Freescale Semiconductor
Application Note

Document Number: AN2411
Rev. 2, 07/2005

Setup and Use of the ARM Interrupt Controller (AITC)

MC9328MX1, MC9328MXL, and MC9328MXS

By: Michael Kjar

1 Abstract

This document describes the use of the ARM Interrupt Controller (AITC) in i.MX processors. It describes the features of the AITC, as well as how to enable interrupts to the ARM9TDMI™ core and in the AITC. This document also describes the prioritization of interrupts, and some of the support software needed to initialize the heap and setup the IRQ and FIQ stack pointers.

This document applies to the following i.MX devices, collectively called i.MX throughout:

- MC9328MX1
- MC9328MXL
- MC9328MXS

It is assumed that the readers of this document are familiar with the i.MX processor (especially the AITC module), and the ARM9TDMI core, as well as have some understanding of embedded programming, such as programming in C, and so on. It also assumes the use of the ARM Developer Suite™ (ADS) as it refers to code examples found in the ADS directories. In addition,

Contents

these same principles can be applied when using the Metrowerks' CodeWarrior for the ARM Limited tools suite.

2 Introduction

This section provides an overview of interrupts and how they are handled in the i.MX processor's environment, particularly in the ARM9TDMI core and the AITC.

2.1 Interrupt Sources

In the system-on-a-chip architecture, interrupts usually occur from module sources (peripherals such as UARTS and timers), from external sources (such as the external IRQs), and can also be generated by software in the AITC via the interrupt force registers. Although this document's primary focus is on using the interrupt force registers to force interrupts, this template can be re-used to service interrupts from peripherals and other sources.

2.2 Interrupts to the ARM Core

The usual sequence of events for interrupts is as follows. Interrupts are enabled at the source (such as a peripheral), then enabled in the interrupt controller, and finally, enabled to the ARM core. When an interrupt occurs at the source, its "signal" is routed to the interrupt controller then to the ARM core. The AITC can be enabled or disabled to the ARM core; the interrupt can be assigned a priority level. The AITC collects up to 64 interrupt requests and provides and interface to the ARM core. The ARM core is the final destination for the interrupt. The interrupt will halt the normal processing routines in the ARM core to allow the interrupt request to be serviced.

The ARM core can handle up to five exceptions, however, our focus is on interrupt handling from an IRQ or FIQ request. Refer to the *ARM Architecture Reference Manual* for more information on the other exceptions.

The IRQ, or normal interrupt request, is used for general purpose interrupt handling. It has a lower priority than an FIQ and is masked out when an FIQ sequence is entered. The IRQ is enabled to the core by clearing the I bit in the CPSR and can be disabled by setting this bit. When an IRQ is detected by the core, it vectors to address 0x18 of the vector table and executes the instruction loaded in that address. Normally, the instruction found at 0x18 of the vector table is of the form:

```
LDR PC, IRQ_Handler
```

Refer to Table 1 for a description of the ARM core vector table.

When an IRQ interrupt is detected, the ARM core saves the address of the next instruction to R14_irq, enables SPSR_irq as the CPSR, enters the IRQ mode by setting the mode bits in the CSPR to 10010, disables normal interrupts by setting the I bit in the CPSR, and loads 0x18 into the PC. At address 0x18, an instruction loads the address of the interrupt handler into the PC. When writing interrupts handlers in C, it is imperative to include the "__irq" function declaration keyword. See Section 6 on Interrupt Handlers in C for more information on the "__irq" keyword. Refer to Chapter 5, Handling Processor Exceptions in the *ADS Developer Guide* for a full explanation of the interrupt handling process.

The FIQ is used to support high-speed data transfer or channel process and has a higher priority than IRQ. The FIQ is enabled to the core by clearing the F bit in the CPSR and can be disabled by setting this bit. When an FIQ is detected by the core, it vectors to address 0x1C of the vector table and executes the instruction loaded in that address. Normally, the instruction found at 0x1C of the vector table is of the form:

```
LDR PC, FIQ_Handler
```

When an FIQ interrupt is detected, the ARM core saves the address of the next instruction to R14_irq, enables SPSR_fiq as the CPSR, enters the FIQ mode by setting the mode bits in the CSPR to 10001, disables Normal and fast interrupts by setting the F and I bits in the CPSR, and loads 0x1C into the PC. At address 0x1C, an instruction loads the address of the interrupt handler into the PC.

