Technical Data

MCF 5272I2CUG
Rev. 0, 3/2002

MCF5272 Soft I^2C User's Guide

Eric Ocasio
TECD Applications

To address customer needs, Motorola has developed a set of C-based software I^2C (Inter-IC) functions for the ColdFire® MCF5272 microprocessor, implemented via general-purpose I/O. These functions support master mode and transfers clocked up to 100 kHz.

The I^2C bus is a standard that was introduced by Philips Semiconductors. Its straightforward concept and manufacturing simplicity has made it a widely recognized form of inter-chip communication in embedded systems. Common peripherals such as LCD drivers, memory, and keyboard interfaces can be I^2C compatible.

This document provides information about how to use the Soft I^2C functions and integrate them into a MCF5272-based system. Specifically, it describes the interface, hardware/software configuration and protocol, and how to test the system.

This document discusses the following topics:

Topics

Section I, "Interface Description"
  1.1, "Software Functionality"
  1.2, "Init Function"
  1.3, "Read Function"
  1.4, "Write Function"
  1.5, "Stop Funtion"
  1.6, "Calling Sequences"
  1.7, "Hardware Interface"

Section II, "Functional Tests"
  2.1, "Communication with iPort™ device"
  2.2, "Alternative Interrupt Driven Implementation"

Freescale Semiconductor, Inc.

Table 1 shows acronyms, along with their meanings, used in this document.

Table 1. Acronyms and Abbreviated Terms

Acronym	Meaning
GPIO	General Purpose Input/Output
SDA	I^2C Data Line
SCL	I^2C Clock Line

Table 2 is a list of the documents and tools referenced in this document.

Table 2. References

Title	Order Number
MCF5272 User's Manual	MCF5272UM/D
I^2C-Bus Specification v2.1, January 2000	http://www.semiconductors.philips.com
iPort/AI™ (MIIC-201V) RS-232 to I^2C Host Adapter with ASCII Interface	Manufactured by Micro Computer Control http://www.mcc-us.com

Section I: Interface Description

This section outlines the details of the MCF5272 Soft I^2C functions, including the functionality of standard I^2C read and write features and formats. Each will be analyzed at the parameter level, including a description of the inputs and other parameters. Section I concludes with information about the hardware interface.

1.1 Software Functionality

This section highlights the read and write features and delivery format of the Soft I^2C functions. Metrowerks CodeWarrior™ IDE v3.2 was used to develop all software described in this and subsequent sections of this manual.

1.1.1 Standard Implemented

The Soft I^2C functions, as supplied by Motorola, implement the standard I^2C (version 2.1) read and write procedures. The following is a list of read and write features:

- User-definable slave address
- User-definable read/write buffer
- User-definable byte count
- Programmable transmission frequency
- Status byte modeled after current Motorola processors with on-chip I^2C modules

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

1.1.2 Delivery Format

The source code is provided via the ColdFire® MCF5272 product page located at http://motorola.com/coldfire.

1.2 I²C Init Function (`i2c_init`)

The Soft I²C implementation uses PA9 and PA10, two Port A GPIO pins. These pins were selected because they can be accessed easily through the expansion connector on the M5272C3 evaluation board. The user has the flexibility to use any of the GPIO pins for establishing I²C communications. Section 1.6.2 discusses the procedure for changing which GPIO pins are used for I²C communication.

To initialize the GPIO pins for an I²C transmission, the user should call the `i2c_init` function. This function initializes the GPIO in the PACNT register to select PA9 and PA10, and sets the appropriate values in the PADAT and PADDR registers.

Function code:

```
#define PACNT_init       MCF5272_GPIO_PACNT &= 0xFFC3FFFF
#define PADDR_init       MCF5272_GPIO_PADDR &= 0xF9FF
#define PADAT_init       MCF5272_GPIO_PADAT &= 0xF9FF

void i2c_init(void)
{
       PACNT_init;
       PADDR_init;
       PADAT_init;

       SCL_high;
       SDA_high;
}
```

1.3 I²C Read Function (`i2c_read`)

This function performs a standard I²C read operation. After sending the slave address, the MCF5272 goes into receive mode and waits for a data transfer from the slave device. Once a byte has been received, it is stored in a read buffer previously defined in the main function. The function generates its own start signal, as does the `i2c_write` function. This is case in order to allow for repeated starts. In order to completely terminate a transfer, the `i2c_stop` function must be called after the read.

Function prototype:

```
status = i2c_read(uint8 slave_address, uint8 *buffer, int byte_count,
int freq);
```


Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

1.3.1 Arguments

This section identifies the arguments for the `i2c_read` function. It includes summaries of the input arguments and the read-status byte.

