Freescale Semiconductor

Freescale Semiconductor, Inc.

FINAL

Application Note AN002A

Implementation and performance of pseudo-DMA functions for the M·CORE architecture

Version A

Chris Joffrain
Applications Manager
M•CORE Technology Center

© Freescale Semiconductor, Inc., 2004. All rights reserved.

For More Information On This
Go to: www.freescale

Introduction

The purpose of this document is to describe how to implement a software based DMA function for the M•CORE architecture. It reviews the interrupt handling mechanism and explains how to take advantage of the alternate register file feature. The theoretical maximum transfer rate is investigated and limitations in real applications are discussed.

Overview

Embedded applications are heavily interrupt-driven activities. There is a number of low-level actions that take place within a microcontroller to handle different kinds of interrupts in the most effective way. The first part of this document reviews these tasks. The second part describes how the M•CORE architecture approaches these tasks. The third part investigates the maximum data transfer rate in a pseudo-DMA application. The fourth part evaluates the limitations in real applications and the fifth part explains the hardware requirements.

Terms, Acronyms

Context — refers to the processor environment (memory, registers) available to a user program at a given time.

Alternate register file — refers to set of registers that are switched in place of the normal set of registers seen by the user program. This usually happens when the processor changes its flow of execution because of an interrupt or exception.

Shadow register(s) — refers to registers that automatically receive copies of the Program Counter and Status registers during interrupt and exception processing in order to preserve their original context.

Envelope — refers to the part of software executed between the processor's detection of an interrupt request and the first instruction servicing the interrupt request. The envelope is defined to contain any jump instruction to access a handler and any interrupt recognition, prioritization, arbitration and routing to a proper handler. The envelope also contains any software necessary to mask and re-enable lower priority interrupts.

Handler — refers to the software that is executed to service a specific interrupt. It also contains the necessary actions to make sure that any resource (register, memory) used by it is properly saved unless this resource is dedicated to this handler.

FINAL
Freescale Semiconductor, Inc.

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

Part 1: Interrupt handling

Handling interrupts in an application involves a hardware implementation and a software implementation. The hardware implementation is what the processor and system architecture is capable of doing, by design, to assist any application responding to interrupts. The software support is what the designer must provide, specific to the application and interrupt source, to properly handle the interrupts for this application.

The first phase that must take place occurs at the initialization stage: the set up of all necessary registers, pointers, memory areas and other parameters that ware required to handle interrupts in the application. This is a software component.

The second phase is executed each time an interrupt occurs. This phase consists of the recognition of the interrupt (i.e. synchronizing the system, waiting for the current instruction to terminate, and transitioning to exception processing.) The time required for the current instruction to terminate before the processor can respond to the interrupt is called 'latency'. It varies, depending upon the number of clock cycles required to execute each of the architecture's instructions and the ability of the architecture to abort lengthy multi-cycles instructions. This latency is determined by the definition of the architecture and implemented in hardware.

The third phase is the envelope for the interrupt handler. It deals with what is needed by the handler but not provided by the architecture in hardware. It performs the selection of the proper handler according to the interrupt activated and is executed in two parts. The first action occurs at the beginning of the interrupt servicing and identifies the selected interrupt, saving some of the original context if necessary. The second action occurs at the end of the interrupt servicing and restores the conditions that were in effect before the interrupt occurred. Although essential to the correct working of the application, this phase has no real value for the handling of the interrupt, represents pure overhead, and as a result should be minimal. The lesser software context switching is required, the smaller this overhead will be for the application and the faster the interrupt servicing will be. This will be critical in the search for MBytes per second.

The fourth and final phase is the interrupt handler itself, and examples will be given in the analysis of real applications and how to achieve high pseudo-DMA data rates.

FINAL

Part 2: M•CORE's way to handle interrupts

Details of the M•CORE architecture can be found in the reference manual MCORERM/AD. It describes the registers and their use as well as the instruction set. The case of multiple interrupts and the use of software prioritization, is addressed in the document "Performance Comparison: interrupt handling".

The first task is the initialization. It performs the loading of the vector base register pointing to the exception vector table. Loading of the system stack pointer may be part of this task but is also necessary for the other tasks in the system and is not considered specific to the interrupts in this document.