Table 1. Vector Table

Exception Type	Mode	Address
Reset	Supervisor	0x00000000
Undefined Instructions	Undefined	0x00000004
Software Interrupts (SWI)	Supervisor	0x00000008
Prefetch Abort	Abort	0x0000000C
Data Abort	Abort	0x00000010
IRQ (Normal Interrupt)	IRQ	0x00000018
FIQ (Fast interrupt)	FIQ	0x0000001C

2.3 Overview of the AITC

The interrupt controller of the i.MX processor is called the AITC. The interrupt requests are collected and controlled in the AITC before going to the core. The following is a brief overview of the AITC programming model. Refer to the specific i.MX processor reference manual for a more detailed description of the AITC.

The AITC contains twenty-six 32-bit registers. The following is a description of each register:

- INTCNTL—Configures specific control functions of the AITC.
- NIMASK—Controls the Normal interrupt mask level. All Normal interrupt priority levels at or below what is programmed in the NIMASK register will be masked. Normal interrupt priorities are programmed via the NIPRIORITY[7:0] registers.
- INTENNUM—Provides hardware accelerated enabling of interrupts. This is done by programming this register with the interrupt source that is desired to be enabled. Doing so will immediately enable (set) this interrupt source bit in the INTENABLEH/L register.
- INTDISNUM—Provides hardware accelerated disabling of interrupts. This is done by programming this register with the interrupt source that is desired to be disabled. Doing so will immediately disable (clear) this interrupt source bit in the INTENABLEH/L register.
- INTENABLEH—Used to enable pending interrupt source bits [63–32] to the core.
- INTENABEL—Used to enable pending interrupt source bits [31–0] to the core.

- INTTYPEH—Used to select whether an enabled and pending interrupt source bit [63–32] will create a Normal interrupt or Fast interrupt to the core.
- INTTYPEL—Used to select whether an enabled and pending interrupt source bit [31–0] will create a Normal interrupt or Fast interrupt to the core.
- NIPRIORITY[7:0]—Provides software prioritization of Normal interrupts. Normal interrupts with a higher priority level preempts Normal interrupts with a lower priority level. If two Normal interrupts are programmed with the same priority, the one with the highest source number will be selected.
- NIVECSR—Provides the priority of the highest pending Normal interrupt and provides the source number of the highest pending Normal interrupt.
- FIVECSR—Provides the source number of the highest pending Fast interrupt.
- INTSRCH—Reflects the status of interrupt request inputs (sources 63–32) into the interrupt controller.
- INTSCRL—Reflects the status of interrupt request inputs (sources 31–0) into the interrupt controller.
- INTFRCH—Allows for software generation of interrupts for interrupt sources 63–32.
- INTFRCL—Allows for software generation of interrupts for interrupt sources 31–0.
- NIPNDH—Reflects the source number(s) of pending Normal interrupt requests, for interrupt sources 63–32.
- NIPNDL—Reflects the source number(s) of pending Normal interrupt requests, for interrupt sources 31–0.
- FIPNDH—Reflects the source number(s) of pending Fast interrupt requests, for interrupt sources 63–32.
- FIPNDL—Reflects the source number(s) of pending Fast interrupt requests, for interrupt sources 31–0.

3 Procedure for Enabling Interrupts

This section outlines the procedure for enabling and setting up the i.MX processor to use interrupts.

3.1 Enabling Interrupts to the Core

This subsection describes how to enable interrupts to the ARM core. To enable IRQ interrupts, clear the I bit of the CPSR; to enable FIQ interrupts, clear the F bit of the CPSR. Likewise, to disabled these interrupts, set the respective bits. Example 1 shows how to do this in C.