1.3.1.1 Input Arguments

The following is a list of the 4 inputs to this function:

- **slave_address**: Indicates from which device the MCF5272 microprocessor reads data
- ***buffer**: Points to the location of the read buffer, where bytes will be stored
- **byte_count**: Tells the Soft I^2C function how many bytes will be read from the slave device
- **freq**: Sets the transmission frequency

1.3.1.1.1 slave_address

The slave address is a byte input that determines with which device the Soft I^2C communicates. There is no need to set or clear the LSB in order to set R/W since this is handled within software.

1.3.1.1.2 *buffer

The buffer argument points to the memory location of the read buffer. Before calling any of the I^2C functions, the user must define a read buffer. The size of the buffer is dependent on the specific application implemented by the and, therefore, has no maximum size requirement.

1.3.1.1.3 byte_count

This parameter tells the `i2c_read` function how many bytes to read from the slave device. Again, there is no maximum value, but there must be at least one byte read when calling the function. If byte_count is set to 0, the MCF5272 will not properly terminate communication with the slave device since it will not be able to hold the acknowledge bit high after a byte transfer. If this parameter is larger than the size of the read buffer, data will be lost.

1.3.1.1.4 freq

This argument determines the transmission frequency for the read process. Note that this value is passed as an integer value, and is to be entered in increments of 1. For example, when running from SDRAM with cache disabled on the M5272C3 evaluation board, the value entered is close to its corresponding frequency in kHz (75 is about 75 kHz). Since this calculation is based on a mathematical equation, the frequency will have some margin of error.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

1.3.1.1.5 Read Status Byte

The read function returns a status byte after execution to indicate the status of the transmission. This status byte's structure is similar to the I^2C Status Register (ISR) in other Motorola ColdFire® processors such as the MCF5307 and MCF5206e. The status bits included are IBB (I^2C bus busy bit), ICF (data transferring bit), and RXAK (receive/acknowledge bit). The status byte configuration is outlined in Figure 1.

	7	6	5	4	3	2	1	0
Field	ICF	—	IBB	—	—	—	—	RXAK
Reset	1000_0001							
R/W	Read only							

Figure 1. Status Byte setup

If a read operation is performed successfully, `i2c_read` will return 0xA1 since the bus will remain active until it is released by the `i2c_stop` function.

Table 3. Status Byte Bit Descriptions

Bit Name	Description
ICF	While one byte of data is being transferred, the Data Transferring Bit is cleared. It is set by the falling edge of the 9th clock of a byte transfer. 1 Transfer complete 0 Transfer in progress
IBB	The Bus Busy Bit indicates the status of the bus. When a START signal is detected, the IBB is set. If a STOP signal is detected, it is cleared. 1 Bus is busy 0 Bus is idle
RXAK	The RXAK shows the value of SDA during the acknowledge bit of a bus cycle. If it is low, it indicates an acknowledge signal has been received after the completion of 8 bits data transmission on the bus. If RXAK is high, it means no acknowledge signal has been detected at the 9th clock. 1 No acknowledge received 0 Acknowledge received

1.4 I^2C Write Function (`i2c_write`)

This function performs a standard I^2C write procedure. After generating a start signal and sending the slave address, it begins sending data clocked at the user-defined transmission

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

frequency. As with the i2c_read function, the i2c_stop function should be called to completely terminate the transmission.

Function prototype:

```
status = i2c_write(uint8 slave_address, uint8 *buffer, int byte_count,
int freq);
```

1.4.1 Arguments

This section identifies the parameters for the i2c_write function. It inclues summaries of the input parameters and the write status byte.

1.4.1.1 Input Arguments

The 4 inputs to this function are:

- **slave_address**: Indicates to which device the MCF5272 microprocessor will be writing
- ***buffer**: Pointer to the location of the write buffer, from where bytes will be read
- **byte_count**: Tells the Soft I^2C how many bytes will be written to the slave device
- **freq:** Sets the transmission frequency

1.4.1.1.1 slave_address

The slave address is a byte input that determines with which device the Soft I^2C communicates. There is no need to set or clear the LSB in order to set R/W since this is handled within software.

1.4.1.1.2 *buffer

This argument points to the memory location of the write buffer. Before calling any of the I^2C functions, the user must define a write buffer. The size of the buffer is dependent on the specific application that the user is implementing and, therefore, has no maximum size requirement.

1.4.1.1.3 byte_count

This parameter tells the i2c_write function how many bytes it will send to the slave device. Again, there is no maximum value, but this parameter cannot be larger than the size of the write buffer. In the case where it is, random data will be transferred after the write buffer has been completely cycled through.