The second task is the first portion of the envelope and the initial response to the request for interrupt. Two actions take place during the execution of this task: (1) waiting for the current instruction to terminate and (2) transitioning to the interrupt handler. The majority of

Freescale Semiconductor, Inc.

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

Freescale Semiconductor, Inc.

M•CORE's instructions execute in one clock cycle. The longest M•CORE instruction execution time is for the DIVU or DIVS instruction and can be up to 37 clock cycles. However, M•CORE has a specific feature which allows this waiting time to be limited to a maximum 3 cycles. This is called the 'interrupt control'. Setting the IC interrupt control bit in the status register allows lengthy instructions to be aborted before completion and thus limiting current instruction interrupt latency to a maximum of 3 clock cycles from the time the interrupt is received. Once interrupt processing is initiated, whether the instruction is aborted or allowed to complete, an additional 6 clock cycles are required by the M•CORE processor to synchronize and transition to the interrupt service routine.

The third task, the final portion of the envelope deals with context switching and directing the processor to the execution of a handler. In order to minimize or even eliminate the overhead of saving the context that is interrupted, M•CORE can switch to an alternate register file automatically by way of special encoding of each interrupt entry in the vector table. This means that the 16 user registers, the status register and the program counter of the current context will be kept intact during the processing of the interrupt. If the least significant bit in the interrupt vector is set, the processor automatically switches to the alternate register file upon entry in the interrupt service routine and switches back upon exit from the service routine.

Part 3: M•CORE pseudo-DMA

A pseudo-DMA function is an interrupt-driven, software implementation of a direct memory access control function. Some knowledge of the purpose and principles of a DMA function are required to understand this analysis. In this analysis the main actions taking place during the interrupt are the reading of data from an I/O port and the writing of this data into a buffer (or vice-versa).

Assumptions for implementing this pseudo-DMA are:

- 32-bit wide data bus,
- zero wait state instruction and data memory,
- 50 MHz clock frequency (20nS clock cycle).

FINAL

Scenario A: single channel, single transfer DMA

In this mode, each interrupt will trigger the reading of a word (4 bytes) into a register and the writing of this register into a memory word (4 byte locations). The transfer rate will benefit from transferring 4 bytes at a time. However, the overhead of responding to the interrupt and executing any software will affect the transfer rate.

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

Freescale Semiconductor, Inc.