Example 1. Inline Assembler Enable and Disable IRQ Functions

```
__inline void enable_IRQ(void)
{
    int tmp;
    __asm
    {
        MRS tmp, CPSR
        BIC tmp, tmp, #0x80
        MSR CPSR_c, tmp
    }
}

__inline void disable_IRQ(void)
{
    int tmp;
    __asm
    {
        MRS tmp, CPSR
        ORR tmp, tmp, #0x80
        MSR CPSR_c, tmp
    }
}

__inline void enable_FIQ(void)
{
    int tmp;
    __asm
    {
        MRS tmp, CPSR
        BIC tmp, tmp, #0x40
        MSR CPSR_c, tmp
    }
}

__inline void disable_FIQ(void)
{
    int tmp;
    __asm
    {
        MRS tmp, CPSR
        ORR tmp, tmp, #0x40
        MSR CPSR_c, tmp
    }
}
```

With the inline functions shown in Example 3, the programmer can enable interrupts with the following function calls:

```
enable_IRQ();
enable_FIQ();
```

Alternatively, to disable the interrupts:

```
disable_IRQ();
disable_FIQ();
```

3.2 Setting Up and Enabling Interrupts in the AITC

This section describes setting up interrupts in the AITC. For full details about the AITC, refer to the AITC chapter in the specific i.MX processor reference manual.

The AITC allows you select whether a pending interrupt source will create a normal interrupt (IRQ) or a fast interrupt (FIQ) to the core. This is accomplished via the INTTYPEH and INTTYPEL registers. Each bit in these registers corresponds to an interrupt source available in the system. Setting a bit will select its corresponding interrupt source as a Fast interrupt, where clearing these bit will select its corresponding bit as a Normal interrupt. In the INTTYPEL register, bit 0 corresponds to interrupt source 0, bit 1 corresponds to interrupt source 1, and so on up to bit 31, which corresponds to interrupt source 31. In the INTTYPEH register, bit 0 corresponds to interrupt source 32, bit 1 corresponds to interrupt source 33, and so on up to bit 31, which corresponds to interrupt source 63.

After choosing which type of interrupt a pending interrupt source is, the next step is to enable the interrupt. This can be done via the INTENABLEH and INTENABLEL registers. To enable a pending interrupt to the core, its corresponding interrupt source bit in the INTENABLEH or INTENABLEL must be set. Likewise, to disable the interrupt, clear this bit. In the INTENABLEL register, bit 0 corresponds to interrupt source 0, bit 1 corresponds to interrupt source 1, and so on up to bit 31, which corresponds to interrupt source 31. In the INTENABLEH register, bit 0 corresponds to interrupt source 32, bit 1 corresponds to interrupt source 33, and so on up to bit 31, which corresponds to interrupt source 63.

For example, to select interrupt source bit 10 as a Normal interrupt, clear bit 10 in the INTTYPEL register. Then, to enable this interrupt, set bit 10 in the INTENABLEL register. Likewise, to select interrupt source bit 50 as a Fast interrupt, set bit 18 in the INTTYPEH register. Then, to enable this interrupt, set bit 18 in the INTENABLEH.

The AITC also allows the programmer to prioritize the pending normal interrupt sources to one of sixteen different priority levels. This can be in the NIPRIORITY[7:0] registers. A pending normal interrupt source with a priority level of fifteen is the highest pending normal interrupt source. However, if two pending normal interrupt sources have a priority level of fifteen, the highest pending interrupt source number has priority. In any event, a pending fast interrupt source has priority over all pending normal interrupt sources. The next section lists the priority levels of all pending interrupts.

3.3 Interrupt Prioritization

The following is a list of pending interrupt sources in order of prioritization, from highest to lowest.

- Fast interrupt source 63, 62, 61, … 0
- Normal interrupt source with priority level 15, 14, 13, … 0
- Normal interrupt sources with same priority level, where a higher source number has priority over a lower source number

3.4 Setting Up the IRQ and FIQ Stack

When the ARM core detects an interrupt, it enters into a different processing mode. In the case of an IRQ interrupt, the core enters the IRQ mode and in the case of an FIQ, the core enters FIQ mode. Because these are different modes of operation for the ARM core, the stack must be initialized for each mode in order to assure proper execution in those modes. This can be achieved using an assembly file known as the init.s file. Examples of the init.s file can be found in the ADS tools suite from ARM Ltd., and similarly the Metrowerks' CodeWarrior for ARM Ltd. tools suite. For a complete description of setting up stack pointers, refer to the *ARM Developer Guide*, initializing stack pointers in Chapter 6.

