

# Using MSCAN on the HCS12 Family

By **Rebeca Delgado**  
**Antonio Ramos**  
**Luis Reynoso**  
**RTAC Americas**  
**Mexico 2005**

---

## Overview

This document is intended to provide embedded engineers with a quick reference to getting the MSCAN12 module up and running for any HCS12 MCU. Basic knowledge of the functional description and configuration options will give the user a better understanding on how the MSCAN12 module works.

This application note provides examples which illustrate one use of the MSCAN12 Module within the HCS12 Family of microcontrollers. The examples mentioned are intended to be modified to suit the specific needs of any application.

---

## Introduction

To use MSCAN12 module on the HCS12 Family many factors must be considered according to the implementation. The intention of this paper is to guide the first-time MSCAN12 user to properly configure the module on a basic mode independent from the physical layer and final application. Afterwards, with this reference the embedded engineer can tailor this implementation to more specific needs.

---

## MSCAN12 Description

The module is a communication controller implementing the CAN 2.0 A/B protocol as defined in the BOSCH specification<sup>1</sup>. It supports standard and extended data frames as well as remote frames with a programmable bit rate up to 1 Mbps<sup>2</sup>. The MSCAN12 has a triple transmit buffer scheme which allows multiple messages to be set up in advance and achieve an optimized performance. Received messages are stored in a five stage input FIFO. The module has a configurable hardware identifier filter which may be applied to incoming messages. A successful transmission or a message reception with a matching identifier will be flagged and can generate an interrupt request to the CPU. Separate signaling and interrupt capabilities for all CAN receiver and transmitter error states can be enabled as well.

Some other features on MSCAN12 are:

- Programmable wake-up functionality with integrated low-pass filter. This feature is used to avoid spurious wake-up signals when enabling the low-pass filter. When using this filter the wake-up signal must be asserted for a longer period in order to wake the module up from sleep mode.
- Programmable loop back mode supports self-test operation. This mode allows single node testing without having a CAN network available for node testing. This is used during the development phase making it easier for the user.
- Programmable listen-only mode for monitoring of the CAN bus
- Programmable MSCAN clock source, either bus clock or oscillator clock — This feature gives the user the option to select a clock source according to the frequency and tolerances needed for the application.
- Internal timer for time-stamping of successfully received and transmitted messages
- Low-power operation modes which include “sleep mode” and “low-power mode”

---

## MSCAN12 Registers

In order to get the MSCAN12 Module running, the following registers are configured:

- Control registers:
  - MSCAN12 control registers (CANCTL0, CANCTL1)
- CAN Module and bit timing:
  - MSCAN12 bus timing registers (CANBTR0, CANBTR1)
- Acceptance filters registers:
  - MSCAN12 identifier acceptance control register (CANIDAC)
  - MSCAN12 identifier acceptance registers (CANIDAR0–7)
  - MSCAN12 identifier mask registers (CANIDMR0–7)
- Interrupt registers
  - MSCAN receiver interrupt enable register (CANRIER)

---

1. Please refer to BCANPSV2.0 for a more detailed description of the CAN 2.0 A/B Protocol

2. Depends on the actual bit timing and the clock jitter of the PLL

---

## MSCAN Transmitter Interrupt Enable Register (CANTIER)

### Code and Explanation

The following C code is a basic example using a MC9S12DP256B that initializes the MSCAN12 module at 125 Kbps in loop back mode. This piece of code has a function that sends a standard data frame message with identifier 0x100. The data payload of the message is 8 bytes ('ABCDEFGH'). The reception of a message with matching identifier is handled by an interrupt routine and the transmission of a message is implemented using polling.

Since the module is configured in loop back mode, the MCU treats its own transmitted message as a message received from a remote node. This mode enables self-test operation, independent from any physical layer implementation. Identifier acceptance registers are configured to form four 16-bit filters that accept only data frames with standard identifiers 0x000 or 0x100.

#### *Initialization*

MSCAN initialization is done by the CANInit() routine.

Registers CANCTL1, CANBTR0, CANBTR1, CANIDAC, CANIDAR0–7, and CANIDMR0–7 can only be written by the CPU when the MSCAN is in initialization mode.

First, we have to get the module to enter initialization mode.