Flow chart and resources required

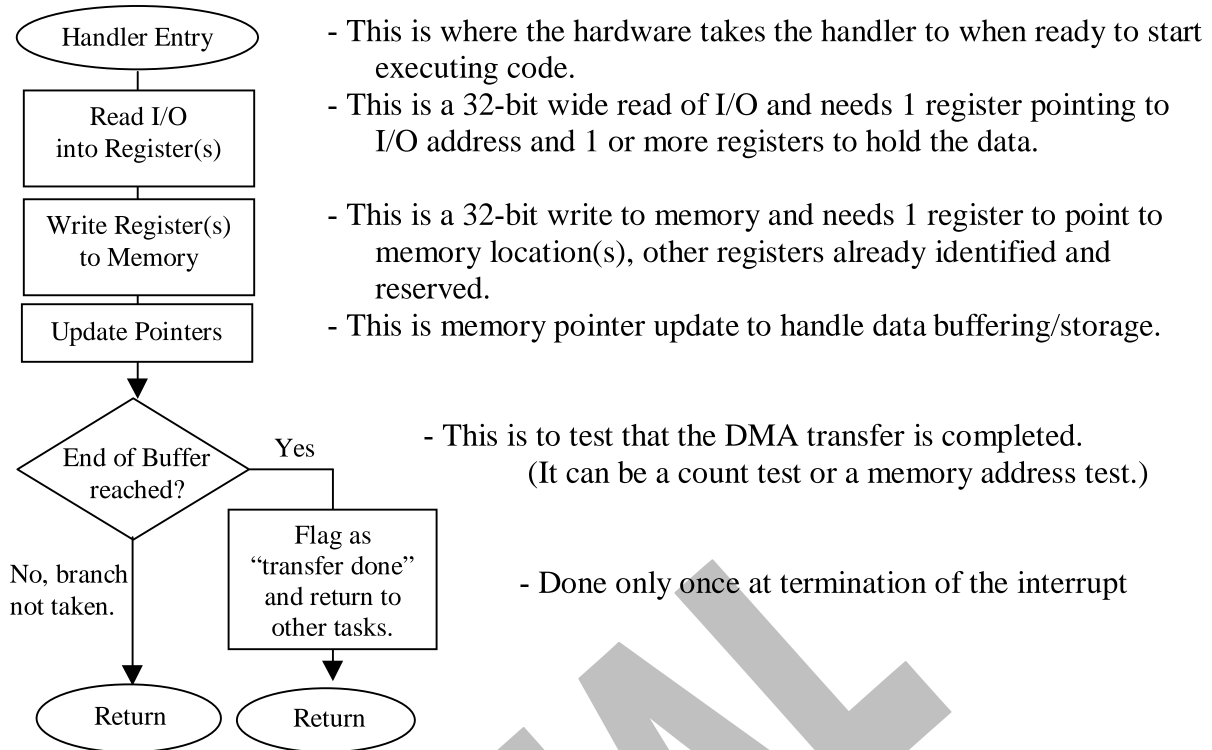

fig 1: basic flow chart for a software DMA

FINAL

This describes the basic requirements for the interrupt handler. It needs 4 registers:

- One data holding register (r4)
- One pointer to the I/O register (r5)
- One pointer to the memory buffer (r6)
- One pointer to the end of the buffer (r7)

These registers are chosen to allow a quick register transfer (STQ and LDQ instructions) if required for a context switch. However, the total execution time of this portion of code being kept to a minimum, it may be preferable to let the handler complete the execution rather than interrupt it in the middle (The penalty paid in overhead execution time is higher than the time to wait.)

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

The following table shows the sequence of instructions required to implement this scenario and the clock cycle count associated.

Task	M•CORE Inst.	Cycles		Comments
response to interrupt	-	6-9		Time to go to handler's first instruction.
read I/O in register	`LD r4,[r5, IO]`	2	*	1 register = 4 bytes. IO= offset to data register
write data to memory	`ST r4, [r6]`	2	*	use register as memory buffer pointer
update pointer	`ADDI r6, 4`	1	*	for 4 bytes
check if end of buffer	`CMP r6, r7`	1		r7 holds the address of last location in buffer
branch is yes (end)	`BPL closure`	1		1 cycle only if not taken
return from exception	`RTE`	3		restore context

(come here when no more DMA required)	`closure:...`
	`.....`
	`RTE`
				will come here only when interrupt service is to be closed. Will do 3 clock cycles in Branch (BPL)

TOTAL:		**16-19**		

The total time to service this 4 bytes DMA request is 16 to 19 clock cycles. Using a 50 MHz clock frequency, this translates to 320-380 nS (average 350 nS) or to a data transfer rate of 10.5 to 12.5 Mbyte/s.

Scenario B: multiple channels, single channel DMA

To implement a multiple channel DMA function, the best performance can be achieved using address this type of application, the best performance achieved using M•CORE's vectored interrupt capability, such that each DMA channel is assigned its own vector and service routine. Note that in this example all DMA channels share the same interrupt input, but that each channel provides a different vector. The code of the handlers will be almost identical, pointers for each channel will point to different I/O and buffer addresses. The overall performance is not affected by the number of interrupts since no extra software (such as interrupt polling) is required.

Scenario C: single channel, multiple transfers DMA and theoretical maximum rate

In this example, the transfer will handle as many bytes/words as possible assuming that the data will be available at the maximum rate (FIFO, buffer, synchronized data link,...)

Requirements

In order to achieve the maximum possible DMA data transfer rate it is necessary to review the minimum requirements and possible constraints:

- Analysis is done with M•CORE dedicated to DMA transfers for the best I/O throughput performance.
- Interrupt will be dedicated to DMA data transfer, with minimum context switching in order to achieve the highest possible burst transfer rate.
- Handler code size and buffer size are not limited but will be evaluated.
- Some hardware requirements are necessary, see part 5: hardware requirements.)

