return the information that was stored as a result of the first read. To use the profiling feature, read the FIFO eight times without using the data to prime the sequence and then begin using the data to get a delayed picture of what addresses were executed.

The information stored into the FIFO on reads of DBGFL (while the FIFO is not armed) is the address of the most recently executed opcode. Storing instantaneous address bus values would be much less useful since you wouldn’t know whether these were data, operand, or instruction accesses.

7.5.7.7 Debug Control Register

This register can be read at any time. The DBGEN and ARM bits can be written at any time. The remaining bits in the register can be written only while ARM = 0.

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	DBGEN	ARM	TAG	BRKEN	RWA	RWAEN	RWB	RWBEN
Write:								
Reset:	0	0	0	0	0	0	0	0

Figure 7-12. Debug Control Register (DBGC)

DBGEN — Debug Module Enable Bit

Used to enable the debug module. DBGEN cannot be set to 1 if the MCU is secure.
1 = DBG enabled
0 = DBG disabled

ARM — Arm Control Bit

Controls whether the debugger is comparing and storing information in the FIFO. A write is used to set this bit (and the ARMF bit) and completion of a debug run automatically clears it. Any debug run can be stopped manually by writing 0 to ARM or to DBGEN.
1 = Debugger armed
0 = Debugger not armed

TAG — Tag/Force Select Bit

Controls whether break requests to the CPU will be tag or force type requests. If BRKEN = 0, this bit has no meaning or effect.
1 = CPU breaks requested as tag type requests
0 = CPU breaks requested as force type requests

BRKEN — Break Enable Bit

Controls whether a trigger event will generate a break request to the CPU. Trigger events can cause information to be stored in the FIFO without generating a break request to the CPU. CPU break requests are issued to the CPU when the comparator(s) and R/W meet the trigger requirements. CPU tag requests must coincide with an opcode fetch so TRGSEL never affects when CPU break requests are issued.

1 = Triggers (before TRGSEL qualification) cause a break request to the CPU
0 = Break requests not enabled

RWA — R/W Comparison Value for Comparator A Bit

When RWAEN = 1, this bit determines whether a read or a write access qualifies comparator A. When RWAEN = 0, RWA and the R/W signal do not affect comparator A.

1 = Comparator A can match only on a read cycle.
0 = Comparator A can match only on a write cycle.

RWAEN — Enable R/W for Comparator A Bit

Controls whether the level of R/W is considered for a comparator A match

1 = R/W is used in comparison A.
0 = R/W is not used in comparison A.

RWB — R/W Comparison Value for Comparator B Bit

When RWBEN = 1, this bit determines whether a read or a write access qualifies comparator B. When RWBEN = 0, RWB and the R/W signal do not affect comparator B.

1 = Comparator B can match only on a read cycle.
0 = Comparator B can match only on a write cycle.

RWBEN — Enable R/W for Comparator B Bit

Controls whether the level of R/W is considered for a comparator B match

1 = R/W is used in comparison B.
0 = R/W is not used in comparison B.

7.5.7.8 Debug Trigger Register

This register can be read at any time, but it can be written only while ARM = 0. Bits 4 and 5 are hardwired to 0s.

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	TRGSEL	BEGIN	0	0	TRG			
Write:			▨	▨				
Reset:	0	0	0	0	0	0	0	0

▨ = Unimplemented or Reserved

Figure 7-13. Debug Trigger Register (DBGT)

TRGSEL — Trigger Type Bit

Controls whether the match outputs from comparators A and B are qualified with the opcode tracking logic in the debug module. A separate control bit (TAG) in DBGC controls whether CPU break requests are qualified with separate opcode tracking logic in the CPU.

If TRGSEL is set, a match signal from comparator A or B must propagate through the opcode tracking logic and a trigger event is only signalled if the opcode at the match address is actually executed. This trigger event stops (BEGIN = 0) or starts (BEGIN = 1) the capture of information into the FIFO.

1 = Trigger if opcode at compare address is executed (tag)
0 = Trigger on access to compare address (force)

BEGIN — Begin/End Trigger Select Bit

Controls whether the FIFO starts filling at a trigger or fills in a circular manner until a trigger ends the capture of information. In event-only trigger modes, this bit is ignored and all debug runs are assumed to be begin-type traces.

1 = Trigger initiates data storage (begin trace)
0 = Data stored in FIFO until trigger (end trace)

TRG3:TRG2:TRG1:TRG0 — Select Trigger Mode Bits

Selects one of nine triggering modes

Table 7-2. Trigger Mode Selection

TRG[3:0]	Triggering Mode
0000	A-only
0001	A OR B
0010	A then B
0011	Event-only B (store data)
0100	A then event-only B (store data)
0101	A AND B data (full mode)
0110	A AND NOT B data (full mode)
0111	Inside range: A ≤ address ≤ B
1000	Outside range: address < A or address > B
1001–1111	No trigger

7.5.7.9 Debug Status Register

This is a read-only status register.

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	AF	BF	ARMF	0	CNT			
Write:	▨	▨	▨	▨	▨	▨	▨	▨
Reset:	0	0	0	0	0	0	0	0

▨ = Unimplemented or Reserved

Figure 7-14. Debug Status Register (DBGS)

AF — Trigger Match A Flag

AF is cleared at the start of a debug run and indicates whether a trigger match A condition was met since arming.

1 = Comparator A match

0 = Comparator A has not matched.

BF — Trigger Match B Flag

BF is cleared at the start of a debug run and indicates whether a trigger match B condition was met since arming.
1 = Comparator B match
0 = Comparator B has not matched.

ARMF — Arm Flag

While DBGEN = 1, this status bit is a read-only image of the ARM bit in DBGC. This bit is set by writing 1 to the ARM control bit in DBGC (while DBGEN = 1) and is automatically cleared at the end of a debug run. A debug run is completed when the FIFO is full (begin trace) or when a trigger event is detected (end trace). A debug run can also be ended manually by writing 0 to the ARM or DBGEN bits in DBGC.
1 = Debugger armed
0 = Debugger not armed

CNT3:CNT2:CNT1:CNT0 — FIFO Valid Count

These bits are cleared at the start of a debug run and indicate the number of words of valid data in the FIFO at the end of a debug run. The value in CNT does not decrement as data is read out of the FIFO. The external debug host is responsible for keeping track of the count as information is read out of the FIFO.

Table 7-3. CNT Status Bits

CNT[3:0]	Valid Words in FIFO
0000	No valid data
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8

7.5.8 Application Information and Examples

Assuming no debug run is already in progress (ARMF = 0), the usual sequence used to setup a new debug run is:

1. Write address or address and data match values to DBGCAH:DBGCAL and/or DBGCBH:DBGCBL.
2. Write to DBGT to:
   - Select a begin/end type trace run (BEGIN = 1/0)
   - Select address/opcode qualification (TRGSEL = 0/1)
   - Select 1 of 9 basic trigger modes (TRG[3:0])
3. Write to DBGC to:
   - Enable the DBG module (DBGEN = 1)
   - Decide whether to request a CPU breakpoint (BRKEN = 1)
   - If so, select a force/tag CPU breakpoint type (TAG = 0/1)
   - Arm the debug run (ARM = 1)
   - Setup and enable optional R/W qualifiers
4. Start the user application program with a GO command through the background debug interface. Although it is technically possible to setup a debug run while the application program is running, it is much more common to stop the user application program so it is in active background mode while the debug run is set up.

Depending on the type of debug run that was set up, the target MCU will finish the debug run and enter active background mode, or the host debugger can monitor the ARMF flag through active background mode commands to determine when the run is finished. After the debug run is finished, the host would:

1. Optionally read DBGS to see how many words of information were captured into the debug FIFO. If the host was reading DBGS to determine when the debug run was finished, it may not be necessary to re-read DBGS to get the CNT[3:0] information. For many debug runs, it is safe to assume the FIFO is full, so it is not always necessary to check the CNT[3:0] bits to determine how much information is in the FIFO.

2. Read the FIFO information by repeatedly reading DBGFH then DBGFL. For some debug runs, the information in the FIFO is not important so it is not necessary to read it out. For event type debug runs (TRG[3:0] = 0:0:1:1 or 0:1:0:0, the upper-half each of FIFO word is unused so it is not necessary to read DBGFH.

The four control bits BEGIN and TRGSEL in DBGT, and BRKEN and TAG in DBGC, determine the basic type of debug run as shown in **Table 7-4**. Some of the 16 possible combinations are not used (refer to the notes at the end of the table).

Table 7-4. Basic Types of Debug Runs

BEGIN	TRGSEL	BRKEN	TAG	Type of Debug Run
0	0	0	x[1]	Fill FIFO until trigger address (No CPU breakpoint — keep running)
0	0	1	0	Fill FIFO until trigger address, then force CPU breakpoint
0	0	1	1	Don’t use[2]
0	1	0	x[1]	Fill FIFO until trigger opcode about to execute (No CPU breakpoint — keep running)
0	1	1	0	Don’t use[3]
0	1	1	1	Fill FIFO until trigger opcode about to execute (trigger causes CPU breakpoint)[4]
1	0	0	x[1]	Start FIFO at trigger address (No CPU breakpoint — keep running)
1	0	1	0	Start FIFO at trigger address, force CPU breakpoint when FIFO full
1	0	1	1	Don’t use[4]
1	1	0	x[1]	Start FIFO at trigger opcode, (No CPU breakpoint — keep running)
1	1	1	0	Start FIFO at trigger opcode, force CPU breakpoint when FIFO full
1	1	1	1	Don’t use[5]

[1] When DBGEN = 0, TAG is don’t care (x in the table).

[2] In end trace configurations (BEGIN = 0) where a CPU breakpoint is enabled (BRKEN = 1), TRGSEL should agree with TAG. In this case, where TRGSEL = 0 to select no opcode tracking qualification and TAG = 1 to specify a tag-type CPU breakpoint, the CPU breakpoint would not take effect until sometime after the FIFO stopped storing values. Depending on program loops or interrupts, the delay could be very long.

[3] In end trace configurations (BEGIN = 0) where a CPU breakpoint is enabled (BRKEN = 1), TRGSEL should agree with TAG. In this case, where TRGSEL = 1 to select opcode tracking qualification and TAG = 0 to specify a force-type CPU breakpoint, the CPU breakpoint would erroneously take effect before the FIFO stopped storing values and the debug run would not complete normally.

[4] In begin trace configurations (BEGIN = 1) where a CPU breakpoint is enabled (BRKEN = 1), TAG should not be set to 1. In begin trace debug runs, the CPU breakpoint corresponds to the FIFO full condition which does not correspond to a taggable instruction fetch.

7.5.8.1 Orientation to the Debugger Examples

The following sections describe how to setup debug runs for several common situations. Each of these examples starts with a table similar to the one shown here:

DBGCAH:DBGCAL	RWAEN[1]	RWA[1]	DBGCBH:DBGCBL	RWBEN[1]	RWB[1]	DBGT	DBGC
Opcode address	0	x	Not used	x	x	$00	$D0

[1] RWAEN, RWA, RWBEN, and RWB are actually bits in DBGC. They are broken out in this table for reference.

To set up and use a debug run like that described in each example, write the values in the table to the registers named in the heading for each column. The registers should be written in left-to-right order. The RWAEN, RWA, RWBEN, and RWB values are shown in separate columns of the table for convenience, but these are actually control bits in the DBGC register. These bit values are already reflected in the value for DBGC at the right end of the table and these bits get written when DBGC is written.

Just below this table in each example section, the trigger mode is shown and a description of the contents of the FIFO after the debug run is shown. The trigger mode can be derived from the low-order four bits of the DBGT value shown near the right of the table, but it is listed separately for easier reference. After explaining the details and purpose of each example case, variations are discussed.

7.5.8.2 Example 1: Stop Execution at Address A

DBGCAH:DBGCAL	RWAEN[1]	RWA[1]	DBGCBH:DBGCBL	RWBEN[1]	RWB[1]	DBGT	DBGC
Trigger address A	0	x	Not used	x	x	$00	$D0

[1] RWAEN, RWA, RWBEN, and RWB are actually bits in DBGC. They are broken out in this table for reference.

Trigger mode: A-only
FIFO contents: Not used in this example

This is a simple hardware breakpoint where the CPU will stop executing the application program and enter active background mode as soon as the application program makes any access to the selected address. It generates a force-type breakpoint to the CPU on the first access (R/W is don’t care) to the address stored in comparator A (DBGCAH:DBGCAL). The FIFO, comparator B, and DBGS are not used for this example.

An end trace is used because begin-type traces cause the breakpoint to the CPU to be related to the FIFO full condition rather than the selected trigger conditions.

Variation: To consider only read accesses or only write accesses, change the DBGC value so RWAEN = 1 and use RWA to select reads (1) or writes (0).

7.5.8.3 Example 2: Stop Execution at the Instruction at Address A

DBGCAH:DBGCAL	RWAEN[1]	RWA[1]	DBGCBH:DBGCBL	RWBEN[1]	RWB[1]	DBGT	DBGC
Trigger opcode A	0	x	Not used	x	x	$80	$F0

[1] RWAEN, RWA, RWBEN, and RWB are actually bits in DBGC. They are broken out in this table for reference.

Trigger mode: A-only
FIFO contents: Not used in this example

This example uses a tag-type breakpoint to the CPU to set a single instruction breakpoint at address A. The address stored to comparator A (DBGCAH:DBGCAL) must be the address of an instruction opcode. When the selected instruction is about to execute, the CPU will go to active background mode rather than execute the tagged instruction. 0 is written to RWAEN because in order for the instruction to be entered into the CPU's instruction queue it has to be a read access, so there is no need to check R/W. The FIFO, comparator B, and DBGS are not used for this example.

An end trace is used because begin-type traces cause the breakpoint to the CPU to be related to the FIFO full condition rather than the selected trigger conditions. Since this is an end-type trace and we want a tag-type breakpoint to the CPU, we must also specify a tag-type trigger (TRGSEL = 1). If the specified address is not the address of an instruction opcode, no breakpoint will occur.

7.5.8.4 Example 3: Stop Execution at the Instruction at Address A or Address B

DBGCAH:DBGCAL	RWAEN[1]	RWA[1]	DBGCBH:DBGCBL	RWBEN[1]	RWB[1]	DBGT	DBGC
Trigger opcode A	0	x	Trigger opcode B	0	x	$81	$F0

[1] RWAEN, RWA, RWBEN, and RWB are actually bits in DBGC. They are broken out in this table for reference.

Trigger mode: A or B
FIFO contents: Not used in this example

This example uses tag-type breakpoints to the CPU to set two instruction breakpoints, one at address A and the other at address B. The addresses stored to comparator A (DBGCAH:DBGCAL) and comparator B (DBGCBH:DBGCBL) must be the addresses of instruction opcodes. When either of the selected instructions is about to execute, the CPU will go to active background mode rather than execute the tagged instruction. 0 is written to RWAEN and RWBEN because in order for the instruction to be entered into the CPU's instruction queue it has to be a read access, so there is no need to check R/W. The FIFO and DBGS are not used for this example.

An end trace is used because begin-type traces cause the breakpoint to the CPU to be related to the FIFO full condition rather than the selected trigger conditions. Since this is an end-type trace and we want a tag-type breakpoint to the CPU, we must also specify a tag-type trigger (TRGSEL = 1). If the specified addresses are not the addresses of instruction opcodes, no breakpoint will occur.

7.5.8.5 Example 4: Begin Trace at the Instruction at Address A

DBGCAH:DBGCAL	RWAEN[1]	RWA[1]	DBGCBH:DBGCBL	RWBEN[1]	RWB[1]	DBGT	DBGC
Trigger opcode A	0	x	Not used	x	x	$C0	$D0

[1] RWAEN, RWA, RWBEN, and RWB are actually bits in DBGC. They are broken out in this table for reference.

Trigger mode: A-only
FIFO contents: Information from the next eight changes of flow starting from the third bus cycle after the instruction at address A began to execute.

This is an example of a begin-trace debug run that starts filling the FIFO when the instruction at address A is executed and ends when the FIFO is full (has stored eight change-of-flow addresses). Because of a delay in the debug logic, the first possible change-of-flow address that will be captured into the FIFO is the third bus cycle after the trigger event that starts the debug run. If the address when the instruction that caused the trigger, or either of the next two bus cycles is a change-of-flow address, it will not be captured as one of the eight change-of-flow addresses in the FIFO for this debug run.

A force-type CPU breakpoint is specified because this breakpoint is associated with the FIFO full condition and not a taggable opcode. The CPU breakpoint causes the target MCU to go to active background mode as soon as the FIFO is full. Typically, a host development system would then read the contents of the FIFO in order to reconstruct what happened during the debug run.

7.5.8.6 Example 5: End Trace to Stop After A-Then-B Sequence

DBGCAH:DBGCAL	RWAEN[1]	RWA[1]	DBGCBH:DBGCBL	RWBEN[1]	RWB[1]	DBGT	DBGC
Trigger opcode A	0	x	Trigger opcode B	0	x	$82	$F0

[1] RWAEN, RWA, RWBEN, and RWB are actually bits in DBGC. They are broken out in this table for reference.

Trigger mode: A Then B
FIFO contents: Information from the last eight changes of flow ending when the instruction at address B begins to execute.

This is an example of an end-trace debug run that ends when the instruction at B executes, but only after the instruction at A has executed at least once. The sequential nature of the trigger ensures that the trigger will occur only when you have followed a certain path through your program. In the previous begin trace example, we may have missed a change-of-flow address (counting the trigger event itself). This example suggests a way to use the first two change-of-flow events from that debug run to specify the A-then-B sequence that ends this debug run. Any change-of-flow event missed during the earlier debug run should be in the FIFO for this debug run.

Since change-of-flow addresses represent addresses where the CPU is going to try to start executing instructions, they should always be the address of an executable instruction. In the case of program runaway, if a change-of-flow address points at an illegal opcode, the CPU will still fetch it into its instruction pipe and try to execute it even though the illegal opcode detect logic will intervene to force an exception.

An end trace is used because begin-type traces cause the breakpoint to the CPU to be related to the FIFO full condition rather than the selected trigger conditions. Since this is an end-type trace and we want a tag-type breakpoint to the CPU, we must also specify a tag-type trigger (TRGSEL = 1). In an end trace, if the instruction at the trigger address is a change of flow, it will be captured as the last FIFO entry for that debug run.

7.5.8.7 Example 6: Begin Trace On Write of Data B to Address A

DBGCAH:DBGCAL	RWAEN[1]	RWA[1]	DBGCBH:DBGCBL	RWBEN[1]	RWB[1]	DBGT	DBGC
Trigger address A	1	0	xx:Trigger data B	0	x	$45	$C4

[1] RWAEN, RWA, RWBEN, and RWB are actually bits in DBGC. They are broken out in this table for reference.

Trigger mode: A AND B Data (Full Mode)
FIFO contents: Information from the next eight changes of flow starting three cycles after the trigger.

This example shows a begin trace debug run that starts when the address in comparator A and the data in the low half of comparator B both match in the same bus cycle. This is a force-type trigger so address A can be the address of a control register or a program variable. When the FIFO has captured the next eight change-of-flow addresses, the debug run ends, but since no CPU breakpoint is specified (BRKEN = 0), the MCU continues to execute the application program. Typically, in this type of debug run, the host development system would monitor the debug status register (DBGS) to determine when the debug run was finished. The host would then read the results of the debug run from the FIFO.

This demonstrates that debugging can be done without disturbing real-time operation of an application program. The background debug commands have a very small impact since the active background mode commands steal a bus cycle whenever they need to access target memory. This impact is never greater than one bus cycle per active background mode command and background memory access commands take at least 528 BDC clock cycles and usually have significant gaps between adjacent commands.

Variation: The A AND NOT B Data trigger mode can be used for a useful variation of this example. Suppose you are debugging a program and you suspect some control register is being overwritten with an unexpected value by some erroneous code. You can setup an end trace where the comparator A is set to the address of the suspicious register and comparator B is setup with the correct data you expect in the register. When the debug run ends, the FIFO will show the last eight changes of flow leading to the offending instruction.

7.5.8.8 Example 7: Capture the First Eight Values Read From Address B

DBGCAH:DBGCAL	RWAEN[1]	RWA[1]	DBGCBH:DBGCBL	RWBEN[1]	RWB[1]	DBGT	DBGC
Not used	x	x	Trigger address B	1	1	$43	$C3

[1] RWAEN, RWA, RWBEN, and RWB are actually bits in DBGC. They are broken out in this table for reference.

Trigger mode: Event-Only B (Store Data)
FIFO contents: The first eight data values read from address B are stored into the low half of the FIFO data words. The high-order eight bits of each FIFO word are unused and read as logic 0s.

This is an event-only trigger mode so the BEGIN control bit is ignored and all debug runs are treated as begin-type traces. This mode is used to capture the data involved in a read or write access to a specific address such as the address of a particular control register or program variable.

It would be inappropriate to set TRGSEL = 1 with this trigger mode because the trigger address is normally not the address of an executable instruction.

7.5.8.9 Example 8: Capture Values Written to Address B After Address A Read

DBGCAH:DBGCAL	RWAEN[1]	RWA[1]	DBGCBH:DBGCBL	RWBEN[1]	RWB[1]	DBGT	DBGC
Qualifier address A	1	1	Trigger address B	1	0	$44	$CD

[1] RWAEN, RWA, RWBEN, and RWB are actually bits in DBGC. They are broken out in this table for reference.

Trigger mode: A Then Event-Only B Data (Store Data)
FIFO contents: The first eight data values written to address B after address A was read. The high-order eight bits of each FIFO word are unused and read as logic 0s.

As in the previous example, this is an event-only trigger mode so the BEGIN control bit is ignored and all debug runs are treated as BEGIN-type traces. In this example, address A must be detected as a qualifying condition before the FIFO begins to capture data values for each write access to trigger address B.

Variation: If TRGSEL = 1, comparator A is qualified by opcode tracking logic so that the A trigger will not occur until the instruction at address A is about to execute. This debug example could be used to detect erroneous writes to a control register after the reset initialization routine was finished. To set up such a run, store the address of one of the last instructions of the reset initialization routine in comparator A and store the address of a selected control register in the low-order half of comparator B. After running the application program, the host debug system can read the DBGS status register to determine whether any values have been written to the selected control register address.

7.5.8.10 Example 9: Trigger On Any Execution Within a Routine

DBGCAH:DBGCAL	RWAEN[1]	RWA[1]	DBGCBH:DBGCBL	RWBEN[1]	RWB[1]	DBGT	DBGC
Opcode address A	0	x	Opcode address B	0	x	$87	$F0

[1] RWAEN, RWA, RWBEN, and RWB are actually bits in DBGC. They are broken out in this table for reference.

Trigger mode: Inside Range (A ≤ Address ≤ B)
FIFO contents: The last eight change of flow addresses before the CPU executed an instruction between address A and address B (inclusive).

This debug run is an end trace that stops if the CPU attempts to execute any instruction within the range specified by address A and address B. Comparator A would be set to the address of the first instruction in the routine to be monitored, and comparator B would be set to the address of the last instruction in the routine. TRGSEL = 1, so comparisons are qualified by opcode tracking logic. R/W is not used to qualify either comparator. When the debug run ends, the CPU will breakpoint to active background mode. An external debug host system can read out the contents of the FIFO to reconstruct instructions leading to the trigger condition.

7.5.8.11 Example 10: Trigger On Any Attempt To Execute Outside FLASH

DBGCAH:DBGCAL	RWAEN[1]	RWA[1]	DBGCBH:DBGCBL	RWBEN[1]	RWB[1]	DBGT	DBGC
Opcode address A	0	x	Opcode address B	0	x	$88	$F0

[1] RWAEN, RWA, RWBEN, and RWB are actually bits in DBGC. They are broken out in this table for reference.

Trigger mode: Outside Range (Address < A or Address > B)
FIFO contents: The last eight change of flow addresses before the CPU executed an instruction that was not between address A and address B.

This example can be used to detect when a program goes outside the expected range. For example, in a program runaway case, you could set comparator A to the address of the first instruction in the FLASH memory and comparator B to the address of the last instruction in the FLASH memory. The debug run will end when the CPU attempts to execute an instruction from any address outside the range of the user program in FLASH memory. After the debug run, the FIFO can be read to reconstruct the last eight changes of flow prior to the erroneous attempt to execute from an address outside the FLASH.

