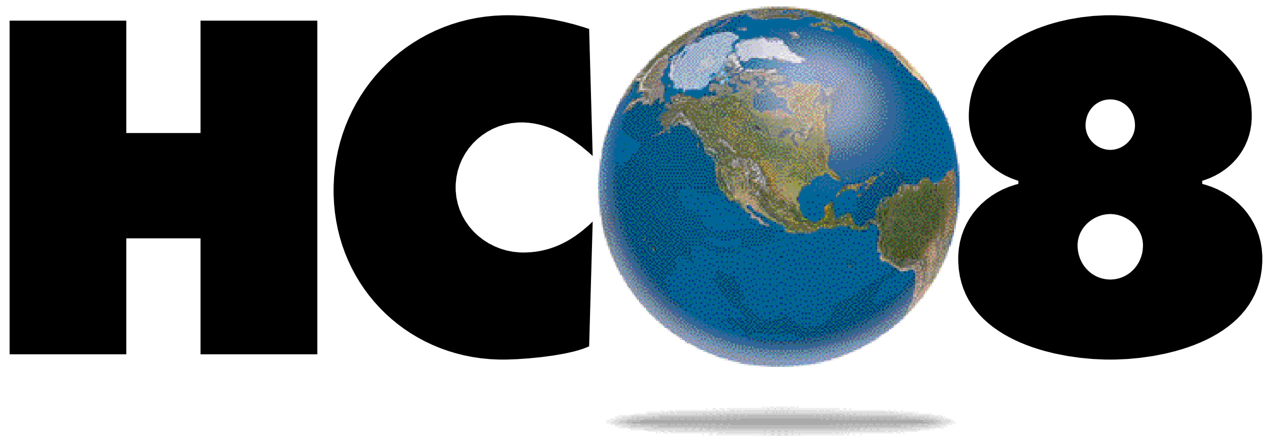
DMA08RM/AD
Rev. 1.0



DMA08
DIRECT MEMORY ACCESS

REFERENCE MANUAL

MOTOROLA

DMA08

Direct Memory Access
Reference Manual

© Motorola, Inc., 1996; All Rights Reserved

Motorola reserves the right to make changes without further notice to any products herein to improve reliability, function, or design. Motorola does not assume any liability arising out of the application or use of any product or circuit described herein; neither does it 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 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 Motorola logo are registered trademarks of Motorola, Inc.

Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

List of Sections

Introduction 17

Overview and Features 21

DMA Module Description 25

DMA Transfers 35

DMA Registers 49

DMA Application Examples 67

DMA Module 105

DMA08 Version B 109

Glossary 111

Index 125

Preface

All M68HC08 microcontrollers are modular, customer-specified designs. To meet customer requirements, Motorola is constantly designing new modules and creating new versions of existing modules.

The *DMA08 Reference Manual* introduces version A of the DMA08, the direct memory access (DMA) of the Motorola HC08 Family. Version B is described in Appendix A. Information concerning future versions of the DMA08 will be included in appendices in later versions of the reference manual.

Revision History

This table summarizes differences between this revision and the previous revision of this reference manual.

Previous Revision	Original Release
Current Revision	1.0
Date	08/96
Changes	Format and organizational changes
Location	Throughout

Table of Contents

Introduction

Contents 17
Introduction 17
The DMA08 Module 20

Overview and Features

Contents 21
Introduction 21
Features 23

DMA Module Description

Contents 25
Introduction 26
Source and Destination Base Addresses 27
Address Extension Module 28
Byte Count 29
Block Length 30
Arithmetic Logic Unit (ALU) 31
DMA Control and Status 31
Memory Stretch 31
Low-Power Modes 32
    Wait Mode 32
    Stop Mode 32
Breakpoints 33
DMA in Expanded Mode 33

DMA Transfers

Contents 35
DMA Operation 36
Transfer Types 37
Cycle-By-Cycle Operation 40
Byte Transfers 40
DMA Activity During a Byte Transfer 41
Word Transfers 42
DMA Activity During a Word Transfer 42
DMA Transfer Latency 44
Example of the DMA Transfer Programming Procedure 44
Address Calculation 46
Bandwidth Control 47

DMA Registers

Contents 49
Introduction 50
DMA Register Latency 51
DMA Module Registers 52
DMA Control Register 1 52
DMA Status and Control Register 54
DMA Control Register 2 57
Individual DMA Channel Registers 59
DMA Channel Control Register 59
DMA Source Base Address Registers 62
DMA Destination Base Address Registers 63
DMA Block Length Register 64
DMA Byte Count Register 65

DMA Application Examples

Contents 67

Introduction 68

Software-Initiated Block Transfer 69
- A – Simple, Small Block Transfer 69
- B – Flexible, Large Block Transfer 71
- Summary 73

