

# A Utility for Programming Single FLASH Array HCS12 MCUs, with Minimum RAM Overhead

By **Jim Williams**  
8/16-Bit Applications Engineering  
Austin, Texas

---

## Introduction

The intent of this application note is to introduce a standard method for implementing HCS12 FLASH erase and programming functions for devices with single FLASH array implementations. The example presented here enables the HCS12 user to program / erase FLASH locations conveniently from user code.

In the HCS12 Family, there are different implementations of the various FLASH memories in several devices. For example, the MC9S12DP256 device has four 64K array blocks implementing the 256K total FLASH space. (For more detailed information on FLASH array block(s), see Freescale application note AN1837.) On the MC9S12DP256 device, the user could code program and erase routines to reside in one of the four array block(s) and call this code to perform functions on the other three blocks. This solution becomes limited when the application demands that the fourth array block(s) be re-program-mable, or if the application is based on smaller FLASH devices such as the single array MC9S12C32.

## Flash.c

Overcoming these limitations is the focus of this application note. The described method is based on a concept presented in Freescale application note AN2548 entitled *HCS12 Serial Monitor* but elaborated here such that it has a callable C wrapper and supports a more powerful implementation. The idea behind this concept is to create a small assembly routine that can be copied to the device's software stack, executed from there, and then removed. This concept of programming and erasing from the stack is important in the embedded controller environment because memory resources are directly related to device cost. Alternative methods involve holding a routine in FLASH memory and copying it to RAM space for execution. In order for this to work, a

portion of the user RAM must be reserved to hold this routine. This buffer space will never be available for system use. This is a waste of a precious resource for a function that might be performed once or twice in a product's lifetime.

The FLASH software is contained in two files:

- The Flash.c routines are C callable functions interfacing to the Do\_On\_Stack.asm routine.
- The DoOnStack routine handles the movement of the SpSub routine to the system stack and then the SpSub handles the actual programming and erasing.

Limitations:

Assuming that the interrupt service routines are stored in the preserved upper bank. Interrupts will be disabled for ~20 ms during the erase function and ~46  $\mu$ s for the programming (code execution time ignored and assuming a 200 kHz FLASH clock).

---

## Flash.c

The following functions are contained the flash.c file to facilitate FLASH programming of the HCS12 FLASH:

- Flash\_Init
- Flash\_Write\_Word
- Flash\_Erase\_Sector
- Flash\_Write\_Block
- Flash\_Erase\_Block

Details of the calling and return values are shown below.

### Flash\_Init()

The function Flash\_Init must be called prior to utilizing any of the FLASH programming or erase functions. This function initializes the clock speed of the HCS12 FLASH state machine. Its prototype is:

```
void Flash_Init(unsigned long oscclk);
```

It is called, as shown below, where 8000 kHz (8 MHz) is the oscillator frequency of the system.

```
(void) Flash_Init(8000);
```

The Flash\_Init function does not require any assembly level calls, and only needs to be called once during application setup. Additional calls of the function will not affect system operation as the register affected is write once. The initialization will remain valid until the next system reset.

## Flash\_Erase\_Sector()

The Flash\_Erase\_Sector function erases the entire sector pointed to by the passed address. Care must be taken to ensure that there is no data or code of value remaining in this sector. Currently HCS12 devices either have a sector size of 512 or 1024 bytes. All of these sector locations will be erased to the un-programmed state by this function.

For code updating it is not usually necessary to save any part of a sector, but for data storage areas this might be important. The application software has two choices:

1. Buffer the sector in RAM either on the stack or in a buffer, or
2. Use the Flash\_Write\_Block function (described below) to move any valid data to an unused FLASH sector while the current sector is erased.

In an embedded environment the second option is most often the best choice.