---

```
...
void CANInit (void)
    CAN0CTL0 = 0x01;                /* Enter Initialization Mode */
    while (! (CAN0CTL1&0x01)) {};   /* Wait for Initialization Mode
                                   acknowledge (INITRQ bit = 1) */
...
```

---

When going into initialization mode the module has to abort any ongoing transmission and lose bus synchronization. When the module has successfully entered initialization mode it activates the INITRQ bit at the CANCTL1 register. Once in initialization mode, the module is ready to accept the new set up.

---

```
...
    CAN0CTL1 = 0xA0;
    CANOBTR0 = 0xC7
    CAN0BTR1 = 0x3A
...
```

---

# MSCAN Transmitter Interrupt Enable Register (CANTIER)

CAN0CTL1, CAN0BTR0, and CANBTR1 registers deal with MSCAN Module’s timing and configuration. For this example, the module is working in loop back mode. This option is selected at the CAN0CTL1 register. See [Figure 1](#).

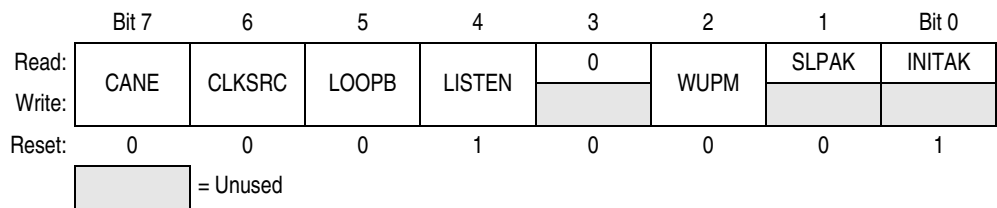


Figure 1. CAN0CTL1 Register

Loading CAN0CTL1 with the hexadecimal value of ‘A0’ results in loop back mode functioning. It also enables the module and sets the clock source to be the oscillator clock. Selecting the appropriate clock source is crucial for successful communications over the CAN bus. When working at high bit rates, a 50% duty cycle of the clock and low jitter is required. When using the PLL to boost the bus speed it is recommended to use the oscillator clock rather than the PLL generated clock to meet those requirements. The next step is calculating bit timings based on the clock selection.

The Module’s frequency is specified by the MSCAN bit timing registers. As the CAN specification states, bits are divided into segments. These segments are divided into single units of time called time quanta. In order to set the bus frequency it is necessary to set the length of the time quantum and how many time quanta are contained in the segments.

The first segment is always one time quantum long. This segment is called the synchronization segment and it is the period where signal transitions in the bus may occur. Following the synchronization segment there are the propagation time segment and the phase buffer segment 1. The propagation time segment is used to compensate for signal delays across the network and can be considered as the physical bus delay.

For this CAN implementation, the propagation time segment and phase buffer segment 1 are combined into one segment called timing segment 1. The last segment is the phase buffer segment 2 which in this implementation is called timing segment 2. These phase buffer segments could be shortened or lengthened in order to maintain synchronization using the re-synchronization jump width (SJW). [Figure 2](#) shows the different timing segments in the module and shows the size selected for each segment in this example.

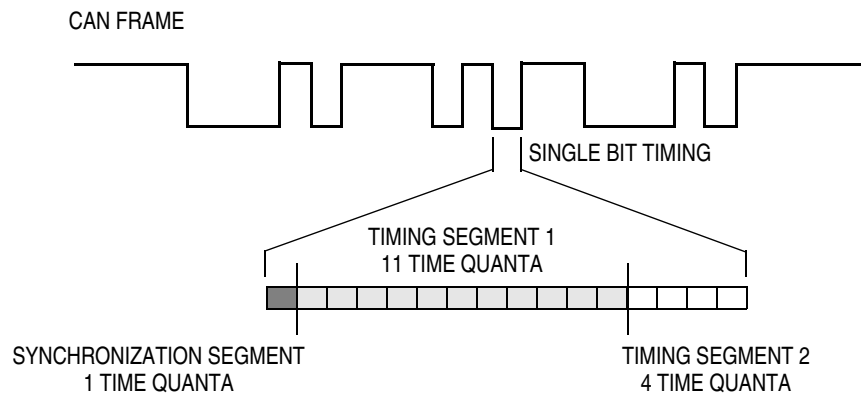


Figure 2. Time Quanta Frequency and Individual Bit Segment Sizes

For this example we selected timing segment 1 to be 11 time quanta and time segment 2 to be 4 time quanta. These values are stored at the CANBTR1 register in the following way:

|        | Bit 7 | 6      | 5      | 4      | 3      | 2      | 1      | Bit 0  |
|--------|-------|--------|--------|--------|--------|--------|--------|--------|
| Read:  | SAMP  | TSEG22 | TSEG21 | TSEG20 | TSEG13 | TSEG12 | TSEG11 | TSEG10 |
| Write: |       |        |        |        |        |        |        |        |

**Figure 3. CANBTR1 Register**

The resulting value for the register would be 0x3A. Having selected the above configuration, it is necessary to select an appropriate baud rate prescaler. The baud rate prescaler is used to divide the module's input frequency and the result is the time quantum's frequency. According to the following formulas:

$$\text{Number of Time Quanta} = \text{SYNC\_SEG} + \text{PROP\_SEG} + \text{PHASE\_SEG1} + \text{PHASE\_SEG2}$$

$$\text{TimingSegment1} = \text{PROP\_SEG} + \text{PHASE\_SEG1}$$

$$\text{TimingSegment2} = \text{PHASE\_SEG2}$$

$$\text{Number of Time Quanta} = 1 + \text{TimeSegment1} + \text{TimeSegment2}$$

$$\text{Bit Time} = \frac{(\text{Prescale value})}{f_{\text{CANCLK}}} \cdot \text{Number of Time Quanta}$$

$$\text{Bit Time} = \frac{(\text{Prescale value})}{f_{\text{CANCLK}}} \cdot (1 + \text{Time Segment 1} + \text{Time Segment 2})$$

This sample application was running on a board with an external oscillator at 16 MHz. The desired bit rate is 125 Kbps. Filling these values into the formula, we have the following:

$$\frac{1}{125 \text{ kHz}} = \frac{(\text{Prescale value})}{16 \text{ MHz}} \cdot 16$$

The resulting value for the prescaler would be eight in order to have a 125-kHz frequency.

The first two bits of the CANBTR0 register are used to select the synchronization jump width. This jump width selects the amount of time quanta that the phase buffer segments could be lengthened or shortened in order to keep up with the rest of the network. This amount of time quanta may not be longer than time segment 2.

|        | Bit 7 | 6    | 5    | 4    | 3    | 2    | 1    | Bit 0 |
|--------|-------|------|------|------|------|------|------|-------|
| Read:  | SJW1  | SKW0 | BRP5 | BRP4 | BRP3 | BRP2 | BRP1 | BPR0  |
| Write: |       |      |      |      |      |      |      |       |
| Reset: | 0     | 0    | 0    | 0    | 0    | 0    | 0    | 0     |

**Figure 4. CANBTR0 Register**

The resulting value for CANBTR0 would be 0xC7. In order to adjust the module to other frequencies the Baud Rate Prescaler or the number of Time Quanta in each segment could be modified.

## Filters

The MSCAN Module has eight pairs of filter registers. Each pair consist of an 8-bit mask register and an 8-bit acceptance register. These eight pairs of registers can be configured at the CAN0IDAC register to work in four different ways.

**Table 1. Table of Acceptance Modes**

| IDAM1 | IDAM0 | Identifier Acceptance Mode     |
|-------|-------|--------------------------------|
| 0     | 0     | Two 32-bit acceptance filters  |
| 0     | 1     | Four 16-bit acceptance filters |
| 1     | 0     | Eight 8-bit acceptance filters |
| 1     | 1     | Filter closed                  |

On reception, each message is written into the background receive buffer. The CPU is only signaled to read the message if it passes the criteria in the identifier acceptance and identifier mask registers. The background receive buffer gets masked by the identifier mask register and the result is compared with the identifier acceptance register. This means that the identifier mask register specifies which of the corresponding bits in the identifier acceptance register are relevant for acceptance filtering. The values for setting the acceptance and mask registers should be formatted to fit the identifier registers format which is explained at the message storage model section. This format depends on the type of identifiers that are being used (standard or extended). Later on this note we will review the standard and the extended identifier format for the identifier registers.

