Application Note

AN2400/D
Rev. 0, 12/2002

HCS12 NVM Guidelines

MOTOROLA
intelligence everywhere™

digital dna™

by Stuart Robb
Applications Engineering
Motorola, East Kilbride

Introduction

The HCS12 is the next generation of the industry standard 68HC12 16-bit microcontrollers. The HCS12 is built around a high performance CPU with bus frequencies of up to 25MHz, and is complemented by on-chip peripherals such as timers, analogue-to-digital converters and advanced serial communications modules such as CAN, SPI, SCI and IIC.

HCS12 microcontrollers incorporate advanced, third generation, non-volatile Flash EEPROM memory that is used to store the application program code and constant data. The Flash memory can be erased and reprogrammed many times over and is ideally suited to the development phase of a product. Flash memory is also suitable for the production phase as product inventories can be reduced by having a common microcontroller for similar products. Any software changes, upgrades or fixes can be implemented immediately during production, without the delay and costs associated with a new ROM mask. Furthermore, products in the field can be reprogrammed as required without having to replace the microcontroller. Over the product lifespan, Flash offers significant potential cost savings when compared to ROM.

Various sizes of Flash memory are available, from 32k bytes to 512k bytes, to suit the requirements of different applications.

Most HCS12 microcontrollers also incorporate EEPROM that may be used to store data variables. The EEPROM on HCS12 microcontrollers is constructed using the same basic technology as the Flash memory.

This paper is intended to give the reader an understanding of how the non-volatile memory (NVM) on the HCS12 works and guidelines on how best to make use of it. Code snippets for all NVM user commands are included, in both ‘C’ and assembly language.

The reader should refer to the relevant Flash (or EEPROM) Block User Guide for complete details of all Flash/EEPROM registers. The reader should also refer to the relevant microcontroller User Guide for current Flash/EEPROM electrical specifications, particularly data retention, write-erase cycles and erase/programming timings.

The author gratefully acknowledges the contributions provided by many colleagues, in particular Derek Beattie, Ally Gorman and Andy Birnie.

Split-Gate Flash Memory

HCS12 microcontrollers incorporate advanced, 0.25µm non-volatile memory technology called Split-Gate Flash (SGF). The same basic technology is used for both Flash and EEPROM on HCS12 microcontrollers.

Split-Gate Flash Memory Structure

The Flash memory is organised in a basic grid of rows and columns. At the intersection of each row and column is a split-gate transistor, which has both a control gate and a floating gate, as depicted in **Figure 1. Split-Gate Flash Transistor**. The floating gate is electrically insulated from both the control gate and the drain-source channel. However, capacitive coupling causes the floating gate both to influence and be influenced by the potential at the source.

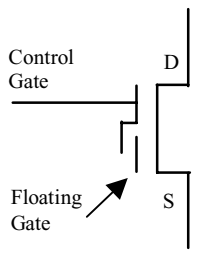

Figure 1. Split-Gate Flash Transistor

Each split-gate transistor corresponds to one bit of Flash memory, called a bitcell. Sixteen bitcells are grouped together to form each word. **Figure 2** illustrates one quarter of a word. Control logic decodes each CPU address to select the appropriate Flash cells to be read or written, and applies the required voltages on the drain, source and control gate to read, program or erase each cell.

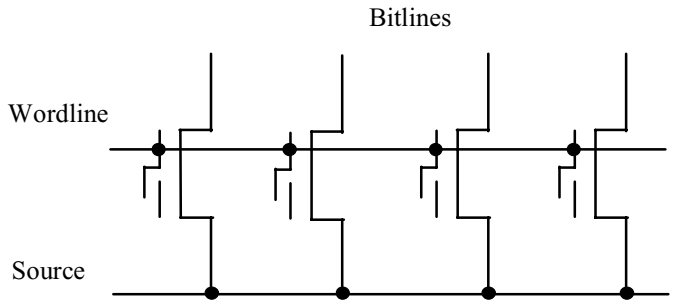

Figure 2. One quarter of a Split-Gate Flash Word

To program a word, a high positive voltage (>>VDD) is applied to the cell sources. The control gates have ~VDD applied through the wordline. Cells that are to be programmed to '0' have a low voltage (~0V) applied to the drain through the bitlines, as depicted in **Figure 3**. A high electric field is created at the gap between the floating gate and the control gate which causes some electrons in the source-drain channel to be injected into the floating gate, leaving the floating gate with a slightly negative charge. Cells that are to remain in the erased state have a voltage (~VDD) applied to the drain through the bitline. This changes the electric field and no electrons are injected into the floating gate. When the high voltage is switched off, the floating gate retains its charge indefinitely. Programming is a quick process, taking only a few microseconds per word.

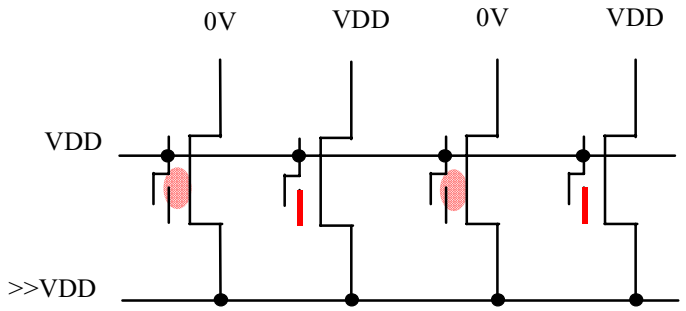

Figure 3. Programming a '0101' pattern into Split-Gate Flash

To read a word, the cells sources are connected to VSS (0V) and ~VDD is applied to the control gates through the wordline, as shown in **Figure 4**. Current flows through the drain to the source only if the control gate is positively charged (erased state), so the bitline current is sensed to determine whether the cell should read as a '1' (erased) or '0' (programmed).

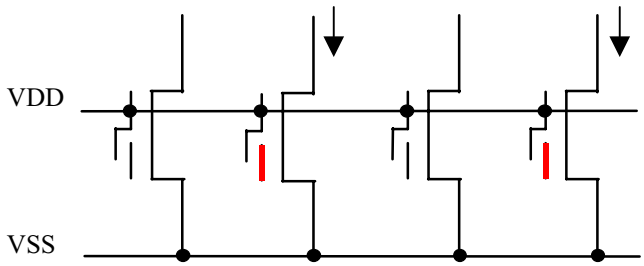

Figure 4. Reading a '0101' pattern in Split-Gate Flash

To erase the Flash, the sources and bitlines are connected to VSS (0V) and a high positive voltage (>>VDD) is applied to the control gate through the wordline, as illustrated in **Figure 5**. A high electric field is created between the floating gate and the control gate. This causes Fowler-Nordheim tunnelling of electrons from the floating gate to the control gate, leaving the floating gate with a net positive charge. This is a slow process, requiring up to 20ms to erase a sector.

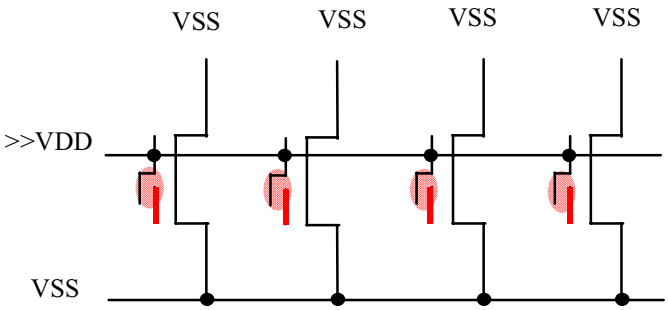

Figure 5. Erasing a nibble of Split-Gate Flash

NVM Programming and Erasure

Programming and erasure of Flash and EEPROM memory is controlled by a command state machine. The command state machine supervises the writing sequence of all commands and verifies the validity of the command sequence. The state machine is also responsible for applying the appropriate voltages to the Flash block for the required length of time. The state machine requires a timebase between 150kHz and 200kHz that is derived from the microcontroller oscillator clock by means of a programmable prescaler. Valid commands are listed in **Table 2. Valid Flash/EEPROM Commands**. Error flags in the Flash or EEPROM Status register indicate any errors. No commands can be executed unless all error flags are cleared (in all blocks).

All the voltages required for programming and erasure are generated by on-chip charge-pumps. Each separate NVM block has an independent command

state machine and charge-pump, thus allowing multiple Flash blocks and EEPROM to be programmed or erased simultaneously.

The Flash and EEPROM are programmed in units of aligned words, i.e. two bytes at a time. The data word is written to an even address, i.e. bit 0 of the address is clear. This will result in the bytes at the even address and the even address plus one being programmed.

The Flash memory is erased either in 512 byte sectors (1024 byte sectors for the 128k byte block), or as a mass erase of an entire block. A sector is a distinct division of Flash: 512 byte sectors start at addresses $x000, $x200, $x400, $x600, $x800, $xA00, $xC00 and $xE00. 1024 bytes sectors start at addresses $x000, $x400, $x800 and $xC00.

The EEPROM memory is erased either in 4 byte sectors, or as a mass erase of the entire block. A sector is a distinct division of EEPROM: EEPROM sectors start at addresses $xxx0, $xxx4, $xxx8 and $xxxC.

The command register, address register and data registers are buffered to allow pipelined programming. Pipelined programming allows the next address, data and command to be loaded while the current command is still executing, thus reducing the overall programming time.

Flash (but not EEPROM) also has a mode called Burst programming. Burst programming is invoked by pipelining program commands for words on the same Flash row. A row is 64 bytes on 32k and 64k byte Flash blocks and 128 bytes on the 128k Flash block. Burst programming reduces the programming time by keeping the high voltage generation switched on between program commands on the same row. Burst programming is approximately twice as fast as single word programming.

Flash and EEPROM are programmed and erased in very similar ways, and so the following description applies equally to both. The two main registers used during programming and erase operations are the Flash or EEPROM Status register and the Flash or EEPROM Command register. The Flash or EEPROM Clock Divider Register must be correctly initialised before programming or erasure can begin.

Flash Clock Divider Register

7	6	5	4	3	2	1	0
FDIVLD	PRDIV8	FDIV5	FDIV4	FDIV3	FDIV2	FDIV1	FDIV0

Figure 6. Flash Clock Divider Register (FCLKDIV)

The Flash Clock Divider register is shared between all Flash Blocks.

The EEPROM Clock Divider Register has identical bit definitions (bit names start with E instead of F).

The command state machine requires a timebase that is derived from the microcontroller oscillator clock via a programmable prescaler. The oscillator is used as a clock source so that programming and erasure is independent of changes in the MCU bus frequency, in low power modes for example. The prescaler value is configured by the FCLKDIV register for Flash and the ECLKDIV register for EEPROM. These registers *must* be written with an appropriate value before programming or erasure can commence. The value written to the register is chosen so that the timebase is between 150kHz and 200kHz. A flowchart for determining the correct value is shown in **Figure 7. PRDIV8 and FDIV bits Determination Procedure**. An incorrect value can result in incomplete programming or erasure, or damage to the Flash or EEPROM due to overstress. Furthermore, the microcontroller bus clock must be 1MHz or greater during programming or erasure.

FDIVLD – Clock Divider Loaded

1 = Register has been written since reset.

0 = Register has not been written, program/erase is not possible.