The prototype for the Flash\_Erase\_Sector function is:

```
signed char Flash_Erase_Sector(unsigned int *far far_address);
```

Returned values are:

Successful (1)

Errors:

Flash\_Odd\_Access (-1)

Access\_Error (-3)

Protection\_Error (-4)

Not\_StartofSector\_Error (-5)

It may be called in several ways:

```
(void) Flash_Erase_Sector ((unsigned int *far) 0x3FF400);
```

Or:

```
(void) Flash_Erase_Sector (Start_of_Sector);
```

Or:

```
(void) Flash_Erase_Sector (&Flash_Parameter1);
```

In the first example, the absolute address must be type cast with the \*far type so that the compiler recognizes to treat this as a far pointer. (24 bit)

The address passed to the Flash\_Erase\_Sector function is checked to make sure that it is the first address of the sector. This is done as a safety precaution. This check may be removed from the code without issue if the programmer does not wish this restriction.

### Flash\_Write\_Word()

The Flash\_Write\_Word function writes a single 16-bit word of data, passed to the function, to the address passed to the function.

The prototype for the Flash\_Write\_Word function is:

```
signed char Flash_Write_Word (unsigned int *far address, \
                             unsigned int data);
```

Returned values are:

Successful	(1)
------------	-----

Errors:

Flash_Odd_Access	(-1)
Flash_Not_Erased	(-2)
Access_Error	(-3)
Protection_Error	(-4)

It may be called as above in several ways:

```
(void) Flash_Write_Word ((unsigned int *far)
0x3FF400),0xABCD);
```

Or:

```
(void) Flash_Write_Word (&Flash_Parameter1,&Ram_data1);
```

In the first example, the absolute address again must be type cast with the \*far type so that the compiler recognizes to treat this as a far pointer. (24 bit)

The second example moves a storage word in RAM to a word in FLASH. This function is restricted to word values only. To move byte data, the application must build the desired word from two desired bytes. This must be done as the HCS12 FLASH only supports word writes. Additionally, long operands (32 bit) must be split by the application into two separate function calls.

**Flash\_Erase\_Block()**

The Flash\_Erase\_Block function is an extension of the Flash\_Erase\_Sector function it erases multiple sectors starting at the passed address on the first sector, to the end of the sector of the ending address passed to the function. Care must be taken with this function as noted above with the Flash\_Erase\_Sector function as it erases a location from the start address to the end address + the sector size. (512 or 1024 bytes).

The prototype is:

```
signed char Flash_Erase_Block (unsigned int *far start_address,\
                               unsigned int *far end_address);
```

Returned values are:

Successful (1)

Errors:

Flash\_Odd\_Access (-1)  
 Access\_Error (-3)  
 Protection\_Error (-4)  
 Not\_StartofSector\_Error (-5)

It may be called as above in several ways:

```
(void) Flash_Erase_Block ((unsigned int *far) 0x3CB400, \
                           (unsigned int *far) 0x3CBC00);
```

Or:

```
(void) Flash_Erase_Block(Start_of_Sector1,Start_of_Sector3);
```

Assuming in the second example that the Start\_of\_Sector1 and Start\_of\_Sector3 pointers are initialized correctly, the above call will erase all sectors from Start\_of\_Sector1 to the end of the sector containing Start\_of\_Sector3.

This function only supports a 16K bank per call. Addresses should never cross a page boundary. (0x7FFF-0x8000 or 0xBFFF-0xC000) Doing so will cause unexpected erasures of memory.

### Flash\_Write\_Block()

The Flash\_Write\_Block function will move a number of words from a source address to a destination address in FLASH memory.

The prototype for the Flash\_Write\_Block function is:

```
signed char Flash_Write_Block(unsigned int *far address_source,\
                             unsigned int *far far_address_destination,\
                             unsigned int count);
```

Returned values are:

Successful	(1)
------------	-----

Errors:

Flash_Odd_Access	(-1)
Flash_Not_Erased	(-2)
Access_Error	(-3)
Protection_Error	(-4)

An example of a simple absolute call of the function:

```
(void) Flash_Write_Block ((unsigned int *far) 0x2000, (unsigned int *far)
0x3FF400,4);
```

This will move 4 hex words from RAM at 0x2000 to FLASH at 0x3FF400. Although not the most useful of functions, it demonstrates the format of the call with banked parameters.