7.5.9 Hardware Breakpoints and ROM Patching

The BRKEN control bit in the DBGC register may be set to 1 to allow any of the trigger conditions described in **7.5.6 Trigger Modes** to be used to generate a hardware breakpoint request to the CPU. In the case of ROM patching, you would never use the FIFO and you should always specify an end trace so the CPU break request coincides with the selected trigger conditions rather than the FIFO full condition. The TAG bit in DBGC controls whether the breakpoint request will be treated as a tag-type breakpoint or a force-type breakpoint. A tag breakpoint causes the current opcode to be marked as it enters the instruction queue. If a tagged opcode reaches the end of the pipe, the CPU executes a BGND instruction to go to active background mode rather than executing the tagged opcode. A force-type breakpoint causes the CPU to finish the current instruction and then go to active background mode.

If the background mode has not been enabled (ENBDM = 1) by a serial WRITE_CONTROL command through the BKGD pin, the CPU will execute an SWI instruction instead of going to active background mode. If the user has taken appropriate steps to prepare for this case, it can be used to implement a form of ROM patching.

ROM patching is a technique that allows program bugs in ROM or other non-volatile memory to be replaced by different program instructions to repair the bug. The mechanism is based on the MCU detecting it is about to execute an instruction at the location of a bug. Instead of executing that instruction, hardware breakpoint logic generates a breakpoint request to the CPU. The CPU knows it is not connected to a development system because the ENBDM control bit in BDCSCR equals 0. So instead of going to active background mode, the CPU executes an SWI instruction. The SWI service routine fetches the address of the repair code from some non-volatile memory location and executes that instead of the bug code. At the end of the repair code, the stack pointer can be adjusted and an ordinary jump instruction can return execution to a location past the original bug.

Alternatively, the repair code could alter the return address of the stack and execute an RTI to return to a point after the original bug.

Appendix A. Instruction Set Details

A.1 Introduction

This section contains detailed information for all HCS08 Family instructions. The instructions are arranged in alphabetical order with the instruction mnemonic set in larger type for easy reference.

A.2 Nomenclature

This nomenclature is used in the instruction descriptions throughout this section.

Operators

( )	=	Contents of register or memory location shown inside parentheses
←	=	Is loaded with (read: “gets”)
&	=	Boolean AND
|	=	Boolean OR
⊕	=	Boolean exclusive-OR
×	=	Multiply
÷	=	Divide
:	=	Concatenate
+	=	Add
–	=	Negate (two’s complement)
«	=	Sign extend

CPU registers

A	=	Accumulator
CCR	=	Condition code register
H	=	Index register, higher order (most significant) eight bits
X	=	Index register, lower order (least significant) eight bits
PC	=	Program counter

PCH = Program counter, higher order (most significant) eight bits

PCL = Program counter, lower order (least significant) eight bits

SP = Stack pointer

Memory and addressing

M = A memory location or absolute data, depending on addressing mode

M:M + $0001= A 16-bit value in two consecutive memory locations. The higher-order (most significant) eight bits are located at the address of M, and the lower-order (least significant) eight bits are located at the next higher sequential address.

rel = The relative offset, which is the two’s complement number stored in the last byte of machine code corresponding to a branch instruction

Condition code register (CCR) bits

V = Two’s complement overflow indicator, bit 7

H = Half carry, bit 4

I = Interrupt mask, bit 3

N = Negative indicator, bit 2

Z = Zero indicator, bit 1

C = Carry/borrow, bit 0 (carry out of bit 7)

Bit status BEFORE execution of an instruction (*n* = 7, 6, 5, ... 0)

For 2-byte operations such as LDHX, STHX, and CPHX, *n* = 15 refers to bit 15 of the 2-byte word or bit 7 of the most significant (first) byte.

M*n* = Bit *n* of memory location used in operation

A*n* = Bit *n* of accumulator

H*n* = Bit *n* of index register H

X*n* = Bit *n* of index register X

b*n* = Bit *n* of the source operand (M, A, or X)

Bit status AFTER execution of an instruction

For 2-byte operations such as LDHX, STHX, and CPHX, *n* = 15 refers to bit 15 of the 2-byte word or bit 7 of the most significant (first) byte.

R*n* = Bit *n* of the result of an operation (*n* = 7, 6, 5, … 0)

CCR activity figure notation

–	=	Bit not affected
0	=	Bit forced to 0
1	=	Bit forced to 1
↕	=	Bit set or cleared according to results of operation
U	=	Undefined after the operation

Machine coding notation

dd	=	Low-order eight bits of a direct address $0000–$00FF (high byte assumed to be $00)
ee	=	Upper eight bits of 16-bit offset
ff	=	Lower eight bits of 16-bit offset or 8-bit offset
ii	=	One byte of immediate data
jj	=	High-order byte of a 16-bit immediate data value
kk	=	Low-order byte of a 16-bit immediate data value
hh	=	High-order byte of 16-bit extended address
ll	=	Low-order byte of 16-bit extended address
rr	=	Relative offset

Source forms

The instruction detail pages provide only essential information about assembler source forms. Assemblers generally support a number of assembler directives, allow definition of program labels, and have special conventions for comments. For complete information about writing source files for a particular assembler, refer to the documentation provided by the assembler vendor.

Typically, assemblers are flexible about the use of spaces and tabs. Often, any number of spaces or tabs can be used where a single space is shown on the glossary pages. Spaces and tabs are also normally allowed before and after commas. When program labels are used, there must also be at least one tab or space before all instruction mnemonics. This required space is not apparent in the source forms.

Everything in the source forms columns, *except expressions in italic characters*, is literal information which must appear in the assembly source file exactly as shown. The initial 3- to 5-letter mnemonic is always a literal expression. All commas, pound signs (#), parentheses, and plus signs (+) are literal characters.

The definition of a legal label or expression varies from assembler to assembler. Assemblers also vary in the way CPU registers are specified. Refer to assembler documentation for detailed information. Recommended register designators are a, A, h, H, x, X, sp, and SP.

n — Any label or expression that evaluates to a single integer in the range 0–7

opr8i — Any label or expression that evaluates to an 8-bit immediate value

opr16i — Any label or expression that evaluates to a 16-bit immediate value

opr8a — Any label or expression that evaluates to an 8-bit value. The instruction treats this 8-bit value as the low order eight bits of an address in the direct page of the 64-Kbyte address space ($00xx).

opr16a — Any label or expression that evaluates to a 16-bit value. The instruction treats this value as an address in the 64-Kbyte address space.

oprx8 — Any label or expression that evaluates to an unsigned 8-bit value; used for indexed addressing

oprx16 — Any label or expression that evaluates to a 16-bit value. Since the HCS08 has a 16-bit address bus, this can be either a signed or an unsigned value.

rel — Any label or expression that refers to an address that is within –128 to +127 locations from the next address after the last byte of object code for the current instruction. The assembler will calculate the 8-bit signed offset and include it in the object code for this instruction.

Cycle-by-cycle execution

This information is found in the tables at the bottom of each instruction glossary page. Entries show how many bytes of information are accessed from different areas of memory during the course of instruction execution. With this information and knowledge of the bus frequency, a user can determine the execution time for any instruction in any system.

A single letter code in the column represents a single CPU cycle. There are cycle codes for each addressing mode variation of each instruction. Simply count code letters to determine the execution time of an instruction.

This list explains the cycle-by-cycle code letters:

f — Free cycle. This indicates a cycle where the CPU does not require use of the system buses. An f cycle is always one cycle of the system bus clock.
p — Program byte access
r — 8-bit data read
s — Stack 8-bit data (push)
w — 8-bit data write
u — Unstack 8-bit data (pull)
v — Vector fetch. Vectors are always fetched as two consecutive 8-bit accesses (v v) with the high byte first.

Address modes

INH = Inherent (no operands)
IMM = 8-bit or 16-bit immediate
DIR = 8-bit direct
EXT = 16-bit extended
IX = 16-bit indexed no offset
IX+ = 16-bit indexed no offset, post increment (CBEQ and MOV only)
IX1 = 16-bit indexed with 8-bit offset from H:X
IX1+ = 16-bit indexed with 8-bit offset, post increment (CBEQ only)
IX2 = 16-bit indexed with 16-bit offset from H:X
REL = 8-bit relative offset
SP1 = Stack pointer relative with 8-bit offset
SP2 = Stack pointer relative with 16-bit offset

A.3 Convention Definitions

Set refers specifically to establishing logic level 1 on a bit or bits.

Cleared refers specifically to establishing logic level 0 on a bit or bits.

A specific bit is referred to by mnemonic and bit number. A7 is bit 7 of accumulator A. **A range of bits** is referred to by mnemonic and the bit numbers that define the range. A [7:4] are bits 7 to 4 of the accumulator.

Parentheses indicate the contents of a register or memory location, rather than the register or memory location itself. (A) is the contents of the accumulator. In Boolean expressions, parentheses have the traditional mathematical meaning.

A.4 Instruction Set

The following pages summarize each instruction, including operation and description, condition codes and Boolean formulae, and a table with source forms, addressing modes, machine code, and cycles.

ADC Add with Carry ADC

Operation

A ← (A) + (M) + (C)

Description

Adds the contents of the C bit to the sum of the contents of A and M and places the result in A. This operation is useful for addition of operands that are larger than eight bits.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↕	1	1	↕	—	↕	↕	↕

V: $A7\&M7\&\overline{R7}\ |\ \overline{A7}\&\overline{M7}\&R7$
Set if a two’s complement overflow resulted from the operation; cleared otherwise

H: $A3\&M3\ |\ M3\&\overline{R3}\ |\ \overline{R3}\&A3$
Set if there was a carry from bit 3; cleared otherwise

N: R7
Set if MSB of result is 1; cleared otherwise

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is \$00; cleared otherwise

C: $A7\&M7\ |\ M7\&\overline{R7}\ |\ \overline{R7}\&A7$
Set if there was a carry from the most significant bit (MSB) of the result; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
ADC *#opr8i*	IMM	A9	ii	2	pp
ADC *opr8a*	DIR	B9	dd	3	rpp
ADC *opr16a*	EXT	C9	hh ll	4	prpp
ADC *oprx16*,X	IX2	D9	ee ff	4	prpp
ADC *oprx8*,X	IX1	E9	ff	3	rpp
ADC ,X	IX	F9		3	rfp
ADC *oprx16*,SP	SP2	9ED9	ee ff	5	pprpp
ADC *oprx8*,SP	SP1	9EE9	ff	4	prpp

ADD Add without Carry ADD

Operation $A \leftarrow (A) + (M)$

Description Adds the contents of M to the contents of A and places the result in A

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↕	1	1	↕	—	↕	↕	↕

V: $A7\&M7\&\overline{R7} \mid \overline{A7}\&\overline{M7}\&R7$
Set if a two's complement overflow resulted from the operation; cleared otherwise

H: $A3\&M3 \mid M3\&\overline{R3} \mid \overline{R3}\&A3$
Set if there was a carry from bit 3; cleared otherwise

N: R7
Set if MSB of result is 1; cleared otherwise

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is \$00; cleared otherwise

C: $A7\&M7 \mid M7\&\overline{R7} \mid \overline{R7}\&A7$
Set if there was a carry from the MSB of the result; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
ADD *#opr8i*	IMM	AB	ii	2	pp
ADD *opr8a*	DIR	BB	dd	3	rpp
ADD *opr16a*	EXT	CB	hh ll	4	prpp
ADD *oprx16*,X	IX2	DB	ee ff	4	prpp
ADD *oprx8*,X	IX1	EB	ff	3	rpp
ADD ,X	IX	FB		3	rfp
ADD *oprx16*,SP	SP2	9EDB	ee ff	5	pprpp
ADD *oprx8*,SP	SP1	9EEB	ff	4	prpp

AIS Add Immediate Value (Signed) to Stack Pointer AIS

Operation

SP ← (SP) + (16 « M)

Description

Adds the immediate operand to the stack pointer (SP). The immediate value is an 8-bit two's complement signed operand. The 8-bit operand is sign-extended to 16 bits prior to the addition. The AIS instruction can be used to create and remove a stack frame buffer that is used to store temporary variables.

This instruction does not affect any condition code bits so status information can be passed to or from a subroutine or C function and allocation or deallocation of space for local variables will not disturb that status information.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycle, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
AIS *#opr8i*	IMM	A7	ii	2	pp

AIX Add Immediate Value (Signed) to Index Register AIX

Operation

H:X ← (H:X) + (16 « M)

Description

Adds an immediate operand to the 16-bit index register, formed by the concatenation of the H and X registers. The immediate operand is an 8-bit two's complement signed offset. The 8-bit operand is sign-extended to 16 bits prior to the addition.

This instruction does not affect any condition code bits so index register pointer calculations do not disturb the surrounding code which may rely on the state of CCR status bits.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
AIX *#opr8i*	IMM	AF	ii	2	pp

AND Logical AND AND

Operation A ← (A) & (M)

Description Performs the logical AND between the contents of A and the contents of M and places the result in A. Each bit of A after the operation will be the logical AND of the corresponding bits of M and of A before the operation.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
0	1	1	—	—	↕	↕	—

V: 0
Cleared

N: R7
Set if MSB of result is 1; cleared otherwise

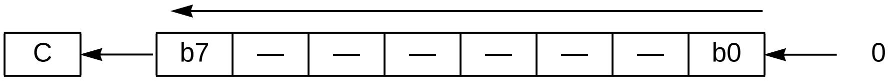
Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is $00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
AND *#opr8i*	IMM	A4	ii	2	pp
AND *opr8a*	DIR	B4	dd	3	rpp
AND *opr16a*	EXT	C4	hh ll	4	prpp
AND *oprx16*,X	IX2	D4	ee ff	4	prpp
AND *oprx8*,X	IX1	E4	ff	3	rpp
AND ,X	IX	F4		3	rfp
AND *oprx16*,SP	SP2	9ED4	ee ff	5	pprpp
AND *oprx8*,SP	SP1	9EE4	ff	4	prpp

ASL Arithmetic Shift Left (Same as LSL) ASL

Operation



Description

Shifts all bits of A, X, or M one place to the left. Bit 0 is loaded with a 0. The C bit in the CCR is loaded from the most significant bit of A, X, or M. This is mathematically equivalent to multiplication by two. The V bit indicates whether the sign of the result has changed.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↕	1	1	—	—	↕	↕	↕

V: $R7 \oplus b7$
Set if the exclusive-OR of the resulting N and C flags is 1; cleared otherwise

N: R7
Set if MSB of result is 1; cleared otherwise

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is \$00; cleared otherwise

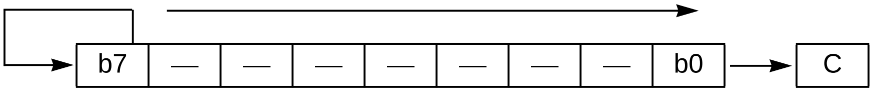
C: b7
Set if, before the shift, the MSB of A, X, or M was set; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
ASL *opr8a*	DIR	38	dd	5	rfwpp
ASLA	INH (A)	48		1	p
ASLX	INH (X)	58		1	p
ASL *oprx8*,X	IX1	68	ff	5	rfwpp
ASL ,X	IX	78		4	rfwp
ASL *oprx8*,SP	SP1	9E68	ff	6	prfwpp

ASR Arithmetic Shift Right ASR

Operation



Description

Shifts all bits of A, X, or M one place to the right. Bit 7 is held constant. Bit 0 is loaded into the C bit of the CCR. This operation effectively divides a two’s complement value by 2 without changing its sign. The carry bit can be used to round the result.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↕	1	1	—	—	↕	↕	↕

V: $R7 \oplus b0$
Set if the exclusive-OR of the resulting N and C flags is 1; cleared otherwise

N: R7
Set if MSB of result is 1; cleared otherwise

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is \$00; cleared otherwise

C: b0
Set if, before the shift, the LSB of A, X, or M was set; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
ASR *opr8a*	DIR	37	dd	5	rfwpp
ASRA	INH (A)	47		1	p
ASRX	INH (X)	57		1	p
ASR *oprx8*,X	IX1	67	ff	5	rfwpp
ASR ,X	IX	77		4	rfwp
ASR *oprx8*,SP	SP1	9E67	ff	6	prfwpp

BCC Branch if Carry Bit Clear (Same as BHS) BCC

Operation

If (C) = 0, PC ← (PC) + $0002 + *rel*

Simple branch

Description

Tests state of C bit in CCR and causes a branch if C is clear. BCC can be used after shift or rotate instructions or to check for overflow after operations on unsigned numbers. See the **BRA** instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BCC *rel*	REL	24	rr	3	ppp

See the **BRA** instruction for a summary of all branches and their complements.

BCLR *n* — Clear Bit *n* in Memory — BCLR *n*

Operation

M*n* ← 0

Description

Clear bit *n* (*n* = 7, 6, 5, … 0) in location M. All other bits in M are unaffected. In other words, M can be any random-access memory (RAM) or input/output (I/O) register address in the $0000 to $00FF area of memory. (Direct addressing mode is used to specify the address of the operand.) This instruction reads the specified 8-bit location, modifies the specified bit, and then writes the modified 8-bit value back to the memory location.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BCLR 0,*opr8a*	DIR (b0)	11	dd	5	rfwpp
BCLR 1,*opr8a*	DIR (b1)	13	dd	5	rfwpp
BCLR 2,*opr8a*	DIR (b2)	15	dd	5	rfwpp
BCLR 3,*opr8a*	DIR (b3)	17	dd	5	rfwpp
BCLR 4,*opr8a*	DIR (b4)	19	dd	5	rfwpp
BCLR 5,*opr8a*	DIR (b5)	1B	dd	5	rfwpp
BCLR 6,*opr8a*	DIR (b6)	1D	dd	5	rfwpp
BCLR 7,*opr8a*	DIR (b7)	1F	dd	5	rfwpp

BCS Branch if Carry Bit Set (Same as BLO) BCS

Operation

If (C) = 1, PC ← (PC) + $0002 + *rel*

Simple branch

Description

Tests the state of the C bit in the CCR and causes a branch if C is set. BCS can be used after shift or rotate instructions or to check for overflow after operations on unsigned numbers. See the **BRA** instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BCS *rel*	REL	25	rr	3	ppp

See the **BRA** instruction for a summary of all branches and their complements.

BEQ Branch if Equal BEQ

Operation

If (Z) = 1, PC ← (PC) + $0002 + *rel*

Simple branch; may be used with signed or unsigned operations

Description

Tests the state of the Z bit in the CCR and causes a branch if Z is set. Compare instructions perform a subtraction with two operands and produce an internal result without changing the original operands. If the two operands were equal, the internal result of the subtraction for the compare will be zero so the Z bit will be equal to one and the BEQ will cause a branch.

This instruction can also be used after a load or store without having to do a separate test or compare on the loaded value. See the **BRA** instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BEQ *rel*	REL	27	rr	3	ppp

See the **BRA** instruction for a summary of all branches and their complements.

BGE Branch if Greater Than or Equal To BGE

Operation

If (N ⊕ V) = 0, PC ← (PC) + $0002 + *rel*

For signed two's complement values
if (Accumulator) ≥ (Memory), then branch

Description

If the BGE instruction is executed immediately after execution of a CMP, CPHX, CPX, SBC, or SUB instruction, the branch occurs if and only if the two's complement number in the A, X, or H:X register was greater than or equal to the two's complement number in memory.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BGE *rel*	REL	90	rr	3	ppp

See the **BRA** instruction for a summary of all branches and their complements.

BGND Background BGND

Operation Enter active background debug mode (if allowed by ENBDM = 1)

Description This instruction is used to establish software breakpoints during debug by replacing a user opcode with this opcode. BGND causes the user program to stop and the CPU enters active background mode (provided it has been enabled previously by a serial WRITE_CONTROL command from a host debug pod). The CPU remains in active background mode until the debug host sends a serial GO, TRACE1, or TAGGO command to return to the user program. This instruction is never used in normal user application programs. If the ENBDM control bit in the BDC status/control register is clear, BGND is treated as an illegal opcode.

Condition Codes and Boolean Formulae None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BGND	INH	82		5+	fp...ppp

BGT Branch if Greater Than BGT

Operation

If (Z) | (N ⊕ V) = 0, PC ← (PC) + $0002 + *rel*

For signed two's complement values
if (Accumulator) > (Memory), then branch

Description

If the BGT instruction is executed immediately after execution of a CMP, CPHX, CPX, SBC, or SUB instruction, the branch will occur if and only if the two's complement number in the A, X, or H:X register was greater than the two's complement number in memory.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BGT *rel*	REL	92	rr	3	ppp

See the **BRA** instruction for a summary of all branches and their complements.

BHCC Branch if Half Carry Bit Clear BHCC

Operation If (H) = 0, PC ← (PC) + $0002 + *rel*

Description Tests the state of the H bit in the CCR and causes a branch if H is clear. This instruction is used in algorithms involving BCD numbers that were originally written for the M68HC05 or M68HC08 devices. The DAA instruction in the HCS08 simplifies operations on BCD numbers so BHCC and BHCS should not be needed in new programs. See the **BRA** instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BHCC *rel*	REL	28	rr	3	ppp

See the **BRA** instruction for a summary of all branches and their complements.

BHCS — Branch if Half Carry Bit Set — BHCS

Operation

If (H) = 1, PC ← (PC) + $0002 + *rel*

Description

Tests the state of the H bit in the CCR and causes a branch if H is set. This instruction is used in algorithms involving BCD numbers that were originally written for the M68HC05 or M68HC08 devices. The DAA instruction in the HCS08 simplifies operations on BCD numbers so BHCC and BHCS should not be needed in new programs. See the **BRA** instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BHCS *rel*	REL	29	rr	3	ppp

See the **BRA** instruction for a summary of all branches and their complements.

BHI Branch if Higher BHI

Operation

If (C) | (Z) = 0, PC ← (PC) + $0002 + *rel*

For unsigned values, if (Accumulator) > (Memory), then branch

Description

Causes a branch if both C and Z are cleared. If the BHI instruction is executed immediately after execution of a CMP, CPHX, CPX, SBC, or SUB instruction, the branch will occur if the unsigned binary number in the A, X, or H:X register was greater than unsigned binary number in memory. Generally not useful after CLR, COM, DEC, INC, LDA, LDHX, LDX, STA, STHX, STX, or TST because these instructions do not affect the carry bit in the CCR. See the **BRA** instruction for details of the execution of the branch.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BHI *rel*	REL	22	rr	3	ppp

See the **BRA** instruction for a summary of all branches and their complements.

BHS Branch if Higher or Same (Same as BCC) BHS

Operation

If (C) = 0, PC ← (PC) + $0002 + *rel*

For unsigned values, if (Accumulator) ≥ (Memory), then branch

Description

If the BHS instruction is executed immediately after execution of a CMP, CPHX, CPX, SBC, or SUB instruction, the branch will occur if the unsigned binary number in the A, X, or H:X register was greater than or equal to the unsigned binary number in memory. Generally not useful after CLR, COM, DEC, INC, LDA, LDHX, LDX, STA, STHX, STX, or TST because these instructions do not affect the carry bit in the CCR. See the **BRA** instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BHS *rel*	REL	24	rr	3	ppp

See the **BRA** instruction for a summary of all branches and their complements.

BIH Branch if IRQ Pin High BIH

Operation

If IRQ pin = 1, PC ← (PC) + $0002 + *rel*

Description

Tests the state of the external interrupt pin and causes a branch if the pin is high. See the **BRA** instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BIH *rel*	REL	2F	rr	3	ppp

See the **BRA** instruction for a summary of all branches and their complements.

BIL Branch if IRQ Pin Low BIL

Operation

If IRQ pin = 0, PC ← (PC) + $0002 + *rel*

Description

Tests the state of the external interrupt pin and causes a branch if the pin is low. See the **BRA** instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BIL *rel*	REL	2E	rr	3	ppp

See the **BRA** instruction for a summary of all branches and their complements.