1.4.1.1.4 freq

This argument determines the transmission frequency for the read process. Note that this value is passed as an integer value, and is to be entered in increments of 1. Unfortunately, since this calculation is based on a mathematical equation, the frequency will have some margin of error.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

However, for example, when running from SDRAM with cache disabled on the M5272C3 reference board, the entered value is close to its corresponding frequency in kHz (that is, 75 is about 75 kHz).

1.4.1.1.5 Write Status Byte

The `i2c_write` function also returns a status byte after execution to indicate the status of the transmission. Please see Table 3 in Section 1.3.1.1.5 for a complete description of the individual bits.
As shown in Figure 1, the reset value of the status byte is 0x81. If a transmission is successful, the function will return 0xA0. It will NOT return 0xA1 because the RXAK bit will not be set. This is because the slave device always pulls SDA low on the last clock cycle. After the i2c_stop function is called, the status byte will be returned to its reset value of 0x81.

1.5 I^2C Stop Function (`i2c_stop`)

This generates an I^2C stop signal. There are no inputs to this function, however it does return a status byte. As mentioned in both the Read and Write sections, this function MUST be called after the last read/write is performed in order to properly terminate the transmission.

Function prototype:

```
status i2c_stop(void);
```

1.5.1 Stop Status Byte

The stop status byte is identical to the status byte in the read and write functions. After `i2c_stop` has been executed, it sets the status byte to 0x81. For a complete description of the individual status byte bits, see Table 3 in Section 1.3.1.1.5.

1.6 Calling Sequences

The following section describes how the functions should be called. There are only two steps to the process: GPIO initialization and reads/writes.

1.6.1 Read/Write Calls

After GPIO initialization, the I^2C communication process is very straightforward. Calling either the `i2c_read` or `i2c_write` functions starts the communication process since the start signal is built into the functions. Also, consecutive reads and writes can be performed without calling for a stop. When the transmission is finished, a call to `i2c_stop` terminates communication. This process can be repeated as many times as is necessary.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

1.6.2 Changing Parameters

Since it may be inconvenient for some users to use the GPIO pins that are set up by default, it is possible, and extremely simple, to change which pins are used. As mentioned in Section 1.2, PA9 and PA10 were selected for SDA and SCL in this example code. In order to modify which GPIO pins are used, the user should change the values in the following seven macros (shown in their default setup) that appear in the "i2c.h" file.

```
#define PACNT_init       MCF5272_GPIO_PACNT &= 0xFFC3FFFF
#define PADDR_init       MCF5272_GPIO_PADDR &= 0xF9FF
#define PADAT_init       MCF5272_GPIO_PADAT &= 0xF9FF
#define SDA_high         MCF5272_GPIO_PADDR &= 0xFBFF
#define SDA_low          MCF5272_GPIO_PADDR |= 0x0400
#define SCL_high         MCF5272_GPIO_PADDR &= 0xFDFF
#define SCL_low          MCF5272_GPIO_PADDR |= 0x0200
```

Each of the above macros reads one of the Port A registers and performs a logical AND/OR with its contents. The result sets or clears the appropriate bits in the register, leaving the other bits unchanged. In the following example, the macros are set to use PA12 and PA0 as SDA and SCL, respectively. Table 4 shows values that should be used in the macros to get the desired result.

Table 4. Example Values for Changing Macros

Macro	Register	Desired Value (Binary)	Logical Operator Performed	Value to enter in macro (Hex)
PACNT_init	PACNT	XXXX_XX00_XXXX_XXXX XXXX_XXXX_XXXX_XX00	AND	0xFCFFFFFC
PADAT_init	PADAT	XXX0_XXXX_XXXX_XXX0	AND	0xEFFE
PADDR_init	PADDR	XXX0_XXXX_XXXX_XXX0	AND	0xEFFE
SDA_high	PADDR	XXX0_XXXX_XXXX_XXXX	AND	0xEFFF
SDA_low	PADDR	XXX1_XXXX_XXXX_XXXX	OR	0x1000
SCL_high	PADDR	XXXX_XXXX_XXXX_XXX0	AND	0xFFFE
SCL_low	PADDR	XXXX_XXXX_XXXX_XXX1	OR	0x0001

In order to use a different GPIO port (for example, Port B instead of Port A), the `MCF5272_GPIO_PXXXX` macros (located in "mcf5272.h") should be switched.

1.7 Hardware Interface

This section discusses reasons for using the GPIO and details some issues that had to be addressed in order to make these functions work in software.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

1.7.1 Why GPIO?

The GPIO were used because they are relatively easy for a user to program. PA9 and PA10 were used as the default because they are easily accessible on the M5272C3 evaluation board. As mentioned in Section 1.6.2, it is extremely easy to change which GPIO pins are used.