PRDIV8 – Prescaler Divide by 8

FDIV[5:0] – Clock Divider Bits

Table 1. Clock Divider Values

PRDIV8	FDIV5	FDIV4	FDIV3	FDIV2	FDIV1	FDIV0	Prescaler
0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	2
0	0	0	0	0	1	0	3
0	…	…	…	…	…	…	…
0	1	1	1	1	1	1	64
1	0	0	0	0	0	0	8
1	0	0	0	0	0	1	16
1	0	0	0	0	1	0	24
1	…	…	…	…	…	…	…
1	1	1	1	1	1	1	512

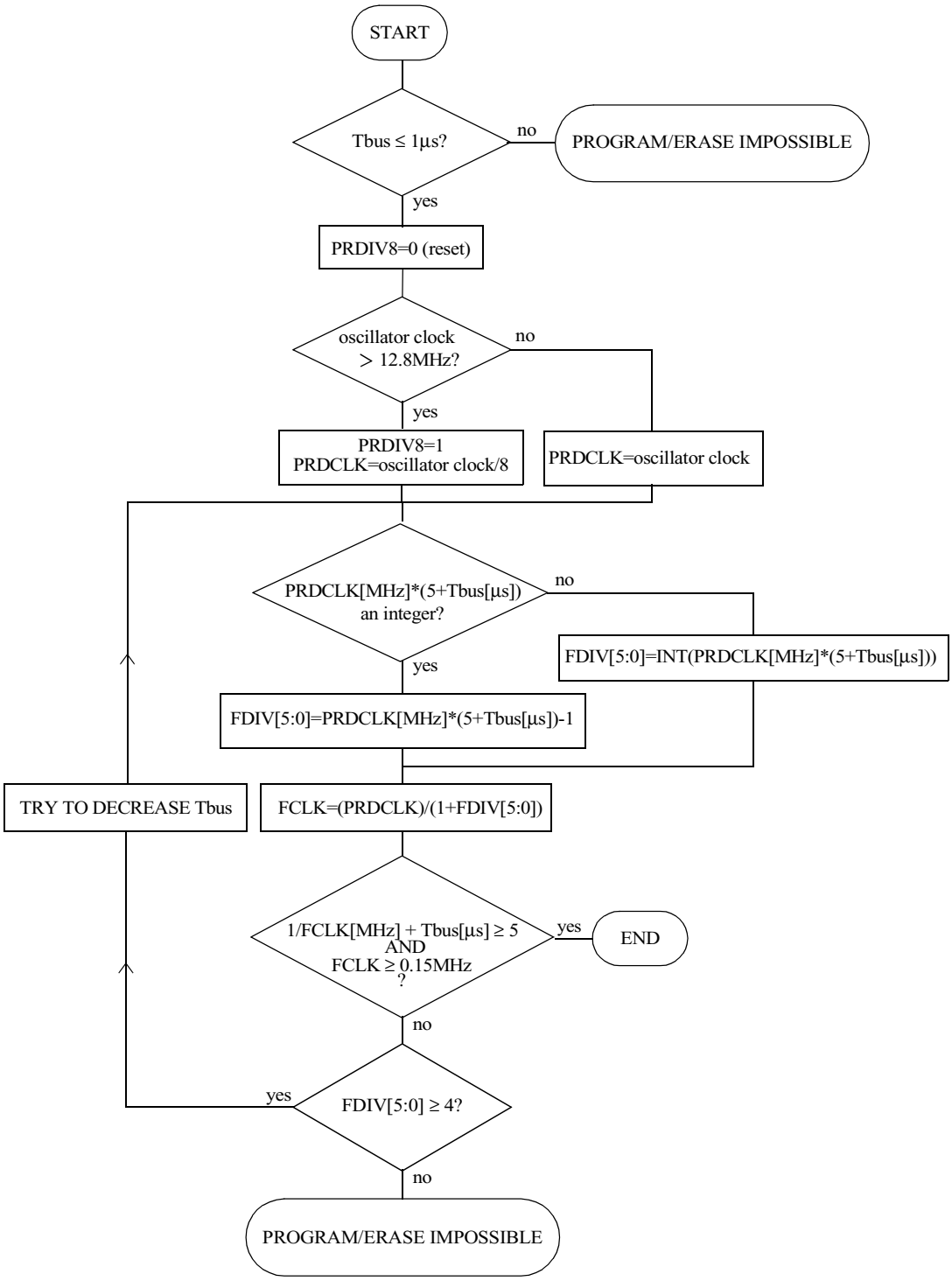

Figure 7. PRDIV8 and FDIV bits Determination Procedure

Flash Status Register

7	6	5	4	3	2	1	0
CBEIF	CCIF	PVIOL	ACCERR	0	BLANK	0	0

Figure 8. Flash Status Register (FSTAT)

NOTE: *CBEIF, PVIOL and ACCER are cleared by writing a '1' to the respective bit. CCIF, BLANK and bits 3, 1 and 0 are read only (write has no effect).*

The Flash Status register is banked on microcontrollers that have multiple Flash blocks. That is, each Flash block has an independent Flash Status register but each Flash Status register is accessed at the same address. The active Block is selected by means of the BKSEL bits in the FCNFG register.

BSET and BCLR instructions should not be used on this register. BSET and BCLR are 'read-modify-write' instructions. This means that the register is read, the read value is modified by setting or clearing the specified bits, and then the resulting value is written back to the register. This may result in bits being cleared unintentionally by the 'write 1 to clear' nature of the CBEIF, PVIOL and ACCERR bits.

The EEPROM Status register has identical bit definitions.

CBEIF — Command Buffer Empty Interrupt Flag

The CBEIF flag indicates whether the address, data and command buffers are empty so that a new command sequence can be started. A command is launched by writing a '1' to CBEIF (to clear the bit). CBEIE in the FCNFG/ECNFG register must be set to enable an interrupt request.

1 = Buffers are ready to accept a new command.
0 = Buffers are full.

CCIF — Command Complete Interrupt Flag

The CCIF flag is cleared by hardware when CBEIF is cleared and is automatically set when all commands have been completed. This flag is read only. The Flash or EEPROM cannot be read when CCIF is clear. CCIE in the FCNFG/ECNFG register must be set to enable an interrupt request.

1 = All commands completed.
0 = Command in progress.

PVIOL — Protection Violation

The PVIOL flag indicates that an attempt was made to program or erase a protected area of Flash or EEPROM. The PVIOL flag must be cleared by writing a '1' to PVIOL before starting a new command sequence.

1 = Protection violation has occurred.
0 = No protection violation.

ACCERR — Access Error

The ACCERR flag indicates that an illegal access to the Flash or EEPROM has occurred. The ACCERR flag must be cleared by writing a ‘1’ to ACCERR before starting a new command sequence.

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

BLANK — Erased Verification

The BLANK flag indicates the result of an Erase Verify command. The BLANK flag is cleared by hardware when the CBEIF flag is cleared. This flag is read only.

1 = Flash/EEPROM verified as erased by Erase Verify command.
0 = If an Erase Verify command has been executed, the Flash or EEPROM is not erased.

Flash Command Register

Valid commands which may be written to the Flash or EEPROM Command registers in Normal modes are shown in **Table 2. Valid Flash/EEPROM Commands**. Any other value will cause the ACCERR bit in ESTAT to be set.

The Flash Command register is banked on microcontrollers that have multiple Flash blocks. That is, each Flash block has an independent Flash Command register but each Flash Command register is accessed at the same address. The active Block is selected by means of the BKSEL bits in the FCNFG register.

Table 2. Valid Flash/EEPROM Commands

Command	Name	Description
$05	Erase Verify	BLANK bit in ESTAT will be set on command completion if the entire EEPROM block is erased.
$20	Program	Program a word (2 bytes)
$40	Sector Erase[1]	Erase a sector.
$41	Mass Erase	Erase an entire block.
$60	Sector Modify[2]	Erase a sector (4 bytes), program a word (2 bytes)

1. An erase sector is 4 bytes for EEPROM, 1024 bytes for a 128k byte Flash block and 512 bytes for all other Flash blocks.
2. The Sector Modify command is applicable to EEPROM only.

Flash/EEPROM Command Sequence

NOTE: ***A Flash or EEPROM word must be erased before it is programmed.***

The general command sequence begins with an initialisation sequence followed by the command write sequence. The initialisation sequence is described in the individual sections on Flash or EEPROM programming.

If the CBEIF flag in the FSTAT/ESTAT register is set, the command write sequence can begin. The following command write sequence must be strictly adhered to and no intermediate writes to the Flash/EEPROM block or Flash/EEPROM registers are permitted. Flash/EEPROM registers, but not the Flash/EEPROM block being programmed/erased, may be read during the command write sequence.

A command sequence can be aborted prior to being launched by writing $00 to the FSTAT/ESTAT register, causing the ACCERR flag to be set. Once launched, a command cannot be safely stopped prior to completion. Avoid executing a CPU STOP instruction whilst an NVM command is running.

1. Write the data word to be programmed to the word aligned Flash/EEPROM address (address bit 0 clear). The data and address are stored in internal buffers. For erase and erase-verify commands, the data value is irrelevant. For mass erase and erase-verify commands, the address can be any valid address for the Flash/EEPROM block. For the sector erase command, the address can be anywhere in the desired sector.

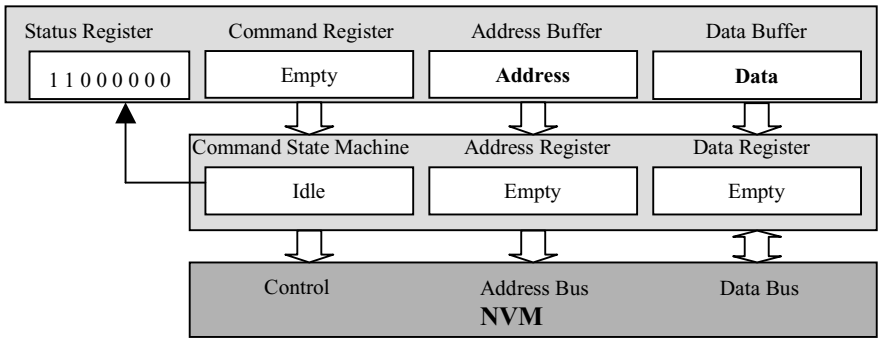

Figure 9. NVM Address and Data Write

2. Write the desired command to the FCMD/ECMD register. Valid commands are listed in **Table 2. Valid Flash/EEPROM Commands**.

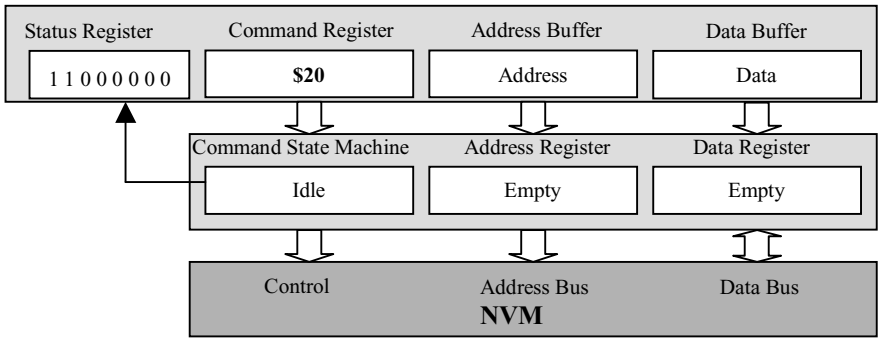

Figure 10. NVM Program Command Write

3. Launch the command by writing $80 to the FSTAT/ESTAT register to clear the CBEIF bit.

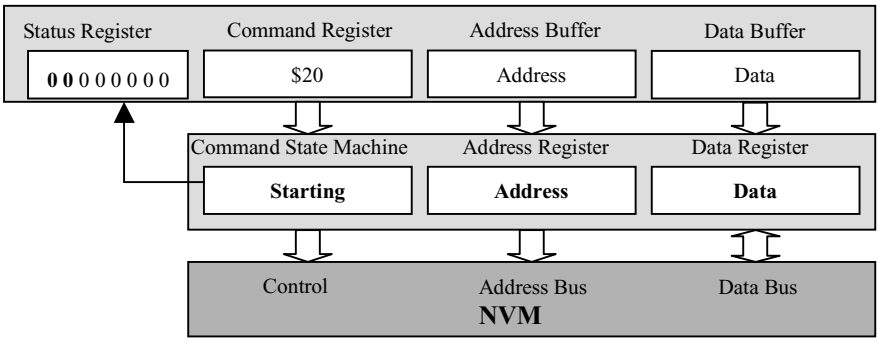

Figure 11. NVM Command Launch

If the command is accepted by the state machine, the CCIF bit will be cleared indicating that the command is in progress and the CBEIF bit will be set again indicating that the command, address and data buffers are ready to accept the next command sequence for pipelined operation. If the command is not accepted by the state machine, the CCIF bit will not be cleared and the ACCERR or PVIOL bit will be set indicating an error.

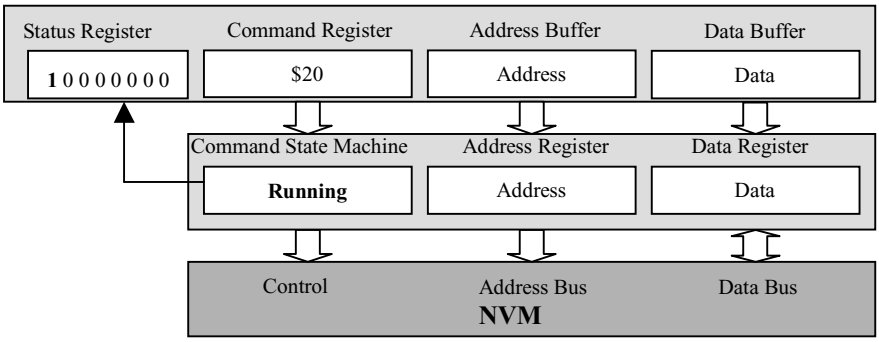

Figure 12. NVM Command Running

For pipelined operation, the next command sequence can begin as soon as the CBEIF bit is set. The command sequence is identical, beginning with the data write, followed by the command write and finally the command launch.