BIT Bit Test BIT

Operation (A) & (M)

Description

Performs the logical AND comparison of the contents of A and the contents of M and modifies the condition codes accordingly. Neither the contents of A nor M are altered. (Each bit of the result of the AND would be the logical AND of the corresponding bits of A and M.)

This instruction is typically used to see if a particular bit, or any of several bits, in a byte are 1s. A mask value is prepared with 1s in any bit positions that are to be checked. This mask may be in accumulator A or memory and the unknown value to be checked will be in memory or the accumulator A, respectively. After the BIT instruction, a BNE instruction will branch if any bits in the tested location that correspond to 1s in the mask were 1s.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
0	1	1	—	—	↕	↕	—

V: 0
Cleared

N: R7
Set if MSB of result is 1; cleared otherwise

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is \$00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BIT *#opr8i*	IMM	A5	ii	2	pp
BIT *opr8a*	DIR	B5	dd	3	rpp
BIT *opr16a*	EXT	C5	hh ll	4	prpp
BIT *oprx16*,X	IX2	D5	ee ff	4	prpp
BIT *oprx8*,X	IX1	E5	ff	3	rpp
BIT ,X	IX	F5		3	rfp
BIT *oprx16*,SP	SP2	9ED5	ee ff	5	pprpp
BIT *oprx8*,SP	SP1	9EE5	ff	4	prpp

BLE Branch if Less Than or Equal To BLE

Operation

If $(Z) \mid (N \oplus V) = 1$, $PC \leftarrow (PC) + \$0002 + rel$

For signed two’s complement numbers
if (Accumulator) ≤ (Memory), then branch

Description

If the BLE instruction is executed immediately after execution of a CMP, CPHX, CPX, SBC, or SUB instruction, the branch will occur if and only if the two’s complement in the A, X, or H:X register was less than or equal to the two’s complement number in memory.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BLE *rel*	REL	93	rr	3	ppp

See the **BRA** instruction for a summary of all branches and their complements.

BLO Branch if Lower BLO

Operation

If (C) = 1, PC ← (PC) + $0002 + *rel*

For unsigned values, if (Accumulator) < (Memory), then branch

Description

If the BLO instruction is executed immediately after execution of a CMP, CPHX, CPX, SBC, or SUB instruction, the branch will occur if the unsigned binary number in the A, X, or H:X register was less than the unsigned binary number in memory. Generally not useful after CLR, COM, DEC, INC, LDA, LDHX, LDX, STA, STHX, STX, or TST because these instructions do not affect the carry bit in the CCR. See the **BRA** instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BLO *rel*	REL	25	rr	3	ppp

See the **BRA** instruction for a summary of all branches and their complements.

BLS — Branch if Lower or Same — BLS

Operation

If (C) | (Z) = 1, PC ← (PC) + \$0002 + *rel*

For unsigned values, if (Accumulator) ≤ (Memory), then branch

Description

Causes a branch if (C is set) or (Z is set). If the BLS instruction is executed immediately after execution of a CMP, CPHX, CPX, SBC, or SUB instruction, the branch will occur if and only if the unsigned binary number in the A, X, or H:X register was less than or equal to the unsigned binary number in memory. Generally not useful after CLR, COM, DEC, INC, LDA, LDHX, LDX, STA, STHX, STX, or TST because these instructions do not affect the carry bit in the CCR. See the **BRA** instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycle, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BLS *rel*	REL	23	rr	3	ppp

See the **BRA** instruction for a summary of all branches and their complements.

BLT Branch if Less Than (Signed Operands) BLT

Operation

If (N ⊕ V) = 1, PC ← (PC) + \$0002 + *rel*

For signed two’s complement numbers
if (Accumulator) < (Memory), then branch

Description

If the BLT instruction is executed immediately after execution of a CMP, CPHX, CPX, SBC, or SUB instruction, the branch will occur if and only if the two’s complement number in the A, X, or H:X register was less than the two’s complement number in memory. See the **BRA** instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BLT *rel*	REL	91	rr	3	ppp

See the **BRA** instruction for a summary of all branches and their complements.

BMC Branch if Interrupt Mask Clear BMC

Operation

If (I) = 0, PC ← (PC) + $0002 + *rel*

Description

Tests the state of the I bit in the CCR and causes a branch if I is clear (if interrupts are enabled). See the **BRA** instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BMC *rel*	REL	2C	rr	3	ppp

See the **BRA** instruction for a summary of all branches and their complements.

BMI Branch if Minus BMI

Operation

If (N) = 1, PC ← (PC) + $0002 + *rel*

Simple branch; may be used with signed or unsigned operations

Description

Tests the state of the N bit in the CCR and causes a branch if N is set.

Simply loading or storing A, X, or H:X will cause the N condition code bit to be set or cleared to match the most significant bit of the value loaded or stored. The BMI instruction can be used after such a load or store without having to do a separate test or compare instruction before the conditional branch. See the **BRA** instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BMI *rel*	REL	2B	rr	3	ppp

See the **BRA** instruction for a summary of all branches and their complements.

BMS Branch if Interrupt Mask Set BMS

Operation If (I) = 1, PC ← (PC) + $0002 + *rel*

Description Tests the state of the I bit in the CCR and causes a branch if I is set (if interrupts are disabled). See **BRA** instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BMS *rel*	REL	2D	rr	3	ppp

See the **BRA** instruction for a summary of all branches and their complements.

BNE Branch if Not Equal BNE

Operation

If (Z) = 0, PC ← (PC) + $0002 + *rel*

Simple branch, may be used with signed or unsigned operations

Description

Tests the state of the Z bit in the CCR and causes a branch if Z is clear

Following a compare or subtract instruction, the branch will occur if the arguments were not equal. This instruction can also be used after a load or store without having to do a separate test or compare on the loaded value. See the **BRA** instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BNE *rel*	REL	26	rr	3	ppp

See the **BRA** instruction for a summary of all branches and their complements.

BPL Branch if Plus BPL

Operation

If (N) = 0, PC ← (PC) + \$0002 + *rel*

Simple branch

Description

Tests the state of the N bit in the CCR and causes a branch if N is clear

Simply loading or storing A, X, or H:X will cause the N condition code bit to be set or cleared to match the most significant bit of the value loaded or stored. The BPL instruction can be used after such a load or store without having to do a separate test or compare instruction before the conditional branch. See the **BRA** instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BPL *rel*	REL	2A	rr	3	ppp

See the **BRA** instruction for a summary of all branches and their complements.

BRA Branch Always BRA

Operation

PC ← (PC) + $0002 + *rel*

Description

Performs an unconditional branch to the address given in the foregoing formula. In this formula, *rel* is the two’s-complement relative offset in the last byte of machine code for the instruction and (PC) is the address of the opcode for the branch instruction.

A source program specifies the destination of a branch instruction by its absolute address, either as a numerical value or as a symbol or expression which can be numerically evaluated by the assembler. The assembler calculates the 8-bit relative offset *rel* from this absolute address and the current value of the location counter.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BRA *rel*	REL	20	rr	3	ppp

The table on the facing page is a summary of all branch instructions.

The BRA description continues next page.

BRA Branch Always (Continued) BRA

Branch Instruction Summary

Table A-1 is a summary of all branch instructions.

Table A-1. Branch Instruction Summary

Branch				Complementary Branch			Type
Test	Boolean	Mnemonic	Opcode	Test	Mnemonic	Opcode	
r>m	(Z) | (N⊕V)=0	BGT	92	r≤m	BLE	93	Signed
r≥m	(N⊕V)=0	BGE	90	r<m	BLT	91	Signed
r=m	(Z)=1	BEQ	27	r≠m	BNE	26	Signed
r≤m	(Z) | (N⊕V)=1	BLE	93	r>m	BGT	92	Signed
r<m	(N⊕V)=1	BLT	91	r≥m	BGE	90	Signed
r>m	(C) | (Z)=0	BHI	22	r≤m	BLS	23	Unsigned
r≥m	(C)=0	BHS/BCC	24	r<m	BLO/BCS	25	Unsigned
r=m	(Z)=1	BEQ	27	r≠m	BNE	26	Unsigned
r≤m	(C) | (Z)=1	BLS	23	r>m	BHI	22	Unsigned
r<m	(C)=1	BLO/BCS	25	r≥m	BHS/BCC	24	Unsigned
Carry	(C)=1	BCS	25	No carry	BCC	24	Simple
result=0	(Z)=1	BEQ	27	result≠0	BNE	26	Simple
Negative	(N)=1	BMI	2B	Plus	BPL	2A	Simple
I mask	(I)=1	BMS	2D	I mask=0	BMC	2C	Simple
H-Bit	(H)=1	BHCS	29	H=0	BHCC	28	Simple
IRQ high	—	BIH	2F	—	BIL	2E	Simple
Always	—	BRA	20	Never	BRN	21	Uncond.

r = register: A, X, or H:X (for CPHX instruction)   m = memory operand

During program execution, if the tested condition is true, the two's complement offset is sign-extended to a 16-bit value which is added to the current program counter. This causes program execution to continue at the address specified as the branch destination. If the tested condition is not true, the program simply continues to the next instruction after the branch.

BRCLR *n* Branch if Bit *n* in Memory Clear BRCLR *n*

Operation

If bit *n* of M = 0, PC ← (PC) + $0003 + *rel*

Description

Tests bit *n* (*n* = 7, 6, 5, … 0) of location M and branches if the bit is clear. M can be any RAM or I/O register address in the $0000 to $00FF area of memory because direct addressing mode is used to specify the address of the operand.

The C bit is set to the state of the tested bit. When used with an appropriate rotate instruction, BRCLR *n* provides an easy method for performing serial-to-parallel conversions.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
—	1	1	—	—	—	—	↕

C: Set if M*n* = 1; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BRCLR 0,*opr8a*,*rel*	DIR (b0)	01	dd rr	5	rpppp
BRCLR 1,*opr8a*,*rel*	DIR (b1)	03	dd rr	5	rpppp
BRCLR 2,*opr8a*,*rel*	DIR (b2)	05	dd rr	5	rpppp
BRCLR 3,*opr8a*,*rel*	DIR (b3)	07	dd rr	5	rpppp
BRCLR 4,*opr8a*,*rel*	DIR (b4)	09	dd rr	5	rpppp
BRCLR 5,*opr8a*,*rel*	DIR (b5)	0B	dd rr	5	rpppp
BRCLR 6,*opr8a*,*rel*	DIR (b6)	0D	dd rr	5	rpppp
BRCLR 7,*opr8a*,*rel*	DIR (b7)	0F	dd rr	5	rpppp

BRN Branch Never BRN

Operation PC ← (PC) + $0002

Description Never branches. In effect, this instruction can be considered a 2-byte no operation (NOP) requiring three cycles for execution. Its inclusion in the instruction set provides a complement for the **BRA** instruction. The BRN instruction is useful during program debugging to negate the effect of another branch instruction without disturbing the offset byte.

This instruction can be useful in instruction-based timing delays. Instruction-based timing delays are usually discouraged because such code is not portable to systems with different clock speeds.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BRN *rel*	REL	21	rr	3	ppp

See the **BRA** instruction for a summary of all branches and their complements.

BRSET *n* Branch if Bit *n* in Memory Set BRSET *n*

Operation If bit *n* of M = 1, PC ← (PC) + $0003 + *rel*

Description Tests bit *n* (*n* = 7, 6, 5, … 0) of location M and branches if the bit is set. M can be any RAM or I/O register address in the $0000 to $00FF area of memory because direct addressing mode is used to specify the address of the operand.

The C bit is set to the state of the tested bit. When used with an appropriate rotate instruction, BRSET *n* provides an easy method for performing serial-to-parallel conversions.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
—	1	1	—	—	—	—	↕

C: Set if M*n* = 1; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BRSET 0,*opr8a*,*rel*	DIR (b0)	00	dd rr	5	rpppp
BRSET 1,*opr8a*,*rel*	DIR (b1)	02	dd rr	5	rpppp
BRSET 2,*opr8a*,*rel*	DIR (b2)	04	dd rr	5	rpppp
BRSET 3,*opr8a*,*rel*	DIR (b3)	06	dd rr	5	rpppp
BRSET 4,*opr8a*,*rel*	DIR (b4)	08	dd rr	5	rpppp
BRSET 5,*opr8a*,*rel*	DIR (b5)	0A	dd rr	5	rpppp
BRSET 6,*opr8a*,*rel*	DIR (b6)	0C	dd rr	5	rpppp
BRSET 7,*opr8a*,*rel*	DIR (b7)	0E	dd rr	5	rpppp

BSET *n* — Set Bit *n* in Memory — BSET *n*

Operation

M*n* ← 1

Description

Set bit *n* (*n* = 7, 6, 5, … 0) in location M. All other bits in M are unaffected. M can be any RAM or I/O register address in the $0000 to $00FF area of memory because direct addressing mode is used to specify the address of the operand. This instruction reads the specified 8-bit location, modifies the specified bit, and then writes the modified 8-bit value back to the memory location.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BSET 0,*opr8a*	DIR (b0)	10	dd	5	rfwpp
BSET 1,*opr8a*	DIR (b1)	12	dd	5	rfwpp
BSET 2,*opr8a*	DIR (b2)	14	dd	5	rfwpp
BSET 3,*opr8a*	DIR (b3)	16	dd	5	rfwpp
BSET 4,*opr8a*	DIR (b4)	18	dd	5	rfwpp
BSET 5,*opr8a*	DIR (b5)	1A	dd	5	rfwpp
BSET 6,*opr8a*	DIR (b6)	1C	dd	5	rfwpp
BSET 7,*opr8a*	DIR (b7)	1E	dd	5	rfwpp

BSR Branch to Subroutine BSR

Operation

PC ← (PC) + \$0002	Advance PC to return address
Push (PCL); SP ← (SP) – \$0001	Push low half of return address
Push (PCH); SP ← (SP) – \$0001	Push high half of return address
PC ← (PC) + *rel*	Load PC with start address of requested subroutine

Description

The program counter is incremented by 2 from the opcode address (so it points to the opcode of the next instruction which will be the return address). The least significant byte of the contents of the program counter (low-order return address) is pushed onto the stack. The stack pointer is then decremented by 1. The most significant byte of the contents of the program counter (high-order return address) is pushed onto the stack. The stack pointer is then decremented by 1. A branch then occurs to the location specified by the branch offset. See the **BRA** instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
BSR *rel*	REL	AD	rr	5	ssppp

CBEQ Compare and Branch if Equal CBEQ

Operation

For DIR or IMM modes: if (A) = (M), PC ← (PC) + $0003 + *rel*
Or for IX+ mode: if (A) = (M); PC ← (PC) + $0002 + *rel*
Or for SP1 mode: if (A) = (M); PC ← (PC) + $0004 + *rel*
Or for CBEQX: if (X) = (M); PC ← (PC) + $0003 + *rel*

Description

CBEQ compares the operand with the accumulator (or index register for CBEQX instruction) against the contents of a memory location and causes a branch if the register (A or X) is equal to the memory contents. The CBEQ instruction combines CMP and BEQ for faster table lookup routines and condition codes are not changed.

The IX+ variation of the CBEQ instruction compares the operand addressed by H:X to A and causes a branch if the operands are equal. H:X is then incremented regardless of whether a branch is taken. The IX1+ variation of CBEQ operates the same way except that an 8-bit offset is added to H:X to form the effective address of the operand.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
CBEQ *opr8a*,*rel*	DIR	31	dd rr	5	rpppp
CBEQA #*opr8i*,*rel*	IMM	41	ii rr	4	pppp
CBEQX #*opr8i*,*rel*	IMM	51	ii rr	4	pppp
CBEQ *oprx8*,X+,*rel*	IX1+	61	ff rr	5	rpppp
CBEQ ,X+,*rel*	IX+	71	rr	5	rfppp
CBEQ *oprx8*,SP,*rel*	SP1	9E61	ff rr	6	prpppp

CLC Clear Carry Bit CLC

Operation

C bit ← 0

Description

Clears the C bit in the CCR. CLC may be used to set up the C bit prior to a shift or rotate instruction that involves the C bit. The C bit can also be used to pass status information between a subroutine and the calling program.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
—	1	1	—	—	—	—	0

C: 0
Cleared

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
CLC	INH	98		1	p

CLI Clear Interrupt Mask Bit CLI

Operation

I bit ← 0

Description

Clears the interrupt mask bit in the CCR. When the I bit is clear, interrupts are enabled. The I bit actually changes to zero at the end of the cycle where the CLI instruction executes. This is too late to recognize an interrupt that arrived before or during the CLI instruction so if interrupts were previously disabled, the next instruction after a CLI will always be executed even if there was an interrupt pending prior to execution of the CLI instruction.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
—	1	1	—	0	—	—	—

I: 0
Cleared

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
CLI	INH	9A		1	p

CLR Clear CLR

Operation

A ← $00
Or M ← $00
Or X ← $00
Or H ← $00

Description

The contents of memory (M), A, X, or H are replaced with zeros.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
0	1	1	—	—	0	1	—

V: 0
Cleared

N: 0
Cleared

Z: 1
Set

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
CLR *opr8a*	DIR	3F	dd	5	rfwpp
CLRA	INH (A)	4F		1	p
CLRX	INH (X)	5F		1	p
CLRH	INH (H)	8C		1	p
CLR *oprx8*,X	IX1	6F	ff	5	rfwpp
CLR ,X	IX	7F		4	rfwp
CLR *oprx8*,SP	SP1	9E6F	ff	6	prfwpp

CMP Compare Accumulator with Memory CMP

Operation (A) – (M)

Description Compares the contents of A to the contents of M and sets the condition codes, which may then be used for arithmetic (signed or unsigned) and logical conditional branching. The contents of both A and M are unchanged.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↕	1	1	—	—	↕	↕	↕

V: $A7\&\overline{M7}\&\overline{R7}\ |\ \overline{A7}\&M7\&R7$
Set if a two's complement overflow resulted from the operation; cleared otherwise. Literally read, an overflow condition occurs if a positive number is subtracted from a negative number with a positive result, or, if a negative number is subtracted from a positive number with a negative result.

N: R7
Set if MSB of result is 1; cleared otherwise

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is \$00; cleared otherwise

C: $\overline{A7}\&M7\ |\ M7\&R7\ |\ R7\&\overline{A7}$
Set if the unsigned value of the contents of memory is larger than the unsigned value of the accumulator; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form		Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
			Opcode	Operand(s)		
CMP	*#opr8i*	IMM	A1	ii	2	pp
CMP	*opr8a*	DIR	B1	dd	3	rpp
CMP	*opr16a*	EXT	C1	hh ll	4	prpp
CMP	*oprx16*,X	IX2	D1	ee ff	4	prpp
CMP	*oprx8*,X	IX1	E1	ff	3	rpp
CMP	,X	IX	F1		3	rfp
CMP	*oprx16*,SP	SP2	9ED1	ee ff	5	pprpp
CMP	*oprx8*,SP	SP1	9EE1	ff	4	prpp

COM Complement (One's Complement) COM

Operation

$A \leftarrow \overline{A}$ = \$FF – (A)
Or $X \leftarrow \overline{X}$ = \$FF – (X)
Or $M \leftarrow \overline{M}$ = \$FF – (M)

Description

Replaces the contents of A, X, or M with the one's complement. Each bit of A, X, or M is replaced with the complement of that bit.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
0	1	1	—	—	↕	↕	1

V: 0
Cleared

N: R7
Set if MSB of result is 1; cleared otherwise

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is \$00; cleared otherwise

C: 1
Set

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
COM *opr8a*	DIR	33	dd	5	rfwpp
COMA	INH (A)	43		1	p
COMX	INH (X)	53		1	p
COM *oprx8*,X	IX1	63	ff	5	rfwpp
COM ,X	IX	73		4	rfwp
COM *oprx8*,SP	SP1	9E63	ff	6	prfwpp

CPHX Compare Index Register with Memory CPHX

Operation (H:X) – (M:M + $0001)

Description CPHX compares index register (H:X) with the 16-bit value in memory and sets the condition codes, which may then be used for arithmetic (signed or unsigned) and logical conditional branching. The contents of both H:X and M:M + $0001 are unchanged.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↕	1	1	—	—	↕	↕	↕

V: $H7\&\overline{M15}\&\overline{R15} \mid \overline{H7}\&M15\&R15$
Set if a two’s complement overflow resulted from the operation; cleared otherwise

N: R15
Set if MSB of result is 1; cleared otherwise

Z: $\overline{R15}\&\overline{R14}\&\overline{R13}\&\overline{R12}\&\overline{R11}\&\overline{R10}\&\overline{R9}\&\overline{R8}$
$\&\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if the result is $0000; cleared otherwise

C: $\overline{H7}\&M15 \mid M15\&R15 \mid R15\&\overline{H7}$
Set if the absolute value of the contents of memory is larger than the absolute value of the index register; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
CPHX *opr16a*	EXT	3E	hh ll	6	prrfpp
CPHX *#opr16i*	IMM	65	jj kk	3	ppp
CPHX *opr8a*	DIR	75	dd	5	rrfpp
CPHX *oprx8*,SP	SP1	9EF3	ff	6	prrfpp

CPX Compare X (Index Register Low) with Memory CPX

Operation (X) – (M)

Description Compares the contents of X to the contents of M and sets the condition codes, which may then be used for arithmetic (signed or unsigned) and logical conditional branching. The contents of both X and M are unchanged.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↕	1	1	—	—	↕	↕	↕

V: $X7\&\overline{M7}\&\overline{R7} \mid \overline{X7}\&M7\&R7$
Set if a two's complement overflow resulted from the operation; cleared otherwise

N: R7
Set if MSB of result of the subtraction is 1; cleared otherwise

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is \$00; cleared otherwise

C: $\overline{X7}\&M7 \mid M7\&R7 \mid R7\&\overline{X7}$
Set if the unsigned value of the contents of memory is larger than the unsigned value in the index register; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
CPX *#opr8i*	IMM	A3	ii	2	pp
CPX *opr8a*	DIR	B3	dd	3	rpp
CPX *opr16a*	EXT	C3	hh ll	4	prpp
CPX *oprx16*,X	IX2	D3	ee ff	4	prpp
CPX *oprx8*,X	IX1	E3	ff	3	rpp
CPX ,X	IX	F3		3	rfp
CPX *oprx16*,SP	SP2	9ED3	ee ff	5	pprpp
CPX *oprx8*,SP	SP1	9EE3	ff	4	prpp

DAA Decimal Adjust Accumulator DAA

Operation $(A)_{10}$

Description Adjusts the contents of the accumulator and the state of the CCR carry bit after an ADD or ADC operation involving binary-coded decimal (BCD) values, so that there is a correct BCD sum and an accurate carry indication. The state of the CCR half carry bit affects operation. Refer to **Table A-2** for details of operation.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
U	1	1	—	—	↕	↕	↕

V: U

Undefined

N: R7
Set if MSB of result is 1; cleared otherwise

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is \$00; cleared otherwise

C: Set if the decimal adjusted result is greater than 99 (decimal); refer to **Table A-2**

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
DAA	INH	72		1	p

The DAA description continues next page.

DAA Decimal Adjust Accumulator (Continued) DAA

Table A-2 shows DAA operation for all legal combinations of input operands. Columns 1–4 represent the results of ADC or ADD operations on BCD operands. The correction factor in column 5 is added to the accumulator to restore the result of an operation on two BCD operands to a valid BCD value and to set or clear the C bit. All values in this table are hexadecimal.