Example 2 is an example from the init.s file of setting up stack pointers for the different operating modes of the ARM processor, including the IRQ and FIQ stack pointers (note, the stack pointer of the Supervisor mode must always be initialized).

Example 2. Equating Stack Variable Locations

```
RAM_Limit    EQU   0x10020000; change this to accommodate another memory base
SVC_Stack    EQU   RAM_Limit; 6144 byte SVC stack at top of memory
IRQ_Stack    EQU   RAM_Limit-6144; followed by IRQ stack
FIQ_Stack    EQU   IRQ_Stack-128; followed by FIQ stack
USR_Stack    EQU   FIQ_Stack-128; followed by USR stack
```

4 Scatter Loading

This section describes the concept of scatter loading and why it is used.

4.1 Using Scatter Files to Place the Vector Table in Memory

Scatter loading requires the use of a scatter file. A scatter file has the extension *.scf. The scatter file is used to tell the linker where to load files or objects in memory. For detailed reference information on the linker and scatter-loading, refer to the *ARM Developer Suite Linker and Utilities Guide*.

It is necessary to place the vector table in a known area of memory, normally at address 0x0. The Example 3 provides the details of the vector table in assembly, which can be found in the file vectors.s.

Example 3. Vector Table in Assembly

```
LDR     PC, Reset_Addr
LDR     PC, Undefined_Addr
LDR     PC, SWI_Addr
LDR     PC, Prefetch_Addr
LDR     PC, Abort_Addr
NOP          ; Reserved vector
LDR     PC, IRQ_Addr
LDR     PC, FIQ_Addr
```

Example 4 is an example of what is contained in a scat.scf file:

Example 4. Example Scat.scf File

```
ROM_LOAD 0x0
{
ROM_EXEC 0x0
     {
        vectors.o (Vect, +First)
        * (+RO)
     }
        RAM 0x00040000
     {
        * (+RW,+ZI)
     }
}
```

In Example 4, the vectors.o object is placed “First” at address 0x0. “First” is a pseudo-attribute in the scatter-load description file to mark the first input section in an execution region. The vector table region is then followed by the Read Only (RO) region.

To allow the ARM linker to use the scatter file (scat.scf), you must include this in the linker section of the settings dialogue box in your project. Under settings, go to ARM linker under the Linker settings. In the dialogue box, under Linktype, choose scattered. Then, in the Scatter description file box, choose the desired scat.scf file.

4.2 Finding Examples of Scatter Files

Scatter file examples, as well as many other examples of how to set up interrupts, can be found in your *ADS_install_directory/Examples/rom_integrator.*

5 Initializing the Heap Using retarget.c

One caveat uncovered in dealing with the AITC was the propensity for the heap to overwrite the vector table. This is especially true when using *printf* statements (from the *standard io* library). However, the lesson learned here is the importance of initializing the heap. Therefore, when initialing the heap, it is best to place it in an area where it will not corrupt the vector table or any other part of your code.

In the *ADS_install_directory/Examples/rom_integrator* directory, there are examples of retarget.c files. In the retarget.c file, there’s a function called: *__user_initial_stackheap.* This function is called from the C run-time library file *__main*. This function is used to initialize the heap, and can also be used to initialize the stack. Example 5 is as example of the *__user_initial_stackheap* function.

Example 5. __user_initial_stackheap Function

```
__value_in_regs struct __initial_stackheap __user_initial_stackheap( unsigned R0, unsigned
SP, unsigned R2, unsigned SL)
{
    struct __initial_stackheap config;
    config.heap_base = 0x10010000; // change this accommodate another base
    config.stack_base = SP;
return config;
}
```

To use this function in the retarget.c file, make sure to include this file in your build project. To find out more information on initializing the heap and stack, as well as to find more detailed information on the retarget.c file, refer to the ARM *ADS Compiler and Libraries Guide*.