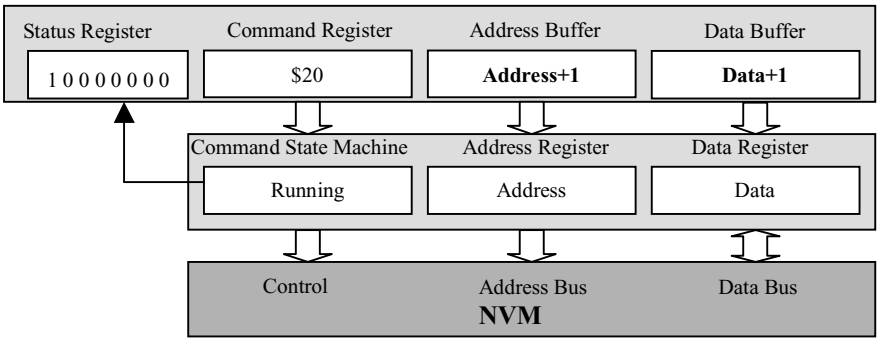

Figure 13. Pipelined NVM Block Write

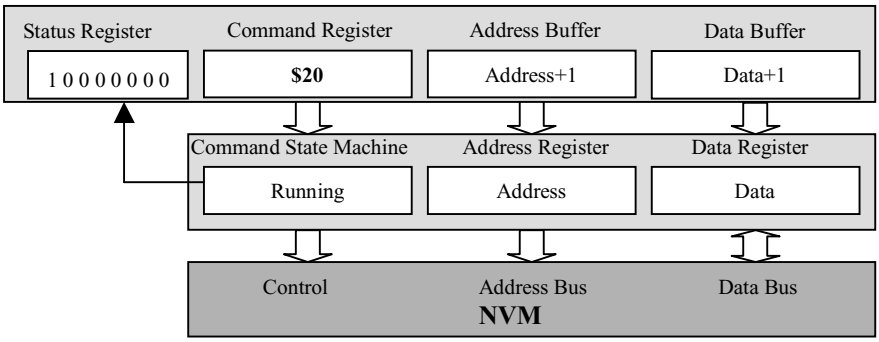

Figure 14. Pipelined Program Command Write

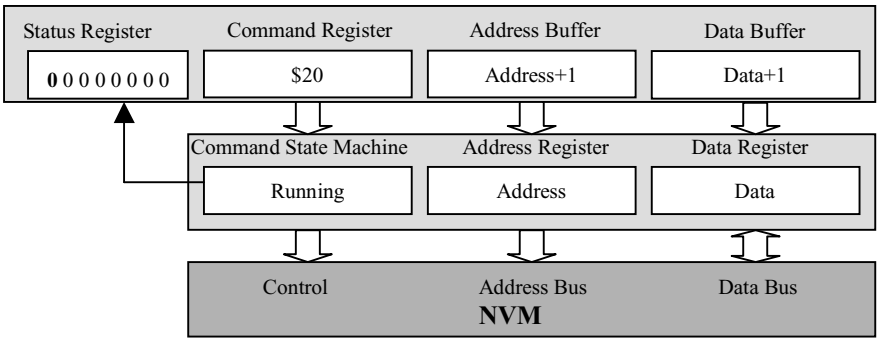

Figure 15. Pipelined Command Launch

As soon as the running command has completed, the new command is started with the new address and data. The command state machine is then ready to accept the next command, as indicated by the CBEIF bit being set again, as in **Figure 12. NVM Command Running**.

The completion of the command is indicated by the CCIF bit being set. The CCIF bit is set only when all active and pending commands for the Flash or EEPROM block have been completed. The Flash or EEPROM block cannot be read when the CCIF bit is clear.

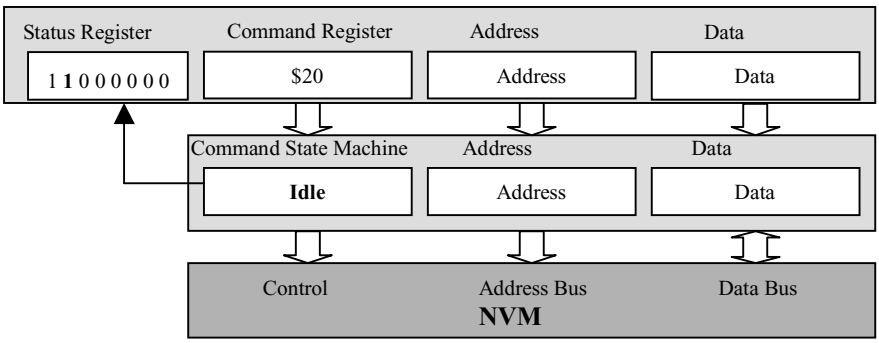

Figure 16. NVM Command Complete

Flash Memory

Introduction

Non-volatile Flash memory is used to store the application program code and constant data. Once programmed, the Flash memory retains the code until it is erased and reprogrammed. Flash memory can be erased and reprogrammed many times over, refer to the microcontroller Electrical Specifications for current data retention and write/erase endurance figures

The standard Flash block sizes are 32k bytes, 64k bytes and 128k bytes. A microcontroller may have a single Flash block of 32k, 64k or 128k bytes, or may have multiple blocks, up to a total of 512k bytes. Each Flash block can be programmed independently and simultaneously, thus enabling microcontroller programming times to be minimised.

The Flash memory is programmed in units of aligned words, i.e. two bytes at a time. The data word is written to an even address, i.e. bit 0 of the address is clear. This will result in the bytes at the even address and the even address plus one being programmed.

The Flash erase sector size is 512 bytes for 32k and 64k byte Flash blocks. To erase an entire sector, any data value is written to any Flash address within the required sector. That is to say, only address bits [15:9] are required to determine the erase sector. For a 128k byte Flash block, the erase sector size is 1024 bytes and address bits [15:10] determine the required sector.

The command register, address register and data registers are buffered to allow pipelined programming. Pipelined programming allows the next address, data and command to be loaded while the current command is still executing, thus reducing the overall programming time.

Flash has a mode called Burst programming. Burst programming is invoked by pipelining program commands for words on the same Flash row. A row is 64 bytes on 32k and 64k byte Flash blocks and 128 bytes on the 128k Flash block. Burst programming reduces the programming time by keeping the high voltage generation switched on between program commands on the same row. Burst programming isapproximately twice as fast as single word programming.

Flash may also be used to emulate EEPROM on microcontrollers that do not have EEPROM, refer to application note AN2302/D for details and example software.

The Flash control registers are located at the register base address + $100 to $10F. The register base address is set by the INITRG register, the base address after a reset is $0000.

Table 3. Flash Register Summary

Address	Name
$x100	Flash Clock Divider Register (FCLKDIV)
$x101	Flash Security Register (FSEC)
$x102	Flash Test Mode Register (FTSTMOD)
$x103	Flash Configuration Register (FCNFG)
$x104	Flash Protection Register (FPROT)[1]
$x105	Flash Status Register (FSTAT)[1]
$x106	Flash Command Register (FCMD)[1]
$x107	Reserved
$x108–$x109	16-bit Address Buffer (FADDR)[1]
$x10A–$x10B	16-bit Data Buffer (FDATA)[1]

1. For microcontrollers that have multiple Flash Blocks, each Flash block has a separate Protection register (FPROT), Status register (FSTAT), Command register (FCMD), Address register (FADDR) and Data register (FDATA). However, these registers are banked, i.e. the registers for each Flash block share the same address. The active bank of registers is selected by the BKSEL bits in the unbanked Flash Configuration register (FCNFG). Thus with the BKSEL bits cleared, Flash Block 0 is selected, and accesses to address $x105 will access the Flash Status register of Flash Block 0

Furthermore, when programming through the page window, $8000 to $BFFF, the PPAGE register must be configured to select a page within the selected flash block.

The general command sequence is described in **Flash/EEPROM Command Sequence**. **Flash/EEPROM Command Sequence** must be preceded with the following initialisation sequence:

1. If the FDIVLD bit is clear, initialise the FCLKDIV register.
2. Verify that all ACCERR and PVIOL flags in the FSTAT register are clear. If the microcontroller has multiple Flash blocks, the FSTAT contents must be checked for all combinations of the BKSEL bits in the FCNFG register.
3. If the microcontroller has multiple Flash blocks, write the BKSEL bits in the FCNFG register to select the bank of registers corresponding to the Flash block to be programmed, erased or verified.
4. Write the core PPAGE register to select the desired page to be programmed if programming in the $8000 to $BFFF range. There is no need to set PPAGE if programming outwith this range, or if the microcontroller does not have a page window.

Illegal Flash Operations

The ACCERR flag will be set during the command write sequence if any of the following illegal operations are performed causing the command write sequence to immediately abort:

1. Writing to the Flash address space before initializing FCLKDIV.
2. If the microcontroller has multiple Flash blocks, writing to the Flash address space in the range $8000–$BFFF when PPAGE register does not select a 16K bytes page in the Flash block selected by the BKSEL bits in the FCNFG register.
3. If the microcontroller has multiple Flash blocks, writing to the Flash address space $4000–$7FFF or $C000–$FFFF with the BKSEL bits in the FCNFG register not selecting Flash block 0.
4. Writing a misaligned word or a byte to the valid Flash address space.
5. Writing to the Flash address space while CBEIF is not set.
6. Writing a second word to the Flash address space before executing a valid command on the previously written word.
7. Writing to any Flash register other than FCMD after writing a word to the Flash address space.
8. Writing a second command to the FCMD register before executing the previously written command.
9. Writing an invalid user command to the FCMD register in user mode.
10. Writing to any Flash register other than FSTAT (to clear CBEIF) after writing to the command register, FCMD.
11. If the microcontroller enters STOP mode while a command is in progress, the command is aborted and any pending command is aborted.
12. A “0” is written to the CBEIF bit in the FSTAT register.

The ACCERR flag will not be set if any Flash register is read during the command sequence.

If the Flash array is read during execution of an algorithm (i.e. CCIF bit in the FSTAT register is clear) the read will return non valid data and the ACCERR flag will not be set.

If an ACCERR flag is set in any of the FSTAT registers the Command State Machine is locked. It is not possible to launch another command on any block until the ACCERR flag is cleared.

The PVIOL flag will be set during the command write sequence after the word write to the Flash address space if any of the following illegal operations are performed, causing the command sequence to immediately abort:

1. Writing a Flash address to program in a protected area of the Flash.
2. Writing a Flash address to erase in a protected area of the Flash.

3. Writing the mass erase command to FCMD while any protection is enabled.

If a PVIOL flag is set in any of the FSTAT registers the Command State Machine is locked. It is not possible to launch another command on any block until the PVIOL flag is cleared.

Parallel Flash Block Programming

On microcontrollers that have multiple Flash blocks, the programming time can be reduced by programming the Flash blocks in parallel. This is possible because each Flash block has independent command state machines, registers, buffers and charge pumps.

The general procedure for programming multiple Flash blocks in parallel is shown in **Figure 17. Parallel Flash Block Programming**. The programming software has to supply the programming algorithm with data words for each Flash block in turn.