DMA Service of Serial Communications 73
- A – Transmitting a Buffered Message Using the CPU 74
- B – Servicing the SCI Transmitter Using the DMA 75
- Summary 76

DMA Timer Servicing 77
- A – Generating a Pseudo Buffered PWM 77
- B – Buffering Input Captures for Period Calculation 81
- Summary 83

Full Assembler Listings 84
- Listing 1 – Fixed Block Length Transfer .84
- Listing 2 – Variable Block Length Transfer 86
- Listing 3 – SCI Transmitter 90
- Listing 4 – SCI Transmitter 92
- Listing 5 – Timer Output Compare 94
- Listing 6 – PWM Generation 96
- Listing 7 – Timer Input Capture 98
- Listing 8 – Period Measurement 101

DMA Module

Contents 105

Introduction 106

708XL36 DMA Registers 107

708XL36 DMA Transfer Source Mapping 108

708XL36 Peripheral Interrupt Prioritization .108

DMA08 Version B

DMA Version B 109

Glossary

Glossary 111

Index

Index 125

List of Figures

Figure	Title	Page
1	Simplified Block Diagram of a System with DMA	18
2	Diagram of the MC68HC708XL36 Layout	20
3	DMA Module Functional Block Diagram	22
4	DMA Module Functional Block Diagram	26
5	Address Extension Registers	28
6	DMA Operation	30
7	DMA Operation	36
8	Write-Clearing Interrupts at 100% Bandwidth	39
9	Software-Initiated Transfers/Read-Clearing Service Request Flags Behavior with 100% Bandwidth	39
10	MCU Bus Activity During a DMA Byte Transfer	40
11	MCU Bus Activity During a DMA Word Transfer (100% Bandwidth)	42
12	MCU Bus Activity During a DMA Word Transfer (50% Bandwidth)	42
13	DMA and CPU Use of the IBUS	47
14	DMA Control Register (DC1)	52
15	DMA Status and Control Register (DSC)	54
16	DMA Control Register 2 (DC2)	57
17	DMA Channel Control Register (D0C)	59
18	DMA Source Base Address Registers (D0SH and D0SL)	62
19	DMA Destination Base Address Registers (D0DH and D0DL)	63
20	DMA Block Length Register (D0BL)	64
21	DMA Byte Count Register (D0BC)	65

Figure	Title	Page
22	Minimum PWM High Time	80
23	Diagram of the MC68HC708XL36 Layout	106
24	MCU Bus Activity During DMA Byte and Word Transfers (50% Bandwidth)	110

List of Tables

Table	Title	Page
1	Byte Transfer Activity	41
2	Word Transfer Activity	43
3	DMA/CPU Bus Bandwidth Sharing	47
4	DMA Transfer Source Selection	57
5	DMA Channel Control Register	59
6	DMA Word Transfer	61
7	Relative Performance in Two Block Transfer Methods	70
8	Relative Performance in Two Block Transfer Methods	72
9	Relative Performance in DMA and CPU Transfer Methods	76
10	MC68HC708XL36 DMA Registers	107
11	DTS Bits	108
12	MC68HC708XL36 Peripheral Interrupt Prioritization	108
13	DMA/CPU Bus Bandwidth Sharing (DMA08 Version B)	109

Introduction

Contents

Introduction 17
The DMA08 Module 20

Introduction

Direct memory access (DMA) is a method of data transfer whereby large amounts of information may be stored in and retrieved from memory and/or buffers without the need for central processing unit (CPU) intervention. This method is the first example of coprocessing associated with the HC08 Family; the technique traditionally has been used in large, complex, multi-chip computer systems to move blocks of data around the system. Its use in single-chip and embedded control systems is a more recent development brought about by the demands for higher performance and ever increasing integration of functions onto a single silicon chip. The on-chip M68HC08 DMA module (DMA08) takes two bus cycles to transfer a byte of data and four bus cycles to transfer a 16-bit word.