These filters are applied to the identifier registers depending on the selected acceptance mode. Filters are applied to the identifier registers IDR0–IDR3. When working as two 32-bit filters, each of the mask and acceptance filter bytes is applied to the corresponding identifier register's byte. When using four 16-bit filters the acceptance and mask registers are grouped in pairs and are applied to the two most significant bytes of the identifier registers. Single byte filters are applied only to IDR0. For more information on acceptance filters you may refer to Section 4.3 of the MSCAN Block Guide V2.15. To set the acceptance filters to work as four 16-bit filters results in a 0x10 value written to the CAN0IDAC.

The firmware uses some macro definitions to fill the values of the mask and acceptance registers. These macros are used to define an acceptance code of 0x100 with the correct format to fit into the acceptance registers.

Mask registers are used to determine which bits of the acceptance registers would be used to filter the incoming messages. If the mask register contains a 1 in certain bits it means that the corresponding bit on the acceptance register won't be used for the data filtering. If the bit contains a 0 it means that the value of the corresponding bit of the acceptance register must match with the corresponding bit of the incoming message's identifier value. For standard identifiers the lowest three bits of the mask register should be set to 1.

```
...
/* Acceptance Code Definitions */
#define ACC_CODE_ID100 0x2000
#define ACC_CODE_ID100_HIGH ((ACC_CODE_ID100@0xFF00)>>8)
#define ACC_CODE_ID100_LOW (ACC_CODE_ID100@0x00FF)
```

```
/* Mask Code Definitions */
#define MASK_CODE_ST_ID 0x0007
#define MASK_CODE_ST_ID_HIGH ((MASK_CODE_ST_ID&0xFF00)>>8)
#define MASK_CODE_ST_ID_LOW (MASK_CODE_ST_ID@0xFF)
...
```

In this mode there are four filter acceptance codes with their respective mask codes. Each filter has two bytes in order to add up the whole 16-bit codes. With the selected configuration the received messages would generate a filter 0 hit upon their arrival.

```
...
/* Acceptance Filters */
CAN0IDAC = 0x10; /* Set four 16-bit Filters

CAN0IDAR0 = ACC_CODE_ID100_HIGH: //|\ 16-bit Filter 0
CAN0IDMR0 = MASK_CODE_ST_ID_HIGH; //| \__ Accepts Standard Data Frame Msg
CAN0IDAR1 = ACC_CODE_ID100_LOW; //| / with ID 0x100
CAN0IDMR1 = MASK_CODE_ST_ID_LOW; //|/

/* Acceptance Filters */
CAN0IDAC = 0x10; /* Set four 16-bit Filters

CAN0IDAR2 = 0x00; //|\ 16-bit Filter 1
CAN0IDMR2 = MASK_CODE_ST_ID_HIGH; //| \__ Accepts Standard Data Frame Msg
CAN0IDAR3 = 0x00; //| / with ID 0x100
CAN0IDMR3 = MASK_CODE_ST_ID_LOW; //|/

CAN0IDAR4 = 0x00; //|\ 16-bit Filter 2
CAN0IDMR4 = MASK_CODE_ST_ID_HIGH; //| \__ Accepts Standard Data Frame Msg
CAN0IDAR5 = 0x00; //| / with ID 0x100
CAN0IDMR5 = MASK_CODE_ST_ID_LOW; //|/

CAN0IDAR6 = 0x00; //|\ 16-bit Filter 3
CAN0IDMR6 = MASK_CODE_ST_ID_HIGH; //| \__ Accepts Standard Data Frame Msg
CAN0IDAR7 = 0x00; //| / with ID 0x100
CAN0IDMR7 = MASK_CODE_ST_ID_LOW; //|/

CAN0CTL0 = 0x00 /* Exit Initialization Mode Request */
while ((CAN0CTL1&0x00) != 0) {} /* Wait for Normal Mode */
...
```

The last two instructions of the code are used to exit the module from its initialization mode. The normal mode request is made and then the program waits for the module to return to normal mode. This is the end of the initialization routine. Now, we are going to look into the data transactions (Tx/Rx) and how are they handled by the sample application.