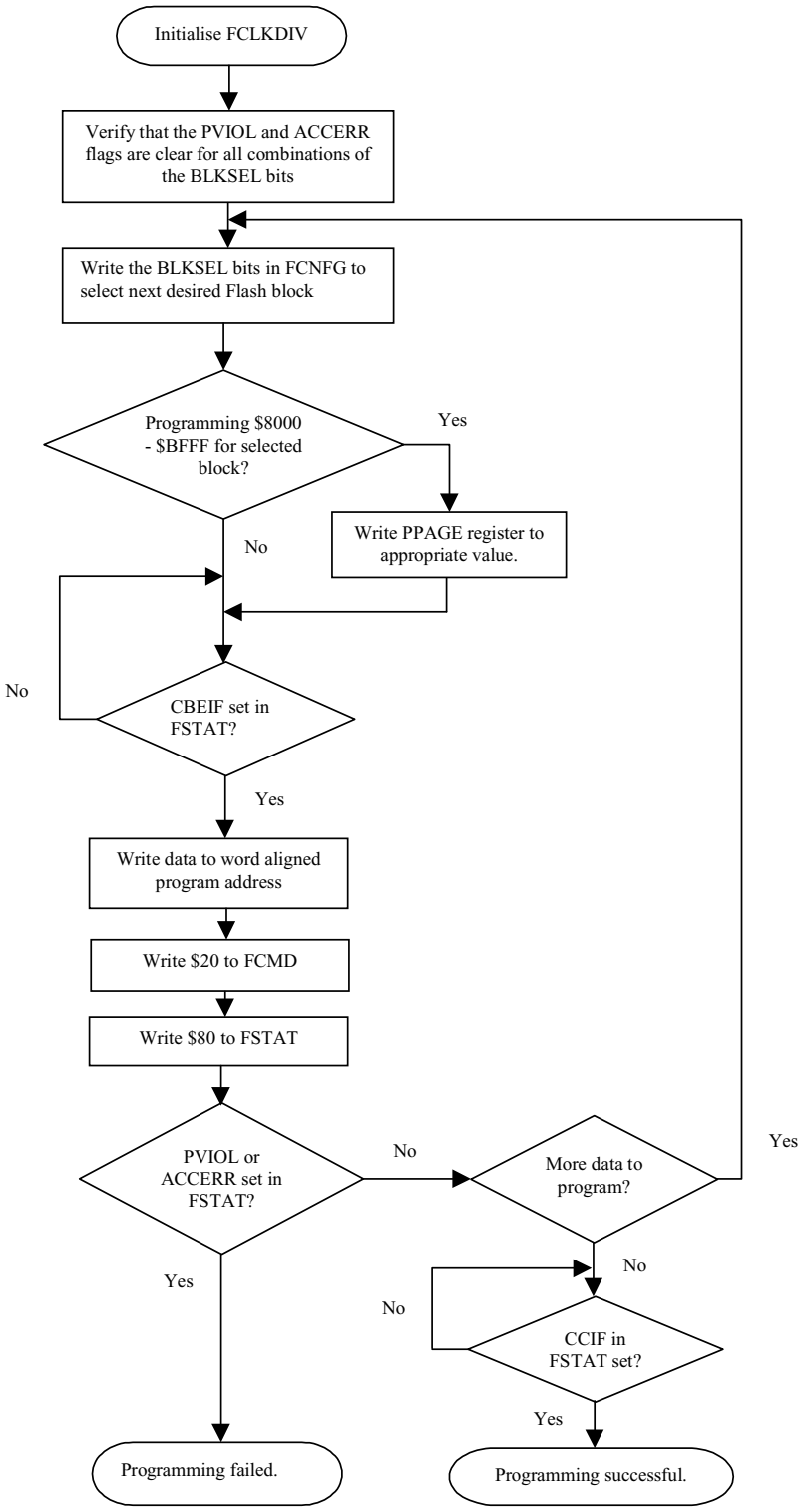

Figure 17. Parallel Flash Block Programming

Flash Memory Paging

The Program Counter on the HCS12 family of microcontrollers is a 16-bit register, which means that the directly addressable space is limited to 64k bytes. Larger memory sizes are accessed by using a technique called *paging*, or memory *banks*. As implemented on the HCS12 family, the entire flash memory is divided into pages 16k bytes in size. One fixed page is always accessible at $4000 to $7FFF and another fixed page is always accessible at $C000 to $FFFF. The region of addresses from $8000 to $BFFF are designated to be the page window. Individual pages of memory are accessible through this window, the desired page being selected by means of the PPAGE register. Only one complete page is accessible through the window at any time. **Figure 18** shows the memory map and memory paging scheme for the HCS12DP256.

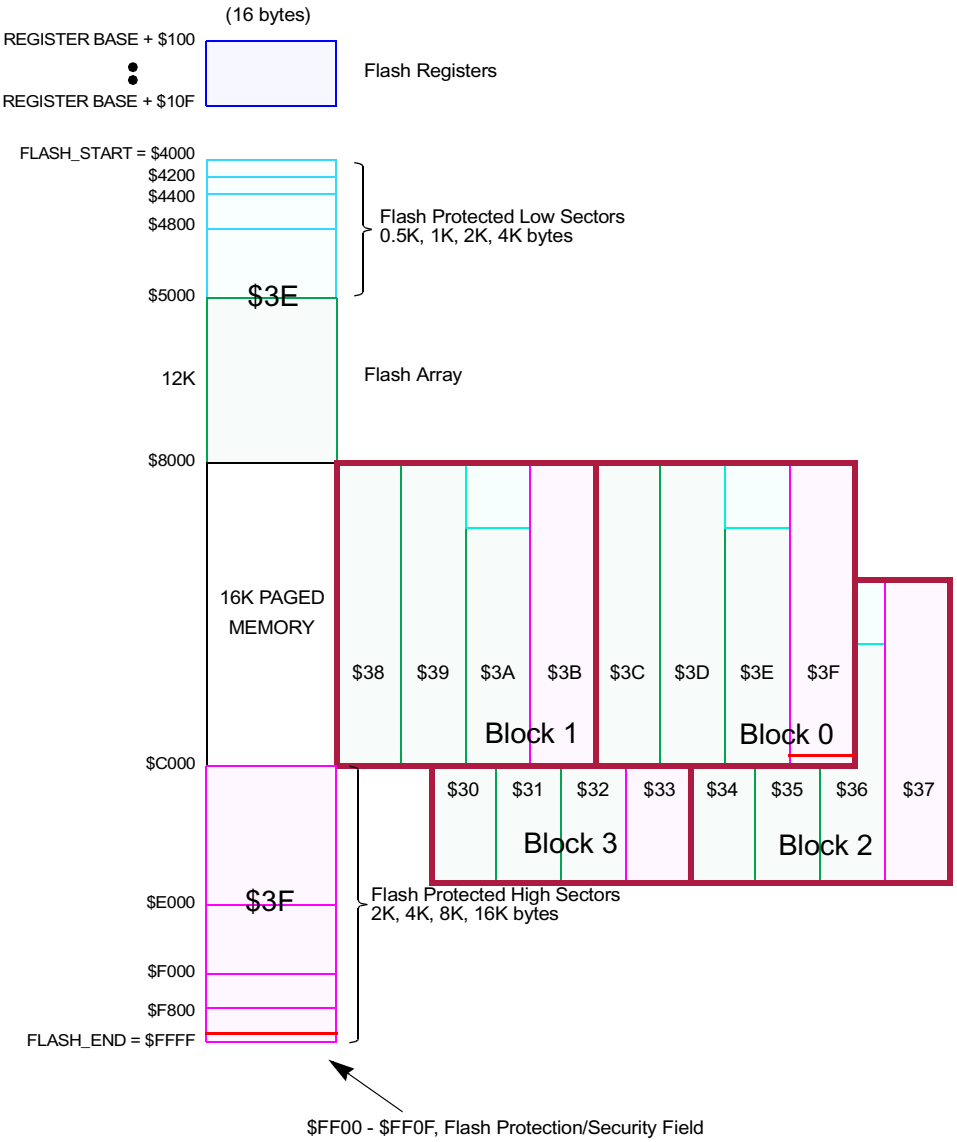

Figure 18. MC9S12DP256 Memory Map

NOTE: *$30-$3F correspond to the PPAGE register content.*

Support for the paging mechanism is built into the HCS12 instruction set, with the CALL and RTC instructions. The CALL instruction is like the JSR instruction (jump to subroutine), but the CALL instruction automatically handles the PPAGE register to transfer control to a subroutine in paged memory. The CALL instruction requires one more byte of memory and three extra clock cycles to execute than the JSR instruction. The CALL instruction need only be used when a subroutine in a different page than the one currently selected is to be called. Modern compilers for the HCS12, such as produced by Metrowerks, are capable of determining whether a CALL or JSR instruction needs to be used and can also arrange routines in memory to minimise the use of the CALL instruction. The extra byte required for a CALL instruction is therefore only used when required, offering enhanced code density over 24-bit instruction sets.

The RTC instruction is the equivalent to the RTS instruction, but restores the PPAGE register to its value prior to the previous CALL instruction. The RTC instruction requires one byte of memory, like the RTS instruction, but two extra clock cycles to execute.

Non-Paged Flash Memory

On HCS12 microcontrollers that employ flash memory paging, there are two regions of addresses that access fixed regions of flash memory. These regions are $4000 to $7FFF and $C000 to $FFFF. Each of these regions corresponds to a fixed page, usually the two highest numbered pages. These pages may also be accessed through the page window, although this is not normally done. Because these fixed pages are permanently present in the memory map, they are used to store certain items that must be accessible at all times and cannot be stored in paged memory.

Vector Table

In order to service an interrupt or restart after a reset, the CPU must fetch the address of the interrupt service routine (ISR) or the reset address. These addresses are called vectors because they redirect the CPU to the appropriate place to start executing code. The list of ISR addresses is called the Interrupt Vector Table and this table also includes the reset vectors. The address of the vector table is fixed in the design of the microcontroller and is located at addresses $FF80 to $FFFF. As an interrupt or reset is by nature an asynchronous event the vector table must be permanently accessible, hence the table is stored at a non-paged address.

Interrupt Service Routines

An interrupt service routine (ISR) contains code that is executed in order to process an interrupt. The CPU obtains the address of the appropriate ISR from the Interrupt Vector Table. As an interrupt is by nature an asynchronous event each ISR must be permanently accessible and located in non-paged memory. It is acceptable for an ISR to call subroutines that are located in paged memory, if the time delay incurred by the page switch is acceptable.

Start-up Code

Start-up code that is executed in the event of a reset must be permanently accessible and therefore must be located in non-paged memory.

Constant Data

For constant data to be accessible to a function, both the function and the data must be present in the memory map simultaneously. This is most easily achieved by locating the constant data in non-paged memory, either in Flash or EEPROM. If there is a large amount of constant data then sometimes an alternative must be sought. There are a number of possibilities, for example the constant data could be located in paged memory, with the functions that access it located either in the same page or in non-paged memory.

If the data and the functions that access it are located on different pages, then the data must be accessed through an intermediate function which handles the page switching. This is to be avoided if at all possible due the inefficiency of this method.

Unused Flash

Surplus Flash memory that is not used by the application is often left in the erased state ($FF). However, the state of unused Flash does sometimes have an effect on the microcontroller behaviour. For example, a severe occurrence of Electromagnetic Interference (EMI) may cause the microcontroller to behave erratically. One possible effect is that the Program Counter may become corrupted and then the CPU may read any address, including unused Flash for the next instruction. This is called code-runaway. The value in the unused Flash will determine what happens next.

If the Flash is erased, the $FF values will be interpreted as a LDS $FFFF instruction. The CPU will continue reading increasing Flash addresses until it reaches some code, and will then behave unpredictably. This situation is normally undesirable and it is preferable to force a microcontroller reset as soon as possible.

A simple solution is to fill unused Flash with $3F, the op-code for the SWI instruction. As this is a single byte instruction, it will always be executed correctly. The microcontroller will fetch the Software Interrupt vector and execute the code at this address, as this interrupt cannot be masked. The code in the interrupt service routine could shut down the microcontroller in an orderly manner before forcing a reset. A reset can be forced by enabling the COP watchdog and then writing an illegal value to the ARMCOP register.

If the Software Interrupt is required by the application, an alternative solution is to use the ‘Unimplemented Instruction’ interrupt. As all page 1 op-codes are valid, it is necessary to choose a page 2 op-code. Filling unused Flash with $18A7 will cause an ‘Unimplemented Instruction’ interrupt if the CPU attempts to execute this op-code. $18 is the pre-byte to select page 2, and $A7 is unimplemented on page 2. If the $A7 is read first, this will be interpreted as a NOP instruction and the next $18A7 will be read.

Filling unused Flash with the op-code for the STOP instruction, $183E, does not give a good solution for two reasons. First, the STOP instruction is disabled by the 'S' bit in the CPU Condition Codes register. If the S bit is set, the STOP instruction is treated like a 2-cycle NOP and the microcontroller does not stop. Furthermore, the $3E op-code may read first, and this corresponds to the WAI instruction. The WAI instruction stops the CPU but not the peripherals, so the op-code for the next STOP instruction is not executed.

Flash Memory Protection

Each Flash block may be protected against accidental erasure or programming. Flash protection is controlled by a Flash Protection register (FPROT). On microcontrollers that have multiple Flash blocks, there is a separate Flash Protection register for each Flash block. In this case the Flash Protection registers share a common address, with the active register selected by means of the Bank Select bits within the Flash Configuration register. During the microcontroller reset sequence, the Flash Protection registers for each Flash block are loaded from programmed bytes within a Flash block. For example, for the MC9S12DP256, location $FF0A controls protection for block three, $FF0B controls protection for block two, $FF0C controls protection for block one and $FF0D controls protection for block zero, as shown in **Table 4. Flash Protection and Security Memory Locations for MC9S12DP256**. The values of each FPROT register determine whether the entire block or just subsections are protected from being accidentally erased or programmed. Software can write to the FPROT registers to *increase* the amount of protected Flash by clearing additional bits in the register. It is possible to decrease the amount of protected Flash by setting bits in the FPROT register only in special modes.