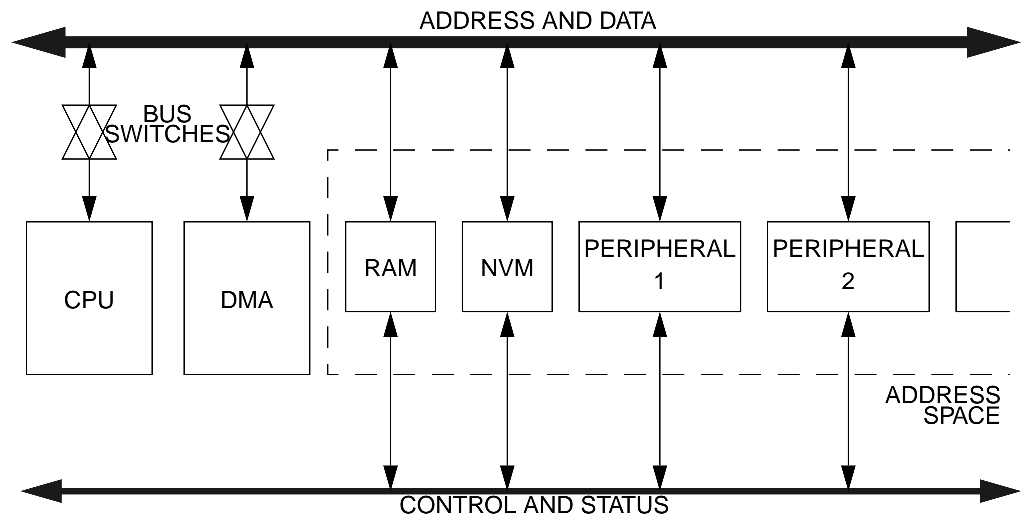


Figure 1. Simplified Block Diagram of a System with DMA

While the use of DMA techniques was relatively obvious and straightforward when large amounts of memory were involved, the usefulness of DMA in an M68HC08 application is, perhaps, less clear. The DMA exists principally because it works with whole blocks of data, rather than on a byte-by-byte basis. CPU intervention is, therefore, only required at block boundaries and not at the end of each byte transfer, thus providing fast, low-latency servicing of peripheral functions, such as serial communications interface (SCI) transmit data, serial peripheral interface (SPI) receive data, etc.

For example: It is highly advantageous (in terms of response time and software size and complexity) to be able to continually write data to an SPI communications module with minimal CPU involvement. This approach can be taken with many peripheral functions, for example, servicing SPI and SCI modules and supporting a timer with data required for input capture and output compare functions. The DMA module also may be used simply to move blocks of memory around, as in the traditional high-end use (including reordering data). Or it can replace a CPU interrupt if the service routine is purely a data transfer, that is, when no data manipulation is involved.

Whether software- or interrupt-driven, data input or output requires considerable CPU involvement. For example, when data is required by a peripheral function, the CPU must first fetch the data from memory and

then write it to the appropriate location. Conversely, when a peripheral signals that it has data to transfer, the CPU must stop what it is doing, read the data and store it at the correct address. The following tasks contribute to the CPU interrupt overhead (limiting the overall transfer rate between the CPU and the peripheral):

- Stacking and unstacking CPU registers
- Loading interrupt vectors
- Loading address pointers
- Reading/writing transfer data
- Incrementing address pointers
- Storing address pointers
- Clearing interrupt flags
- Returning from interrupt

The normal CPU interrupt overhead is 16 cycles. The term “latency” is used to describe the delay between the request for an action and the action’s actual start. The DMA operates with a much lower latency than the CPU and, therefore, can improve system performance.

For a DMA transfer to occur within a system, the DMA subsystem must take control of the system bus, thus temporarily replacing the CPU as the bus master. In general terms, there are two basic methods of DMA operation: cycle stealing and CPU halt. By halting CPU execution, the DMA has access to the system buses all of the time and, therefore, will be able to transfer the data in the minimum time (at the expense of any CPU tasks). In most situations, this method is only really practicable for transferring small amounts of data. For larger amounts of data, however, the DMA can be allocated a proportion of the bus bandwidth. This means that both DMA and CPU effectively process data concurrently by sharing their use of the bus. The overall DMA transfer rate is thereby reduced, but CPU tasks continue to progress.

By using the DMA to move data, the load on the CPU is reduced, as is the amount of executable code required for each move, thus giving faster response times and more efficient use of the available program memory space.

The DMA08 Module

The DMA08 is a coprocessor module for the M68HC08 that can take control of the system address and data buses at any bus cycle boundary. It is designed to allow the transfer of data between any two CPU addressable locations with a minimum of latency. Normal interrupt latency is nine bus cycles, with a total overhead of 16 bus cycles. DMA transfers (when enabled) can have a latency of only two cycles.

The DMA implementation on the M68HC08 is based on a modular approach: One or more independent DMA channels may be provided in a given hardware implementation. Data transfer may be either interrupt- or software-driven.

The modular design approach means that it is an easy matter to expand the DMA from one to seven channels. Figure 2 shows the layout of the 708XL36 die and illustrates the common module height and its advantages. The 708XL36 has three DMA channels; more could be added by expanding the DMA module horizontally.