A more representative call of the function might be something like:

```
(void) Flash_Write_Block ( (unsigned int *far) &Ram_Storage_Area, \
                          (unsigned int *far) &Flash_Storage_Area, \
                          sizeof (Ram_Storage_Area)/2);
```

The software source code for these FLASH functions is shown below.

```

/*****
 *
 *          Freescale
 *
 *   DESCRIPTION:   S12 single array Flash routines
 *   SOURCE:       flash.c
 *   COPYRIGHT:    © 04/2004  Made in the USA
 *   AUTHOR:      rat579
 *   REV. HISTORY: (none)
 *
 *****/
#include "projectglobals.h"
#include "flash.h"

extern DoOnStack(unsigned int *far address);

/*****
/** Function Name: Flash_Init
/** Description : Initialize Flash NVM for programming
/**             FCLKDIV based on passed sys/2 (FlashClk) frequency, then
/**             uprotect the array, and finally ensure PVIOL and
/**             ACCERR are cleared by writing to them.
/**
 *****/
volatile int flash_init(unsigned long FlashClk)
{
    unsigned char fclk_val;

    if ((MCF_CFM_CFMCLKD && MCF_CFM_CFMCLKD_DIVLD) != 1)
    {
        /* Next, initialize FCLKDIV register to ensure we can program/erase */
        if (FlashClk >= 12000) {
            fclk_val = (unsigned char) (FlashClk/8/200 + 1);
            /* PRDIV8 set since above 12 MHz clock */
            MCF_CFM_CFMCLKD |= fclk_val | MCF_CFM_CFMCLKD_PDIV8;
        }
        else
        {
            fclk_val = (unsigned char) (FlashClk/200 + 1);
            MCF_CFM_CFMCLKD |= fclk_val;
        }
    }

    MCF_CFM_CFMSTAT = 0x00; /*Disable all protection (if LOCK not set) */
    MCF_CFM_CFMSTAT |= (PVIOL|ACCERR); /* Clear any errors */
    return(0);
}

/*****
/** Function Name: Flash_Write_Word
/** Description : Program a given Flash location using address and data
/**             passed from calling function.
/**
 *****/
signed char Flash_Write_Word(unsigned int *far far_address, unsigned int data)
{
    unsigned int* address;
    address = (unsigned int*)far_address;                // strip page off
    Flash.fstat.byte = (ACCERR | PVIOL);                // clear errors
    if((unsigned int)address & 0x0001) {return Flash_Odd_Access;} // Aligned word?
    if(*far_address != 0xFFFF) {return Flash_Not_Erased;} // Is it erased?
}

```

## Flash.c

```

        (*address) = data;          // Store desired data to address being programmed

Flash.fcmd.byte = PROG;            // Store programming command in FCMD
(void)DoOnStack(far_address);     // just passed for PPAGE
if (Flash.fstat.bit.accerr) {return Access_Error;}
if (Flash.fstat.bit.pviol) {return Protection_Error;}
return 1;
}
/*****
/* Function Name: Flash_Write_Block
/* Description : Program a range of Flash location using address and data
/*                pointers passed from calling function.
/*
/*
/*****
signed char Flash_Write_Block(unsigned int *far address_source,\
                             unsigned int *far far_address_destination,\
                             unsigned int count)
{
unsigned long i; // long supports > 64K words
signed char ret_val;

    for (i = 0;i<count;i++)
    {
        ret_val = Flash_Write_Word(far_address_destination++, *address_source++);
        if (ret_val == Access_Error) {return Access_Error;}
        if (ret_val == Protection_Error) {return Protection_Error;}
    }
    return 1;
}
/*****
/* Function Name: Flash_Erase_Sector
/* Description : Erases a given Flash sector using address
/*                passed from calling function.
/*
/*
/*****
signed char Flash_Erase_Sector(unsigned int *far far_address)
{
unsigned int* address;
    address = (unsigned int*)far_address;          // strip page off
    if((unsigned int)address & 0x0001) {return Flash_Odd_Access;} // Aligned word?
    if((unsigned int)address % Flash_Sector_Size !=0) {return Not_StartofSector_Error;}
    Flash.fstat.byte = (ACCERR | PVIOL);          // clear errors
    (*address) = 0xFFFF; /* Dummy store to page to be erased */

    Flash.fcmd.byte=ERASE;
    (void)DoOnStack(far_address);
    if (Flash.fstat.bit.accerr) {return Access_Error;}
    if (Flash.fstat.bit.pviol) {return Protection_Error;}
    return 1;
}
/*****
/* Function Name: Flash_Erase_Block
/* Description : Erases a range of Flash sectors using address
/*                pointers passed from calling function.
/*
/*
/* S12 have max 64 page and max of 32 sectors per page (800 total sectors)
/*****
signed char Flash_Erase_Block(unsigned int *far start_address,\
                             unsigned int *far end_address)
{
unsigned int i;
unsigned int count;
unsigned long address;