The flow chart in scenario A (figure 1) applies here too. The handler needs 4 registers as pointers, 1 register as offset value and as many registers as possible as data transfer registers. It is important to remember the following two assumptions (see also part 5, hardware requirements):

FINAL

Freescale Semiconductor, Inc.

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

- One assumption is that this interrupt handler will not call any subroutine and for this reason will not require R15 register as a dedicated Link Register. R15 is therefore free to be used as a data transfer register.
- The second assumption is that R0 register can be used as a buffer pointer register. It is usually used as the Stack Pointer but is needed in the STM and LDM instructions. For this matter, R0' register will be used as the interrupt handler's pointer for the buffer operations occurring with the LDM/STM instructions.

Other considerations

Since M•CORE, in this analysis, is dedicated to pseudo-DMA function, it is assumed that no other interrupt will be permitted during the transfer. As a consequence, it is not considered to have any context saving activity. Of the 5 registers required it is possible to combine the memory pointer with the stack pointer and consider all data transfers as stacking/de-stacking operations. This leaves the 11 registers of the shadow bank available for the data transfer with the use of the LDM (Load Multiple Registers) and STM (Store Multiple registers) instructions. They are very efficient instructions as they consume 1 cycle plus only 1 cycle per 32-bit register being transferred.

Summary of the resources allocation (alternate register file):

- r0' Dynamic memory pointer during transfers to or from memory
- r1' Memory address holding register
- r2' I/O address holding register
- r3' Offset value holding register (contains the number of bytes moved = 44)
- r4' End of buffer pointer holding register
- r5' to r15' Serve as 11 data dynamic transfer registers (holding 44 bytes)

FINAL

Remarks:

It is better to use registers to hold the pointers rather than have these pointers in memory or scratch registers because register based instructions consume only one cycle to execute.

LDM and STM instructions require both the use of the same register r0 as the base pointer. As a consequence it is necessary to reload r0 for each LDM and STM operation with the proper addresses. The role of r1 and r2 is to hold these addresses.

It is preferable to use r1 as a memory address holding register and r2 as an I/O address register rather than store the addresses in memory or in scratch registers because it would consume two clock cycles to retrieve each address.

R3 is necessary to hold the offset with a value of 44 (11 registers of 4 bytes each.) The instruction ADDI (add immediate) would spare this register but cannot be used since the maximum value that we can "add immediate" is 32 (which gives only 8 registers.) Limiting to 8 data registers and using the 'add immediate" instruction results in lower data throughput. Likewise, executing two ADDI instructions to add the value 44 to the register results in lower data throughput.

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

Freescale Semiconductor, Inc.

Performance analysis

The table below describes the instructions executed in the handler, from the entry after recognition and jump to the handler until the return to the normal program. The instructions marked as (*) will be discussed later.

Task	M•CORE Instructions	Cycles		Comments
response to interrupt	-	6-9		time to reach the first instruction.
get pointer to I/O	`mov r0, r1`	1	*	restore r0 pointer with I/O address
read I/O in registers	`ldm r5-r15, (r0,0)`	12	*	read 44 bytes into 11 registers
get pointer to stack	`mov r0, r2`	1	*	restore r0 pointer with buffer address
push registers to stack	`stm r5-r15, (r0,0)`	12	*	write 44 bytes from 11 registers
update pointer	`addu r2, r3`	1	*	add R3 to R2 (update pointer)
check if end of stack	`cmp r2, r4`	1		has r2 reached the end of buffer?
branch is yes (end)	`bpl closure`	1		no, go wait for next interrupt
return from exception	`rte`	3		restore context

(here other task when finished)	`closure:....`
	`here, DMA is finished`
	`(RTE)`
				will execute only once when service is terminated.

TOTAL:		**38-41**		**for 44 bytes transfered**

Details

The list above indicates the time taken to transfer 44 bytes (11 registers) from an I/O port to memory (which is accessed as a stack.) The total time to execute the interrupt is 38-41 cycles or 760-820 nS (average 790 nS) for a 50MHz processor clock frequency.
This give a **burst DMA** transfer rate of 53.6 to 57.9 MByte/s.

This takes only 8 active instructions (16 bytes) including the termination instructions.