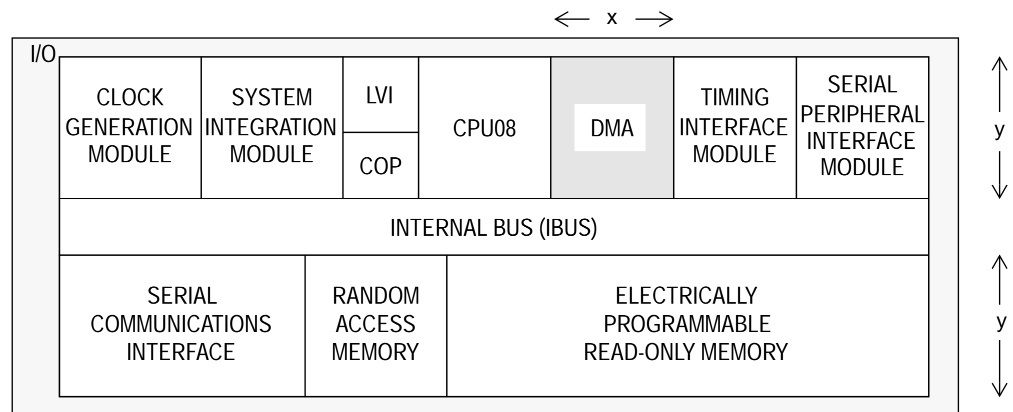


Figure 2. Diagram of the MC68HC708XL36 Layout

Overview and Features

Contents

Introduction 21
Features 22

Introduction

The M68HC08 direct memory access module (DMA08) is constructed in a modular fashion to ensure flexibility and ease of use, while shortening the design effort required each time a DMA module is specified for a new device. From one to seven DMA channels may be specified for a particular implementation; each channel is independent and is enabled only when required. If the DMA module is not enabled or is enabled but not active, it does not consume bus cycles.

There are two versions of the DMA08 module. Version A, implemented in the MC68HC708XL36, is described in the main body of this book. Version B of the DMA08 operates differently in word mode. Version B is described in DMA08 Version B on page 109.

In this manual, the modular nature of the DMA is exploited and only one channel (channel 0) is discussed in detail. For information on precise register addresses, bit names and positions and channel assignments, refer to the specific device's data sheet.

Figure 3 is a simplified representation of the DMA module, showing its functional blocks as perceived by the user. The system control logic enables the DMA to select the set of registers for the DMA channel required, controls the incrementing or decrementing of source and destination addresses and controls the movement of data within the DMA module.

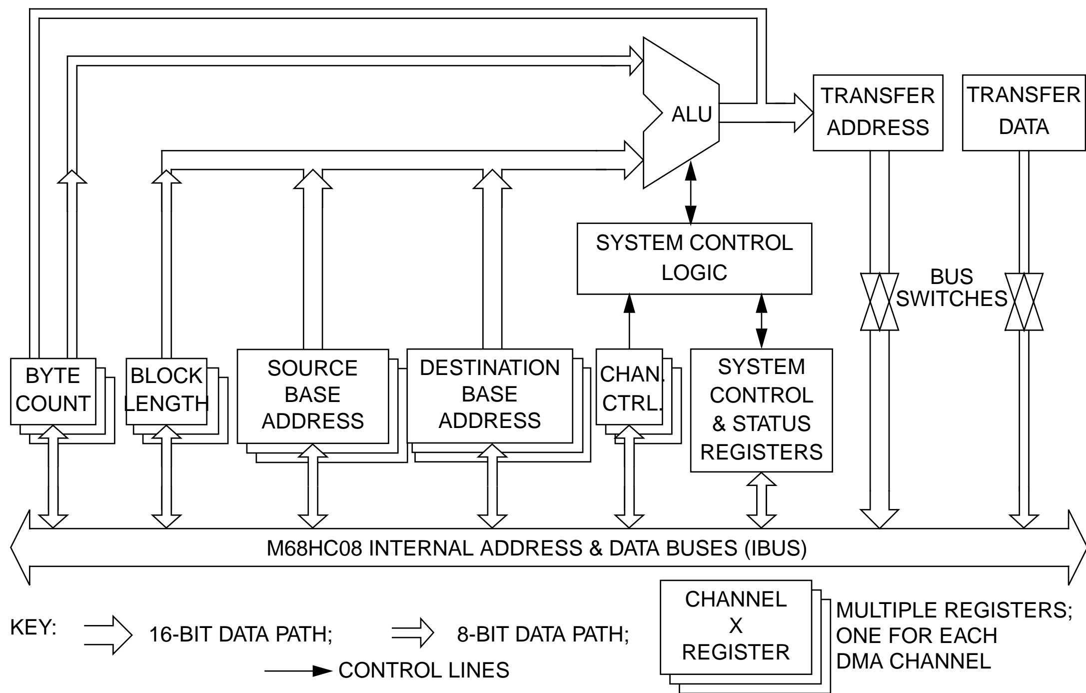


Figure 3. DMA Module Functional Block Diagram

Features

The following are features of the DMA08:

- Modular and Expandable Architecture; Up to Seven Independent Channels with Eight Transfer Source Inputs May Be Specified
- Interrupt-Driven Operation Without CPU Intervention
- Data Transfer Rates of up to 4 Mbytes/sec (8 MHz bus); One Byte Transferred Every Two Bus Cycles
- Low Latency for Data Moves (Two Cycles)
- Separate 16-Bit Source and Destination Addresses
- Byte or Word Transfer Capability
- Single Block or Loop (Repeated Block) Transfer Options
- Programmable Block Length, up to 256 Bytes
- Optional CPU Interrupt Request on Completion of Block Transfer or on Loop Restart
- Programmable DMA Bus Bandwidth. 25, 50, 67, or 100% of Total Bus Bandwidth Can Be Allocated to the DMA Module
- Programmable DMA Transfer Priority, DMA Transfers Can Either Take Priority Over CPU Interrupts, or CPU Interrupts Can Halt the DMA transfer
- Arbitration of Priorities in Multichannel Implementation
- Automatic Peripheral Flag Clearing During DMA Transfer Mode
- Programmable DMA Enable During Wait Mode
- Memory Stretch Capability For Interfacing to Slow Memory
- Built-in Support for Optional Memory Address Extension Module, Allowing the DMA to Access up to 16 M Address Space
- Breakpoint Feature Capable of Halting DMA Operation

DMA Module Description

Contents

Introduction 26
Source and Destination Base Addresses 27
Address Extension Module 28
Byte Count 29
Block Length 30
Arithmetic Logic Unit (ALU) 31
DMA Control and Status 31
Memory Stretch 31
Low-Power Modes 32
  Wait Mode 32
  Stop Mode 32
Breakpoints 33
DMA in Expanded Mode 33

Introduction

Figure 4 shows a simplified representation of the DMA module from a user's perspective. As shown in Figure 4, all the registers may be both read and written by the CPU. The system control logic enables the DMA to select the set of registers for the required DMA channel, controls the incrementing or decrementing of source and destination addresses, and controls the movement of data within the DMA module. DMA access to the M68HC08 bus and the ratio of DMA to CPU cycles is also controlled.

The function of each of the blocks is discussed in the following paragraphs. How to program a DMA transfer is covered in DMA Transfers on page 35 and details of the registers are given in DMA Registers on page 49.

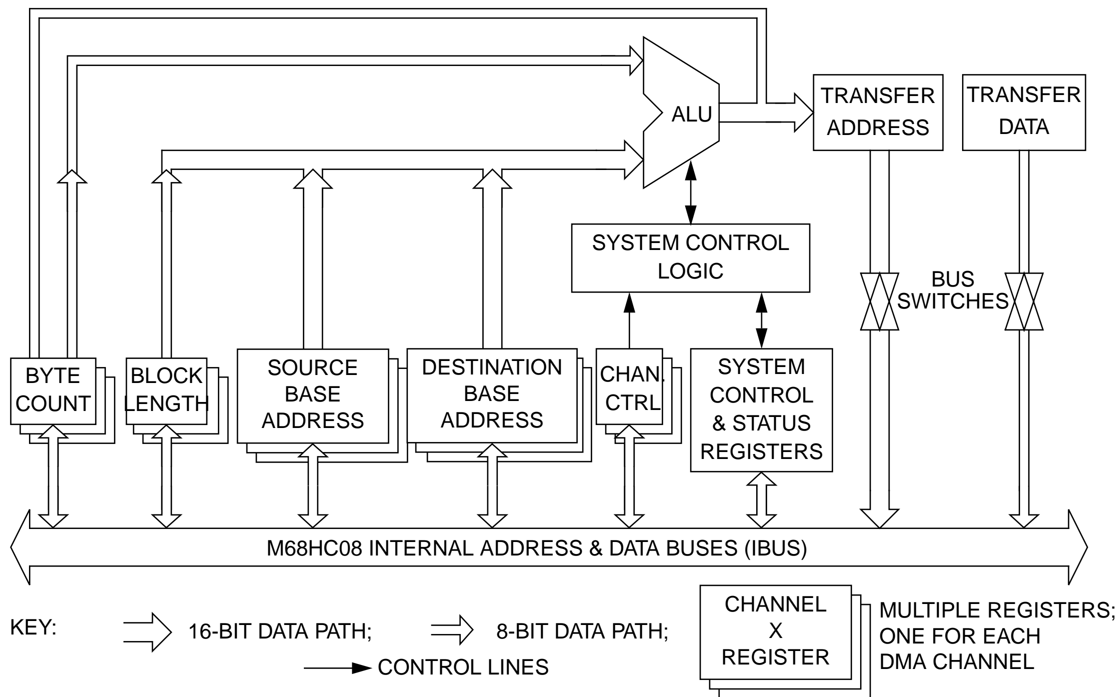


Figure 4. DMA Module Functional Block Diagram

Source and Destination Base Addresses

Each channel has a 16-bit source base and a 16-bit destination base address register. These registers are used to contain the *base of addresses* for the data transfer process. The CPU writes the desired addresses to these registers before the DMA transfer starts. Writing to any of these locations clears the byte count register.