1.7.2 Using GPIO on Open Collector Lines

Using the GPIO pins on open collector lines requires that a 0 be written to the pin's corresponding data bit in the port data register. Once this is done, switching the pin between input and output mode in the data direction register leaves the line high or pulls it low. When the DDR is cleared to 0 (input mode), the SDA/SCL remains high. When the DDR is set to 1 (output mode), the SDA/SCL is pulled low since the data register has a 0 written to it. The SDA and SCL macros (shown below) are based on this concept.

```
#define SDA_high        MCF5272_GPIO_PADDR &= 0xFBFF
#define SDA_low         MCF5272_GPIO_PADDR |= 0x0400
#define SCL_high        MCF5272_GPIO_PADDR &= 0xFDFF
#define SCL_low         MCF5272_GPIO_PADDR |= 0x0200
```

Section II: Functional Test

This section details successful I^2C communication between the Soft I^2C and a device with I^2C-capable hardware. This conformance testing was performed using a M5272C3 evaluation board and an iPort™ MIIC-201V I^2C tool (see Table 2 for details about this device). All software was written using the Metrowerks CodeWarrior™ IDE for ColdFire and was run from SDRAM with the cache disabled.

2.1 The iPort™ Device

The easiest way to test the Soft I^2C is to use a device that provides a simple PC graphical user interface and does not require extensive programming. The iPort™ is such a device. It connects through the serial port of a machine and provides a simple user interface that allows the user to select the various parameters for an I^2C transfer. Another key feature of the iPort™ is that it requires absolutely no programming. This is the device that was used throughout the build and testing phases of the Soft I^2C.

2.1.1 Setting Up the MCF5272

Once the "i2c.h" and "i2c.c" files are incorporated into a project, initializing I^2C communication is very simple. The following program sets up a read and write buffer before calling the I^2C functions. Once the transmission starts, it sends the contents of the write buffer to the iPort™, followed by a read of the data provided by the iPort™. After the read, it writes the data that was

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

read back to the iPort™. It finishes by again sending the contents of the write buffer. The following code is used for this procedure:

```
void main (void)
{
      uint8 read_buffer[12];
      uint8 write_buffer[12] = {0xE3, 0x56, 0xC2, 0xFE, 0x00, 0xFF,
                                0x53, 0xB1, 0x7C, 0x42, 0xF9, 0xEE};
      uint8 status = 0x81;
      i2c_init();

      /* Note: The status is not being monitored in this example. */
      status = i2c_write(0x6E,write_buffer,12,75);
      status = i2c_read(0x6E,read_buffer,12,75);
      status = i2c_write(0x6E,read_buffer,12,75);
      status = i2c_write(0x6E,write_buffer,12,75);
      status = i2c_stop();
}
```

Figure 2 shows the information that the iPort™ device has logged.