Table A-2. DAA Function Summary

1	2	3	4	5	6
Initial C-Bit Value	**Value of A[7:4]**	**Initial H-Bit Value**	**Value of A[3:0]**	**Correction Factor**	**Corrected C-Bit Value**
0	0–9	0	0–9	00	0
0	0–8	0	A–F	06	0
0	0–9	1	0–3	06	0
0	A–F	0	0–9	60	1
0	9–F	0	A–F	66	1
0	A–F	1	0–3	66	1
1	0–2	0	0–9	60	1
1	0–2	0	A–F	66	1
1	0–3	1	0–3	66	1

DBNZ Decrement and Branch if Not Zero DBNZ

Operation

A ← (A) – $01
Or M ← (M) – $01
Or X ← (X) – $01

For DIR or IX1 modes: PC ← (PC) + $0003 + *rel* if (result) ≠ 0
Or for INH or IX modes: PC ← (PC) + $0002 + *rel* if (result) ≠ 0
Or for SP1 mode: PC ← (PC) + $0004 + *rel* if (result) ≠ 0

Description

Subtract 1 from the contents of A, M, or X; then branch using the relative offset if the result of the subtraction is not $00. DBNZX only affects the low order eight bits of the H:X index register pair; the high-order byte (H) is not affected.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
DBNZ *opr8a*,*rel*	DIR	3B	dd rr	7	rfwpppp
DBNZA *rel*	INH	4B	rr	4	fppp
DBNZX *rel*	INH	5B	rr	4	fppp
DBNZ *oprx8*,X,*rel*	IX1	6B	ff rr	7	rfwpppp
DBNZ ,X, *rel*	IX	7B	rr	6	rfwppp
DBNZ *oprx8*,SP,*rel*	SP1	9E6B	ff rr	8	prfwpppp

DEC Decrement DEC

Operation

A ← (A) – $01
Or X ← (X) – $01
Or M ← (M) – $01

Description

Subtract 1 from the contents of A, X, or M. The V, N, and Z bits in the CCR are set or cleared according to the results of this operation. The C bit in the CCR is not affected; therefore, the BLS, BLO, BHS, and BHI branch instructions are not useful following a DEC instruction.

DECX only affects the low-order byte of index register pair (H:X). To decrement the full 16-bit index register pair (H:X), use AIX # –1.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↕	1	1	—	—	↕	↕	—

V: $\overline{R7}$ & A7
Set if there was a two’s complement overflow as a result of the operation; cleared otherwise. Two’s complement overflow occurs if and only if (A), (X), or (M) was $80 before the operation.

N: R7
Set if MSB of result is 1; cleared otherwise

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is $00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
DEC *opr8a*	DIR	3A	dd	5	rfwpp
DECA	INH (A)	4A		1	p
DECX	INH (X)	5A		1	p
DEC *oprx8*,X	IX1	6A	ff	5	rfwpp
DEC ,X	IX	7A		4	rfwp
DEC *oprx8*,SP	SP1	9E6A	ff	6	prfwpp

DEX is recognized by assemblers as being equivalent to DECX.

DIV Divide DIV

Operation A ← (H:A) ÷ (X); H ← Remainder

Description Divides a 16-bit unsigned dividend contained in the concatenated registers H and A by an 8-bit divisor contained in X. The quotient is placed in A, and the remainder is placed in H. The divisor is left unchanged.

An overflow (quotient > $FF) or divide-by-0 sets the C bit, and the quotient and remainder are indeterminate.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
—	1	1	—	—	—	↕	↕

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result (quotient) is $00; cleared otherwise

C: Set if a divide-by-0 was attempted or if an overflow occurred; cleared otherwise

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Detail
		Opcode	Operand(s)		Access
DIV	INH	52		6	fffffp

EOR Exclusive-OR Memory with Accumulator EOR

Operation $A \leftarrow (A \oplus M)$

Description Performs the logical exclusive-OR between the contents of A and the contents of M and places the result in A. Each bit of A after the operation will be the logical exclusive-OR of the corresponding bits of M and A before the operation.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
0	1	1	—	—	↕	↕	—

V: 0
Cleared

N: R7
Set if MSB of result is 1; cleared otherwise

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is \$00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
EOR *#opr8i*	IMM	A8	ii	2	pp
EOR *opr8a*	DIR	B8	dd	3	rpp
EOR *opr16a*	EXT	C8	hh ll	4	prpp
EOR *oprx16*,X	IX2	D8	ee ff	4	prpp
EOR *oprx8*,X	IX1	E8	ff	3	rpp
EOR ,X	IX	F8		3	rfp
EOR *oprx16*,SP	SP2	9ED8	ee ff	5	pprpp
EOR *oprx8*,SP	SP1	9EE8	ff	4	prpp

INC Increment INC

Operation

A ← (A) + $01
Or X ← (X) + $01
Or M ← (M) + $01

Description

Add 1 to the contents of A, X, or M. The V, N, and Z bits in the CCR are set or cleared according to the results of this operation. The C bit in the CCR is not affected; therefore, the BLS, BLO, BHS, and BHI branch instructions are not useful following an INC instruction.

INCX only affects the low-order byte of index register pair (H:X). To increment the full 16-bit index register pair (H:X), use AIX #1.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↕	1	1	—	—	↕	↕	—

V: $\overline{A7}$&R7
Set if there was a two’s complement overflow as a result of the operation; cleared otherwise. Two’s complement overflow occurs if and only if (A), (X), or (M) was $7F before the operation.

N: R7
Set if MSB of result is 1; cleared otherwise

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is $00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
INC *opr8a*	DIR	3C	dd	5	rfwpp
INCA	INH (A)	4C		1	p
INCX	INH (X)	5C		1	p
INC *oprx8*,X	IX1	6C	ff	5	rfwpp
INC ,X	IX	7C		4	rfwp
INC *oprx8*,SP	SP1	9E6C	ff	6	prfwpp

INX is recognized by assemblers as being equivalent to INCX.

JMP Jump JMP

Operation

PC ← effective address

Description

A jump occurs to the instruction stored at the effective address. The effective address is obtained according to the rules for extended, direct, or indexed addressing.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
JMP *opr8a*	DIR	BC	dd	3	ppp
JMP *opr16a*	EXT	CC	hh ll	4	pppp
JMP *oprx16*,X	IX2	DC	ee ff	4	pppp
JMP *oprx8*,X	IX1	EC	ff	3	ppp
JMP ,X	IX	FC		3	ppp

JSR Jump to Subroutine JSR

Operation

PC ← (PC) + *n*;
n = 1, 2, or 3 depending on address mode

Push (PCL); SP ← (SP) – $0001	Push low half of return address
Push (PCH); SP ← (SP) – $0001	Push high half of return address
PC ← effective address	Load PC with start address of requested subroutine

Description

The program counter is incremented by *n* so that it points to the opcode of the next instruction that follows the JSR instruction (*n* = 1, 2, or 3 depending on the addressing mode). The PC is then pushed onto the stack, eight bits at a time, least significant byte first. The stack pointer points to the next empty location on the stack. A jump occurs to the instruction stored at the effective address. The effective address is obtained according to the rules for extended, direct, or indexed addressing.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
JSR *opr8a*	DIR	BD	dd	5	ssppp
JSR *opr16a*	EXT	CD	hh ll	6	pssppp
JSR *oprx16*,X	IX2	DD	ee ff	6	pssppp
JSR *oprx8*,X	IX1	ED	ff	5	ssppp
JSR ,X	IX	FD		5	ssppp

LDA Load Accumulator from Memory LDA

Operation

A ← (M)

Description

Loads the contents of the specified memory location into A. The N and Z condition codes are set or cleared according to the loaded data; V is cleared. This allows conditional branching after the load without having to perform a separate test or compare.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
0	1	1	—	—	↕	↕	—

V: 0
Cleared

N: R7
Set if MSB of result is 1; cleared otherwise

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is $00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
LDA *#opr8i*	IMM	A6	ii	2	pp
LDA *opr8a*	DIR	B6	dd	3	rpp
LDA *opr16a*	EXT	C6	hh ll	4	prpp
LDA *oprx16*,X	IX2	D6	ee ff	4	prpp
LDA *oprx8*,X	IX1	E6	ff	3	rpp
LDA ,X	IX	F6		3	rfp
LDA *oprx16*,SP	SP2	9ED6	ee ff	5	pprpp
LDA *oprx8*,SP	SP1	9EE6	ff	4	prpp

LDHX Load Index Register from Memory LDHX

Operation

H:X ← (M:M + $0001)

Description

Loads the contents of the specified memory location into the index register (H:X). The N and Z condition codes are set according to the data; V is cleared. This allows conditional branching after the load without having to perform a separate test or compare.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
0	1	1	—	—	↕	↕	—

V: 0
Cleared

N: R15
Set if MSB of result is 1; cleared otherwise

Z: $\overline{R15}\&\overline{R14}\&\overline{R13}\&\overline{R12}\&\overline{R11}\&\overline{R10}\&\overline{R9}\&\overline{R8}$
$\&\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if the result is $0000; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
LDHX *#opr16i*	IMM	45	jj kk	3	ppp
LDHX *opr8a*	DIR	55	dd	4	rrpp
LDHX *opr16a*	EXT	32	hh ll	5	prrpp
LDHX ,X	IX	9EAE		5	prrfp
LDHX *oprx16*,X	IX2	9EBE	ee ff	6	pprrpp
LDHX *oprx8*,X	IX1	9ECE	ff	5	prrpp
LDHX *oprx8*,SP	SP1	9EFE	ff	5	prrpp

LDX Load X (Index Register Low) from Memory LDX

Operation

X ← (M)

Description

Loads the contents of the specified memory location into X. The N and Z condition codes are set or cleared according to the loaded data; V is cleared. This allows conditional branching after the load without having to perform a separate test or compare.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
0	1	1	—	—	↕	↕	—

V: 0
Cleared

N: R7
Set if MSB of result is 1; cleared otherwise

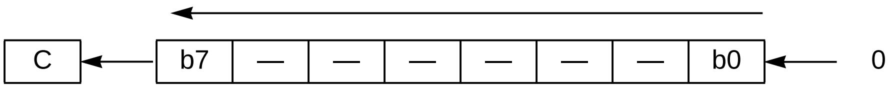
Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is $00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
LDX *#opr8i*	IMM	AE	ii	2	pp
LDX *opr8a*	DIR	BE	dd	3	rpp
LDX *opr16a*	EXT	CE	hh ll	4	prpp
LDX *oprx16*,X	IX2	DE	ee ff	4	prpp
LDX *oprx8*,X	IX1	EE	ff	3	rpp
LDX ,X	IX	FE		3	rfp
LDX *oprx16*,SP	SP2	9EDE	ee ff	5	pprpp
LDX *oprx8*,SP	SP1	9EEE	ff	4	prpp

LSL — Logical Shift Left (Same as ASL) — LSL

Operation



Description

Shifts all bits of the A, X, or M one place to the left. Bit 0 is loaded with a 0. The C bit in the CCR is loaded from the most significant bit of A, X, or M.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↕	1	1	—	—	↕	↕	↕

V: $R7 \oplus b7$
Set if the exclusive-OR of the resulting N and C flags is 1; cleared otherwise

N: R7
Set if MSB of result is 1; cleared otherwise

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is \$00; cleared otherwise

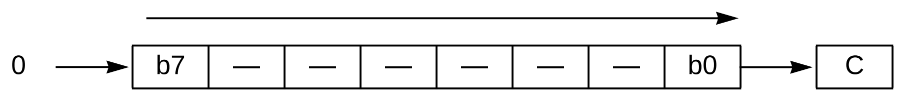
C: b7
Set if, before the shift, the MSB of A, X, or M was set; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
LSL *opr8a*	DIR	38	dd	5	rfwpp
LSLA	INH (A)	48		1	p
LSLX	INH (X)	58		1	p
LSL *oprx8*,X	IX1	68	ff	5	rfwpp
LSL ,X	IX	78		4	rfwp
LSL *oprx8*,SP	SP1	9E68	ff	6	prfwpp

LSR Logical Shift Right LSR

Operation



Description

Shifts all bits of A, X, or M one place to the right. Bit 7 is loaded with a 0. Bit 0 is shifted into the C bit.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↕	1	1	—	—	0	↕	↕

V: $0 \oplus b0 = b0$
Set if the exclusive-OR of the resulting N and C flags is 1; cleared otherwise. Since N = 0, this simplifies to the value of bit 0 before the shift.

N: 0
Cleared

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is $00; cleared otherwise

C: b0
Set if, before the shift, the LSB of A, X, or M, was set; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
LSR *opr8a*	DIR	34	dd	5	rfwpp
LSRA	INH (A)	44		1	p
LSRX	INH (X)	54		1	p
LSR *oprx8*,X	IX1	64	ff	5	rfwpp
LSR ,X	IX	74		4	rfwp
LSR *oprx8*,SP	SP1	9E64	ff	6	prfwpp

MOV Move MOV

Operation

$(M)_{Destination} \leftarrow (M)_{Source}$

Description

Moves a byte of data from a source address to a destination address. Data is examined as it is moved, and condition codes are set. Source data is not changed. The accumulator is not affected.

The four addressing modes for the MOV instruction are:

1. IMM/DIR moves an immediate byte to a direct memory location.
2. DIR/DIR moves a direct location byte to another direct location.
3. IX+/DIR moves a byte from a location addressed by H:X to a direct location. H:X is incremented after the move.
4. DIR/IX+ moves a byte from a direct location to one addressed by H:X. H:X is incremented after the move.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
0	1	1	—	—	↕	↕	—

V: 0
Cleared

N: R7
Set if MSB of result is set; cleared otherwise

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is \$00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
MOV *opr8a*,*opr8a*	DIR/DIR	4E	dd dd	5	rpwpp
MOV *opr8a*,X+	DIR/IX+	5E	dd	5	rfwpp
MOV #*opr8i*,*opr8a*	IMM/DIR	6E	ii dd	4	pwpp
MOV ,X+,*opr8a*	IX+/DIR	7E	dd	5	rfwpp

MUL Unsigned Multiply MUL

Operation

X:A ← (X) × (A)

Description

Multiplies the 8-bit value in X (index register low) by the 8-bit value in the accumulator to obtain a 16-bit unsigned result in the concatenated index register and accumulator. After the operation, X contains the upper eight bits of the 16-bit result and A contains the lower eight bits of the result.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
—	1	1	0	—	—	—	0

H: 0
Cleared

C: 0
Cleared

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
MUL	INH	42		5	ffffp

NEG Negate (Two's Complement) NEG

Operation

A ← – (A)
Or X ← – (X)
Or M ← – (M);
this is equivalent to subtracting A, X, or M from $00

Description

Replaces the contents of A, X, or M with its two's complement. Note that the value $80 is left unchanged.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↕	1	1	—	—	↕	↕	↕

V: M7&R7
Set if a two's complement overflow resulted from the operation; cleared otherwise. Overflow will occur only if the operand is $80 before the operation.

N: R7
Set if MSB of result is 1; cleared otherwise

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is $00; cleared otherwise

C: R7|R6|R5|R4|R3|R2|R1|R0
Set if there is a borrow in the implied subtraction from 0; cleared otherwise. The C bit will be set in all cases except when the contents of A, X, or M was $00 prior to the NEG operation.

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
NEG *opr8a*	DIR	30	dd	5	rfwpp
NEGA	INH (A)	40		1	p
NEGX	INH (X)	50		1	p
NEG *oprx8*,X	IX1	60	ff	5	rfwpp
NEG ,X	IX	70		4	rfwp
NEG *oprx8*,SP	SP1	9E60	ff	6	prfwpp

NOP No Operation NOP

Operation

Uses one bus cycle

Description

This is a single-byte instruction that does nothing except to consume one CPU clock cycle while the program counter is advanced to the next instruction. No register or memory contents are affected by this instruction.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
NOP	INH	9D		1	p

NSA Nibble Swap Accumulator NSA

Operation

A ← (A[3:0]:A[7:4])

Description

Swaps upper and lower nibbles (4 bits) of the accumulator. The NSA instruction is used for more efficient storage and use of binary-coded decimal operands.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
NSA	INH	62		1	p

ORA Inclusive-OR Accumulator and Memory ORA

Operation A ← (A) | (M)

Description Performs the logical inclusive-OR between the contents of A and the contents of M and places the result in A. Each bit of A after the operation will be the logical inclusive-OR of the corresponding bits of M and A before the operation.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
0	1	1	—	—	↕	↕	—

V: 0
Cleared

N: R7
Set if MSB of result is 1; cleared otherwise

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is \$00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
ORA *#opr8i*	IMM	AA	ii	2	pp
ORA *opr8a*	DIR	BA	dd	3	rpp
ORA *opr16a*	EXT	CA	hh ll	4	prpp
ORA *oprx16*,X	IX2	DA	ee ff	4	prpp
ORA *oprx8*,X	IX1	EA	ff	3	rpp
ORA ,X	IX	FA		3	rfp
ORA *oprx16*,SP	SP2	9EDA	ee ff	5	pprpp
ORA *oprx8*,SP	SP1	9EEA	ff	4	prpp

PSHA — Push Accumulator onto Stack — PSHA

Operation

Push (A); SP ← (SP) – $0001

Description

The contents of A are pushed onto the stack at the address contained in the stack pointer. The stack pointer is then decremented to point to the next available location in the stack. The contents of A remain unchanged.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
PSHA	INH	87		2	sp

PSHH Push H (Index Register High) onto Stack PSHH

Operation

Push (H); SP ← (SP) – $0001

Description

The contents of H are pushed onto the stack at the address contained in the stack pointer. The stack pointer is then decremented to point to the next available location in the stack. The contents of H remain unchanged.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
PSHH	INH	8B		2	sp

PSHX — Push X (Index Register Low) onto Stack — PSHX

Operation

Push (X); SP ← (SP) – $0001

Description

The contents of X are pushed onto the stack at the address contained in the stack pointer (SP). SP is then decremented to point to the next available location in the stack. The contents of X remain unchanged.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
PSHX	INH	89		2	sp

PULA Pull Accumulator from Stack PULA

Operation

SP ← (SP + $0001); pull (A)

Description

The stack pointer (SP) is incremented to address the last operand on the stack. The accumulator is then loaded with the contents of the address pointed to by SP.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
PULA	INH	86		3	ufp

PULH Pull H (Index Register High) from Stack PULH

Operation

SP ← (SP + $0001); pull (H)

Description

The stack pointer (SP) is incremented to address the last operand on the stack. H is then loaded with the contents of the address pointed to by SP.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
PULH	INH	8A		3	ufp

PULX Pull X (Index Register Low) from Stack PULX

Operation

SP ← (SP + $0001); pull (X)

Description

The stack pointer (SP) is incremented to address the last operand on the stack. X is then loaded with the contents of the address pointed to by SP.

Condition Codes and Boolean Formulae

None affected

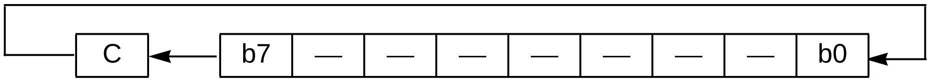
V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
PULX	INH	88		3	ufp

ROL Rotate Left through Carry ROL

Operation



Description

Shifts all bits of A, X, or M one place to the left. Bit 0 is loaded from the C bit. The C bit is loaded from the most significant bit of A, X, or M. The rotate instructions include the carry bit to allow extension of the shift and rotate instructions to multiple bytes. For example, to shift a 24-bit value left one bit, the sequence (ASL LOW, ROL MID, ROL HIGH) could be used, where LOW, MID, and HIGH refer to the low-order, middle, and high-order bytes of the 24-bit value, respectively.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↕	1	1	—	—	↕	↕	↕

V: $R7 \oplus b7$
Set if the exclusive-OR of the resulting N and C flags is 1; cleared otherwise

N: R7
Set if MSB of result is 1; cleared otherwise

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is \$00; cleared otherwise

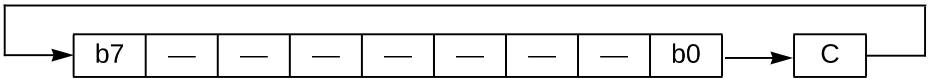
C: b7
Set if, before the rotate, the MSB of A, X, or M was set; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
ROL *opr8a*	DIR	39	dd	5	rfwpp
ROLA	INH (A)	49		1	p
ROLX	INH (X)	59		1	p
ROL *oprx8*,X	IX1	69	ff	5	rfwpp
ROL ,X	IX	79		4	rfwp
ROL *oprx8*,SP	SP1	9E69	ff	6	prfwpp

ROR Rotate Right through Carry ROR

Operation



Description

Shifts all bits of A, X, or M one place to the right. Bit 7 is loaded from the C bit. Bit 0 is shifted into the C bit. The rotate instructions include the carry bit to allow extension of the shift and rotate instructions to multiple bytes. For example, to shift a 24-bit value right one bit, the sequence (LSR HIGH, ROR MID, ROR LOW) could be used, where LOW, MID, and HIGH refer to the low-order, middle, and high-order bytes of the 24-bit value, respectively.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↕	1	1	—	—	↕	↕	↕

V: $R7 \oplus b0$
Set if the exclusive-OR of the resulting N and C flags is 1; cleared otherwise

N: R7
Set if MSB of result is 1; cleared otherwise

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is \$00; cleared otherwise

C: b0
Set if, before the shift, the LSB of A, X, or M was set; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
ROR *opr8a*	DIR	36	dd	5	rfwpp
RORA	INH (A)	46		1	p
RORX	INH (X)	56		1	p
ROR *oprx8*,X	IX1	66	ff	5	rfwpp
ROR ,X	IX	76		4	rfwp
ROR *oprx8*,SP	SP1	9E66	ff	6	prfwpp

RSP Reset Stack Pointer RSP

Operation

SP ← $FF

Description

For M68HC08 compatibility, the HCS08 RSP instruction only sets the least significant byte of SP to $FF. The most significant byte is unaffected.

In most M68HC05 MCUs, RAM only goes to $00FF. In most HCS08s, however, RAM extends beyond $00FF. Therefore, do not locate the stack in direct address space which is more valuable for commonly accessed variables. In new HCS08 programs, it is more appropriate to initialize the stack pointer to the address of the last location (highest address) in the on-chip RAM, shortly after reset. This code segment demonstrates a typical method for initializing SP.

```
LDHX   #RamLast+1   ; Point at next addr past RAM
TXS                 ; SP <-(H:X)-1
```

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
RSP	INH	9C		1	p

RTI Return from Interrupt RTI

Operation

SP ← SP + $0001; pull (CCR)	Restore CCR from stack
SP ← SP + $0001; pull (A)	Restore A from stack
SP ← SP + $0001; pull (X)	Restore X from stack
SP ← SP + $0001; pull (PCH)	Restore PCH from stack
SP ← SP + $0001; pull (PCL)	Restore PCL from stack

Description

The condition codes, the accumulator, X (index register low), and the program counter are restored to the state previously saved on the stack. The I bit will be cleared if the corresponding bit stored on the stack is 0, the normal case. If this instruction causes the I bit to change from 1 to 0, a one bus cycle delay is imposed before interrupts are allowed. This ensures that the next instruction after an RTI instruction will always be executed, even if an interrupt was pending before the RTI instruction was executed and bit 3 of the CCR value on the stack cleared.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↕	1	1	↕	↕	↕	↕	↕

Set or cleared according to the byte pulled from the stack into CCR.

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
RTI	INH	80		9	uuuuufppp

RTS Return from Subroutine RTS

Operation

SP ← SP + $0001; pull (PCH) Restore PCH from stack
SP ← SP + $0001; pull (PCL) Restore PCL from stack

Description

The stack pointer is incremented by 1. The contents of the byte of memory that is pointed to by the stack pointer are loaded into the high-order byte of the program counter. The stack pointer is again incremented by 1. The contents of the byte of memory that are pointed to by the stack pointer are loaded into the low-order eight bits of the program counter. Program execution resumes at the address that was just restored from the stack.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
RTS	INH	81		6	uufppp

SBC Subtract with Carry SBC

Operation A ← (A) – (M) – (C)

Description Subtracts the contents of M and the contents of the C bit of the CCR from the contents of A and places the result in A. This is useful for multi-precision subtract algorithms involving operands with more than eight bits.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↕	1	1	—	—	↕	↕	↕

V: $A7\&\overline{M7}\&\overline{R7} \mid \overline{A7}\&M7\&R7$
Set if a two’s complement overflow resulted from the operation; cleared otherwise. Literally read, an overflow condition occurs if a positive number is subtracted from a negative number with a positive result, or, if a negative number is subtracted from a positive number with a negative result.