Scenario D: Doing more

It is possible to consider transferring more bytes during the execution of the handler. It requires executing the first five instructions (marked * in the tables) several times. But rather than creating a loop using a counter with compare and branch instructions, it is more efficient to duplicate these five instructions. The cost is the addition of 5 instructions (10 bytes) and 27 clock cycles for every 44 data bytes transferred.

The following table summarizes the performance obtained for different number of data bytes transferred. In the formulas, N is the number of times the five instructions are repeated.

Data bytes transferred	Intructions (bytes) (3 + 5*N)	clock cycles (14 + 27*N)	Transfer Rate (@ 50MHz)	comments
44	8 (16 bytes)	41 cy	53.6 MByte/s	N=1
220	28 (56 bytes)	149 cy	73.8 MByte/s	N=5
440	53 (106 bytes)	284 cy	77.4 MByte/s	N=10
880	103 (203 bytes)	554 cy	79.4 MByte/s	N=20
2,200	253 (506 bytes)	1,364 cy	80.6 MByte/s	N=50

FINAL

Freescale Semiconductor, Inc.

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

Part 4: Real applications

Some applications may call for some data processing in the handler. This execution time must be added to the time spent in this handler. Some applications may require data to be transferred to and from memory once the data has been processed. This means that for an application implementing both transfers directions the transfer rate will drop to half the values presented here (twice as much time spent for the same bytes.)

Example: An I/O controller is designed to receive blocks of data from an I/O interface in burst mode (fixed size blocks), perform some operations on the data or route the data, then send this block to another I/O in burst mode (fixed size blocks.) The software in the handler can be written to accommodate any block size.

Part 6: Hardware requirements

In order to support the software handler described above, it is necessary to look at what is required from the hardware. Two points need consideration, the support of LDM and STM instructions and the concept of a burst mode in the handler.

The two instructions LDM and STM provide a multiple register transfer capability by pointing to an address, load/store data from/to that address and repeat for other registers while incrementing the address pointer. To exploit the benefit of STM/LDM instructions, it is necessary to

- Map the I/O as a word wide register, able to transfer 4 bytes in one access,
- Map the I/O such that the same word wide register responds to the 11 different addresses accessed by the Load or Store instruction when executed. An easy way to do this is to map the I/O register as responding to 16 contiguous word addresses, thereby allowing the I/O registers to act as a FIFO data register.

To maximize data I/O throughput, the burst mode executes data transfers between I/O and memory via a 44 byte block (the content of 11 registers.) The software does not include any test on the availability of the data and it is assumed that the interrupt will be generated only when the bytes that are going to be transferred, are available. This can be implemented with a FIFO (first in-first out) mechanism or with a device able to provide data at the required rate and for the exact amount of data (fixed length block-based transmission.) In this case, the handler can be adjusted to transfer the exact amount of bytes. If necessary error recovery exception handling and variable block length I/O transfer handling can be added via separate unique exception vectors. Handling dedicated vectors and exception handling routines optimize I/O throughput while still offering error recovery and sub-block size transfers at the beginning and end of an I/O transfer.

FINAL
Freescale Semiconductor, Inc.

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

Conclusion

The table below summarizes the clock cycles and data transfer rates calculated in different DMA modes for the M•CORE architecture.

DMA type	MCore (50MHz)	comments
Single channel, single transfer	16 – 19 cycles 10.5 – 12.5 MByte/s	4 bytes transfered
Multiple channels, single transfer	16 – 19 cycles/channel 10.5 – 12.5 MByte/s	Arm’s penalty is interrupt polling and masking
Single channel, multiple transfers (BURST mode)	38 – 41 cycles (44 bytes) 53.6 – 57.9 MByte/s	Arm’s penalty: an incomplete alternate register file

In the multiple channels mode, the transfer rate indicated is the maximum rate each channel can achieve WHEN NO OTHER CHANNEL IS ACTIVE. If several channels, say 5 channels were interrupting each other, the rate would be divided by 5.

All these transfer rates indicated are valid only during the transfer. Any other activity out of the interrupt will make the transfer rate decrease in proportion. For example, if the single channel-single transfer gets only half the time of the processor, the transfer rates will be divided by two.

FINAL

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 which 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 Semiconductor, Inc.

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