The state of these registers on reset is undefined. See DMA Source Base Address Registers on page 62 and DMA Destination Base Address Registers on page 63 for details.

NOTE: *Source base and destination base registers are unchanged by the transfer process. It is not advisable to write to these registers while transfers on the corresponding channel are enabled.*

Address Extension Module

Support for this optional HC08 Family module is built into the DMA. By programming and using the 8-bit DMA source base and destination base extension registers provided in the address extension module (ADX), the DMA has the capability to access up to 16 M of address space. In the ADX module, each channel has a separate source base and destination base address extension register pair.

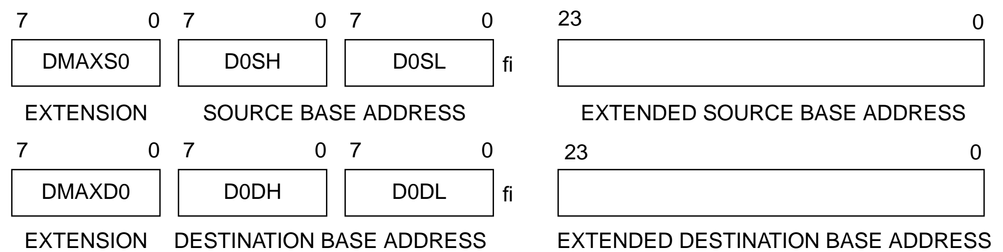


Figure 5. Address Extension Registers

When the address extension capability is used, the DMA addresses are formed by concatenating the source base (or destination base) address register and the relevant address extension register to form the complete address. The address extension register provides the most significant bits of the address. The DMA operation has no other change.

Byte Count

The byte count register is usually written to by the arithmetic logic unit (ALU) of the DMA, although it is also accessible by the CPU. In normal operation, the ALU increments the byte count register after every byte that is transferred. See DMA Byte Count Register on page 65 for details.

This register is cleared:

- by reset
- at the end of a byte transfer, when its contents match those of the block length register
- when either the source base or destination base registers are written
- by writing zero to it

Block Length

The block length register is used to define the number of bytes to be moved in a particular transfer operation. Prior to the start of a transfer, the block length register should be written by the CPU.

When the content of the byte count register, which gets incremented by one with each byte transferred, matches that of the block length register, the current transfer is complete. Subsequent action depends on the state of the loop bit; either the transfer restarts or control reverts to the CPU.

The state of this register on reset is undefined. See DMA Block Length Register on page 64.

NOTE: *A block length of zero results in a block transfer of 256 bytes.*

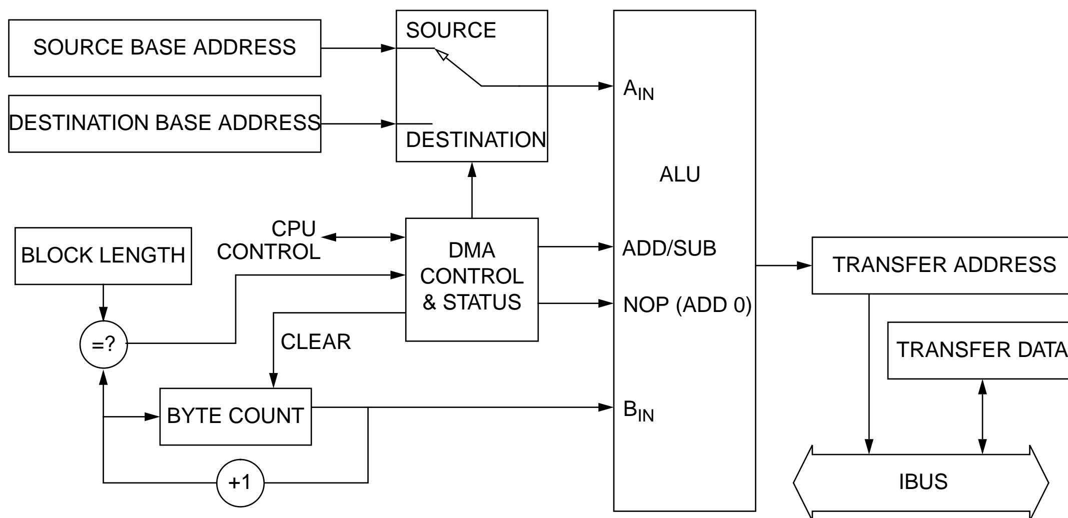


Figure 6. DMA Operation

Arithmetic Logic Unit (ALU)

The ALU is used for calculation of the actual source and destination transfer addresses and for incrementing the byte count register. The ALU is a 16-bit subsystem and therefore can operate on the 16-bit source and destination address registers in a single cycle.

Under the direction of the system control logic, the DMA combines the content of the source (or destination) base address register with that of the byte count register to form a transfer source (or destination) address. This address is stored in the temporary address register and is output on the internal bus (IBUS) at the appropriate time.