Table 4. Flash Protection and Security Memory Locations for MC9S12DP256

Address	Description
$FF00 – $FF07	Security Backdoor Key
$FF08 – $FF09	Reserved
$FF0A	Flash Block 3 Protection Byte
$FF0B	Flash Block 2 Protection Byte
$FF0C	Flash Block 1 Protection Byte
$FF0D	Flash Block 0 Protection Byte
$FF0E	Reserved
$FF0F	Security Byte

Each Flash block can be entirely protected, or can have one or two separate protected areas. One of these areas, known as the lower protected block, starts at a point 32k bytes below the maximum Flash block address and is extendable towards higher addresses. The other, known as the upper protected block,

ends at the top of the Flash block and is extendable towards lower addresses. The lower and upper protected blocks do not meet up. In general, the upper protected area of Flash Block 0 is used to hold bootloader code since it contains the reset and interrupt vectors. The lower protected area of Block 0 and the protected areas of the other Flash blocks can be used for critical parameters that would not change when program firmware is updated.

On some microcontrollers, it is also possible to protect the area between the upper and lower areas. This feature allows a small area of Flash to remain unprotected when using Flash to emulate EEPROM. Refer to the relevant Flash Block Guide for details.

Trying to program or erase any of the protected areas will result in a protection violation error and bit PVIOL will be set in the Flash Status Register FSTAT. A mass erase of an entire Flash block is only possible if protection of that block is fully disabled.

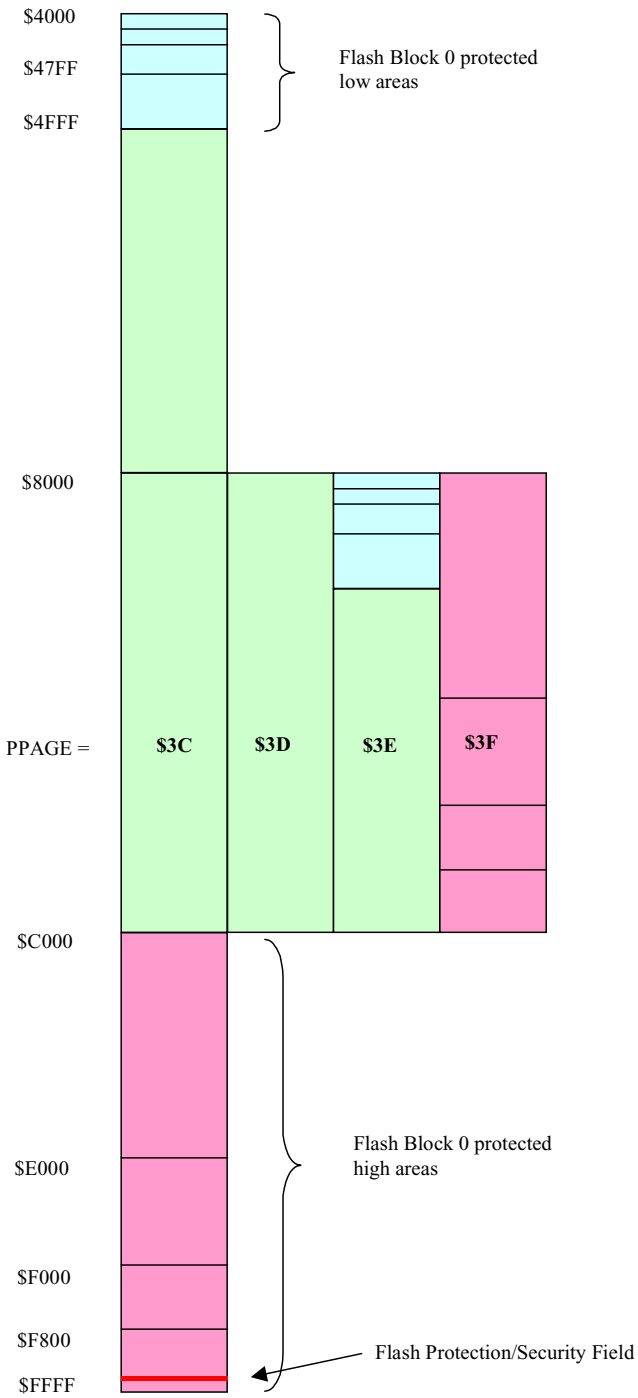

Figure 19. 64k Flash Block 0 Protection Areas

7	6	5	4	3	2	1	0
FPOPEN	NV6	FPHDIS	FPHS1	FPHS0	FPLDIS	FPLS1	FPLS0

Figure 20. Flash Protection Register (FPROT)

The Flash Protection Register for each Flash block is loaded from the appropriate Flash memory bytes during a reset sequence, **Table 4. Flash Protection and Security Memory Locations for MC9S12DP256** lists these for the MC9S12DP256. The erased state of these Flash memory bytes is $FF, which corresponds to Flash protection disabled.

The FPOPEN bit in the FPROT register determines whether the entire Flash block is protected. When the FPOPEN bit is erased, the remainder of the bits in the register determine the state of protection and the size of each protected block. In its programmed state the entire Flash block is protected and the state of the remaining bits within the FPROT register is irrelevant.

The FPHDIS and FPLDIS bits determine the protection state of the upper and lower areas within each Flash block respectively. The erased state of these bits allows erasure and programming of the two protected areas and renders the state of the FPHS[1:0] and FPLS[1:0] bits immaterial. When either of these bits is programmed, the FPHS[1:0] and FPLS[1:0] bits determine the size of the upper and lower protected areas. **Table 5. Flash Protection High Bits** and **Table 6. Flash Protection Low Bits** summarize the combinations of the FPHS[1:0] and FPLS[1:0] bits and the size of the protected area selected by each for a 64k byte Flash block.

On some microcontrollers, the FPHDIS, FPHS[1:0], FPLDIS and FPLS[1:0] bits do have an effect on the selection of the protected Flash areas when FPOPEN = 0. This allows upper or lower areas to be unprotected while the rest of the Flash is protected. This feature allows a small area of Flash to remain unprotected when using Flash to emulate EEPROM. Refer to the relevant Flash Block Guide for details.

Trying to program or erase any of the protected areas will result in a protection violation error and bit PVIOL will be set in the Flash Status Register FSTAT. A mass erase of an entire block is only possible if the protection for that block is fully disabled, i.e. FOPEN = 1, or FPHDIS = 1 and FPLDIS = 1.

NOTE: *The Flash protection memory locations are located within the upper protected block of Flash Block 0. Therefore if the upper protected block of Flash Block 0 is protected, or if the whole of Flash Block 0 is protected, then the Flash protection memory locations themselves are protected and can no longer be changed.*

FPOPEN — Flash Protection Open
1 = The Flash block protection depends on the FPHDIS, FPHS[1:0], FPLDIS and FPLS[1:0] bits.
0 = The entire Flash block is protected against programming and erasure. The FPHDIS, FPHS[1:0], FPLDIS and FPLS[1:0] bits have no effect.

NOTE: *On some microcontrollers, the FPHDIS, FPHS[1:0], FPLDIS and FPLS[1:0] bits do have an effect on the selection of the protected Flash areas when FPOPEN = 0. Refer to the relevant Flash Block Guide for details.*

FPHDIS — Flash Protection High address range Disable.
1 = The high address range protection is disabled.
0 = The high address range protection is enabled and the size of the protected area depends on the FPHS[1:0] bits.

FPHS[1:0] — Flash Protection High address Size.

The FPHS[1:0] bits determine the size of the high address range protected area. **Table 5. Flash Protection High Bits** gives the available sizes for a 64k byte Flash Block.

FPLDIS — Flash Protection Low address range Disable.
1 = The low address range protection is disabled.
0 = The low address range protection is enabled and the size of the protected area depends on the FPLS[1:0] bits.

FPLS[1:0] — Flash Protection Low address Size.

The FPLS[1:0] bits determine the size of the low address range protected area. **Table 6. Flash Protection Low Bits** gives the available sizes for a 64k byte Flash Block.

Table 5. Flash Protection High Bits

FPHS[1:0]	Protected Size (64k Block)
00	2k bytes
01	4k bytes
10	8k bytes
11	16k bytes

Table 6. Flash Protection Low Bits

FPLS[1:0]	Protected Size (64k Block)
00	512 bytes
01	1k bytes
10	2k bytes
11	4k bytes

Protected Application Example

The values loaded into each FPROT register during a reset sequence determine the minimum level of protection: the application software can write to the FPROT registers to *increase* (but not decrease) the amount of protected Flash by clearing additional bits in the register. This feature is useful for applications which are required to have the ability to reprogram the Flash. For example, after each reset, start-up code would determine whether reprogramming is required or not. If reprogramming is required, serial communication software would download and run first a Flash erase routine and then a Flash program routine followed by the new Flash code.

The reset vector, start-up code and communication software (bootloader) should be protected so that reprogramming is always possible, even if the reprogramming process is interrupted or corrupted in some way. The start-up code and bootloader should be located in the higher protected area of Flash Block 0 (page $3F) and address $FF0D is programmed to a suitable value to permanently protect the required size. Note that this will protect the entire vector table, so future code revisions will need to have interrupt service routines located at constant, defined, addresses.

If reprogramming is not required, or when reprogramming has been completed, the whole of the Flash could be protected by software clearing the FPOPEN bit in each FPROT register. This will protect the main application from unintentional corruption until the next reset.

On the other hand, if Flash reprogramming is never required, the FPOPEN bit in each Flash protection byte should be programmed to ‘0’. This will give permanent Flash protection.

Flash Program and Erase Routines

Flash Program Command

The following code segment demonstrates how to program a number of words of Flash using the pipelined programming command. For words that are on the same Flash row, this will invoke Burst programming, reducing the programming time by half.

The leftmost column contains C code (variable definitions in **Appendix A ‘C’ Variable Definitions**), the centre column contains equivalent assembly code assuming the register base address is $0000, and the rightmost column contains comments.

Prerequisites: FCLKDIV must be configured correctly, the Flash words to be programmed must be erased and not protected, the first Flash address must be word aligned (bit 0 = 0). If the Flash program address is in the range $8000 to $BFFF, the PPAGE register must be written to select the desired page. If the microcontroller has multiple Flash blocks, the ACCERR and PVIOL flags in all other blocks must be clear and the BKSEL bits in the FCNFG register must be

written to select the desired block for programming. This code snippet will not program over page boundaries.

Registers: X contains first word aligned Flash address to be programmed, Y contains address of first word of data.

Stack Pointer → number of words to be programmed.

Stack Pointer + 2 → return address.

C function local variables: `UINT16* progAddr, UINT16* dataAddr, UINT16 wordsToDo`.

On return, accumulator B contains 0 if the command executed correctly, or 1 if the command failed.

`FSTAT.byte = ACCERR|PVIOL;`	`MOVB #$30,$105`	`Clear error flags on selected block`
`while(wordsToDo != 0)` `{`	`BRA fepsc`	`Check if any more words to be programmed`
`if(FSTAT.bit.cbeif == 1)` `{`	`fepbt:` `BRCLR $105,#$80,fepsc`	`Check command buffer is empty`
`*progAddr++ = *dataAddr++;`	`LDD 2,Y+` `STD 2,X+`	`Write data word to Flash address, increment addresses`
`FCMD.byte = PROG;`	`MOVB #$40,$106`	`Write program command`
`FSTAT.byte = CBEIF;`	`MOVB #$80,$105`	`Write '1' to CBEIF to launch the command`

`if((FSTAT.byte & (ACCERR|PVIOL))!= 0)`
`{`
`return(FAIL);`
`}`
	`BRSET $105,#$30,fepf`	`Command failed if either error flag set`

`wordsToDo--;`	`DEC 0,SP`	`One word less to be programmed`

`}`
`}`
	`fepsc:`
	`LDD 0,SP`
	`BNE fepbt`
		`Any more words to be programmed?`

`while(FSTAT.bit.ccif != 1)`
`{`
`}`
	`BRCLR $105,#$40,*+0`	`Wait for last command to finish: this is optional, but the Flash block cannot be accessed until CCIF is set.`

`return(PASS);` `}`	`CLRB` `BRA feprtn`	`Successful, return`

	`fepf:`
	`LDAB #1`
	`feprtn:`
	`RTS`
		`Fail, return`

Flash Sector Erase Command

The following code segment demonstrates how to erase a sector (512 bytes, or 1024 bytes on a 128k byte Flash block) of Flash.

The leftmost column contains C code (variable definitions in **Appendix A 'C' Variable Definitions**), the centre column contains equivalent assembly code

assuming the register base address is $0000, and the rightmost column contains comments.