```



```

signed char ret_val;

count = (((unsigned int)end_address)-((unsigned int)start_address))/Flash_Sector_Size)+1;

address = (unsigned long)start_address;

    for (i = 0;i < count;i++)
    {
        ret_val = Flash_Erase_Sector((unsigned int *far)address);
        if (ret_val == Access_Error) {return Access_Error;}
        if (ret_val == Protection_Error) {return Protection_Error;}
        address = address+Flash_Sector_Size;
    }
    return 1;
}

```

---

```

/*****
 *                               Freescale
 *
 *   DESCRIPTION:   Header file for S12 single array Flash routines
 *   SOURCE:       flash.h
 *   COPYRIGHT:    © 04/2004  Made in the USA
 *   AUTHOR:      rat579
 *   REV. HISTORY: (none)
 *
 *****/
#ifndef FLASH_H          /* Prevent duplicated includes */
#define FLASH_H

/* Functions from flash.c */
void Flash_Init(unsigned long oscclk);
signed char Flash_Write_Word(unsigned int *far address, unsigned int data);
signed char Flash_Erase_Sector(unsigned int *far far_address);
signed char Flash_Write_Block(unsigned int *far address_source,\
                             unsigned int *far far_address_destination,\
                             unsigned int count);
signed char Flash_Erase_Block(unsigned int *far start_address,\
                             unsigned int *far end_address);

/* Error codes */
#define Flash_Odd_Access          -1
#define Flash_Not_Erased          -2
#define Access_Error              -3
#define Protection_Error          -4
#define Not_StartofSector_Error  -5

#endif /*FLASH_H*/

```

---

---

## Do\_On\_Stack.asm

The DoOnStack: subroutine (which is located in the FLASH), copies a small routine (SpSub:) onto the stack (in RAM) and then passes control to that subroutine on the stack. When the operation is finished, an RTS returns control from the DoOnStack: routine. This de-allocates the space used by the small stack routine and then returns to the program from which it was called.

The first line in DoOnStack: saves the B register on the stack. (This register contains the PPAGE value and must be saved.) The SpMoveLoop copies the SpSub: routine onto the stack (with a series of PSHD instructions) starting with the last word of SpSub: and ending with the push of the first word of SpSub: onto the stack. At this point, the stack pointer points to the location of the first word of the stacked SpSub: routine.

The TFR in the next line copies the SP into the IX register so IX points to the start of the copy of SpSub: on the stack. The next line pre-loads A with a mask corresponding to the CBEIF bit (command buffer empty interrupt flag [0x80]) which will be used to start the FLASH command. (Setting the CBEIF causes the command loaded into the FCMD register to start execution.)

The CALL 0,x,00 instruction jumps to the copy of SpSub: that is now located on the stack. Due to this use of SP relative addressing, the DoOnStack: routine may be located in banked or non-banked memory. This is because the CALL instruction stores the return PPAGE address needed for a return from banked or non-banked memory on the stack, allowing program execution to continue.