6 Interrupt Handlers In C

As previously stated, when writing interrupt handlers in C, it is imperative to include the “__irq” keyword in the function name. Example 6 shows an example of a compiled interrupt handler using the “__irq” keyword. In the example code, the section labeled “interrupt handler code here”, is where the branch to the interrupt service routine takes place.

Example 6. Using the __irq keyword

```
        STMFDsp!,{r0-r4,r12,lr}
        ;interrupt handler code here
        ADD  sp,sp,#4
        LDMFDsp!,{r0-r4,r12,lr}
        SUBS pc,lr,#4
```

Example 7 shows a compiled interrupt handler without using the __irq keyword.

Example 7. Not Using the __irq keyword

```
            STMFD    sp!,{r4,lr}
            ;interrupt handler code here
            LDMFD    sp!,{r4,pc}
```

Example 8 illustrates the interrupt handler used in the AITC test code for the i.MX processor. Refer to this test code as it provides examples of the topics discussed thus far.

Example 8. __irq Handler Functions

```
void __irq IRQ_Handler(void)
{
        short vectNum;
        vectNum = NIVECSR >> 16;// determine highest pending normal interrupt
        vect_IRQ[vectNum]();  // find the pointer to correct ISR in the look up table


void __irq FIQ_Handler(void)
{
        short vectNum;
        vectNum = FIVECSR & 0x0000003F;// determine highest pending fast interrupt
        vect_FIQ[vectNum]();// find the pointer to correct ISR in the look up table
}
```

7 Setup the GPIO as an Interrupt Source

Frequently, users need an interrupt mechanism to employ an external interrupt source. The i.MX processor allows the user to configure one or more GPIO lines as interrupt sources.

There are four GPIO ports on an i.MX processor. These are Port A, Port B, Port C, and Port D. Each GPIO signal on each port is multiplexed with another signal on the i.MX processor. Refer to Chapter 2 Signals Descriptions and Pin Assignments in the i.MX reference manual for list of GPIO signals and their multiplexed functions. Refer to the GPIO chapter for more details on the GPIO module and the available port signals.

Each GPIO port may contain up to 32 GPIO signals. These signals for each port are ORed together to form one interrupt signal per port to the i.MX processor's interrupt controller (AITC). Referring to the Interrupt Controller chapter in the reference manual there is a table depicting each interrupt source assignment. For the GPIO in particular, the following are listed:

- Port A which is called GPIO_INT_PORTA is assigned interrupt source number 11
- Port B which is called GPIO_INT_PORTB is assigned interrupt source number 12
- Port C which is called GPIO_INT_PORTC is assigned interrupt source number 13
- Port D which is called GPIO_INT_PORTD is assigned interrupt source number 62

To re-iterate the procedure for enabling interrupts, these steps are to:

1. Set up and enable the interrupt source (in this case the GPIO module)
2. Set up and enable the interrupts in the AITC
3. Enabled the interrupts to the ARM core (IRQ or FIQ)

7.1 Enabling Interrupts in the GPIO Module

The previous sections discuss how to enable interrupts in the AITC and to the ARM core. This section focuses on enabling interrupts in the GPIO module.

To enable the interrupts in the GPIO module, the user must first determine which GPIO Port signal to use that will not interfere with their other functionality. Then the following procedure must be followed to enable the GPIO for interrupt operation:

1. For each pin [i] that is to be used as a GPIO, set bit [i] in the Port A, B, C, or D GPIO in use register (GIUS_A, GIUS_B, GIUS_C, or GIUS_D).
2. For each pin [i], configure that bit [i] as an input by clearing the desired bit [i] in the data direction register (DDR_A, DDR_B, DDR_C, or DDR_D).
3. Configure the pin [i] for the desired external interrupt condition in the corresponding port interrupt configuration register. There are two configuration bits per pin [i] in the interrupt configuration register allowing the choice of positive or negative edge sensitive, or positive or negative level sensitive. Refer to the reference manual for more details.
4. Configure the interrupt mask register (IMR_A, IMR_B, IMR_C, or IMR_D) to unmask the desired pin [i] interrupt by setting the corresponding bit [i]. When an interrupt occurs and the corresponding bit is set (active), the corresponding bit in the interrupt status register (ISR_A, ISR_B, ISR_C, or ISR_D) will be set.
5. Option: It may be desirable to disable the pull up in the pull up enable register (PUEN_A, PUEN_B, PUEN_C, or PUEN_D) depending on the interrupt and system conditions used to generate the interrupt.
6. After the interrupt has been triggered, clear the corresponding bit in the interrupt status register in the interrupt service routine.