Prerequisites: FCLKDIV must be configured correctly, the sector to be erased must not be protected, the Flash address must be word aligned (bit 0 = 0). If the Flash address is in the range $8000 to $BFFF, the PPAGE register must be written to select the desired page. If the microcontroller has multiple Flash blocks, the ACCERR and PVIOL flags in all other blocks must be clear and the BKSEL bits in the FCNFG register must be written to select the desired block for sector erase.

Registers: X contains word aligned Flash address within the sector to be erased.

Stack Pointer → return address.

C function local variables: `UINT16* sectorAddr, UINT16 dummy`.

On return, accumulator B contains 0 if the command executed correctly, or 1 if the command failed.

`FSTAT.byte = ACCERR|PVIOL;`	`MOVB  #$30,$105`	`Clear error flags on selected block`
`if(FSTAT.bit.cbeif == 1)` `{`	`BRCLR $105,#$80,fesef`	`Check command buffer is empty`
`  *sectorAddr = dummy;`	`STD   0,X`	`Write any data to Flash sector address`
`  FCMD.byte = ERASE;`	`MOVB  #$40,$106`	`Write sector erase command`
`  FSTAT.byte = CBEIF;`	`MOVB  #$80,$105`	`Write '1' to CBEIF to launch the command`

`  if((FSTAT.byte & (ACCERR|PVIOL))!= 0)`
`  {`
`    return(FAIL);`
`  }`
	`BRSET $105,#$30,fesef`	`Command failed if either error flag set`

`  while(FSTAT.bit.ccif != 1)`
`  {`
`  }`
	`BRCLR $105,#$40,*+0`	`Wait for command to finish: this is optional, but the Flash block cannot be accessed until CCIF is set.`

`  return(PASS);` `}`	`CLRB` `BRA  fesertn`	`Successful, return`

`else`
`{`
`  return(FAIL);`
`}`
	`fesef:`
	`LDAB  #1`
	`fesertn:`
	`RTS`
		`Fail, return`

Flash Mass Erase Command

The following code segment demonstrates how to erase an entire Flash block.

Note: A mass erase of the entire block is only possible when the FPLDIS, FPHDIS and FOPEN bits are set, see section **Parallel Flash Block Programming**.

The leftmost column contains C code (variable definitions in **Appendix A 'C' Variable Definitions**), the centre column contains equivalent assembly code assuming the register base address is $0000, and the rightmost column contains comments.

Prerequisites: FCLKDIV must be configured correctly, the flash block to be erased must not be protected, the Flash address must be word aligned (bit 0 = 0). If the Flash address is in the range $8000 to $BFFF, the PPAGE register must be written to select any page in the Flash block. If the microcontroller has multiple Flash blocks, the ACCERR and PVIOL flags in all other blocks must be clear and the BKSEL bits in the FCNFG register must be written to select the desired block for mass erase.

Registers: X contains word aligned Flash address within the Flash block to be erased.

Stack Pointer → return address.

C function local variables: `UINT16* flashAddr, UINT16 dummy`.

On return, accumulator B contains 0 if the command executed correctly, or 1 if the command failed.

`FSTAT.byte = ACCERR|PVIOL;`	`MOVB  #$30,$105`	`Clear error flags on selected block`
`if(FSTAT.bit.cbeif == 1)` `{`	`BRCLR $105,#$80,femef`	`Check command buffer is empty`
`  *flashAddr = dummy;`	`STD  0,X`	`Write any data to Flash block address`
`  FCMD.byte = MASS_ERASE;`	`MOVB  #$41,$106`	`Write mass erase command`
`  FSTAT.byte = CBEIF;`	`MOVB  #$80,$105`	`Write '1' to CBEIF to launch the command`

`  if((FSTAT.byte & (ACCERR|PVIOL))!= 0)`
`  {`
`    return(FAIL);`
`  }`
	`BRSET $105,#$30,femef`	`Command failed if either error flag set`

`  while(FSTAT.bit.ccif != 1)`
`  {`
`  }`
	`BRCLR $105,#$40,*+0`	`Wait for command to finish: this is optional, but the Flash block cannot be accessed until CCIF is set.`

`  return(PASS);` `}`	`CLRB` `BRA  femertn`	`Successful, return`

`else`
`{`
`  return(FAIL);`
`}`
	`femef:`
	`LDAB  #1`
	`femertn:`
	`RTS`
		`Fail, return`

Flash Erase-Verify Command

The following code segment demonstrates how to verify that a Flash block is erased using the erase verify command.

The leftmost column contains C code (variable definitions in **Appendix A 'C' Variable Definitions**), the centre column contains equivalent assembly code

assuming the register base address is $0000, and the rightmost column contains comments.

Prerequisites: FCLKDIV must be configured correctly, the Flash address must be word aligned (bit 0 = 0). If the Flash address is in the range $8000 to $BFFF, the PPAGE register must be written to select any page in the Flash block. If the microcontroller has multiple Flash blocks, the ACCERR and PVIOL flags in all other blocks must be clear and the BKSEL bits in the FCNFG register must be written to select the desired block for erase verify.

Registers: X contains word aligned Flash address within the Flash block to be verified.

Stack Pointer → return address.

C function local variables: `UINT16* flashAddr, UINT16 dummy`.

On return, accumulator B contains 0 if the command executed correctly and the Flash block verified as erased, or 1 if the command failed or the Flash block did not verify as erased.

`FSTAT.byte = ACCERR|PVIOL;`	`MOVB  #$30,$105`	`Clear error flags on selected block`
`if(FSTAT.bit.cbeif == 1)` `{`	`BRCLR $105,#$80,feevf`	`Check command buffer is empty`
`  *flashAddr = dummy;`	`STD   0,X`	`Write any data to Flash sector address`
`  FCMD.byte = ERASE;`	`MOVB  #$05,$106`	`Write erase verify command`
`  FSTAT.byte = CBEIF;`	`MOVB  #$80,$105`	`Write '1' to CBEIF to launch the command`

`  if((FSTAT.byte & (ACCERR|PVIOL))!= 0)`
`  {`
`    return(FAIL);`
`  }`
	`BRSET $105,#$30,feevf`	`Command failed if either error flag set`

`  while(FSTAT.bit.ccif != 1)`
`  {`
`  }`
	`BRCLR $105,#$40,*+0`	`Wait for command to finish: the BLANK bit is not valid until CCIF is set.`

`  if(FSTAT.bit.BLANK == 1)`	`BRCLR $105,#$04,feevf`	`Check BLANK bit`

`  {`
`    return(PASS);`
`  }`
	`CLRB`
	`BRA  feevrtn`
		`Successful, return`

`  else`
`  {`
`    return(FAIL)`
`  }`
`}`
`else`
`{`
`  return(FAIL);`
`}`
	`feevf:`
	`LDAB  #1`
	`feevrtn:`
	`RTS`
		`Fail, return`

EEPROM

Introduction

Most HCS12 microcontrollers also incorporate EEPROM that may be used to store data variables. HCS12 microcontrollers that do not have EEPROM may use Flash to emulate EEPROM, refer to application note AN2302/D for details and example software. The EEPROM on HCS12 microcontrollers is constructed using the same basic technology as the Flash memory, but with some adjustments to make it more suitable to data storage applications. The most obvious of these is the erase sector size, which is 4 bytes.

Once programmed, the EEPROM retains data until it is erased and reprogrammed. The EEPROM can be erased and reprogrammed many times over, refer to the microcontroller Electrical Specifications for current data retention and write/erase endurance figures.

The available sizes of EEPROM blocks are 1K, 2K and 4K bytes. The position of the EEPROM block within the microcontroller address space is set by the core INITEE register. The EEPROM control registers are located at the register base address + $110 to $11F. The register base address is set by the core INITRG register, the base address after a reset is $0000. If the EEPROM is located at an address that overlaps the RAM, registers, or Flash, then the EEPROM takes priority over Flash but the RAM and registers take priority over EEPROM.

Address	Name
$x110	EEPROM Clock Divider Register (ECLKDIV)
$x111–$x112	Reserved
$x113	EEPROM Configuration Register (ECNFG)
$x114	EEPROM Protection Register (EPROT)
$x115	EEPROM Status Register (ESTAT)
$x116	EEPROM Command Register (ECMD)
$x117–$x119	16-Bit Address Buffer (EADDR)
$x11A–$x11B	16-Bit Data Buffer (EDATA)

Figure 21. EEPROM Register Summary

The EEPROM is programmed in units of aligned words, i.e. two bytes at a time. The data word is written to an even address, i.e. bit 0 of the address is clear. This will result in the bytes at the even address and the even address plus one being programmed.

The EEPROM erase sector size is four bytes. To erase a sector, any data value is written to any EEPROM address within the required sector. That is to say, only address bits [15:2] are used to determine the erase sector.

The command register, EEPROM address register and EEPROM data registers are buffered to allow pipelined programming. Pipelined programming reduces the programming time by allowing the next address, data and command to be loaded while the current command is still executing.

The general command sequence is described in **Flash/EEPROM Command Sequence**. **Flash/EEPROM Command Sequence** must be preceded with the following initialisation sequence:

1. If the EDIVLD bit is clear, initialise the ECLKDIV register.
2. Verify that all ACCERR and PVIOL flags in the ESTAT register are clear.

Illegal EEPROM Operations

The ACCERR flag will be set during the command write sequence if any of the following illegal operations are performed causing the command write sequence to immediately abort:

1. Writing to the EEPROM address space before initializing ECLKDIV.
2. Writing a misaligned word or a byte to the valid EEPROM address space.
3. Writing to the EEPROM address space while CBEIF is not set.
4. Writing a second word to the EEPROM address space before executing a program or erase command on the previously written word.
5. Writing to any EEPROM register other than ECMD after writing a word to the EEPROM address space.
6. Writing a second command to the ECMD register before executing the previously written command.
7. Writing an invalid user command to the ECMD register in user mode.
8. Writing to any EEPROM register other than ESTAT (to clear CBEIF) after writing to the command register, ECMD.
9. If the microcontroller enters STOP mode and a program or erase command is in progress, the command is aborted and any pending command is aborted.
10. A “0” is written to the CBEIF bit in the ESTAT register.

The ACCERR flag will not be set if any EEPROM register is read during the command sequence.

If the EEPROM array is read during execution of an algorithm (i.e. CCIF bit in the ESTAT register is clear) the read will return non valid data and the ACCERR flag will not be set.

When an ACCERR flag is set in the ESTAT register the Command State Machine is locked. It is not possible to launch another command until the ACCERR flag is cleared.

The PVIOL flag will be set during the command write sequence after the word write to the EEPROM address space and the command sequence will be aborted if any of the following illegal operations are performed.

1. Writing a EEPROM address to program in a protected area of the EEPROM.
2. Writing a EEPROM address to erase in a protected area of the EEPROM.
3. Writing the mass erase command to ECMD while any protection is enabled.

When the PVIOL flag is set in the ESTAT register the Command State Machine is locked. It is not possible to launch another command until the PVIOL flag is cleared.

Storage of Variables in EEPROM

Traditionally, EEPROM is characterised by the ability to program and erase individual bytes. This means that variables can be allocated to EEPROM without regard to their size or order and the permitted number of write/erase cycles for each variable was equal to the specified number of write/erase cycles for a byte EEPROM.

However, the implementation of EEPROM of the HCS12 family of microcontrollers means that this is no longer the case. The smallest unit that can be programmed is an aligned word (2 bytes), and the smallest unit that can be erased is a sector of 4 bytes. This has implications for the way that variables are allocated to EEPROM if the maximum number of write/erase cycles is to be realised.

For example, if a sector of EEPROM contains 4 different variables each 1 byte long, then each time a variable is updated requires that the whole sector is erased and reprogrammed. This means that the specified maximum write/erase cycles for the sector is *shared* between all 4 variables.

Various methods of data storage are examined in the following sections, grouped according to frequency of update.

Infrequently Updated Data Variables

Data variables that are updated infrequently may be allocated into EEPROM or Flash, so long as the total number of updates (over the product lifetime) for all variables in each sector does not exceed the specified maximum for the sector. These variables can be packed into EEPROM or Flash without ‘gaps’ to ensure maximum utilisation. It may be advantageous to store certain variables, such as end-of-line configuration data or end-of-life diagnostic data, in Flash due to the faster programming time that may be achieved using burst programming.

Frequently Updated Variables

Data variables which are updated frequently may need to be allocated a whole EEPROM sector each. This ensures that the total number of permitted updates

of these variables matches the specified maximum of write erase cycles of a sector. This may be done by defining the variable to be of size 4 bytes (e.g. type ‘long’) *and* ensuring that it is aligned to a sector boundary by correct linking. If the ‘natural’ size of the variable is less than 4 bytes, the remaining bytes in the sector are unused, and the variable should be cast to its ‘natural’ size in expressions to avoid unnecessary computation.