The SpSub: subroutine is written in a position-independent manner so it can be copied to a new location (on the stack) and will still execute as expected. SpSub: is such a short subroutine that it is easy to make it position independent.

The first task of the SpSub: routine is to get the current PPAGE register value, which will be required for the return of calling functions and save it in the unused IY register. This must be accomplished via the SEX (signed extended transfer) instruction as the high byte of the IY register must be cleared for future restoration.

Next, the PPAGE address is passed from the calling C function and pushed onto the stack, recovered, and stored into the PPAGE register, selecting the desire page to be programmed / erased.

At this point the I bit in the CCR may or may not be set to mask interrupts. At this time interrupts must be disabled, as the FLASH and hence the interrupt vector table, will go off-line during programming / erase operations. The next line saves the CCR (and the I bit) in the B register for restoration after the FLASH command is completed. This is the absolute minimum that interrupts can be disabled. (About 50  $\mu$ s for a word program and 20 ms for a sector erase function.)

SpSub: completes the FLASH command by setting the CBEIF bit in FSTAT (via the mask saved in the A register above). The series of NOPs in the routine is used to ensure that the FLASH state machine has registered the command before polling begins. This delay is required so the internal FLASH command sequencer can properly update the CBEIF and CCIF flags in FSTAT. Execution stays in the ChkDone loop until the command finishes (CCIF becomes set).

At this point, the calling function PPAGE value is restored from the IY register and the CCR is restored from to B register. As the FLASH is back in the memory map and we can return to DoOnStack: (which is in FLASH), the RTC in the SpSub: returns to the DoOnStack: routine.

Upon return, the DoOnStack: de-allocates the space used by the small stack routine and then returns to the program from which it was called.

Refer to the following code listing.

```

;*****
;*
;*           Freescale
;*
;*   DESCRIPTION:   S12 single array Flash routines
;*   SOURCE:       flash.c
;*   COPYRIGHT:    © 04/2004  Made in the USA
;*   AUTHOR:      rat579
;*   REV. HISTORY: 060304 - fixed CCR return value and optimized
;*                   in SpSub routine
;*
;*****/
;*****/
; Local defines
;*****
        CBEIF      EQU      $80
        FSTAT      EQU      $105
        FCMD       EQU      $106
        CCIF       EQU      $40
        PAGE_ADDR  EQU      $30
                xdef DoOnStack
;*****
;* DoOnStack - copy SpSub onto stack and call it (see also SpSub)
;* De-allocates the stack space used by SpSub after returning from it.
;* Allows final steps in a 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 word-program or erase commands
;*
;* Calling Convention:
;*         jsr     DoOnStack
;*
;* Uses 32 bytes on stack + 3 bytes if Call instruction used
;*****
DoOnStack:
        pshb                                ;save B - PPAGE
        ldx    #SpSubEnd-2                  ;point at last word to move to stack
SpmoveLoop: ldd    2,x-                      ;read from flash
        pshd                                ;move onto stack
        cpx    #SpSub-2                    ;past end?
        bne    SpmoveLoop                  ;loop till whole sub on stack
        tfr    sp,x                        ;point to sub on stack
        ldaa   #CBEIF                      ;preload mask to register command
        call   0,x,00                      ;execute the sub on the stack
        leas  SpSubEnd-SpSub,sp            ;de-allocate space used by sub
        pulb                                ;restore B
        rtc                                ;to flash where DoOnStack was called
                                           ; assume banked calling function

;*****
;* SpSub - register flash command and wait for Flash CCIF
;* this subroutine is copied onto the stack before executing
;* because you can't execute out of flash while a flash command is
;* in progress (see DoOnStack to see how this is used)
;*
;* Note: must be even # of bytes!
;*
;* Uses 32 bytes on stack + 2 bytes for JSR above
;*****
    
```