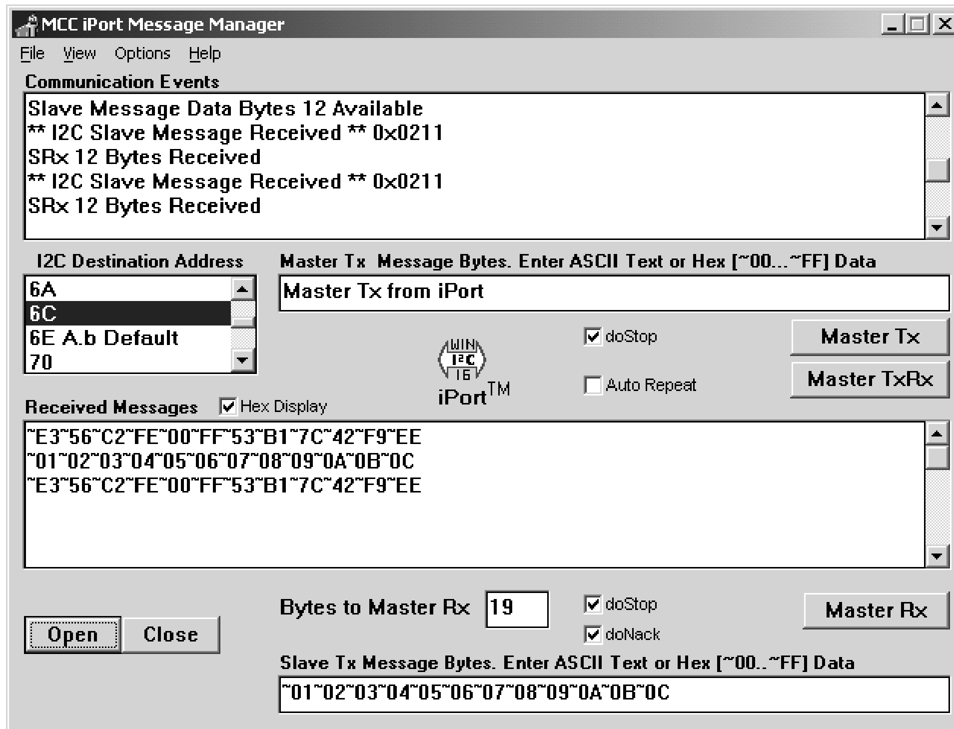

Figure 2. iPort™ Data

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

2.2 Alternative Interrupt-Driven Implementation

An alternative implementation that uses the MCF5272's timer modules can be used for a more precise transmission frequency. The example below outlines a timer implementation of the `i2c_write` function. The main idea of this method is to have the timer module count down for half of the transmission frequency clock cycle and toggle the SCL line in the timer's interrupt service routine (ISR).

2.2.1 Pseudo Code

Following is an example of ISR-based function code for the `i2c_write` function.

Global variables:

`isr_parity` – determines which value the ISR will put on SCL
`isr_count` – counts the number of times the ISR has been called
`isr_done` – indicates that the ISR has finished all operations for a 9-cycle transmission

```
i2c_write
{
      Initialize variables;
      SCL_low;

      while (i < byte_count)
      {
            isr_parity = 1;
            isr_count = 0;
            isr_done = 0;
            Set up timer registers, TMR, TER, TRR, TCN;
            Put first bit on SDA line;
            Wait for SCL to be released by slave;

            while(isr_done != 1); /* Wait for ISR to finish */

            Update write buffer to point to next byte;
      }

SDA_high;
SCL_low;
return(status);
}
```

The following is an example of the ISR:

```
__interrupt__
timerX_handler (void)
{
      Clear Timer Event Register;

      if (isr_parity == 0)
      {
            if (isr_count == 15)
```


Freescale Semiconductor, Inc.
For More Information On This Product,
Go to: www.freescale.com

```
		{
			SDA_high;
			SCL_high;
			Update status byte depending on ACK signal;
			isr_parity = 1;
			isr_count++;}
		else
		{
			Put data bit on SDA one bit at a time;
			SCL_high;
			isr_parity = 1;
			isr_count++;}
	}

	else
	{
		if (isr_count == 16)
		{
			SCL_low;
			Turn off timer;
			isr_parity = 0;
			isr_done = 1;}

		else
		{
			SCL_low;
			Update any mask used to send data;
			isr_parity = 0;
			isr_count++;}
	}
```

This example shows how to use an ISR to do all of the work in sending the data and clock signals for an I^2C write procedure. A similar procedure could be used to do the same thing for the `i2c_read` function. It should be noted that there is some overhead involved with using an ISR, such as the execution time of the code, that would need to be measured and factored into the timer's reference value. Once this has been addressed, this method can achieve extremely precise clocking for the Soft I^2C.

Section III: Conclusion

This document has outlined how to use the Soft I^2C provided by Motorola. This implementation is designed to be a simple solution for using I^2C hardware with the MCF5272. Section 2.2 discussed higher-level implementation if there is any need for a more precise clocking mechanism. It is important to understand that, when developing the Soft I^2C, the code was run out of SDRAM (with the cache disabled) on an M5272C3 evaluation board. All documented performance (frequency-wise) is based on these conditions. Performance may differ if other configurations are used.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

THIS PAGE INTENTIONALLY LEFT BLANK

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

THIS PAGE INTENTIONALLY LEFT BLANK

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

THIS PAGE INTENTIONALLY LEFT BLANK

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Freescale Semiconductor, Inc.

HOW TO REACH US:

USA / EUROPE / Locations Not Listed:

Motorola Literature Distribution
P.O. Box 5405
Denver, Colorado 80217
1-800-521-6274 or 480-768-2130

JAPAN:

Motorola Japan Ltd.
SPS, Technical Information Center
3-20-1, Minami-Azabu, Minato-ku
Tokyo, 106-8573 Japan
81-3-3440-3569

ASIA/PACIFIC:

Motorola Semiconductors H.K. Ltd.
Silicon Harbour Centre
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
852-26668334

HOME PAGE:
http://motorola.com/semiconductors/

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

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

MOTOROLA and the Stylized M Logo are registered in the US Patent and Trademark Office. All other product or service names are the property of their respective owners. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

© Motorola Inc. 2003

For More Information On This Product,
Go to: www.freescale.com