7.2 Example

Take for example the use of GPIO Port A signals PA0, PA22, and PA23 as interrupt sources. Referring to Chapter 2, *Signals Descriptions and Pin Assignments* in the specific i.MX processor's reference manual, we see that PA0 is multiplexed with A24, PA22 is multiplexed with CS4, and PA23 is multiplexed with CS5. So if this were an actual application, it is assumed that these functions are not needed for the application. Next, assume for this example, PA0 and PA23 are positive level sensitive interrupts while PA22 is a negative edge sensitive interrupt.

Following the conditions stated previously, to enable interrupts in the GPIO module:

1. Configure port A pins PA0, PA22, and PA23 as GPIO, by setting each of the corresponding bits in the GIUS_A register. Therefore:

   GIUS_A = 0x00C00001

2. Configure port A pins PA0, PA22, and PA23 as inputs, by clearing the corresponding bits in the DDR_A register. Using an AND operation, this would be:

   DDR_A &= 0xFF3FFFFE
3. Configure port A pins PA0, PA22, and PA23 for the desired external interrupt condition. In this case, PA0 and PA23 are positive level, so their corresponding bit settings in the interrupt configuration register would be 10, while PA22 is negative edge sensitive, so its corresponding bit setting would be 01. So, the register settings for the Interrupt configuration registers yield:

   ICR1_A = 0x00000002, for PA0 positive level

   ICR2_A = 0x00009000, for PA22 negative edge and PA23 positive level
4. Unmask or enable Port A pins PA0, PA22, and PA23 as interrupts by setting their corresponding bits in the IMR_A register:

   IMR_A = 0x00C00001
5. In this example, disable the pull ups for the PA0, PA22, and PA23 pins. Thus, using an AND operation yields:

   PUEN_A &= 0xFF3FFFFE, bits 22, 23, and 0 are cleared

The next step is to enable the interrupts in the AITC module. In the example given here, we assign these interrupts or normal interrupts or IRQs in the interrupt type registers and not assign any particular priority to these interrupts in the normal interrupt priority registers. However, it must be noted that normal interrupts with a higher source number have a higher priority than normal interrupts with a lower source number, given their priority levels are assigned the same value in the normal interrupt priority register. The interrupts also must be enabled in the AITC via the interrupt enable registers. Thus the settings for the AITC module will be as follows:

INTTYPEL &= 0xFF3FFFFE, use an AND operation to clear bits 23, 22, and 0

INTENABLEL = 0x00C00001, set bits 23, 22, and 0 to enable

The final step is to enable the interrupts to the core. In this case, we would only need to enable normal or IRQ interrupts, by using the sequence described in section 3.1.

8 References

The following i.MX technical reference manuals may be found at the Freescale Semiconductor Inc. World Wide Web site at http://www.freescale.com/imx. These documents may be downloaded directly from the World Wide Web site, or printed versions may be ordered.

MC9328MXS Reference Manual (order number MC9328MXSRM)

MC9328MX1 Reference Manual (order number MC9328MX1RM)

MC9328MXL Reference Manual (order number MC9328MXLRM)

The following ARM Limited documentation such as the ARM920T technical reference manual and the ARM architecture reference manual, may be accessed at: www.arm.com.

ADS Compiler and Libraries Guide, ARM Limited (ARM DUI0067)

ADS Developer Guide, ARM Limited (ARM DUI0056)

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
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

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

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

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

***For Literature Requests Only*:**
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-521-6274 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN2411
Rev. 2
07/2005

freescale™
semiconductor