## Implementation

```

EVEN                                ;Make code start word aliened
SpSub:
    ldab    SpSubEnd-SpSub+2,sp    ;get PPAGE back
    stab    PAGE_ADDR              ;Store the PPAGE address
    tfr     ccr,b                  ;get copy of ccr
    orcc    #$10                  ;disable interrupts
    staa    FSTAT                  ;[Pw0] register command
    nop     ;[0] wait min 4~ from w cycle to r
    nop     ;[0]
    nop     ;[0]
    brclr   FSTAT,CCIF,*          ;[rfPPP] wait for queued commands to finish
    tfr     b,ccr                 ;restore ccr and int condition
    rtc     ;back into DoOnStack in flash
SpSubEnd:

```

---

## Implementation

Pseudo code for upper level implementation could be as shown below. It assumes that a sign char Return\_val is defined to save the status from the FLASH programming, and a word Buffer\_size is defined to save a number of bytes in the buffer. Optimally the Buffer\_size should equal the FLASH sector size (512 or 1024). All must be defined prior to starting the procedure. If this software will be ported to different HCS12 devices, it is suggested a global define Flash\_Sector\_Size as 0x200 or 0x400 be defined. This is done in order to identify the sector size of the device, as this information can not be read from the device.

Word Buffer\_size;

Signed Char Return\_val;

1. Set the Destination location:

```
Destination = 0x3E8000;    // Page 0x3E address 0x8000
```

2. Load a RAM buffer from the receiving device — serial port for instance:

```
Buffer_size = Get_SCI_data (Buffer);    // <- User Software
```

3. Erase target sector:

```
Flash_Erase_Sector (&Destination);
```

4. Make sure there is an even number of bytes:

```
if (Buffer_size & 0x0001)
    Buffer_size ++;
```

5. Program the buffer to FLASH:

```
Return_val = Flash_Write_Block((unsigned int *far)Buffer, \
                                Destination, (Buffer_size /2));
```

6. If flashing was completed, increment FLASH destination for next buffer:

```
if (Return_val > 0)
    Destination += Buffer_size /2;
```

Return to step one as many times as are needed to complete data loading, or until the complete flash page has been programmed.

---

## Conclusion

This code was designed and tested utilizing the Codewarrior development tools for the HC(S)12 with the HCS12 register stationery (header definitions) described in application note entitled *HCS12 Software Stationery* (Freescale document order number, AN2485). As only a single parameter is passed to the assembly routines, porting to other compilers should only require modifications to match target compilers' register stationery format.

## **How to Reach Us:**

### **Home Page:**

[www.freescale.com](http://www.freescale.com)

### **E-mail:**

[support@freescale.com](mailto:support@freescale.com)

### **USA/Europe or Locations Not Listed:**

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

### **Europe, Middle East, and Africa:**

Freescale Halbleiter Deutschland GmbH  
Technical Information Center  
Schatzbogen 7  
81829 Muenchen, Germany  
+44 1296 380 456 (English)  
+46 8 52200080 (English)  
+49 89 92103 559 (German)  
+33 1 69 35 48 48 (French)  
[support@freescale.com](mailto:support@freescale.com)

### **Japan:**

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

### **Asia/Pacific:**

Freescale Semiconductor Hong Kong Ltd.  
Technical Information Center  
2 Dai King Street  
Tai Po Industrial Estate  
Tai Po, N.T., Hong Kong  
+800 2666 8080  
[support.asia@freescale.com](mailto:support.asia@freescale.com)

### **For Literature Requests Only:**

Freescale Semiconductor Literature Distribution Center  
P.O. Box 5405  
Denver, Colorado 80217  
1-800-441-2447 or 303-675-2140  
Fax: 303-675-2150  
[LDCForFreescaleSemiconductor@hibbertgroup.com](mailto:LDCForFreescaleSemiconductor@hibbertgroup.com)

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

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

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

© Freescale Semiconductor, Inc. 2005, 2008. All rights reserved.