Very Frequently Updated Variables

If a data variable must be updated more than the maximum permitted number of write/erase cycles for a sector of EEPROM, then an alternative approach is necessary. First, consider that it may be possible to store the variable in RAM whilst the microcontroller is powered and to update a copy in EEPROM only when a power-down is imminent. If this is not possible and the variable must be stored in EEPROM at all times, then it becomes necessary to allocate a number of separate sectors for the variable. This can be done by creating a circular buffer in EEPROM, with each element of the buffer corresponding to one or more EEPROM sectors. A pointer in RAM can be used to store the address of the most recent data. When the variable is to be updated, the next element of the buffer is written, the ‘old’ data is erased, and the pointer is updated. In this way, the maximum number of updates for the variable becomes the maximum permitted number of write/erase cycles for a sector of EEPROM multiplied by the number of elements in the circular buffer.

EEPROM Protection

The EEPROM block may be protected against accidental erasure or programming. EEPROM protection is controlled by an EEPROM Protection register (EPROT). During the microcontroller reset sequence, the EEPROM Protection register is loaded from the EEPROM Protection byte, located within EEPROM. The EEPROM Protection byte is located within the smallest EEPROM protected area, so protecting EEPROM always protects the EEPROM protection byte, thus guaranteeing the reset state of EEPROM protection. The value of the EPROT register determines whether the entire EEPROM or just subsections are protected from being accidentally erased or programmed. Software can write to the EPROT register to *increase* the amount of protected EEPROM by clearing additional bits in the register. It is possible to decrease the amount of protected EEPROM by setting bits in the EPROT register only in special modes.

7	6	5	4	3	2	1	0
EPOPEN	NV6	NV5	NV4	EPDIS	EP2	EP1	EP0

Figure 22. EEPROM Protection Register (EPROT)

The EEPROM Protection Register is loaded from the EEPROM memory during a reset sequence. The erased state of these EEPROM memory bytes is $FF, which corresponds to EEPROM protection disabled.

The EPOPEN bit in the EPROT register determines whether the entire EEPROM block is protected. When the EPOPEN bit is erased, the remainder of the bits in the register determine the state of protection and the size of the protected block. When the EPOPEN bit is programmed the entire EEPROM block is protected and the state of the remaining bits within the EPROT register is irrelevant.

Trying to program or erase any of the protected areas will result in a protection violation error and bit PVIOL will be set in the EEPROM Status Register ESTAT. A mass erase of the entire EEPROM block is only possible if protection is fully disabled, i.e. EPOPEN = 1 or EPDIS = 1.

NOTE: *The EEPROM protection memory locations are located within the upper protected area of EEPROM block. Therefore if the upper area of EEPROM is protected, or if the whole of EEPROM is protected, then the EEPROM protection memory locations themselves are protected and can no longer be changed.*

EPOPEN — EEPROM Protection Open

1 = The EEPROM block protection depends on the EPDIS and EP[2:0] bits.
0 = The entire EEPROM block is protected against programming and erasure. The EPDIS and EP[2:0] bits have no effect.

EPDIS — EEPROM Protection Disable

1 = The high address range protection is disabled.
0 = The high address range protection is enabled and the size of the protected area depends on the EP[2:0] bits.

EP[2:0] — EEPROM Protection Size

The FPHS[1:0] bits determine the size of the protected area. **Table 5. Flash Protection High Bits** gives the available sizes for a 4k byte EEPROM Block.

Table 7. EEPROM Protection High Bits

FPHS[2:0]	Protected Size (4k Block)
000	64 bytes
001	128 bytes
010	192 bytes
011	256 bytes
100	320 bytes
101	384 bytes
110	448 bytes
111	512 bytes

EEPROM Program and Erase Routines

EEPROM Program Command

The following code segment demonstrates how to program a word (2 bytes) of EEPROM.

The leftmost column contains C code (variable definitions in **Appendix A 'C' Variable Definitions**), the centre column contains equivalent assembly code assuming the register base address is $0000, and the rightmost column contains comments.

Prerequisites: ECLKDIV must be configured correctly, the EEPROM word to be programmed must be erased and not protected, the EEPROM address must be word aligned (bit 0 = 0).

Registers: X contains word aligned EEPROM address to be programmed, D contains new data value.

Stack Pointer → return address.

C function local variables: `UINT16* progAddr, UINT16 data`.

On return, accumulator B contains 0 if the command executed correctly, or 1 if the command failed.

`ESTAT.byte = ACCERR|PVIOL;`	`MOVB  #$30,$115`	`Clear error flags`
`if(ESTAT.bit.cbeif == 1)` `{`	`BRCLR $115,#$80,eepwf`	`Check command buffer is empty`
`  *progAddr = data;`	`STD  0,X`	`Write data to EEPROM aligned word address`
`  ECMD.byte = PROG;`	`MOVB  #$20,$116`	`Write program command`
`  ESTAT.byte = CBEIF;`	`MOVB  #$80,$115`	`Write '1' to CBEIF to launch the command`

`  if((ESTAT.byte & (ACCERR|PVIOL))!= 0)`
`  {`
`    return(FAIL);`
`  }`
	`BRSET $115,#$30,eepwf`	`Command failed if either error flag set`

`  while(ESTAT.bit.ccif != 1)`
`  {`
`  }`
	`BRCLR $115,#$40,*+0`	`Wait for command to finish: this is optional, but the EEPROM cannot be accessed until CCIF is set.`

`  return(PASS);` `}`	`CLRB` `BRA  eepwrtn`	`Successful, return`

`else`
`{`
`  return(FAIL);`
`}`
	`eepwf:`
	`LDAB  #1`
	`eepwrtn:`
	`RTS`
		`Fail, return`

EEPROM Sector Erase Command

The following code segment demonstrates how to erase a sector (4 bytes) of EEPROM.

The leftmost column contains C code (variable definitions in **Appendix A 'C' Variable Definitions**), the centre column contains equivalent assembly code assuming the register base address is $0000, and the rightmost column contains comments.

Prerequisites: ECLKDIV must be configured correctly, the sector to be erased must not be protected, the EEPROM address must be word aligned (bit 0 = 0).

Registers: X contains word aligned EEPROM address within sector to be erased.

Stack Pointer → return address.

C function local variables: `UINT16* sectorAddr, UINT16 dummy`.

On return, accumulator B contains 0 if the command executed correctly, or 1 if the command failed.

`ESTAT.byte = ACCERR|PVIOL;`	`MOVB  #$30,$115`	`Clear error flags`
`if(ESTAT.bit.cbeif == 1)` `{`	`BRCLR $115,#$80,eesef`	`Check command buffer is empty`
`  *sectorAddr = dummy;`	`STD  0,X`	`Write any data to EEPROM sector address`
`  ECMD.byte = ERASE;`	`MOVB  #$40,$116`	`Write sector erase command`
`  ESTAT.byte = CBEIF;`	`MOVB  #$80,$115`	`Write '1' to CBEIF to launch the command`

`  if((ESTAT.byte & (ACCERR|PVIOL))!= 0)`
`  {`
`    return(FAIL);`
`  }`
	`BRSET $115,#$30,eesef`	`Command failed if either error flag set`

`  while(ESTAT.bit.ccif != 1)`
`  {`
`  }`
	`BRCLR $115,#$40,*+0`	`Wait for command to finish: this is optional, but the EEPROM cannot be accessed until CCIF is set.`

`  return(PASS);` `}`	`CLRB` `BRA  eesertn`	`Successful, return`

`else`
`{`
`  return(FAIL);`
`}`
	`eesef:`
	`LDAB  #1`
	`eesertn:`
	`RTS`
		`Fail, return`

EEPROM Mass Erase Command

The following code segment demonstrates how to erase the entire EEPROM.

The leftmost column contains C code (variable definitions in **Appendix A 'C' Variable Definitions**), the centre column contains equivalent assembly code assuming the register base address is $0000, and the rightmost column contains comments.

Prerequisites: ECLKDIV must be configured correctly, no part of EEPROM must be protected, the EEPROM address must be word aligned (bit 0 = 0).

Registers: X contains any word aligned EEPROM address.

Stack Pointer → return address.

C function local variables: `UINT16* eepromAddr, UINT16 dummy`.

On return, accumulator B contains 0 if the command executed correctly, or 1 if the command failed.

`ESTAT.byte = ACCERR|PVIOL;`	`MOVB  #$30,$115`	`Clear error flags`
`if(ESTAT.bit.cbeif == 1)` `{`	`BRCLR $115,#$80,eemef`	`Check command buffer is empty`
`  *eepromAddr = dummy;`	`STD   0,X`	`Write any data to EEPROM address`
`  ECMD.byte = MASS_ERASE;`	`MOVB  #$41,$116`	`Write mass erase command`
`  ESTAT.byte = CBEIF;`	`MOVB  #$80,$115`	`Write '1' to CBEIF to launch the command`

`  if((ESTAT.byte & (ACCERR|PVIOL))!= 0)`
`  {`
`    return(FAIL);`
`  }`
	`BRSET $115,#$30,eemef`	`Command failed if either error flag set`

`  while(ESTAT.bit.ccif != 1)`
`  {`
`  }`
	`BRCLR $115,#$40,*+0`	`Wait for command to finish: this is optional, but the EEPROM cannot be accessed until CCIF is set.`

`  return(PASS);` `}`	`CLRB` `BRA  eemertn`	`Successful, return`

`else`
`{`
`  return(FAIL);`
`}`
	`eemef:`
	`LDAB  #1`
	`eemertn:`
	`RTS`
		`Fail, return`

EEPROM Erase Verify Command

The following code segment demonstrates how to verify whether the entire EEPROM is erased.

The leftmost column contains C code (variable definitions in **Appendix A 'C' Variable Definitions**), the centre column contains equivalent assembly code assuming the register base address is $0000, and the rightmost column contains comments.

Prerequisites: ECLKDIV must be configured correctly, no part of EEPROM must be protected, the EEPROM address must be word aligned (bit 0 = 0).

Registers: X contains any word aligned EEPROM address.

Stack Pointer → return address.

C function local variables: `UINT16* eepromAddr, UINT16 dummy`.

On return, accumulator B contains 0 if the command executed correctly and the EEPROM verified as erased, or 1 if the command failed or the EEPROM was not erased.

`ESTAT.byte = ACCERR|PVIOL;`	`MOVB  #$30,$115`	`Clear error flags`
`if(ESTAT.bit.cbeif == 1)` `{`	`BRCLR $115,#$80,eeevf`	`Check command buffer is empty`
`  *eepromAddr = dummy;`	`STD   0,X`	`Write any data to EEPROM address`
`  ECMD.byte = ERASE_VERIFY;`	`MOVB  #$05,$116`	`Write mass erase command`
`  ESTAT.byte = CBEIF;`	`MOVB  #$80,$115`	`Write '1' to CBEIF to launch the command`

`  if((ESTAT.byte & (ACCERR|PVIOL))!= 0)`
`  {`
`    return(FAIL);`
`  }`
	`BRSET $115,#$30,eeevf`	`Command failed if either error flag set`

`  while(ESTAT.bit.ccif != 1)`
`  {`
`  }`
	`BRCLR $115,#$40,*+0`	`Wait for command to finish: the BLANK flag is not valid until CCIF is set.`

`  if(ESTAT.bit.BLANK == 1)`	`BRCLR $115,#$04,eeevf`	`Check BLANK bit`

`  {`
`    return(PASS);`
`  }`
	`CLRB`
	`BRA  eeevrtn`
		`Successful, return`

`  else`
`  {`
`    return(FAIL)`
`  }`
`}`
`else`
`{`
`  return(FAIL);`
`}`
	`eeevf:`
	`LDAB  #1`
	`eeevrtn:`
	`RTS`
		`Fail, return`

EEPROM Sector Modify Command

The following code segment demonstrates how to reprogram a sector (4 bytes) of EEPROM using the sector modify command followed by a pipelined program command.

The leftmost column contains C code (variable definitions in **Appendix A 'C' Variable Definitions**), the centre column contains equivalent assembly code assuming the register base address is $0000, and the rightmost column contains comments.

Prerequisites: ECLKDIV must be configured correctly, the sector to be programmed must not be protected, the EEPROM address must be word aligned (bit 0 = 0).

Registers: X contains word aligned EEPROM sector address to be reprogrammed, Y contains address of first word of data.

Stack Pointer → return address.

C function local variables: `UINT16* progAddr, UINT16* dataAddr`.

On return, accumulator B contains 0 if the commands executed correctly, or 1 if either command failed.