N: R7
Set if MSB of result is 1; cleared otherwise

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is \$00; cleared otherwise

C: $\overline{A7}\&M7 \mid M7\&R7 \mid R7\&\overline{A7}$
Set if the unsigned value of the contents of memory plus the previous carry are larger than the unsigned value of the accumulator; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
SBC *#opr8i*	IMM	A2	ii	2	pp
SBC *opr8a*	DIR	B2	dd	3	rpp
SBC *opr16a*	EXT	C2	hh ll	4	prpp
SBC *oprx16*,X	IX2	D2	ee ff	4	prpp
SBC *oprx8*,X	IX1	E2	ff	3	rpp
SBC ,X	IX	F2		3	rfp
SBC *oprx16*,SP	SP2	9ED2	ee ff	5	pprpp
SBC *oprx8*,SP	SP1	9EE2	ff	4	prpp

SEC Set Carry Bit SEC

Operation

C bit ← 1

Description

Sets the C bit in the condition code register (CCR). SEC may be used to set up the C bit prior to a shift or rotate instruction that involves the C bit.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
—	1	1	—	—	—	—	1

C: 1
Set

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
SEC	INH	99		1	p

SEI Set Interrupt Mask Bit SEI

Operation

I bit ← 1

Description

Sets the interrupt mask bit in the condition code register (CCR). The microprocessor is inhibited from responding to interrupts while the I bit is set. The I bit actually changes at the end of the cycle where SEI executed. This is too late to stop an interrupt that arrived during execution of the SEI instruction so it is possible that an interrupt request could be serviced after the SEI instruction before the next instruction after SEI is executed. The global I-bit interrupt mask takes effect before the next instruction can be completed.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
—	1	1	—	1	—	—	—

I: 1
Set

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
SEI	INH	9B		1	p

STA Store Accumulator in Memory STA

Operation M ← (A)

Description Stores the contents of A in memory. The contents of A remain unchanged. The N condition code is set if the most significant bit of A is set, the Z bit is set if A was $00, and V is cleared. This allows conditional branching after the store without having to do a separate test or compare.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
0	1	1	—	—	↕	↕	—

V: 0
Cleared

N: A7
Set if MSB of result is 1; cleared otherwise

Z: $\overline{A7}\&\overline{A6}\&\overline{A5}\&\overline{A4}\&\overline{A3}\&\overline{A2}\&\overline{A1}\&\overline{A0}$
Set if result is $00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
STA *opr8a*	DIR	B7	dd	3	wpp
STA *opr16a*	EXT	C7	hh ll	4	pwpp
STA *oprx16*,X	IX2	D7	ee ff	4	pwpp
STA *oprx8*,X	IX1	E7	ff	3	wpp
STA ,X	IX	F7		2	wp
STA *oprx16*,SP	SP2	9ED7	ee ff	5	ppwpp
STA *oprx8*,SP	SP1	9EE7	ff	4	pwpp

STHX — Store Index Register — STHX

Operation

$(M:M + \$0001) \leftarrow (H:X)$

Description

Stores the contents of H in memory location M and then the contents of X into the next memory location (M + $0001). The N condition code bit is set if the most significant bit of H was set, the Z bit is set if the value of H:X was $0000, and V is cleared. This allows conditional branching after the store without having to do a separate test or compare.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
0	1	1	—	—	↕	↕	—

V: 0
Cleared

N: R15
Set if MSB of result is 1; cleared otherwise

Z: $\overline{R15}\&\overline{R14}\&\overline{R13}\&\overline{R12}\&\overline{R11}\&\overline{R10}\&\overline{R9}\&\overline{R8}$
$\&\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if the result is $0000; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
STHX *opr8a*	DIR	35	dd	4	wwpp
STHX *opr16a*	EXT	96	hh ll	5	pwwpp
STHX *oprx8*,SP	SP1	9E FF	ff	5	pwwpp

STOP Enable IRQ Pin, Stop Processing STOP

Operation

I bit ← 0; stop processing

Description

Reduces power consumption by eliminating all dynamic power dissipation. Depending on system configuration, this instruction is used to enter stop1, stop2, or stop3 mode. (See module documentation for the behavior of these modes and module reactions to the stop instruction.)

The external interrupt pin is enabled and the I bit in the condition code register (CCR) is cleared to enable interrupts. Interrupts can be used to exit stop3 only.

Finally, the oscillator is inhibited to put the MCU into the stop condition. In stop1 or stop2 mode, when either the RESET pin or IRQ pin goes low, the reset vector is fetched and the MCU operates as if a POR has occurred. For stop3 mode, if an IRQ, KBI, or RTI interrupt occurs, the associated service routine is executed. Upon stop recovery, normally the MCU defaults to a self-clocked system clock source so there is little or no startup delay.

Some HCS08 derivatives can be configured so the oscillator and real-time interrupt (RTI) module continue to run in stop mode so no external components are needed to make the MCU periodically wake up from stop. Also, if the background debug system is enabled (ENBDM), only stop3 mode is entered and the oscillator continues to run so a host debug system can still force the target MCU into active background mode.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
—	1	1	—	0	—	—	—

I: 0
Cleared

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
STOP	INH	8E		2+stop	fp

STX Store X (Index Register Low) in Memory STX

Operation

M ← (X)

Description

Stores the contents of X in memory. The contents of X remain unchanged. The N condition code is set if the most significant bit of X was set, the Z bit is set if X was $00, and V is cleared. This allows conditional branching after the store without having to do a separate test or compare.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
0	1	1	—	—	↕	↕	—

V: 0
Cleared

N: X7
Set if MSB of result is 1; cleared otherwise

Z: $\overline{X7}\&\overline{X6}\&\overline{X5}\&\overline{X4}\&\overline{X3}\&\overline{X2}\&\overline{X1}\&\overline{X0}$
Set if X is $00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
STX *opr8a*	DIR	BF	dd	3	wpp
STX *opr16a*	EXT	CF	hh ll	4	pwpp
STX *oprx16*,X	IX2	DF	ee ff	4	pwpp
STX *oprx8*,X	IX1	EF	ff	3	wpp
STX ,X	IX	FF		2	wp
STX *oprx16*,SP	SP2	9EDF	ee ff	5	ppwpp
STX *oprx8*,SP	SP1	9EEF	ff	4	pwpp

SUB Subtract SUB

Operation

A ← (A) – (M)

Description

Subtracts the contents of M from A and places the result in A

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↕	1	1	—	—	↕	↕	↕

V: $A7\&\overline{M7}\&\overline{R7} \mid \overline{A7}\&M7\&R7$
Set if a two's complement overflow resulted from the operation; cleared otherwise. Literally read, an overflow condition occurs if a positive number is subtracted from a negative number with a positive result, or, if a negative number is subtracted from a positive number with a negative result.

N: R7
Set if MSB of result is 1; cleared otherwise

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is \$00; cleared otherwise

C: $\overline{A7}\&M7 \mid M7\&R7 \mid R7\&\overline{A7}$
Set if the unsigned value of the contents of memory is larger than the unsigned value of the accumulator; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
SUB *#opr8i*	IMM	A0	ii	2	pp
SUB *opr8a*	DIR	B0	dd	3	rpp
SUB *opr16a*	EXT	C0	hh ll	4	prpp
SUB *oprx16*,X	IX2	D0	ee ff	4	prpp
SUB *oprx8*,X	IX1	E0	ff	3	rpp
SUB X	IX	F0		3	rfp
SUB *oprx16*,SP	SP2	9ED0	ee ff	5	pprpp
SUB *oprx8*,SP	SP1	9EE0	ff	4	prpp

SWI Software Interrupt SWI

Operation

PC ← (PC) + $0001	Increment PC to return address
Push (PCL); SP ← (SP) – $0001	Push low half of return address
Push (PCH); SP ← (SP) – $0001	Push high half of return address
Push (X); SP ← (SP) – $0001	Push index register on stack
Push (A); SP ← (SP) – $0001	Push A on stack
Push (CCR); SP ← (SP) – $0001	Push CCR on stack
Push bit ← 1	Mask further interrupts
PCH ← ($FFFC)	Vector fetch (high byte)
PCL ← ($FFFD)	Vector fetch (low byte)

Description

The program counter (PC) is incremented by 1 to point at the instruction after the SWI. The PC, index register, and accumulator are pushed onto the stack. The condition code register (CCR) bits are then pushed onto the stack, with bits V, H, I, N, Z, and C going into bit positions 7 and 4–0. Bit positions 6 and 5 contain 1s. The stack pointer is decremented by 1 after each byte of data is stored on the stack. The interrupt mask bit is then set. The program counter is then loaded with the address stored in the SWI vector located at memory locations $FFFC and $FFFD. This instruction is not maskable by the I bit.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
—	1	1	—	1	—	—	—

I: 1
Set

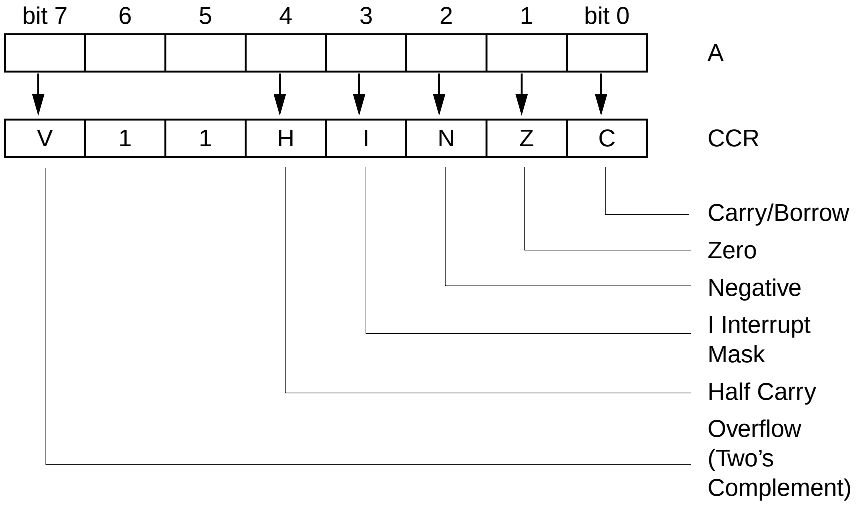
Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
SWI	INH	83		11	ssssssvvfppp

TAP Transfer Accumulator to Processor Status Byte TAP

Operation

CCR ← (A)



Description

Transfers the contents of A to the condition code register (CCR). The contents of A are unchanged. If this instruction causes the I bit to change from 1 to 0, a one bus cycle delay is imposed before interrupts are allowed. This ensures that the next instruction after a TAP instruction will always be executed even if an interrupt was pending before the TAP instruction was executed with bit 3 of accumulator A cleared.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↕	1	1	↕	↕	↕	↕	↕

Set or cleared according to the value that was in the accumulator.

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Details
		Opcode	Operand(s)		
TAP	INH	84		1	p

TAX Transfer Accumulator to X (Index Register Low) TAX

Operation

X ← (A)

Description

Loads X with the contents of the accumulator (A). The contents of A are unchanged.

Condition Codes and Boolean Formulae

None affected

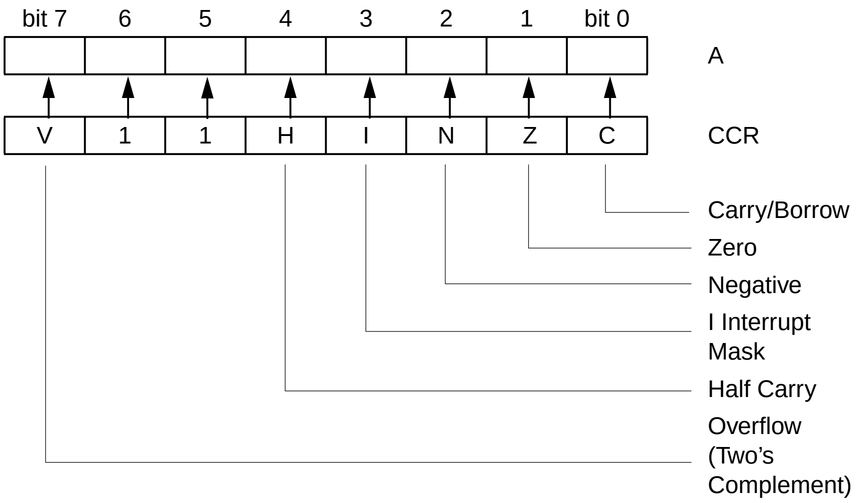
V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
TAX	INH	97		1	p

TPA Transfer Processor Status Byte to Accumulator TPA

Operation A ← (CCR)



Description Transfers the contents of the condition code register (CCR) into the accumulator (A)

Condition Codes and Boolean Formulae None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
TPA	INH	85		1	p

TST Test for Negative or Zero TST

Operation

(A) – \$00
Or (X) – \$00
Or (M) – \$00

Description

Sets the N and Z condition codes according to the contents of A, X, or M. The contents of A, X, and M are not altered.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
0	1	1	—	—	↕	↕	—

V: 0
Cleared

N: M7
Set if MSB of the tested value is 1; cleared otherwise

Z: $\overline{M7}\&\overline{M6}\&\overline{M5}\&\overline{M4}\&\overline{M3}\&\overline{M2}\&\overline{M1}\&\overline{M0}$
Set if A, X, or M contains \$00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Detail
		Opcode	Operand(s)		
TST *opr8a*	DIR	3D	dd	4	rfpp
TSTA	INH (A)	4D		1	p
TSTX	INH (X)	5D		1	p
TST *oprx8*,X	IX1	6D	ff	4	rfpp
TST ,X	IX	7D		3	rfp
TST *oprx8*,SP	SP1	9E6D	ff	5	prfpp

TSX Transfer Stack Pointer to Index Register TSX

Operation

H:X ← (SP) + $0001

Description

Loads index register (H:X) with 1 plus the contents of the stack pointer (SP). The contents of SP remain unchanged. After a TSX instruction, H:X points to the last value that was stored on the stack.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form		Addr. Mode	Machine Code		HCS08 Cycles	Access Details
			Opcode	Operand(s)		
TSX		INH	95		2	fp

TXA Transfer X (Index Register Low) to Accumulator TXA

Operation

A ← (X)

Description

Loads the accumulator (A) with the contents of X. The contents of X are not altered.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Details
		Opcode	Operand(s)		
`TXA`	INH	9F		1	p

TXS Transfer Index Register to Stack Pointer TXS

Operation

SP ← (H:X) – $0001

Description

Loads the stack pointer (SP) with the contents of the index register (H:X) minus 1. The contents of H:X are not altered.

Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Details
		Opcode	Operand(s)		
TXS	INH	94		2	fp

WAIT Enable Interrupts; Stop Processor WAIT

Operation

I bit ← 0; inhibit CPU clocking until interrupted

Description

Reduces power consumption by eliminating dynamic power dissipation in some portions of the MCU. The timer, the timer prescaler, and the on-chip peripherals continue to operate (if enabled) because they are potential sources of an interrupt. Wait causes enabling of interrupts by clearing the I bit in the CCR and stops clocking of processor circuits.

Interrupts from on-chip peripherals may be enabled or disabled by local control bits prior to execution of the WAIT instruction.

When either the $\overline{\text{RESET}}$ or IRQ pin goes low or when any on-chip system requests interrupt service, the processor clocks are enabled, and the reset, IRQ, or other interrupt service request is processed.

Condition Codes and Boolean Formulae

V			H	I	N	Z	C
—	1	1	—	0	—	—	—

I: 0
Cleared

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		HCS08 Cycles	Access Details
		Opcode	Operand(s)		
WAIT	INH	8F		2+wait	fp

Appendix B. Equate File Conventions

B.1 Introduction

This appendix describes the conventions used to create and use device definition files, usually called equate files. The equate file for the first device derivative in the HCS08 Family (9S08GB60_v1.equ) is used as an example and the entire equate file is included in **B.5 Complete Equate File for MC9S08GB60**. Each new member of the HCS08 Family will have a similar equate file available on the Motorola MCU Web site http://www.motorola.com/semiconductors

Equate files do not produce object code, so including this file in an application program does not affect program size. The equate file defines all control register and bit names from the manufacturer's documentation into a form that is understood by the assembler. The equate file also defines some basic system attributes including the beginning and ending addresses of on-chip memory blocks and the name and location of all interrupt vectors. The file is comprised entirely of EQU directives and comments.

All register names and bit names use uppercase characters so they match the spelling and capitalization used in the data sheet and other manuals. To help prevent conflicting register names as new device derivatives are introduced, register names will start with a 2- or 3-character prefix that identifies the module they are located in. For example, the KBI in KBISC indicates this register is located in the keyboard interrupt module (KBI). When more than one copy of a module is included in the MCU derivative, a digit immediately after this prefix indicates which instance of the module the register is located in, such as SCI1C1 and SCI2C1, which refer to the control register number 1 in SCI module 1 and 2, respectively.

Occasionally, two different control bits may have the same name. The most common case occurs when two identical modules are included on the same MCU. In this situation, the matching bit names don't really conflict because the definitions equate the bit name to its bit number and its bit position which are the same for both registers. When two identical modules are included, the register names must include the module number in the name to make each register name unique, but the bit numbers and bit positions can simply be defined once. These definitions are valid regardless of which register is being referenced, so there is no conflict.

In this example, the first two lines identify the status register number 2 for each of the SCI modules. The remaining lines define the bit position and bit number once and these definitions may be used with either register.

```
SCI1S2:      equ    $1D           ;SCI1 status register 2
SCI2S2:      equ    $25           ;SCI2 status register 2
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
RAF:         equ    0             ;(bit #0) Rx active flag
; bit position masks
mRAF:        equ    %00000001     ;receiver active flag
```

As future modules are designed for the HCS08 Family, care will be taken to avoid bit names that are the same in different registers but are not located in the same bit position in all registers where the name appears.

B.2 Memory Map Definition

The first set of EQU directives defines the starting and ending addresses for each on-chip memory block. The main program memory is called "Rom" even if it is actually FLASH memory in the HCS08 Family. For each memory, there is an xxxStart definition and an xxxLast definition. This book uses a combination of uppercase and lowercase letters to break up multiword labels so "RomStart" is the convention rather than "rom_start."

```
RomStart:    equ    $1080         ;start of 60K flash
HighRegs:    equ    $1800         ;start of high page registers
Rom1Start:   equ    $182C         ;start of flash after high regs
RomLast:     equ    $FFFF         ;last flash location
RamStart:    equ    $0080         ;start of 4096 byte RAM
RamLast:     equ    $107F         ;last RAM location
```

B.3 Vector Definitions

The next set of EQU directives defines the location of each interrupt vector starting from the lowest vector address and continuing through the reset vector location at the end of memory ($FFFE:FFFF). The names for each of these vector definitions starts with an uppercase V. Care should be taken to use the same name for these vectors in equate files for other derivatives that reuse a module such as the TPM or SCI.

```
Vrti:        equ    $FFCC         ;RTI (periodic interrupt) vector
Viic:        equ    $FFCE         ;IIC vector
Vatd:        equ    $FFD0         ;analog to digital conversion vector
Vkeyboard:   equ    $FFD2         ;keyboard vector
Vsci2tx:     equ    $FFD4         ;SCI2 transmit vector
Vsci2rx:     equ    $FFD6         ;SCI2 receive vector
Vsci2err:    equ    $FFD8         ;SCI2 error vector
Vsci1tx:     equ    $FFDA         ;SCI1 transmit vector
Vsci1rx:     equ    $FFDC         ;SCI1 receive vector
Vsci1err:    equ    $FFDE         ;SCI1 error vector
Vspi:        equ    $FFE0         ;SPI vector
Vtpm2ovf:    equ    $FFE2         ;TPM2 overflow vector
Vtpm2ch4:    equ    $FFE4         ;TPM2 channel 4 vector
Vtpm2ch3:    equ    $FFE6         ;TPM2 channel 3 vector
Vtpm2ch2:    equ    $FFE8         ;TPM2 channel 2 vector
Vtpm2ch1:    equ    $FFEA         ;TPM2 channel 1 vector
Vtpm2ch0:    equ    $FFEC         ;TPM2 channel 0 vector
Vtpm1ovf:    equ    $FFEE         ;TPM1 overflow vector
Vtpm1ch2:    equ    $FFF0         ;TPM1 channel 2 vector
Vtpm1ch1:    equ    $FFF2         ;TPM1 channel 1 vector
Vtpm1ch0:    equ    $FFF4         ;TPM1 channel 0 vector
Vicg:        equ    $FFF6         ;ICG vector
Vlvd:        equ    $FFF8         ;low voltage detect vector
Virq:        equ    $FFFA         ;IRQ pin vector
Vswi:        equ    $FFFC         ;SWI vector
Vreset:      equ    $FFFE         ;reset vector
```

B.4 Bits Defined in Two Ways

Bit names in the equate files for HCS08 MCUs need to be defined in two separate ways:

- With their bit number (0–7)
- A bit-position mask which is used in instructions such as AND, ORA, BIT, etc.

In the equate file, the bit name is first equated to its bit number (0–7), and then its bit position mask is equated to the bit name with a prefix of lowercase m, as in the next example.

```
SCI1S1:      equ    $1C          ;SCI1 status register 1
SCI2S1:      equ    $24          ;SCI2 status register 1
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
TDRE:        equ    7            ;(bit #7) Tx data register empty
TC:          equ    6            ;(bit #6) transmit complete
RDRF:        equ    5            ;(bit #5) Rx data register full
IDLE:        equ    4            ;(bit #4) idle line detected
OR:          equ    3            ;(bit #3) Rx over run
NF:          equ    2            ;(bit #2) Rx noise flag
FE:          equ    1            ;(bit #1) Rx framing error
PF:          equ    0            ;(bit #0) Rx parity failed
; bit position masks
mTDRE:       equ    %10000000    ;transmit data register empty
mTC:         equ    %01000000    ;transmit complete
mRDRF:       equ    %00100000    ;receive data register full
mIDLE:       equ    %00010000    ;idle line detected
mOR:         equ    %00001000    ;receiver over run
mNF:         equ    %00000100    ;receiver noise flag
mFE:         equ    %00000010    ;receiver framing error
mPF:         equ    %00000001    ;received parity failed
```

The next example shows the bit number variation of a bit definition. The operand field of the BRCLR instruction includes three items separated by commas. RDRF is converted to the number 5 which tells the assembler to use the bit-5 variation of the BRCLR instruction (opcode = $0B). The next item, SCI1S1, tells the assembler the operand to be tested is located at the direct addressing mode address $001C (just 1C in the object code). The last item, waitRDRF, tells the assembler to branch back to the same BRCLR instruction if the RDRF status bit is found to be still clear (0).

```
  450 120A 0B 1C FD  waitRDRF:    brclr  RDRF,SCI1S1,waitRDRF ;loop till RDRF set
```

The next example shows an expression combining the bit masks for the OR, NF, FE, and PF status bits. In this example, the bit names are used with a preceding m to get the bit position mask rather than the bit number. A simple addition operator (+) combines the bit masks. Although a logical OR might have been more correct in this case, not all assemblers use the same character to indicate the logical OR operation so the + is more portable among assemblers. The plus operator can be used in this case because the individual bit masks do not have any overlapping logic 1 bits.

```
                     mOR:         equ    %00001000     ;receiver over run
                     mNF:         equ    %00000100     ;receiver noise flag
                     mFE:         equ    %00000010     ;receiver framing error
                     mPF:         equ    %00000001     ;received parity failed
   “   “   “  “       “            “      “             “
  413                ; BIT example to check several error flags in SCI status reg
  414 11F1 B6 1C                  lda    SCI1S1        ;read SCI status register
  415 11F3 A5 0F                  bit    #(mOR+mNF+mFE+mPF) ;mask of all error flags
  416 11F5 26 00                  bne    sciError      ;branch if any flags set
  417                ; A still contains undisturbed status register
```

The 0F in the object code field of line 415 shows the assembler evaluated (mOR+mNF+mFE+mFF) to $0F. The A5 in the object code field of the same line is the opcode for the immediate addressing mode variation of the BIT instruction.