DMA Control and Status

The operation of the DMA module is defined and monitored by two groups of registers: one group controls the entire DMA module and the other group is specific to the individual DMA channel. See DMA Module Registers on page 52 and DMA Channel Control Register on page 59 for details.

Memory Stretch

For slow, off-chip or on-chip peripherals that require extra bus cycles, the DMA can stretch the address phase of the bus cycle by an integer number of bus cycles. When this operation is required, a stretch signal is supplied to the DMA module for the number of bus cycles that the address needs to operate correctly.

Low-Power Modes

The CPU08 WAIT and STOP instructions put the MCU in low-power consumption standby modes. The effect on DMA operation is explained in the following paragraphs.

Wait Mode

The DMA wait enable bit (DMAWE) controls the activity of the DMA in wait mode.

- DMAWE set. The DMA can execute a transfer during wait mode, whenever it receives a valid DMA service request. If a CPU WAIT instruction is executed during a transfer, the transfer will continue until completion.
- DMAWE clear. The DMA cannot respond to service requests in wait mode. If a CPU WAIT instruction is executed during a transfer, the transfer will be suspended and will resume when the processor exits wait mode.

Stop Mode

The DMA module is inactive during stop mode. A STOP instruction suspends any DMA transfer in progress. If an external interrupt brings the MCU out of stop mode, the suspended DMA transfer resumes. If a reset brings the MCU out of stop mode, the transfer is aborted.

Breakpoints

If an address match occurs on a DMA address, then the BREAK state is not entered until the end of the current CPU instruction. Thus, a DMA transfer cannot be aborted due to a match on a DMA address comparison with one exception. A DMA transfer can be aborted if a DMA transfer is pending after the BREAK state has been entered from an address match on the previous CPU instruction.

DMA in Expanded Mode

The DMA can be used to access external resources. The control of access to these devices is normally done via an EBI (external bus interface) module. In general, if the CPU can address these external components, so can the DMA.

DMA Transfers

Contents

DMA Operation 36
Transfer Types 37
Cycle-By-Cycle Operation 40
Byte Transfers 40
DMA Activity During a Byte Transfer 41
Word Transfers 42
DMA Activity During a Word Transfer 42
DMA Transfer Latency 44
Example of the DMA Transfer Programming Procedure 44
Address Calculation 46
Bandwidth Control 47

DMA Operation

To transfer a byte of data, the DMA first uses the contents of the source base address register and the byte count register (according to the instructions in the control registers) to generate a transfer source address. The DMA then takes control of the IBUS to read a byte of data from the source location at the transfer source address into a temporary register in the DMA. The transfer destination address is then generated in the same manner and the data byte is written to this destination address.

NOTE: *All DMA registers may be read from or written to at any time by the CPU or by the DMA module. Accessing these registers during a transfer can increase transfer latency. See DMA Transfer Latency* on page 44.

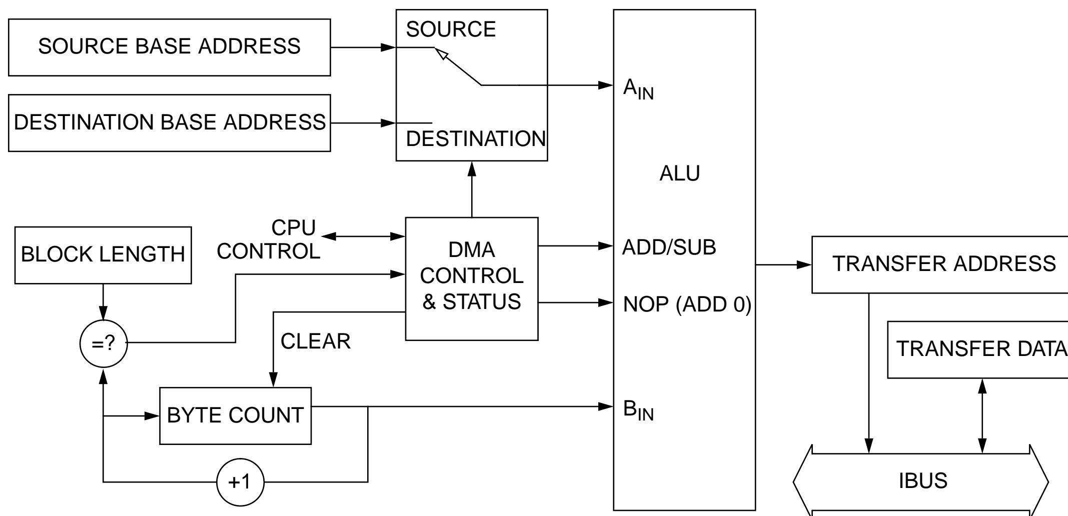


Figure 7. DMA Operation

Transfer Types

The DMA module supports two main types of transfer: hardware interrupt-driven and software-initiated.