*Transmission*

Due to the speeds used at the CAN bus, the MSCAN module has a triple transmit buffer scheme that enables the module to transmit messages over the bus at the maximum speed without releasing the bus. However, only one address area is applicable for transmit process and visible to the programmer. This area is known as the foreground transmit buffer. The transmit process should use the foreground buffer to set up all the required parameters of the frame and then select one of the three available buffers with the CANTBSEL register. This scheme makes the buffer select process easier and also avoids having three different buffer addresses for transmitting.

The following code section shows how the sample application detects if there is an available buffer and then selects the lowest empty buffer to start transmission.

---

```

unsigned char CAN0SendFrame *unsigned log id, _
                                unsigned char priority, _
                                unsigned char length, _
                                unsigned char *txdata )
{
    if (!CAN0TFLG)                /* Is Transmit Buffer full?? */
        return ERR_BUFFER_FULL;
    CAN0TBSEL = CAN0TFLG;         /* Select lowest empty buffer */
    txbuffer = CAN0TBSEL;         /* Backup selected buffer */
    ...

```

---

Having selected the appropriate free buffer then we can fill in the different parts of the CAN frame. The parameters received by the function are the frame's ID, local priority, data length, and data. Starting with the ID registers and then using a software loop to fill the data buffer. The CAN0TXBPR register is used to assign local priority inside the MSCAN Module. Priority in the CAN bus is determined by the identifier of the frame, the priority register selects which of the three transmit buffers is to be transmitted first.

The CAN0TFLG assignment sets the current transmit buffer and starts the frame transmission. The last line waits until the buffer is released when transmission is completed.

---

```

...
/* Load Id to IDR Register */
*((unsigned long *) (&CAN0TXIDR0)) = id;

for (index=0;index<length;index++) {
    *(&CAN0TXDSR0 + index) = txdata[index];    /* Load data to Tx buffer
                                                * Data Segment Registers
                                                */
}

CAN0TXDLR = length;                          /* Set Data Length Code */
CAN0TXBPR = priority;                        /* Set Priority */

CAN0TFLG = txbuffer;                         /* Start transmission */

while ( (CAN0TFLG & txbuffer) != txbuffer);  /* Wait for Transmission
                                                * completion
                                                */

```

---



*Reception*

This sample program implements a simple interrupt function to service the receiver interrupt. The function just reads the data buffer and clears the reception flag in order to keep the module available to receive further messages.

---

```
...
void interrupt CAN0RxISR(void)
{
    unsigned char length, index;
    unsigned char rxdata[8];

    length = {CAN0RXDLR & 0x0F};
    for (index=0; index<length; index++)
        rxdata[index] = *(&CAN0RXDSR0 + index); /* Get received data */

    CAN0RFLG = 0x01;    /* Clear RXF */
...

```

---

*Main Function*

The main function uses the CANInit() function and then waits for bus synchronization. When the bus gets synchronized it sets the CAN0RFLG and CAN0RIER registers to reset receiver flags and to enable the receiver buffer full interruption. After setting the receiver interrupt the program enables the interrupts and starts transmitting messages periodically.

Since the module is working in loop back mode, the transmission is received by the receive buffers and after filter acceptance the receiver buffer full interrupt is triggered.

---

```
...
void main () {

    unsigned char errorflag = NO_ERR;
    unsigned char txbuff[] = "ABCDEFGH";

    CANInit();

    while (!(CAN0CTL0&0x10));

    CAN0RFLG = 0xC3;

    CAN0RIER = 0x01;

    Enable Interrupts;

    for (;;) {
        errorflag = CAN0SendFrame((ST_ID_100), 0x00, sizeof(txbuff)-1, txbuff);
        Delay();
    }
}
...

```

---

### Message Storage Model

Message storage model includes the following registers: identifier registers, data segment registers, data length register, transmit buffer priority register, and time stamp register. The message storage model was designed to provide the programmer an easy way of handling CAN frames for transmission and reception.