B.5 Complete Equate File for MC9S08GB60

The following listing is a complete equate file for the MC9S08GB60 MCU and is a complete example of an equate file for an HCS08 MCU. Each derivative in the HCS08 Family has a similar equate file posted on the Motorola Web site for free downloading.

```
;*******************************************************************************************
;* Title:  9S08GB60_v1.EQU                        (c) MOTOROLA Inc. 2003 All rights reserved.
;*******************************************************************************************
;* Author: Jim Sibigtroth - Motorola TSPG
;*
;* Description: Register and bit name definitions for 9S08GB60
;*
;* Documentation: 9S08GB60 family Data Sheet for register and bit explanations
;* HCS08 Family Reference Manual (HCS08RM1/D) appendix B for explanation of equate files
;*
;* Include Files: none
```

```
;*
;* Assembler:  Metrowerks Code Warrior 3.0 (pre-release)
;*             or P&E Microcomputer Systems - CASMS08 (beta v4.02)
;*
;* Revision History: not yet released
;* Rev #     Date      Who     Comments
;* -----  -----------  ------  --------------------------------------------
;*  1.2    24-Apr-03   J-Sib   correct minor typos in comments
;*  1.1    21-Apr-03   J-Sib   comments and modify for CW 3.0 project
;*  1.0    15-Apr-03   J-Sib   Release version for 9S09GB60
;********************************************************************************************

;****  Memory Map and Interrupt Vectors  ****************************************************
;*
RomStart:    equ    $1080         ;start of 60K flash
HighRegs:    equ    $1800         ;start of high page registers
Rom1Start:   equ    $182C         ;start of flash after high regs
RomLast:     equ    $FFFF         ;last flash location
RamStart:    equ    $0080         ;start of 4096 byte RAM
RamLast:     equ    $107F         ;last RAM location
;
Vrti:        equ    $FFCC         ;RTI (periodic interrupt) vector
Viic:        equ    $FFCE         ;IIC vector
Vatd:        equ    $FFD0         ;analog to digital conversion vector
Vkeyboard:   equ    $FFD2         ;keyboard vector
Vsci2tx:     equ    $FFD4         ;SCI2 transmit vector
Vsci2rx:     equ    $FFD6         ;SCI2 receive vector
Vsci2err:    equ    $FFD8         ;SCI2 error vector
Vsci1tx:     equ    $FFDA         ;SCI1 transmit vector
Vsci1rx:     equ    $FFDC         ;SCI1 receive vector
Vsci1err:    equ    $FFDE         ;SCI1 error vector
Vspi:        equ    $FFE0         ;SPI vector
Vtpm2ovf:    equ    $FFE2         ;TPM2 overflow vector
Vtpm2ch4:    equ    $FFE4         ;TPM2 channel 4 vector
Vtpm2ch3:    equ    $FFE6         ;TPM2 channel 3 vector
Vtpm2ch2:    equ    $FFE8         ;TPM2 channel 2 vector
Vtpm2ch1:    equ    $FFEA         ;TPM2 channel 1 vector
Vtpm2ch0:    equ    $FFEC         ;TPM2 channel 0 vector
Vtpm1ovf:    equ    $FFEE         ;TPM1 overflow vector
Vtpm1ch2:    equ    $FFF0         ;TPM1 channel 2 vector
Vtpm1ch1:    equ    $FFF2         ;TPM1 channel 1 vector
Vtpm1ch0:    equ    $FFF4         ;TPM1 channel 0 vector
Vicg:        equ    $FFF6         ;ICG vector
Vlvd:        equ    $FFF8         ;low voltage detect vector
Virq:        equ    $FFFA         ;IRQ pin vector
Vswi:        equ    $FFFC         ;SWI vector
Vreset:      equ    $FFFE         ;reset vector

;****  Input/Output (I/O) Ports  ************************************************************
;*
PTAD:        equ    $00           ;I/O port A data register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTAD7:       equ    7             ;bit #7
PTAD6:       equ    6             ;bit #6
PTAD5:       equ    5             ;bit #5
```

```
PTAD4:       equ    4             ;bit #4
PTAD3:       equ    3             ;bit #3
PTAD2:       equ    2             ;bit #2
PTAD1:       equ    1             ;bit #1
PTAD0:       equ    0             ;bit #0
; bit position masks
mPTAD7:      equ    %10000000     ;port A bit 7
mPTAD6:      equ    %01000000     ;port A bit 6
mPTAD5:      equ    %00100000     ;port A bit 5
mPTAD4:      equ    %00010000     ;port A bit 4
mPTAD3:      equ    %00001000     ;port A bit 3
mPTAD2:      equ    %00000100     ;port A bit 2
mPTAD1:      equ    %00000010     ;port A bit 1
mPTAD0:      equ    %00000001     ;port A bit 0

PTAPE:       equ    $01           ;I/O port A pullup enable controls
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTAPE7:      equ    7             ;bit #7
PTAPE6:      equ    6             ;bit #6
PTAPE5:      equ    5             ;bit #5
PTAPE4:      equ    4             ;bit #4
PTAPE3:      equ    3             ;bit #3
PTAPE2:      equ    2             ;bit #2
PTAPE1:      equ    1             ;bit #1
PTAPE0:      equ    0             ;bit #0
; bit position masks
mPTAPE7:     equ    %10000000     ;port A bit 7
mPTAPE6:     equ    %01000000     ;port A bit 6
mPTAPE5:     equ    %00100000     ;port A bit 5
mPTAPE4:     equ    %00010000     ;port A bit 4
mPTAPE3:     equ    %00001000     ;port A bit 3
mPTAPE2:     equ    %00000100     ;port A bit 2
mPTAPE1:     equ    %00000010     ;port A bit 1
mPTAPE0:     equ    %00000001     ;port A bit 0

PTASE:       equ    $02           ;I/O port A slew rate control register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTASE7:      equ    7             ;bit #7
PTASE6:      equ    6             ;bit #6
PTASE5:      equ    5             ;bit #5
PTASE4:      equ    4             ;bit #4
PTASE3:      equ    3             ;bit #3
PTASE2:      equ    2             ;bit #2
PTASE1:      equ    1             ;bit #1
PTASE0:      equ    0             ;bit #0
; bit position masks
mPTASE7:     equ    %10000000     ;port A bit 7
mPTASE6:     equ    %01000000     ;port A bit 6
mPTASE5:     equ    %00100000     ;port A bit 5
mPTASE4:     equ    %00010000     ;port A bit 4
mPTASE3:     equ    %00001000     ;port A bit 3
mPTASE2:     equ    %00000100     ;port A bit 2
mPTASE1:     equ    %00000010     ;port A bit 1
mPTASE0:     equ    %00000001     ;port A bit 0
```

```
PTADD:       equ    $03           ;I/O port A data direction register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTADD7:      equ    7             ;bit #7
PTADD6:      equ    6             ;bit #6
PTADD5:      equ    5             ;bit #5
PTADD4:      equ    4             ;bit #4
PTADD3:      equ    3             ;bit #3
PTADD2:      equ    2             ;bit #2
PTADD1:      equ    1             ;bit #1
PTADD0:      equ    0             ;bit #0
; bit position masks
mPTADD7:     equ    %10000000     ;port A bit 7
mPTADD6:     equ    %01000000     ;port A bit 6
mPTADD5:     equ    %00100000     ;port A bit 5
mPTADD4:     equ    %00010000     ;port A bit 4
mPTADD3:     equ    %00001000     ;port A bit 3
mPTADD2:     equ    %00000100     ;port A bit 2
mPTADD1:     equ    %00000010     ;port A bit 1
mPTADD0:     equ    %00000001     ;port A bit 0

PTBD:        equ    $04           ;I/O port B data register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTBD7:       equ    7             ;bit #7
PTBD6:       equ    6             ;bit #6
PTBD5:       equ    5             ;bit #5
PTBD4:       equ    4             ;bit #4
PTBD3:       equ    3             ;bit #3
PTBD2:       equ    2             ;bit #2
PTBD1:       equ    1             ;bit #1
PTBD0:       equ    0             ;bit #0
; bit position masks
mPTBD7:      equ    %10000000     ;port B bit 7
mPTBD6:      equ    %01000000     ;port B bit 6
mPTBD5:      equ    %00100000     ;port B bit 5
mPTBD4:      equ    %00010000     ;port B bit 4
mPTBD3:      equ    %00001000     ;port B bit 3
mPTBD2:      equ    %00000100     ;port B bit 2
mPTBD1:      equ    %00000010     ;port B bit 1
mPTBD0:      equ    %00000001     ;port B bit 0

PTBPE:       equ    $05           ;I/O port B pullup enable controls
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTBPE7:      equ    7             ;bit #7
PTBPE6:      equ    6             ;bit #6
PTBPE5:      equ    5             ;bit #5
PTBPE4:      equ    4             ;bit #4
PTBPE3:      equ    3             ;bit #3
PTBPE2:      equ    2             ;bit #2
PTBPE1:      equ    1             ;bit #1
PTBPE0:      equ    0             ;bit #0
; bit position masks
mPTBPE7:     equ    %10000000     ;port B bit 7
mPTBPE6:     equ    %01000000     ;port B bit 6
mPTBPE5:     equ    %00100000     ;port B bit 5
mPTBPE4:     equ    %00010000     ;port B bit 4
```

```
mPTBPE3:     equ    %00001000     ;port B bit 3
mPTBPE2:     equ    %00000100     ;port B bit 2
mPTBPE1:     equ    %00000010     ;port B bit 1
mPTBPE0:     equ    %00000001     ;port B bit 0

PTBSE:       equ    $06           ;I/O port B slew rate control register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTBSE7:      equ    7             ;bit #7
PTBSE6:      equ    6             ;bit #6
PTBSE5:      equ    5             ;bit #5
PTBSE4:      equ    4             ;bit #4
PTBSE3:      equ    3             ;bit #3
PTBSE2:      equ    2             ;bit #2
PTBSE1:      equ    1             ;bit #1
PTBSE0:      equ    0             ;bit #0
; bit position masks
mPTBSE7:     equ    %10000000     ;port B bit 7
mPTBSE6:     equ    %01000000     ;port B bit 6
mPTBSE5:     equ    %00100000     ;port B bit 5
mPTBSE4:     equ    %00010000     ;port B bit 4
mPTBSE3:     equ    %00001000     ;port B bit 3
mPTBSE2:     equ    %00000100     ;port B bit 2
mPTBSE1:     equ    %00000010     ;port B bit 1
mPTBSE0:     equ    %00000001     ;port B bit 0

PTBDD:       equ    $07           ;I/O port B data direction register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTBDD7:      equ    7             ;bit #7
PTBDD6:      equ    6             ;bit #6
PTBDD5:      equ    5             ;bit #5
PTBDD4:      equ    4             ;bit #4
PTBDD3:      equ    3             ;bit #3
PTBDD2:      equ    2             ;bit #2
PTBDD1:      equ    1             ;bit #1
PTBDD0:      equ    0             ;bit #0
; bit position masks
mPTBDD7:     equ    %10000000     ;port B bit 7
mPTBDD6:     equ    %01000000     ;port B bit 6
mPTBDD5:     equ    %00100000     ;port B bit 5
mPTBDD4:     equ    %00010000     ;port B bit 4
mPTBDD3:     equ    %00001000     ;port B bit 3
mPTBDD2:     equ    %00000100     ;port B bit 2
mPTBDD1:     equ    %00000010     ;port B bit 1
mPTBDD0:     equ    %00000001     ;port B bit 0

PTCD:        equ    $08           ;I/O port C data register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTCD7:       equ    7             ;bit #7
PTCD6:       equ    6             ;bit #6
PTCD5:       equ    5             ;bit #5
PTCD4:       equ    4             ;bit #4
PTCD3:       equ    3             ;bit #3
PTCD2:       equ    2             ;bit #2
PTCD1:       equ    1             ;bit #1
PTCD0:       equ    0             ;bit #0
```

```
; bit position masks
mPTCD7:      equ    %10000000     ;port C bit 7
mPTCD6:      equ    %01000000     ;port C bit 6
mPTCD5:      equ    %00100000     ;port C bit 5
mPTCD4:      equ    %00010000     ;port C bit 4
mPTCD3:      equ    %00001000     ;port C bit 3
mPTCD2:      equ    %00000100     ;port C bit 2
mPTCD1:      equ    %00000010     ;port C bit 1
mPTCD0:      equ    %00000001     ;port C bit 0

PTCPE:       equ    $09           ;I/O port C pullup enable controls
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTCPE7:      equ    7             ;bit #7
PTCPE6:      equ    6             ;bit #6
PTCPE5:      equ    5             ;bit #5
PTCPE4:      equ    4             ;bit #4
PTCPE3:      equ    3             ;bit #3
PTCPE2:      equ    2             ;bit #2
PTCPE1:      equ    1             ;bit #1
PTCPE0:      equ    0             ;bit #0
; bit position masks
mPTCPE7:     equ    %10000000     ;port C bit 7
mPTCPE6:     equ    %01000000     ;port C bit 6
mPTCPE5:     equ    %00100000     ;port C bit 5
mPTCPE4:     equ    %00010000     ;port C bit 4
mPTCPE3:     equ    %00001000     ;port C bit 3
mPTCPE2:     equ    %00000100     ;port C bit 2
mPTCPE1:     equ    %00000010     ;port C bit 1
mPTCPE0:     equ    %00000001     ;port C bit 0

PTCSE:       equ    $0A           ;I/O port C slew rate control register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTCSE7:      equ    7             ;bit #7
PTCSE6:      equ    6             ;bit #6
PTCSE5:      equ    5             ;bit #5
PTCSE4:      equ    4             ;bit #4
PTCSE3:      equ    3             ;bit #3
PTCSE2:      equ    2             ;bit #2
PTCSE1:      equ    1             ;bit #1
PTCSE0:      equ    0             ;bit #0
; bit position masks
mPTCSE7:     equ    %10000000     ;port C bit 7
mPTCSE6:     equ    %01000000     ;port C bit 6
mPTCSE5:     equ    %00100000     ;port C bit 5
mPTCSE4:     equ    %00010000     ;port C bit 4
mPTCSE3:     equ    %00001000     ;port C bit 3
mPTCSE2:     equ    %00000100     ;port C bit 2
mPTCSE1:     equ    %00000010     ;port C bit 1
mPTCSE0:     equ    %00000001     ;port C bit 0

PTCDD:       equ    $0B           ;I/O port C data direction register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTCDD7:      equ    7             ;bit #7
PTCDD6:      equ    6             ;bit #6
PTCDD5:      equ    5             ;bit #5
```

```
PTCDD4:      equ    4             ;bit #4
PTCDD3:      equ    3             ;bit #3
PTCDD2:      equ    2             ;bit #2
PTCDD1:      equ    1             ;bit #1
PTCDD0:      equ    0             ;bit #0
; bit position masks
mPTCDD7:     equ    %10000000     ;port C bit 7
mPTCDD6:     equ    %01000000     ;port C bit 6
mPTCDD5:     equ    %00100000     ;port C bit 5
mPTCDD4:     equ    %00010000     ;port C bit 4
mPTCDD3:     equ    %00001000     ;port C bit 3
mPTCDD2:     equ    %00000100     ;port C bit 2
mPTCDD1:     equ    %00000010     ;port C bit 1
mPTCDD0:     equ    %00000001     ;port C bit 0

PTDD:        equ    $0C           ;I/O port D data register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTDD7:       equ    7             ;bit #7
PTDD6:       equ    6             ;bit #6
PTDD5:       equ    5             ;bit #5
PTDD4:       equ    4             ;bit #4
PTDD3:       equ    3             ;bit #3
PTDD2:       equ    2             ;bit #2
PTDD1:       equ    1             ;bit #1
PTDD0:       equ    0             ;bit #0
; bit position masks
mPTDD7:      equ    %10000000     ;port D bit 7
mPTDD6:      equ    %01000000     ;port D bit 6
mPTDD5:      equ    %00100000     ;port D bit 5
mPTDD4:      equ    %00010000     ;port D bit 4
mPTDD3:      equ    %00001000     ;port D bit 3
mPTDD2:      equ    %00000100     ;port D bit 2
mPTDD1:      equ    %00000010     ;port D bit 1
mPTDD0:      equ    %00000001     ;port D bit 0

PTDPE:       equ    $0D           ;I/O port D pullup enable controls
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTDPE7:      equ    7             ;bit #7
PTDPE6:      equ    6             ;bit #6
PTDPE5:      equ    5             ;bit #5
PTDPE4:      equ    4             ;bit #4
PTDPE3:      equ    3             ;bit #3
PTDPE2:      equ    2             ;bit #2
PTDPE1:      equ    1             ;bit #1
PTDPE0:      equ    0             ;bit #0
; bit position masks
mPTDPE7:     equ    %10000000     ;port D bit 7
mPTDPE6:     equ    %01000000     ;port D bit 6
mPTDPE5:     equ    %00100000     ;port D bit 5
mPTDPE4:     equ    %00010000     ;port D bit 4
mPTDPE3:     equ    %00001000     ;port D bit 3
mPTDPE2:     equ    %00000100     ;port D bit 2
mPTDPE1:     equ    %00000010     ;port D bit 1
mPTDPE0:     equ    %00000001     ;port D bit 0
```

```
PTDSE:       equ    $0E           ;I/O port D slew rate control register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTDSE7:      equ    7             ;bit #7
PTDSE6:      equ    6             ;bit #6
PTDSE5:      equ    5             ;bit #5
PTDSE4:      equ    4             ;bit #4
PTDSE3:      equ    3             ;bit #3
PTDSE2:      equ    2             ;bit #2
PTDSE1:      equ    1             ;bit #1
PTDSE0:      equ    0             ;bit #0
; bit position masks
mPTDSE7:     equ    %10000000     ;port D bit 7
mPTDSE6:     equ    %01000000     ;port D bit 6
mPTDSE5:     equ    %00100000     ;port D bit 5
mPTDSE4:     equ    %00010000     ;port D bit 4
mPTDSE3:     equ    %00001000     ;port D bit 3
mPTDSE2:     equ    %00000100     ;port D bit 2
mPTDSE1:     equ    %00000010     ;port D bit 1
mPTDSE0:     equ    %00000001     ;port D bit 0

PTDDD:       equ    $0F           ;I/O port D data direction register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTDDD7:      equ    7             ;bit #7
PTDDD6:      equ    6             ;bit #6
PTDDD5:      equ    5             ;bit #5
PTDDD4:      equ    4             ;bit #4
PTDDD3:      equ    3             ;bit #3
PTDDD2:      equ    2             ;bit #2
PTDDD1:      equ    1             ;bit #1
PTDDD0:      equ    0             ;bit #0
; bit position masks
mPTDDD7:     equ    %10000000     ;port D bit 7
mPTDDD6:     equ    %01000000     ;port D bit 6
mPTDDD5:     equ    %00100000     ;port D bit 5
mPTDDD4:     equ    %00010000     ;port D bit 4
mPTDDD3:     equ    %00001000     ;port D bit 3
mPTDDD2:     equ    %00000100     ;port D bit 2
mPTDDD1:     equ    %00000010     ;port D bit 1
mPTDDD0:     equ    %00000001     ;port D bit 0

PTED:        equ    $10           ;I/O port E data register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTED7:       equ    7             ;bit #7
PTED6:       equ    6             ;bit #6
PTED5:       equ    5             ;bit #5
PTED4:       equ    4             ;bit #4
PTED3:       equ    3             ;bit #3
PTED2:       equ    2             ;bit #2
PTED1:       equ    1             ;bit #1
PTED0:       equ    0             ;bit #0
; bit position masks
mPTED7:      equ    %10000000     ;port E bit 7
mPTED6:      equ    %01000000     ;port E bit 6
mPTED5:      equ    %00100000     ;port E bit 5
mPTED4:      equ    %00010000     ;port E bit 4
```

```
mPTED3:      equ    %00001000     ;port E bit 3
mPTED2:      equ    %00000100     ;port E bit 2
mPTED1:      equ    %00000010     ;port E bit 1
mPTED0:      equ    %00000001     ;port E bit 0

PTEPE:       equ    $11           ;I/O port E pullup enable controls
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTEPE7:      equ    7             ;bit #7
PTEPE6:      equ    6             ;bit #6
PTEPE5:      equ    5             ;bit #5
PTEPE4:      equ    4             ;bit #4
PTEPE3:      equ    3             ;bit #3
PTEPE2:      equ    2             ;bit #2
PTEPE1:      equ    1             ;bit #1
PTEPE0:      equ    0             ;bit #0
; bit position masks
mPTEPE7:     equ    %10000000     ;port E bit 7
mPTEPE6:     equ    %01000000     ;port E bit 6
mPTEPE5:     equ    %00100000     ;port E bit 5
mPTEPE4:     equ    %00010000     ;port E bit 4
mPTEPE3:     equ    %00001000     ;port E bit 3
mPTEPE2:     equ    %00000100     ;port E bit 2
mPTEPE1:     equ    %00000010     ;port E bit 1
mPTEPE0:     equ    %00000001     ;port E bit 0

PTESE:       equ    $12           ;I/O port E slew rate control register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTESE7:      equ    7             ;bit #7
PTESE6:      equ    6             ;bit #6
PTESE5:      equ    5             ;bit #5
PTESE4:      equ    4             ;bit #4
PTESE3:      equ    3             ;bit #3
PTESE2:      equ    2             ;bit #2
PTESE1:      equ    1             ;bit #1
PTESE0:      equ    0             ;bit #0
; bit position masks
mPTESE7:     equ    %10000000     ;port E bit 7
mPTESE6:     equ    %01000000     ;port E bit 6
mPTESE5:     equ    %00100000     ;port E bit 5
mPTESE4:     equ    %00010000     ;port E bit 4
mPTESE3:     equ    %00001000     ;port E bit 3
mPTESE2:     equ    %00000100     ;port E bit 2
mPTESE1:     equ    %00000010     ;port E bit 1
mPTESE0:     equ    %00000001     ;port E bit 0

PTEDD:       equ    $13           ;I/O port E data direction register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTEDD7:      equ    7             ;bit #7
PTEDD6:      equ    6             ;bit #6
PTEDD5:      equ    5             ;bit #5
PTEDD4:      equ    4             ;bit #4
PTEDD3:      equ    3             ;bit #3
PTEDD2:      equ    2             ;bit #2
PTEDD1:      equ    1             ;bit #1
PTEDD0:      equ    0             ;bit #0
```

```
; bit position masks
mPTEDD7:     equ    %10000000     ;port E bit 7
mPTEDD6:     equ    %01000000     ;port E bit 6
mPTEDD5:     equ    %00100000     ;port E bit 5
mPTEDD4:     equ    %00010000     ;port E bit 4
mPTEDD3:     equ    %00001000     ;port E bit 3
mPTEDD2:     equ    %00000100     ;port E bit 2
mPTEDD1:     equ    %00000010     ;port E bit 1
mPTEDD0:     equ    %00000001     ;port E bit 0

PTFD:        equ    $40           ;I/O port F data register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTFD7:       equ    7             ;bit #7
PTFD6:       equ    6             ;bit #6
PTFD5:       equ    5             ;bit #5
PTFD4:       equ    4             ;bit #4
PTFD3:       equ    3             ;bit #3
PTFD2:       equ    2             ;bit #2
PTFD1:       equ    1             ;bit #1
PTFD0:       equ    0             ;bit #0
; bit position masks
mPTFD7:      equ    %10000000     ;port F bit 7
mPTFD6:      equ    %01000000     ;port F bit 6
mPTFD5:      equ    %00100000     ;port F bit 5
mPTFD4:      equ    %00010000     ;port F bit 4
mPTFD3:      equ    %00001000     ;port F bit 3
mPTFD2:      equ    %00000100     ;port F bit 2
mPTFD1:      equ    %00000010     ;port F bit 1
mPTFD0:      equ    %00000001     ;port F bit 0

PTFPE:       equ    $41           ;I/O port F pullup enable controls
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTFPE7:      equ    7             ;bit #7
PTFPE6:      equ    6             ;bit #6
PTFPE5:      equ    5             ;bit #5
PTFPE4:      equ    4             ;bit #4
PTFPE3:      equ    3             ;bit #3
PTFPE2:      equ    2             ;bit #2
PTFPE1:      equ    1             ;bit #1
PTFPE0:      equ    0             ;bit #0
; bit position masks
mPTFPE7:     equ    %10000000     ;port F bit 7
mPTFPE6:     equ    %01000000     ;port F bit 6
mPTFPE5:     equ    %00100000     ;port F bit 5
mPTFPE4:     equ    %00010000     ;port F bit 4
mPTFPE3:     equ    %00001000     ;port F bit 3
mPTFPE2:     equ    %00000100     ;port F bit 2
mPTFPE1:     equ    %00000010     ;port F bit 1
mPTFPE0:     equ    %00000001     ;port F bit 0

PTFSE:       equ    $42           ;I/O port F slew rate control register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTFSE7:      equ    7             ;bit #7
PTFSE6:      equ    6             ;bit #6
PTFSE5:      equ    5             ;bit #5
```

```
PTFSE4:      equ    4             ;bit #4
PTFSE3:      equ    3             ;bit #3
PTFSE2:      equ    2             ;bit #2
PTFSE1:      equ    1             ;bit #1
PTFSE0:      equ    0             ;bit #0
; bit position masks
mPTFSE7:     equ    %10000000     ;port F bit 7
mPTFSE6:     equ    %01000000     ;port F bit 6