`ESTAT.byte = ACCERR|PVIOL;`	`MOVB  #$30,$115`	`Clear error flags`
`if(ESTAT.bit.cbeif == 1)` `{`	`BRCLR $115,#$80,eesmf`	`Check command buffer is empty`
`  *progAddr = *dataAddr;`	`LDD   0,Y` `STD   0,X`	`Write first data word to first word of EEPROM sector`
`  ECMD.byte = SECTOR_MODIFY;`	`MOVB  #$60,$116`	`Write sector modify command`
`  ESTAT.byte = CBEIF;`	`MOVB  #$80,$115`	`Write '1' to CBEIF to launch the command`

`  if((ESTAT.byte & (ACCERR|PVIOL))!= 0)`
`  {`
`    return(FAIL);`
`  }`
	`BRSET $115,#$30,eesmf`	`Command failed if either error flag set`

`  while(ESTAT.bit.ccif != 1)`
`  {`
`  }`
	`BRCLR $115,#$80,*+0`	`Wait for buffers to empty`

`  *(progAddr+1) = *(dataAddr+1);`	`LDD   2,Y` `STD   2,X`	`Write second data word to second word of EEPROM sector`
`  ECMD.byte = PROG`	`MOVB  #$20,$116`	`Write program command`
`  ESTAT.byte = CBEIF;`	`MOVB  #$80,$115`	`Write '1' to CBEIF to launch the command`

`  if((ESTAT.byte & (ACCERR|PVIOL))!= 0)`
`  {`
`    return(FAIL);`
`  }`
	`BRSET $115,#$30,eesmf`	`Command failed if either error flag set`

`  while(ESTAT.bit.ccif != 1)`
`  {`
`  }`
	`BRCLR $115,#$40,*+0`	`Wait for command to finish: this is optional, but the EEPROM cannot be accessed until CCIF is set.`

`  return(PASS);` `}`	`CLRB` `BRA  eesmrtn`	`Successful, return`

`else`
`{`
`  return(FAIL);`
`}`
	`eesmf:`
	`LDAB  #1`
	`eesmrtn:`
	`RTS`
		`Fail, return`

NVM Security

HCS12 microcontrollers offer a memory security feature. This security feature is designed to prevent unauthorised access to the non-volatile memory. Note that memory security is not the same as memory protection, which is designed to prevent accidental modification of the NVM and is discussed in section **Flash Memory Protection**.

Secured Operation

The memory contents are secured by programming the security bits within the Flash Options/Security byte at address $FF0F. The Flash Options/Security byte is located within the Flash memory at address $FF0F and is erased and programmed like any other Flash location. On devices that have a memory page window, the Flash Options/Security byte is also available at address $BF0F by selecting page $3F with the PPAGE register. The contents of this byte are copied into the Flash Security Register (FSEC) during a reset sequence.

The Flash sector $FE00 to $FFFF must be erased before the Flash Options/Security byte is programmed. The Flash is programmed by aligned word only, so address $FF0E must be written as the word address to be programmed, to program the Flash Options/Security byte. The Flash Options/Security byte can only be erased or programmed when this sector is not protected (see **Flash Memory Protection**).

	7	6	5	4	3	2	1	0
$FF0F	KEYEN1	KEYEN0	NV5	NV4	NV3	NV2	SEC1	SEC0

Figure 23. Flash Options/Security Byte

KEYEN[1:0] – Backdoor Key Enable Bits

The KEYEN[1:0] bits define the enabling of the Backdoor Key access, as shown in **Table 8. Backdoor Key States**.

Table 8. Backdoor Key States

KEYEN[1:0]	Description
00	Disabled
01	Disabled
10	Enabled
11	Disabled

NOTE: *Some older versions of HCS12 have only a single Backdoor Key Enable bit, KEYEN, equivalent to KEYEN1. In this case, KEYEN = 0 means disabled and KEYEN = 1 means enabled.*

SEC[1:0] – Memory Security Bits

The SEC[1:0] bits define the security state of the microcontroller at reset as shown in **Table 9. Security States**.

Table 9. Security States

SEC[1:0]	Description
00	Secured
01	Secured
10	Unsecured
11	Secured

Secured operation takes effect on the next reset after programming the security bits to a secure value. When enabled, secured operation has the following effects on the microcontroller:

Normal Single Chip Mode

- Background Debug Module (BDM) operation is completely disabled.
- Flash and EEPROM commands PROG, Mass Erase, Sector Erase, Erase Verify and Sector Modify remain enabled.

Special Single Chip Mode

- BDM firmware commands are disabled.
- BDM hardware commands are restricted to the register space.
- Flash and EEPROM commands limited to MASS ERASE only.

Expanded Modes

- BDM operation is completely disabled.
- External access to internal Flash and EEPROM is disabled.
- Internal visibility (IVIS) and CPU pipe (IPIPE) information is disabled.
- Flash and EEPROM commands cannot be executed from external memory in Normal Expanded mode.

By these actions, unauthorised access to the EEPROM and Flash memory contents can be prevented. However, it must be realised that the security of the EEPROM and Flash memory contents also depends on the design of the application program. For example, if the application has the capability of downloading code through a serial port and then executing that code (e.g. an application containing bootloader code), then this capability could potentially be used to read the EEPROM and Flash memory contents even when the microcontroller is in the secure state. In this example, the security of the

application could be enhanced by requiring a challenge/response authentication before any code can be downloaded.

Unsecuring the Microcontroller

When secure, the microcontroller can be unsecured by one of the following methods.

Backdoor Key Access

In Normal modes (Single Chip and Expanded), security can be temporarily disabled by means of the Backdoor Key access method. This method requires that:

- The Backdoor Key at $FF00 to $FF07 has been programmed to a valid value.
- The KEYEN[1:0] bits within the Flash Options/Security byte select 'enabled'.
- In Single Chip mode, the application program programmed into the microcontroller must be designed to have the capability to write to the Backdoor Key locations.

The Backdoor Key values themselves would not normally be stored within the application data, and so the application program would have to be designed to receive the Backdoor Key values from an external source, through a serial port for example. It is not possible to download the backdoor keys using Background Debug Mode.

The Backdoor Key Access method is useful because it allows debugging of a secured microcontroller, without having to erase the Flash. This is particularly useful for failure analysis.

NOTE: *No word of the Backdoor Key is allowed to have the value $0000 or $FFFF.*

Backdoor Key Access Sequence:

1. Set the KEYACC bit in the Flash Configuration register FCNFG.
2. Write the first 16-bit word of the backdoor key to $FF00.
3. Write the second 16-bit word of the backdoor key to $FF02.
4. Write the third 16-bit word of the backdoor key to $FF04.
5. Write the fourth 16-bit word of the backdoor key to $FF06.
6. Clear the KEYACC bit in the Flash Configuration register FCNFG.

If all four 16-bit words match the Flash contents at $FF00 to $FF07, the microcontroller will be unsecured and the security bits SEC[1:0] in the Flash Security register FSEC will be forced to the unsecured state, '10'. The contents of the Flash Options/Security byte are not changed by this procedure, and so the microcontroller will revert to the secure state after the next reset, unless further action is taken as detailed below.

If any of the four 16-bit words do not match the Flash contents at $FF00 to $FF07, the microcontroller will remain secured.

Reprogramming the Security Bits

In Normal Single Chip Mode, security can also be disabled by means of erasing and reprogramming the security bits within Flash Options/Security byte to the unsecured value. As the erase operation will erase the entire sector from $FE00 to $FFFF, the Backdoor Key and the interrupt vectors will also be erased and so this method is not recommended for Normal Single Chip mode. The application software can only erase and program the Flash Options/Security byte if the Flash sector containing the Flash Options/Security byte is not protected (see Flash Protection). Thus Flash protection is a useful means of preventing this method. The microcontroller will enter the unsecured state after the next reset following the programming of the security bits to the unsecured value.

This method requires:

- That the application software previously programmed into the microcontroller has been designed to have the capability to erase and program the Flash Options/Security byte, or
- That security is first disabled using the Backdoor Key method, allowing BDM to be used to issue commands to erase and program the Flash Options/Security byte, and
- The Flash sector containing the Flash Options/Security byte is not protected.

Complete Memory Erase (Special modes)

The microcontroller can be unsecured in Special modes by erasing the entire EEPROM and Flash contents.

When a secure microcontroller is reset into Special Single Chip mode, the BDM firmware verifies whether the EEPROM and Flash are erased. If any EEPROM or Flash address is not erased, only BDM hardware commands are enabled. BDM hardware commands can then be used to write to the EEPROM and Flash registers, and so to Mass Erase the EEPROM and all Flash blocks.

When next reset into Special Single Chip mode, the BDM firmware will again verify whether all EEPROM and Flash are erased, and this being the case, will enable all BDM commands, allowing the Flash Options/Security byte to be programmed to the unsecured value. The security bits SEC[1:0] in the Flash Security register will indicate the unsecure state following the next reset.

Special Single Chip Erase and Unsecure:

1. Reset into Special Single Chip mode
2. Write an appropriate value to the ECLKDIV register for correct timing.
3. Write $FF to the EPROT register to disable protection.

4. Write $30 to the ESTAT register to clear the PVIOL and ACCERR bits.
5. Write $0000 to the EDATA register ($011A–$011B)
6. Write $0000 to the EADDR register ($0118–$0119)
7. Write $41 (Mass Erase) to the ECMD register.
8. Write $80 to the ESTAT register to clear CBEIF.
9. Write an appropriate value to the FCLKDIV register for correct timing.
10. Write $00 to the FCNFG register to select Flash block 0.
11. Write $10 to the FTSTMOD register ($0102) to set the WRALL bit, so the following writes affect all flash blocks.
12. Write $FF to the FPROT register to disable protection.
13. Write $30 to the FSTAT register to clear the PVIOL and ACCERR bits.
14. Write $0000 to the FDATA register ($010A–$010B)
15. Write $0000 to the FADDR register ($0108–$0109)
16. Write $41 (Mass Erase) to the FCMD register.
17. Write $80 to the FSTAT register to clear CBEIF.
18. Wait until all CCIF flags are set.
19. Reset back into Special Single Chip mode
20. Write an appropriate value to the FCLKDIV register for correct timing.
21. Write $00 to the FCNFG register to select Flash block 0.
22. Write $FF to the FPROT register to disable protection.
23. Write $FFBE to Flash address $FF0E
24. Write $20 (Program) to the FCMD register.
25. Write $80 to the FSTAT register to clear CBEIF.
26. Wait until the CCIF flag in FSTAT is are set.
27. Reset into any mode.

Appendix A ‘C’ Variable Definitions

```
#define PASS 0u                                          /*function return values */
#define FAIL 1u

#define REG_BASE 0x0000                                   /*register base address */

#define BLANK                   0x04                      /*FSTAT/ESTAT bit masks */
#define ACCERR                  0x10
#define PVIOL                   0x20
#define CCIF                    0x40
#define CBEIF                   0x80
```

```
#define ERASE_VERIFY                0x05                                /*FCMD/ECMD commands*/
#define PROG                        0x20
#define ERASE                       0x40
#define MASS_ERASE                  0x41
#define MODIFY                      0x60

typedef unsigned char               UINT8;                                    /*basic types */
typedef unsigned short              UINT16;
typedef signed char                 INT8;
typedef signed short                INT16;

typedef union                                                          /*MCU register types */
{
  UINT8          byte;
  struct
  {
    UINT8                           :2;                                        /*not used */
    UINT8 blank                     :1;                                 /*blank verify flag */
    UINT8                           :1;                                        /*not used */
    UINT8 accerr                    :1;                                 /*access error flag */
    UINT8 pviol                     :1;                          /*protection violation flag */
    UINT8 ccif                      :1;                      /*command complete interrupt flag */
    UINT8 cbeif                     :1;                  /*command buffer empty interrupt flag */
  }bit;
}tFSTAT;

typedef union
{
  UINT8          byte;
  struct
  {
    UINT8                           :8;                                 /*bitfield not used */
  }bit;
}tFCMD;

                                                                      /*MCU register variables */
volatile tFSTAT                     FSTAT   @(REG_BASE + 0x105);     /*Flash status register */
volatile tFCMD                      FCMD    @(REG_BASE + 0x106);      /*Flash command buffer */
volatile tFSTAT                     ESTAT   @(REG_BASE + 0x115);    /*EEPROM status register */
volatile tFCMD                      ECMD    @(REG_BASE + 0x116);     /*EEPROM command buffer */
```

HOW TO REACH US:

USA/EUROPE/LOCATIONS NOT LISTED:

Motorola Literature Distribution;
P.O. Box 5405, Denver, Colorado 80217
1-303-675-2140 or 1-800-441-2447

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

TECHNICAL INFORMATION CENTER:

1-800-521-6274

HOME PAGE:

http://motorola.com/semiconductors

This product incorporates Superflash® technology licensed from SST