This section explains the identifier register structure. These memory addresses were designed to handle standard and extended frame formats. The identifier registers (IDR0–3) add up a total of 32 bits which, when used as an extended identifier format includes 29 bits for the identifier and three flags:

- Extended identifier flag
- Remote transmission request flag
- Substitute remote request flag

| Register Name |        | Bit 7 | 6    | 5    | 4        | 3        | 2    | 1    | Bit 0 | Address |
|---------------|--------|-------|------|------|----------|----------|------|------|-------|---------|
| DR0           | Read:  | ID28  | ID27 | ID26 | ID25     | ID24     | ID23 | ID22 | ID21  | \$_x0   |
|               | Write: |       |      |      |          |          |      |      |       |         |
| DR1           | Read:  | ID20  | ID19 | ID18 | SRR (=1) | IDE (=1) | ID17 | ID16 | ID15  | \$_x1   |
|               | Write: |       |      |      |          |          |      |      |       |         |
| DR2           | Read:  | ID14  | ID13 | ID12 | ID11     | ID10     | ID9  | ID8  | ID7   | \$_x2   |
|               | Write: |       |      |      |          |          |      |      |       |         |
| DR3           | Read:  | ID6   | ID5  | ID4  | ID3      | ID2      | ID1  | IDS0 | RTR   | \$_x3   |
|               | Write: |       |      |      |          |          |      |      |       |         |

**Figure 5. Identifier Registers for Extended Identifier Mode**

The distribution of the flags inside the IDR1 is due to compatibility between standard identifiers and extended identifiers. In order to obtain the original identifier (standard or extended) it is necessary to have some bit manipulation routines which read the identifier registers and arrange the result into a single 29-bit or 11-bit value depending on the identifier's length.

For the standard frame format, only 11 bits are used to state the identifier and only two flags are included:

- Extended identifier flag
- Remote transmission request flag

| Register Name |        | Bit 7 | 6   | 5   | 4   | 3        | 2   | 1   | Bit 0 | Address |
|---------------|--------|-------|-----|-----|-----|----------|-----|-----|-------|---------|
| IDR0          | Read:  | ID10  | ID9 | ID8 | ID7 | ID6      | ID5 | ID4 | ID3   | \$_x0   |
|               | Write: |       |     |     |     |          |     |     |       |         |
| IDR1          | Read:  | ID2   | ID1 | ID0 | RTR | IDE (=0) |     |     |       | \$_x1   |
|               | Write: |       |     |     |     |          |     |     |       |         |
| IDR2          | Read:  |       |     |     |     |          |     |     |       | \$_x2   |
|               | Write: |       |     |     |     |          |     |     |       |         |
| IDR3          | Read:  |       |     |     |     |          |     |     |       | \$_x3   |
|               | Write: |       |     |     |     |          |     |     |       |         |

= Unused

**Figure 6. Identifier Registers for Standard Identifier Mode**

For this example we used the following macro to define the value being loaded into the identifier registers. The desired value for the identifier is 0x100. The lowest two bytes of the identifier registers are unused hence set to zero. The upper two bytes of these registers contain the identifier value shifted five times to the left. The resulting value for the four identifier registers is 0x20000000.

---

```
...
/* ID Definition */
#define ST_ID_100 0x20000000 /* Standard ID 0x100 formatted to be loaded
                             * in IDRx Registers in Tx Buffer
                             */
...
```

---



---

## Considerations

This example code was developed using Metrowerks CodeWarrior IDE version 3.0 for HCS12, and was expressly made for the MC9S12DP256B.

### NOTE

*There may be changes needed in the code if it is to be used in another MCU. Also, depending on the HC9S12 MCU being used the addresses will vary according to its corresponding memory map.*

---

## References

Refer to the following documents for more information pertaining to the subjects covered in this application note.

*S12MSCANV2 — HCS12 Scalable Controller Area Network (MSCAN)*

[http://www.freescale.com/files/microcontrollers/doc/ref\\_manual/S12MSCANV2.pdf](http://www.freescale.com/files/microcontrollers/doc/ref_manual/S12MSCANV2.pdf)

*BCANPSV2.0 — Bosch Controller Area Network (CAN) Version 2.0 Protocol Standard*

[http://www.freescale.com/files/microcontrollers/doc/data\\_sheet/BCANPSV2.pdf](http://www.freescale.com/files/microcontrollers/doc/data_sheet/BCANPSV2.pdf)

## ***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 All rights reserved.