mPTFSE5:     equ    %00100000     ;port F bit 5
mPTFSE4:     equ    %00010000     ;port F bit 4
mPTFSE3:     equ    %00001000     ;port F bit 3
mPTFSE2:     equ    %00000100     ;port F bit 2
mPTFSE1:     equ    %00000010     ;port F bit 1
mPTFSE0:     equ    %00000001     ;port F bit 0


PTFDD:       equ    $43           ;I/O port F data direction register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTFDD7:      equ    7             ;bit #7
PTFDD6:      equ    6             ;bit #6
PTFDD5:      equ    5             ;bit #5
PTFDD4:      equ    4             ;bit #4
PTFDD3:      equ    3             ;bit #3
PTFDD2:      equ    2             ;bit #2
PTFDD1:      equ    1             ;bit #1
PTFDD0:      equ    0             ;bit #0
; bit position masks
mPTFDD7:     equ    %10000000     ;port F bit 7
mPTFDD6:     equ    %01000000     ;port F bit 6
mPTFDD5:     equ    %00100000     ;port F bit 5
mPTFDD4:     equ    %00010000     ;port F bit 4
mPTFDD3:     equ    %00001000     ;port F bit 3
mPTFDD2:     equ    %00000100     ;port F bit 2
mPTFDD1:     equ    %00000010     ;port F bit 1
mPTFDD0:     equ    %00000001     ;port F bit 0

PTGD:        equ    $44           ;I/O port G data register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTGD7:       equ    7             ;bit #7
PTGD6:       equ    6             ;bit #6
PTGD5:       equ    5             ;bit #5
PTGD4:       equ    4             ;bit #4
PTGD3:       equ    3             ;bit #3
PTGD2:       equ    2             ;bit #2
PTGD1:       equ    1             ;bit #1
PTGD0:       equ    0             ;bit #0
; bit position masks
mPTGD7:      equ    %10000000     ;port G bit 7
mPTGD6:      equ    %01000000     ;port G bit 6
mPTGD5:      equ    %00100000     ;port G bit 5
mPTGD4:      equ    %00010000     ;port G bit 4
mPTGD3:      equ    %00001000     ;port G bit 3
mPTGD2:      equ    %00000100     ;port G bit 2
mPTGD1:      equ    %00000010     ;port G bit 1
mPTGD0:      equ    %00000001     ;port G bit 0
```

```
PTGPE:       equ    $45           ;I/O port G pullup enable controls
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTGPE7:      equ    7             ;bit #7
PTGPE6:      equ    6             ;bit #6
PTGPE5:      equ    5             ;bit #5
PTGPE4:      equ    4             ;bit #4
PTGPE3:      equ    3             ;bit #3
PTGPE2:      equ    2             ;bit #2
PTGPE1:      equ    1             ;bit #1
PTGPE0:      equ    0             ;bit #0
; bit position masks
mPTGPE7:     equ    %10000000     ;port G bit 7
mPTGPE6:     equ    %01000000     ;port G bit 6
mPTGPE5:     equ    %00100000     ;port G bit 5
mPTGPE4:     equ    %00010000     ;port G bit 4
mPTGPE3:     equ    %00001000     ;port G bit 3
mPTGPE2:     equ    %00000100     ;port G bit 2
mPTGPE1:     equ    %00000010     ;port G bit 1
mPTGPE0:     equ    %00000001     ;port G bit 0

PTGSE:       equ    $46           ;I/O port G slew rate control register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTGSE7:      equ    7             ;bit #7
PTGSE6:      equ    6             ;bit #6
PTGSE5:      equ    5             ;bit #5
PTGSE4:      equ    4             ;bit #4
PTGSE3:      equ    3             ;bit #3
PTGSE2:      equ    2             ;bit #2
PTGSE1:      equ    1             ;bit #1
PTGSE0:      equ    0             ;bit #0
; bit position masks
mPTGSE7:     equ    %10000000     ;port G bit 7
mPTGSE6:     equ    %01000000     ;port G bit 6
mPTGSE5:     equ    %00100000     ;port G bit 5
mPTGSE4:     equ    %00010000     ;port G bit 4
mPTGSE3:     equ    %00001000     ;port G bit 3
mPTGSE2:     equ    %00000100     ;port G bit 2
mPTGSE1:     equ    %00000010     ;port G bit 1
mPTGSE0:     equ    %00000001     ;port G bit 0

PTGDD:       equ    $47           ;I/O port G data direction register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTGDD7:      equ    7             ;bit #7
PTGDD6:      equ    6             ;bit #6
PTGDD5:      equ    5             ;bit #5
PTGDD4:      equ    4             ;bit #4
PTGDD3:      equ    3             ;bit #3
PTGDD2:      equ    2             ;bit #2
PTGDD1:      equ    1             ;bit #1
PTGDD0:      equ    0             ;bit #0
; bit position masks
mPTGDD7:     equ    %10000000     ;port G bit 7
mPTGDD6:     equ    %01000000     ;port G bit 6
mPTGDD5:     equ    %00100000     ;port G bit 5
```

```
mPTGDD4:     equ    %00010000     ;port G bit 4
mPTGDD3:     equ    %00001000     ;port G bit 3
mPTGDD2:     equ    %00000100     ;port G bit 2
mPTGDD1:     equ    %00000010     ;port G bit 1
mPTGDD0:     equ    %00000001     ;port G bit 0

;****  Interrupt Request Module (IRQ)  ******************************************************
;*
IRQSC:       equ    $14           ;IRQ status and control register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
IRQEDG:      equ    5             ;(bit #5) IRQ pin edge sensitivity
IRQPE:       equ    4             ;(bit #4) IRQ pin enable (PTB5)
IRQF:        equ    3             ;(bit #3) IRQ flag
IRQACK:      equ    2             ;(bit #2) acknowledge IRQ flag
IRQIE:       equ    1             ;(bit #1) IRQ pin interrupt enable
IRQMOD:      equ    0             ;(bit #0) IRQ mode
; bit position masks
mIRQEDG:     equ    %00100000     ;IRQ pin edge sensitivity
mIRQPE:      equ    %00010000     ;IRQ pin enable (PTB5)
mIRQF:       equ    %00001000     ;IRQ flag
mIRQACK:     equ    %00000100     ;acknowledge IRQ flag
mIRQIE:      equ    %00000010     ;IRQ pin interrupt enable
mIRQMOD:     equ    %00000001     ;IRQ mode

;****  Keyboard Interrupt Module (KBI)  *****************************************************
;*
KBISC:       equ    $16           ;KBI status and control register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
KBEDG7:      equ    7             ;rise-hi/fall-low for KBIP7 pin
KBEDG6:      equ    6             ;rise-hi/fall-low for KBIP6 pin
KBEDG5:      equ    5             ;rise-hi/fall-low for KBIP5 pin
KBEDG4:      equ    4             ;rise-hi/fall-low for KBIP4 pin
KBF:         equ    3             ;KBI flag
KBACK:       equ    2             ;acknowledge
KBIE:        equ    1             ;KBI interrupt enable
KBIMOD:      equ    0             ;KBI mode select
; bit position masks
mKBEDG7:     equ    %10000000     ;rise-hi/fall-low for KBIP7 pin
mKBEDG6:     equ    %01000000     ;rise-hi/fall-low for KBIP6 pin
mKBEDG5:     equ    %00100000     ;rise-hi/fall-low for KBIP5 pin
mKBEDG4:     equ    %00010000     ;rise-hi/fall-low for KBIP4 pin
mKBF:        equ    %00001000     ;KBI flag
mKBACK:      equ    %00000100     ;acknowledge
mKBIE:       equ    %00000010     ;KBI interrupt enable
mKBIMOD:     equ    %00000001     ;KBI mode select

KBIPE:       equ    $17           ;KBI pin enable controls
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
KBIPE7:      equ    7             ;bit #7
KBIPE6:      equ    6             ;bit #6
KBIPE5:      equ    5             ;bit #5
KBIPE4:      equ    4             ;bit #4
KBIPE3:      equ    3             ;bit #3
KBIPE2:      equ    2             ;bit #2
KBIPE1:      equ    1             ;bit #1
```

```
KBIPE0:      equ    0             ;bit #0
; bit position masks
mKBIPE7:     equ    %10000000     ;port A bit 7
mKBIPE6:     equ    %01000000     ;port A bit 6
mKBIPE5:     equ    %00100000     ;port A bit 5
mKBIPE4:     equ    %00010000     ;port A bit 4
mKBIPE3:     equ    %00001000     ;port A bit 3
mKBIPE2:     equ    %00000100     ;port A bit 2
mKBIPE1:     equ    %00000010     ;port A bit 1
mKBIPE0:     equ    %00000001     ;port A bit 0

;****  Serial Communications Interface 1&2 (SCI1 & SCI2)  ***********************************
;*
SCI1BDH:     equ    $18           ;SCI1 baud rate register (high)
SCI2BDH:     equ    $20           ;SCI2 baud rate register (high)
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
SBR12:       equ    4             ;(bit #4) baud divide (high)
SBR11:       equ    3             ;(bit #3)  "
SBR10:       equ    2             ;(bit #2)  "
SBR9:        equ    1             ;(bit #1)  "
SBR8:        equ    0             ;(bit #0)  "
; bit position masks
mSBR12:      equ    %00010000     ;high bits of baud rate divider
mSBR11:      equ    %00001000     ; "
mSBR10:      equ    %00000100     ; "
mSBR9:       equ    %00000010     ; "
mSBR8:       equ    %00000001     ; "

SCI1BDL:     equ    $19           ;SCI1 baud rate register (low byte)
SCI2BDL:     equ    $21           ;SCI2 baud rate register (low byte)
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
SBR7:        equ    7             ;(bit #7) baud divide (low)
SBR6:        equ    6             ;(bit #6)  "
SBR5:        equ    5             ;(bit #5)  "
SBR4:        equ    4             ;(bit #4)  "
SBR3:        equ    3             ;(bit #3)  "
SBR2:        equ    2             ;(bit #2)  "
SBR1:        equ    1             ;(bit #1)  "
SBR0:        equ    0             ;(bit #0)  "
; bit position masks
mSBR7:       equ    %10000000     ;low byte of baud rate divider
mSBR6:       equ    %01000000     ; "
mSBR5:       equ    %00100000     ; "
mSBR4:       equ    %00010000     ; "
mSBR3:       equ    %00001000     ; "
mSBR2:       equ    %00000100     ; "
mSBR1:       equ    %00000010     ; "
mSBR0:       equ    %00000001     ; "

SCI1C1:      equ    $1A           ;SCI1 control register 1
SCI2C1:      equ    $22           ;SCI2 control register 1
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
LOOPS:       equ    7             ;(bit #7) loopback mode
SCISWAI:     equ    6             ;(bit #6) SCI stop in wait
RSRC:        equ    5             ;(bit #5) receiver source
```

```
M:           equ    4             ;(bit #4) 9/8 bit data
WAKE:        equ    3             ;(bit #3) wake by addr mark/idle
ILT:         equ    2             ;(bit #2) idle line type; stop/start
PE:          equ    1             ;(bit #1) parity enable
PT:          equ    0             ;(bit #0) parity type
; bit position masks
mLOOPS:      equ    %10000000     ;loopback mode select
mSCISWAI:    equ    %01000000     ;SCI stops in wait mode
mRSRC:       equ    %00100000     ;receiver source
mM:          equ    %00010000     ;9/8 bit data
mWAKE:       equ    %00001000     ;wakeup by addr mark/idle
mILT:        equ    %00000100     ;idle line type; after stop/start
mPE:         equ    %00000010     ;parity enable
mPT:         equ    %00000001     ;parity type even/odd

SCI1C2:      equ    $1B           ;SCI1 control register 2
SCI2C2:      equ    $23           ;SCI2 control register 2
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
TIE:         equ    7             ;(bit #7) transmit interrupt enable
TCIE:        equ    6             ;(bit #6) TC interrupt enable
RIE:         equ    5             ;(bit #5) receive interrupt enable
ILIE:        equ    4             ;(bit #4) idle line interrupt enable
TE:          equ    3             ;(bit #3) transmitter enable
RE:          equ    2             ;(bit #2) receiver enable
RWU:         equ    1             ;(bit #1) receiver wakeup engage
SBK:         equ    0             ;(bit #0) send break
; bit position masks
mTIE:        equ    %10000000     ;transmit interrupt (TDRE) enable
mTCIE:       equ    %01000000     ;transmit complete interrupt enable
mRIE:        equ    %00100000     ;receive interrupt (RDRF) enable
mILIE:       equ    %00010000     ;idle line interrupt (ILIE) enable
mTE:         equ    %00001000     ;transmitter enable
mRE:         equ    %00000100     ;receiver enable
mRWU:        equ    %00000010     ;receiver wakeup engage
mSBK:        equ    %00000001     ;send break characters

SCI1S1:      equ    $1C           ;SCI1 status register 1
SCI2S1:      equ    $24           ;SCI2 status register 1
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
TDRE:        equ    7             ;(bit #7) Tx data register empty
TC:          equ    6             ;(bit #6) transmit complete
RDRF:        equ    5             ;(bit #5) Rx data register full
IDLE:        equ    4             ;(bit #4) idle line detected
OR:          equ    3             ;(bit #3) Rx over run
NF:          equ    2             ;(bit #2) Rx noise flag
FE:          equ    1             ;(bit #1) Rx framing error
PF:          equ    0             ;(bit #0) Rx parity failed
; bit position masks
mTDRE:       equ    %10000000     ;transmit data register empty
mTC:         equ    %01000000     ;transmit complete
mRDRF:       equ    %00100000     ;receive data register full
mIDLE:       equ    %00010000     ;idle line detected
mOR:         equ    %00001000     ;receiver over run
mNF:         equ    %00000100     ;receiver noise flag
mFE:         equ    %00000010     ;receiver framing error
```

```
mPF:         equ    %00000001     ;received parity failed

SCI1S2:      equ    $1D           ;SCI1 status register 2
SCI2S2:      equ    $25           ;SCI2 status register 2
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
RAF:         equ    0             ;(bit #0) Rx active flag
; bit position masks
mRAF:        equ    %00000001     ;receiver active flag

SCI1C3:      equ    $1E           ;SCI1 control register 3
SCI2C3:      equ    $26           ;SCI2 control register 3
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
R8:          equ    7             ;(bit #7) 9th Rx bit
T8:          equ    6             ;(bit #6) 9th Tx bit
TXDIR:       equ    5             ;(bit #5) TxD pin direction?
ORIE:        equ    3             ;(bit #3) Rx over run int. enable
NEIE:        equ    2             ;(bit #2) Rx noise flag int. enable
FEIE:        equ    1             ;(bit #1) Rx framing error int. enable
PEIE:        equ    0             ;(bit #0) Rx parity error int. enable
; bit position masks
mR8:         equ    %10000000     ;9th receive data bit
mT8:         equ    %01000000     ;9th transmit data bit
mTXDIR:      equ    %00100000     ;transmit pin direction?
mORIE:       equ    %00001000     ;receiver over run int. enable
mNEIE:       equ    %00000100     ;receiver noise flag int. enable
mFEIE:       equ    %00000010     ;receiver framing error int. enable
mPEIE:       equ    %00000001     ;received parity error int. enable

SCI1D:       equ    $1F           ;SCI1 data register (low byte)
SCI2D:       equ    $27           ;SCI2 data register (low byte)
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
; read-only Rx data buffer
R7:          equ    7             ;(bit #7) receive data bits
R6:          equ    6             ;(bit #6)  "
R5:          equ    5             ;(bit #5)  "
R4:          equ    4             ;(bit #4)  "
R3:          equ    3             ;(bit #3)  "
R2:          equ    2             ;(bit #2)  "
R1:          equ    1             ;(bit #1)  "
R0:          equ    0             ;(bit #0)  "
; write-only Tx data buffer
T7:          equ    7             ;(bit #7) transmit data bits
T6:          equ    6             ;(bit #6)  "
T5:          equ    5             ;(bit #5)  "
T4:          equ    4             ;(bit #4)  "
T3:          equ    3             ;(bit #3)  "
T2:          equ    2             ;(bit #2)  "
T1:          equ    1             ;(bit #1)  "
T0:          equ    0             ;(bit #0)  "
; bit position masks
; read-only Rx data buffer
mR7:         equ    %10000000     ;receive data bits
mR6:         equ    %01000000     ; "
mR5:         equ    %00100000     ; "
mR4:         equ    %00010000     ; "
```

```
mR3:         equ    %00001000     ; "
mR2:         equ    %00000100     ; "
mR1:         equ    %00000010     ; "
mR0:         equ    %00000001     ; "
; write-only Tx data buffer
mT7:         equ    %10000000     ;transmit data bits
mT6:         equ    %01000000     ; "
mT5:         equ    %00100000     ; "
mT4:         equ    %00010000     ; "
mT3:         equ    %00001000     ; "
mT2:         equ    %00000100     ; "
mT1:         equ    %00000010     ; "
mT0:         equ    %00000001     ; "

;****  Serial Peripheral Interface (SPI)  **************************************************
;*
SPIC1:      equ    $28           ;SPI control register 1
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
SPIE:        equ    7             ;(bit #7) SPI interrupt enable
SPE:         equ    6             ;(bit #6) SPI enable
SPTIE:       equ    5             ;(bit #5) Tx error interrupt enable
MSTR:        equ    4             ;(bit #4) master/slave
CPOL:        equ    3             ;(bit #3) clock polarity
CPHA:        equ    2             ;(bit #2) clock phase
SSOE:        equ    1             ;(bit #1) SS output enable
LSBFE:       equ    0             ;(bit #0) LSB-first enable
; bit position masks
mSPIE:       equ    %10000000     ;SPI interrupt enable
mSPE:        equ    %01000000     ;SPI enable
mSPTIE:      equ    %00100000     ;SPI Tx error interrupt enable
mMSTR:       equ    %00010000     ;master/slave
mCPOL:       equ    %00001000     ;clock polarity
mCPHA:       equ    %00000100     ;clock phase
mSSOE:       equ    %00000010     ;slave select output enable
mLSBFE:      equ    %00000001     ;LSB-first enable

SPIC2:      equ    $29           ;SPI control register 2
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
MODFEN:      equ    4             ;(bit #4) mode fault enable
BIDIROE:     equ    3             ;(bit #3) bi-directional enable
SPISWAI:     equ    1             ;(bit #1) SPI stops in wait
SPCO:        equ    0             ;(bit #0) SPI pin control
; bit position masks
mMODFEN:     equ    %00010000     ;mode fault enable
mBIDIROE:    equ    %00001000     ;bi-directional operation enable
mSPISWAI:    equ    %00000010     ;SPI stops in wait mode
mSPCO:       equ    %00000001     ;SPI pin control

SPIBR:       equ    $2A           ;SPI baud rate select
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
SPPR2:       equ    6             ;(bit #6) SPI baud rate prescale
SPPR1:       equ    5             ;(bit #5)  "
SPPR0:       equ    4             ;(bit #4)  "
SPR2:        equ    2             ;(bit #2) SPI rate selact
SPR1:        equ    1             ;(bit #1)  "
```

```
SPR0:        equ    0             ;(bit #0)  "
; bit position masks
mSPPR2:      equ    %01000000     ;SPI baud rate prescale
mSPPR1:      equ    %00100000     ; "
mSPPR0:      equ    %00010000     ; "
mSPR2:       equ    %00000100     ;SPI rate select
mSPR1:       equ    %00000010     ; "
mSPR0:       equ    %00000001     ; "

SPIS:        equ    $2B           ;SPI status register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
SPRF:        equ    7             ;(bit #7) SPI Rx full flag
SPTEF:       equ    5             ;(bit #5) SPI Tx error flag
MODF:        equ    4             ;(bit #4) mode fault flag
; bit position masks
mSPRF:       equ    %10000000     ;SPI receive buffer full flag
mSPTEF:      equ    %00100000     ;SPI Tx error flag?
mMODF:       equ    %00010000     ;mode fault flag

SPID:        equ    $2D           ;SPI data register


;****  Analog-to-Digital Converter Module (ATD)  ********************************************
;*;
ATDC:        equ    $50           ;atd control tegister
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
ATDPU:       equ    7             ;(bit #7) ATD power up
DJM:         equ    6             ;(bit #6) justification mode; rt/left
RES8:        equ    5             ;(bit #5) ATD resolution select
SGN:         equ    4             ;(bit #4) signed result select
PRS3:        equ    3             ;(bit #3) prescaler rate select (high)
PRS2:        equ    2             ;(bit #2) prescaler rate select
PRS1:        equ    1             ;(bit #1) prescaler rate select
PRS0:        equ    0             ;(bit #0) prescaler rate select (low)
; bit position masks
mATDPU:      equ    %10000000     ;ATD power up
mDJM:        equ    %01000000     ;data justification mode; right/left
mRES8:       equ    %00100000     ;ATD resolution select
mSGN:        equ    %00010000     ;signed result select
mPRS3:       equ    %00001000     ;prescaler rate select (high)
mPRS2:       equ    %00000100     ;prescaler rate select
mPRS1:       equ    %00000010     ;prescaler rate select
mPRS0:       equ    %00000001     ;prescaler rate select (low)

ATDSC:       equ    $51           ;atd ststus and control register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
CCF:         equ    7             ;(bit #7) conversion complete flag
ATDIE:       equ    6             ;(bit #6) ATD interrupt enable
ATDCO:       equ    5             ;(bit #5) ATD continuous conversion
ATDCH4:      equ    4             ;(bit #4) ATD input channel select (high)
ATDCH3:      equ    3             ;(bit #3) ATD input channel select
ATDCH2:      equ    2             ;(bit #2) ATD input channel select
ATDCH1:      equ    1             ;(bit #1) ATD input channel select
ATDCH0:      equ    0             ;(bit #0) ATD input channel select (low)
```

```
; bit position masks
mCCF:         equ    %10000000     ;conversion complete flag
mATDIE:       equ    %01000000     ;ATD interrupt enable
mATDCO:       equ    %00100000     ;ATD continuous conversion
mATDCH4:      equ    %00010000     ;ATD input channel select (high)
mATDCH3:      equ    %00001000     ;prescaler rate select
mATDCH2:      equ    %00000100     ;prescaler rate select
mATDCH1:      equ    %00000010     ;prescaler rate select
mATDCH0:      equ    %00000001     ;prescaler rate select (low)

ATDPE:       equ    $54            ;ATD pin enable register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
ATDPE7:      equ    7             ;(bit #7)
ATDPE6:      equ    6             ;(bit #6)
ATDPE5:      equ    5             ;(bit #5)
ATDPE4:      equ    4             ;(bit #4)
ATDPE3:      equ    3             ;(bit #3)
ATDPE2:      equ    2             ;(bit #2)
ATDPE1:      equ    1             ;(bit #1)
ATDPE0:      equ    0             ;(bit #0)
; bit position masks
mATDPE7:     equ    %10000000     ;ATDPE bit 7
mATDPE6:     equ    %01000000     ;ATDPE bit 6
mATDPE5:     equ    %00100000     ;ATDPE bit 5
mATDPE4:     equ    %00010000     ;ATDPE bit 4
mATDPE3:     equ    %00001000     ;ATDPE bit 3
mATDPE2:     equ    %00000100     ;ATDPE bit 2
mATDPE1:     equ    %00000010     ;ATDPE bit 1
mATDPE0:     equ    %00000001     ;ATDPE bit 0

ATDRH:       equ    $52           ;ATD result register (high)
ATDRL:       equ    $53           ;ATD result register (low)

;****  Inter-Integrated Circuit Module (IIC) ************************************************
;*;
IICA:        equ    $58           ;IIC address register

IICF:        equ    $59           ;IIC frequency divider register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
MULT1:       equ    7             ;(bit #7) IIC multiply factor (high)
MULT0:       equ    6             ;(bit #6) IIC multiply factor (low)
ICR5:        equ    5             ;(bit #5) IIC Divider and Hold bit-5
ICR4:        equ    4             ;(bit #4) IIC Divider and Hold bit-4
ICR3:        equ    3             ;(bit #3) IIC Divider and Hold bit-3
ICR2:        equ    2             ;(bit #2) IIC Divider and Hold bit-2
ICR1:        equ    1             ;(bit #1) IIC Divider and Hold bit-1
ICR0:        equ    0             ;(bit #0) IIC Divider and Hold bit-0
; bit position masks
mMULT1:      equ    %10000000     ;IIC multiply factor (high)
mMULT0:      equ    %01000000     ;IIC multiply factor (low)
mICR5:       equ    %00100000     ;IIC Divider and Hold bit-5
mICR4:       equ    %00010000     ;IIC Divider and Hold bit-4
mICR3:       equ    %00001000     ;IIC Divider and Hold bit-3
mICR2:       equ    %00000100     ;IIC Divider and Hold bit-2
mICR1:       equ    %00000010     ;IIC Divider and Hold bit-1
```

```
mICR0:       equ    %00000001     ;IIC Divider and Hold bit-0

IICC:        equ    $5A           ;IIC control register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
IICEN:       equ    7             ;(bit #7) IIC enable bit
IICIE:       equ    6             ;(bit #6) IIC interrupt enable bit
MST:         equ    5             ;(bit #5) IIC master mode select bit
TX:          equ    4             ;(bit #4) IIC transmit mode select bit
TXAK:        equ    3             ;(bit #3) IIC transmit acknowledge bit
RSTA:        equ    2             ;(bit #2) IIC repeat start bit
; bit position masks
mIICEN:      equ    %10000000     ;IIC enable
mIICIE:      equ    %01000000     ;IIC interrupt enable bit
mMST:        equ    %00100000     ;IIC master mode select bit
mTX:         equ    %00010000     ;IIC transmit mode select bit
mTXAK:       equ    %00001000     ;IIC transmit acknowledge bit
mRSTA:       equ    %00000100     ;IIC repeat start bit

IICS:        equ    $5B           ;IIC status register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
TCF:         equ    7             ;(bit #7) IIC transfer complete flag bit
IIAS:        equ    6             ;(bit #6) IIC addressed as slave bit
BUSY:        equ    5             ;(bit #5) IIC bus busy bit
ARBL:        equ    4             ;(bit #4) IIC arbitration lost bit
SRW:         equ    2             ;(bit #2) IIC slave read/write bit
IICIF:       equ    1             ;(bit #1) IIC interrupt flag bit
RXAK:        equ    0             ;(bit #0) IIC receive acknowledge bit
; bit position masks
mTCF:        equ    %10000000     ;IIC transfer complete flag bit
mIIAS:       equ    %01000000     ;IIC addressed as slave bit
mBUSY:       equ    %00100000     ;IIC bus busy bit
mARBL:       equ    %00010000     ;IIC arbitration lost bit
mSRW:        equ    %00000100     ;IIC slave read/write bit
mIICIF:      equ    %00000010     ;IIC interrupt flag bit
mRXAK:       equ    %00000001     ;IIC receive acknowledge bit

IICD:        equ    $5C           ;IIC data I/O register bits 7:0

;****  Timer/PWM Module 1 (TPM1)  ***** TPM1 has 3 channels *********************************
;****  Timer/PWM Module 2 (TPM2)  ***** TPM2 has 5 channels *********************************
;*
TPM1SC:      equ    $30           ;TPM1 status and control register
TPM2SC:      equ    $60           ;TPM2 status and control register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
TOF:         equ    7             ;(bit #7) tomer overflow flag
TOIE:        equ    6             ;(bit #6) TOF interrupt enable
CPWMS:       equ    5             ;(bit #5) centered PWM select
CLKSB:       equ    4             ;(bit #4) clock select bits
CLKSA:       equ    3             ;(bit #3)  "
PS2:         equ    2             ;(bit #2) prescaler bits
PS1:         equ    1             ;(bit #1)  "
PS0:         equ    0             ;(bit #0)  "
; bit position masks
mTOF:        equ    %10000000     ;timer overflow flag
mTOIE:       equ    %01000000     ;timer overflow interrupt enable
```

```
mCPWMS:      equ    %00100000     ;center-aligned PWM select
mCLKSB:      equ    %00010000     ;clock source select bits
mCLKSA:      equ    %00001000     ; "
mPS2:        equ    %00000100     ;prescaler bits
mPS1:        equ    %00000010     ; "
mPS0:        equ    %00000001     ; "

TPM1CNTH:    equ    $31           ;TPM1 counter (high half)
TPM1CNTL:    equ    $32           ;TPM1 counter (low half)
TPM1MODH:    equ    $33           ;TPM1 modulo register (high half)
TPM1MODL:    equ    $34           ;TPM1 modulo register(low half)

TPM2CNTH:    equ    $61           ;TPM2 counter (high half)
TPM2CNTL:    equ    $62           ;TPM2 counter (low half)
TPM2MODH:    equ    $63           ;TPM2 modulo register (high half)
TPM2MODL:    equ    $64           ;TPM2 modulo register(low half)

TPM1C0SC:    equ    $35           ;TPM1 channel 0 status and control
TPM2C0SC:    equ    $65           ;TPM2 channel 0 status and control
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
CH0F:        equ    7             ;(bit #7) channel 0 flag
CH0IE:       equ    6             ;(bit #6) ch 0 interrupt enable
MS0B:        equ    5             ;(bit #5) mode select B
MS0A:        equ    4             ;(bit #4) mode select A
ELS0B:       equ    3             ;(bit #3) edge/level select B
ELS0A:       equ    2             ;(bit #2) edge/level select A
; bit position masks
mCH0F:       equ    %10000000     ;channel 0 flag
mCH0IE:      equ    %01000000     ;ch 0 interrupt enable
mMS0B:       equ    %00100000     ;mode select B
mMS0A:       equ    %00010000     ;mode select A
mELS0B:      equ    %00001000     ;edge/level select B
mELS0A:      equ    %00000100     ;edge/level select A

TPM1C0VH:    equ    $36           ;TPM1 channel 0 value register (high)
TPM1C0VL:    equ    $37           ;TPM1 channel 0 value register (low)

TPM2C0VH:    equ    $66           ;TPM2 channel 0 value register (high)
TPM2C0VL:    equ    $67           ;TPM2 channel 0 value register (low)

TPM1C1SC:    equ    $38           ;TPM1 channel 1 status and control
TPM2C1SC:    equ    $68           ;TPM2 channel 1 status and control
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
CH1F:        equ    7             ;(bit #7) channel 1 flag
CH1IE:       equ    6             ;(bit #6) ch 1 interrupt enable
MS1B:        equ    5             ;(bit #5) mode select B
MS1A:        equ    4             ;(bit #4) mode select A
ELS1B:       equ    3             ;(bit #3) edge/level select B
ELS1A:       equ    2             ;(bit #2) edge/level select A
; bit position masks
mCH1F:       equ    %10000000     ;channel 1 flag
mCH1IE:      equ    %01000000     ;ch 1 interrupt enable
mMS1B:       equ    %00100000     ;mode select B
mMS1A:       equ    %00010000     ;mode select A
mELS1B:      equ    %00001000     ;edge/level select B
```

```
mELS1A:      equ    %00000100     ;edge/level select A

TPM1C1VH:    equ    $39           ;TPM1 channel 1 value register (high)
TPM1C1VL:    equ    $3A           ;TPM1 channel 1 value register (low)

TPM2C1VH:    equ    $69           ;TPM2 channel 1 value register (high)
TPM2C1VL:    equ    $6A           ;TPM2 channel 1 value register (low)

TPM1C2SC:    equ    $3B           ;TPM1 channel 2 status and control
TPM2C2SC:    equ    $6B           ;TPM2 channel 2 status and control
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
CH2F:        equ    7             ;(bit #7) channel 2 flag
CH2IE:       equ    6             ;(bit #6) ch 2 interrupt enable
MS2B:        equ    5             ;(bit #5) mode select B
MS2A:        equ    4             ;(bit #4) mode select A
ELS2B:       equ    3             ;(bit #3) edge/level select B
ELS2A:       equ    2             ;(bit #2) edge/level select A
; bit position masks
mCH2F:       equ    %10000000     ;channel 2 flag
mCH2IE:      equ    %01000000     ;ch 2 interrupt enable
mMS2B:       equ    %00100000     ;mode select B
mMS2A:       equ    %00010000     ;mode select A
mELS2B:      equ    %00001000     ;edge/level select B
mELS2A:      equ    %00000100     ;edge/level select A

TPM1C2VH:    equ    $3C           ;TPM1 channel 2 value register (high)
TPM1C2VL:    equ    $3D           ;TPM1 channel 2 value register (low)

TPM2C2VH:    equ    $6C           ;TPM2 channel 1 value register (high)
TPM2C2VL:    equ    $6D           ;TPM2 channel 1 value register (low)

TPM2C3SC:    equ    $6E           ;TPM2 channel 3 status and control
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
CH3F:        equ    7             ;(bit #7) channel 3 flag
CH3IE:       equ    6             ;(bit #6) ch 3 interrupt enable
MS3B:        equ    5             ;(bit #5) mode select B
MS3A:        equ    4             ;(bit #4) mode select A
ELS3B:       equ    3             ;(bit #3) edge/level select B
ELS3A:       equ    2             ;(bit #2) edge/level select A
; bit position masks
mCH3F:       equ    %10000000     ;channel 3 flag
mCH3IE:      equ    %01000000     ;ch 3 interrupt enable
mMS3B:       equ    %00100000     ;mode select B
mMS3A:       equ    %00010000     ;mode select A
mELS3B:      equ    %00001000     ;edge/level select B
mELS3A:      equ    %00000100     ;edge/level select A

TPM2C3VH:    equ    $6F           ;TPM2 channel 1 value register (high)
TPM2C3VL:    equ    $70           ;TPM2 channel 1 value register (low)

TPM2C4SC:    equ    $71           ;TPM2 channel 4 status and control
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
CH4F:        equ    7             ;(bit #7) channel 4 flag
CH4IE:       equ    6             ;(bit #6) ch 4 interrupt enable
MS4B:        equ    5             ;(bit #5) mode select B
```

```
MS4A:        equ    4             ;(bit #4) mode select A
ELS4B:       equ    3             ;(bit #3) edge/level select B
ELS4A:       equ    2             ;(bit #2) edge/level select A
; bit position masks
mCH4F:       equ    %10000000     ;channel 4 flag
mCH4IE:      equ    %01000000     ;ch 4 interrupt enable
mMS4B:       equ    %00100000     ;mode select B
mMS4A:       equ    %00010000     ;mode select A
mELS4B:      equ    %00001000     ;edge/level select B
mELS4A:      equ    %00000100     ;edge/level select A

TPM2C4VH:    equ    $72           ;TPM2 channel 1 value register (high)
TPM2C4VL:    equ    $73           ;TPM2 channel 1 value register (low)

****  Internal Clock Generator Module (ICG)  ************************************************
;*
ICGC1:       equ    $48           ;ICG control register 1
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
RANGE:       equ    6             ;(bit #6) frequency range select
REFS:        equ    5             ;(bit #5) reference select
CLKS1:       equ    4             ;(bit #4) clock select bit 1
CLKS0:       equ    3             ;(bit #3) clock select bit 0
OSCSTEN:     equ    2             ;(bit #2) oscillator runs in stop
; bit position masks
mRANGE:      equ    %01000000     ;frequency range select
mREFS:       equ    %00100000     ;reference select
mCLKS1:      equ    %00010000     ;clock mode select (bit-1)
mCLKS0:      equ    %00001000     ;clock mode select (bit 0)
mOSCSTEN:    equ    %00000100     ;enable oscillator in stop mode

ICGC2:       equ    $49           ;ICG control register 2
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
LOLRE:       equ    7             ;(bit #7) loss of lock reset enable
MFD2:        equ    6             ;(bit #6) multiplication factor div
MFD1:        equ    5             ;(bit #5)  "
MFD0:        equ    4             ;(bit #4)  "
LOCRE:       equ    3             ;(bit #3) loss of clock reset enable
RFD2:        equ    2             ;(bit #2) reference divider
RFD1:        equ    1             ;(bit #1)  "
RFD0:        equ    0             ;(bit #0)  "
; bit position masks
mLOLRE:      equ    %10000000     ;loss of lock reset enable
mMFD2:       equ    %01000000     ;multiplication factor divider
mMFD1:       equ    %00100000     ; "
mMFD0:       equ    %00010000     ; "
mLOCRE:      equ    %00001000     ;loss of clock reset enable
mRFD2:       equ    %00000100     ;reference divider bits
mRFD1:       equ    %00000010     ; "
mRFD0:       equ    %00000001     ; "

ICGS1:        equ    $4A           ;ICG status register 1
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
CLKST1:      equ    7             ;(bit #7) clock mode status 1
CLKST0:      equ    6             ;(bit #6) clock mode status 0
REFST:       equ    5             ;(bit #5) reference clock status
```

```
LOLS:        equ    4             ;(bit #4) loss of lock status
LOCK:        equ    3             ;(bit #3) FLL lock status
LOCS:        equ    2             ;(bit #2) loss of clock status
ERCS:        equ    1             ;(bit #1) ext ref clk status
ICGIF:       equ    0             ;(bit #0) ICG interrupt flag
; bit position masks
mCLKST1:     equ    %10000000     ;clock mode status 1
mCLKST0:     equ    %01000000     ;clock mode status 0
mREFST:      equ    %00100000     ;reference clock status
mLOLS:       equ    %00010000     ;loss of lock status
mLOCK:       equ    %00001000     ;FLL lock status
mLOCS:       equ    %00000100     ;loss of clock status
mERCS:       equ    %00000010     ;ext ref clk status
mICGIF:      equ    %00000001     ;ICG interrupt flag

ICGS2:        equ    $4B          ;ICG status register 2
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
DCOS:         equ    0            ;(bit #0) DCO Clock Stable
; bit position masks
mDCOS:        equ    %00000001    ;DCO Clock Stable

ICGFLTU:      equ    $4C          ;ICG filter register (upper 4 bits in bits 3:0)
ICGFLTL:      equ    $4D          ;ICG filter register (lower 8 bits)

ICGTRM:       equ    $4E          ;ICG trim register

;****  System Integration Module (SIM)  *****************************************************
;*
SRS:         equ    $1800         ;SIM reset status register
; bit position masks
mPOR:        equ    %10000000     ;power-on reset
mPIN:        equ    %01000000     ;external reset pin
mCOP:        equ    %00100000     ;COP watchdog timed out
mILOP:       equ    %00010000     ;illegal opcode
mICG:        equ    %00000100     ;illegal address access
mLVD:        equ    %00000010     ;low-voltage detect

SBDFR:       equ    $1801         ;system BDM reset register
; bit position masks
mBDFR:       equ    %00000001     ;BDM force reset

SOPT:        equ    $1802         ;SIM options register (write once)
; bit position masks
mCOPE:       equ    %10000000     ;COP watchdog enable
mCOPT:       equ    %01000000     ;COP time-out select
mSTOPE:      equ    %00100000     ;stop enable
mBKGDPE:     equ    %00000010     ;BDM pin enable

SDIDH:       equ    $1806         ;system device identification 1 register (read-only)
SDIDL:       equ    $1807         ;rev3,2,1,0 + 12-bit ID. GB60 ID = $002
; bit position masks within SDIDH
mREV3:       equ    %10000000     ;device revision identification (high)
mREV2:       equ    %01000000     ;device revision identification
mREV1:       equ    %00100000     ;device revision identification
mREV0:       equ    %00010000     ;device revision identification (low)
```

```
;****  Power Management and Control Module (PMC)  *******************************************
;*
SRTISC:      equ    $1808         ;System RTI ststus and control register
; bit position masks
mRTIF:       equ    %10000000     ;real-time interrupt flag
mRTIACK:     equ    %01000000     ;real-time interrupt acknowledge
mRTICLKS:    equ    %00100000     ;real-time interrupt clock select
mRTIE:       equ    %00010000     ;real-time interrupt enable
mRTIS2:      equ    %00000100     ;real-time interrupt delay select (high)
mRTIS1:      equ    %00000010     ;real-time interrupt delay select
mRTIS0:      equ    %00000001     ;real-time interrupt delay select (low)


SPMSC1:      equ    $1809         ;System power management status and control 1 register
; bit position masks
mLVDF:       equ    %10000000     ;low voltage detect flag
mLVDACK:     equ    %01000000     ;LVD interrupt acknowledge
mLVDIE:      equ    %00100000     ;LVD interrupt enable
mLVDRE:      equ    %00010000     ;LVD reset enable (write once bit)
mLVDSE:      equ    %00001000     ;LDV stop enable (write once bit)
mLVDE:       equ    %00000100     ;LVD enable (write once bit)

SPMSC2:      equ    $180A         ;System power management status and control 2 register
; bit position masks
mLVWF:       equ    %10000000     ;low voltage warning flag
mLVWACK:     equ    %01000000     ;low voltage warning acknowledge
mLVDV:       equ    %00100000     ;low voltage detect voltage select
mLVWV:       equ    %00010000     ;low voltage warning voltage select
mPPDF:       equ    %00001000     ;partial power down flag
mPPDACK:     equ    %00000100     ;partial power down acknowledge
mPDC:        equ    %00000010     ;power down control
mPPDC:       equ    %00000001     ;partial power down control

;****  Debug Module (DBG)  ******************************************************************
;*
DBGCAH:      equ    $1810         ;DBG comparator register A (high)
DBGCAL:      equ    $1811         ;DBG comparator register A (low)
DBGCBH:      equ    $1812         ;DBG comparator register B (high)
DBGCBL:      equ    $1813         ;DBG comparator register B (low)
DBGFH:       equ    $1814         ;DBG FIFO register (high)
DBGFL:       equ    $1815         ;DBG FIFO register (low)

DBGC:        equ    $1816         ;DBG control register
; bit position masks
mDBGEN:      equ    %10000000     ;debug module enable
mARM:        equ    %01000000     ;arm control
mTAG:        equ    %00100000     ;tag/force select
mBRKEN:      equ    %00010000     ;break enable
mRWA:        equ    %00001000     ;R/W compare A value
mRWAEN:      equ    %00000100     ;R/W compare A enable
mRWB:        equ    %00000010     ;R/W compare B value
mRWBEN:      equ    %00000001     ;R/W compare B enable

DBGT:        equ    $1817         ;DBG trigger register
```

```
; bit position masks
mTRGSEL:     equ    %10000000     ;trigger on opcode/access
mBEGIN:      equ    %01000000     ;begin/end trigger
mTRG3:       equ    %00001000     ;trigger mode bits
mTRG2:       equ    %00000100     ; "
mTRG1:       equ    %00000010     ; "
mTRG0:       equ    %00000001     ; "

DBGS:        equ    $1818         ;DBG status register
; bit position masks
mAF:         equ    %10000000     ;trigger A match flag
mBF:         equ    %01000000     ;trigger B match flag
mARMF:       equ    %00100000     ;arm flag
mCNT3:       equ    %00001000     ;count of items in FIFO (high)
mCNT2:       equ    %00000100     ; "
mCNT1:       equ    %00000010     ; "
mCNT0:       equ    %00000001     ;count of items in FIFO (low)

;****  Flash Module (FLASH)  ***************************************************************
;*
FCDIV:       equ    $1820         ;Flash clock divider register
; bit position masks
mDIVLD:      equ    %10000000     ;clock divider loaded
mPRDIV8:     equ    %01000000     ;enable prescale by 8
mDIV5:       equ    %00100000     ;flash clock divider bits (high)
mDIV4:       equ    %00010000     ; "
mDIV3:       equ    %00001000     ; "
mDIV2:       equ    %00000100     ; "
mDIV1:       equ    %00000010     ; "
mDIV0:       equ    %00000001     ;flash clock divider bits (low)

FOPT:        equ    $1821         ;Flash options register
; bit position masks
mKEYEN:      equ    %10000000     ;enable backdoor key to security
mFNORED      equ    %01000000     ;Vector redirection enable
mSEC01:      equ    %00000010     ;security state code (high)
mSEC00:      equ    %00000001     ;security state code (low)

FCNFG:       equ    $1823         ;Flash configuration register
; bit position masks
mKEYACC:     equ    %00100000     ;enable security key writing

FPROT:       equ    $1824         ;Flash protection register
; bit position masks
mFPOPEN:    equ    %10000000     ;open unprotected flash for program/erase
mFPDIS:     equ    %01000000     ;flash protection disable
mFPS2:      equ    %00100000     ;flash protect size select (high)
mFPS1:      equ    %00010000     ;flash protect size select
mFPS0:      equ    %00001000     ;flash protect size select (low)

FSTAT:       equ    $1825         ;Flash status register
; bit position masks
mFCBEF:      equ    %10000000     ;flash command buffer empty flag
mFCCF:       equ    %01000000     ;flash command complete flag
mFPVIOL:     equ    %00100000     ;flash protection violation
```

```
mFACCERR:    equu    %00010000     ;flash access error
mFBLANK:     equ    %00000100     ;flash verified as all blank (erased =$ff) flag

FCMD:        equ    $1826         ;Flash command register
; bit position masks
mFCMD7:      equ    %10000000     ;Flash command (high)
mFCMD6:      equ    %01000000     ; "
mFCMD5:      equ    %00100000     ; "
mFCMD4:      equ    %00010000     ; "
mFCMD3:      equ    %00001000     ; "
mFCMD2:      equ    %00000100     ; "
mFCMD1:      equ    %00000010     ; "
mFCMD0:      equ    %00000001     ;Flash command (low)
; command codes
mBlank:      equ    $05           ;Blank Check command
mByteProg:   equ    $20           ;Byte Program command
mBurstProg:  equ    $25           ;Burst Program command
mPageErase:  equ    $40           ;Page Erase command
mMassErase:  equ    $41           ;Mass Erase command

;****  Flash non-volatile register images  ****************************************************
;*
NVBACKKEY:   equ    $FFB0         ;8-byte backdoor comparison key
; comparison key in $FFB0 through $FFB7

; following 2 registers transfered from flash to working regs at reset

NVPROT:      equ    $FFBD         ;NV flash protection byte
; NVPROT transfers to FPROT on reset

NVICGTRIM:   equ    $FFBE         ;NV ICG Trim Setting
; ICG trim value measured during factory test. User software optionally
; copies to ICGTRM during initialization.

NVOPT:       equ    $FFBF         ;NV flash options byte
; NVFEOPT transfers to FOPT on reset

;****  END  *******************************************************************************
```

HOW TO REACH US:

USA/EUROPE/LOCATIONS NOT LISTED:
Motorola Literature Distribution
P.O. Box 5405
Denver, Colorado 80217
1-800-521-6274 or 480-768-2130

JAPAN:
Motorola Japan Ltd.
SPS, Technical Information Center
3-20-1, Minami-Azabu, Minato-ku
Tokyo 106-8573, Japan
81-3-3440-3569

ASIA/PACIFIC:
Motorola Semiconductors H.K. Ltd.
Silicon Harbour Centre
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
852-26668334

HOME PAGE:
http://motorola.com/semiconductors

MOTOROLA

Information in this document is provided solely to enable system and software implementers to use Motorola 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.

Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola 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 Motorola 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. Motorola does not convey any license under its patent rights nor the rights of others. Motorola 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 Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola 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 Motorola was negligent regarding the design or manufacture of the part.

MOTOROLA and the Stylized M Logo are registered in the US Patent and Trademark Office. All other product or service names are the property of their respective owners. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

HCS08RMv1/D
Rev. 1
6/2003