Hardware interrupt-driven transfers may be initiated by one of the MCU peripheral subsystems, for example, SCI, SPI, timer, A/D, etc. In this type of transfer, one or more of the system's peripherals is configured to generate a request for DMA transfer. (This configuration is by means of a read/write bit in one of the peripheral's control registers.) The DMA service request is passed to the DMA channel, assuming that this transfer source has been allocated to the channel. The DMA module then arbitrates between the channels, where necessary, and begins the highest priority transfer by halting the CPU clocks and taking control of the IBUS. The relative channel priority increases as the DMA channel number decreases. This means that channel 0 has the highest priority, channel 1 the next highest, and so on. Depending on the state of the DMAP bit, CPU interrupts from other modules may be recognized within a block transfer and the transfer is suspended as a consequence. The transfer will then resume only if the transfer enable bit for the channel is set again. See DMA Status and Control Register on page 54 for further information on the operation of the DMAP bit.

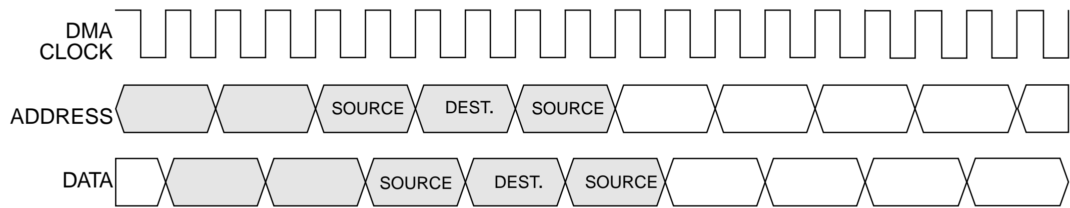
Software-initiated transfers are similar in operation to those that are hardware interrupt-driven. The major difference is that software-initiated transfers are started by setting a bit in the DMA control register (DC2), whereas hardware-initiated transfers are started by a peripheral module setting a flag bit. Software-initiated transfers may be used to initiate a DMA data block transfer. As before, whichever transfer source bit is used to initiate the transfer must have been assigned to an enabled DMA channel.

Interrupt-driven DMA transfers automatically clear the peripheral's DMA flag bit as the DMA reads or writes specific registers within the peripheral. The user must take special care with registers that are "write-cleared" with DMA bus bandwidths of 100%. Under these conditions the number of CPU cycles between one channel finishing and another starting can vary depending on when the interrupt was cleared.

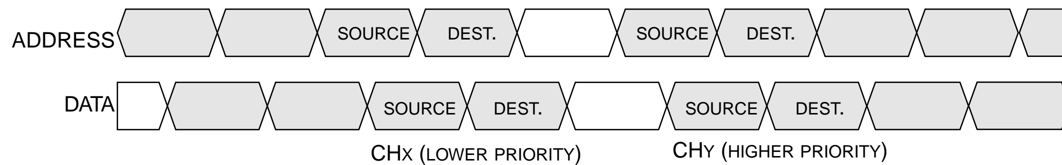
Figure 12 (a) shows a standard 100% bandwidth write-clearing transfer. The DMA08 is not cleared for the next transfer, which is subsequently aborted, until after the write to the destination. The DMA, therefore, issues an unused source read.

Figure 12 (b), Figure 12 (c), and Figure 12 (d) show how the DMA operation is affected under different circumstances.

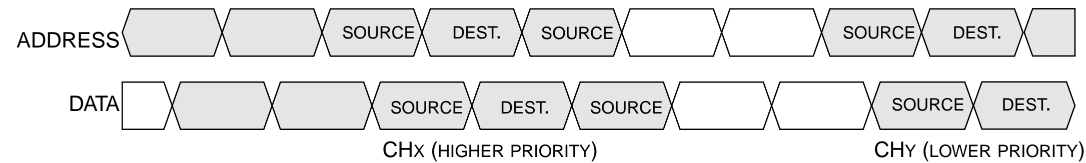
Software-initiated transfers and read-clearing interrupts have DMA behavior as shown in Figure 9.



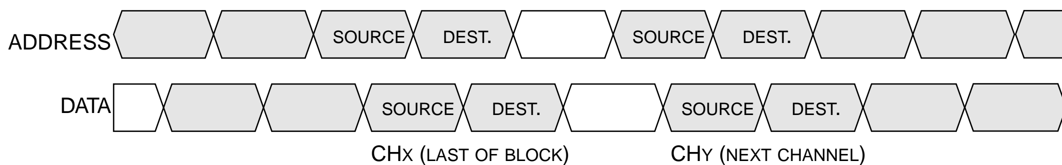
(a) Standard 100% Bandwidth with Write-Clearing DMA Interrupt Flag



(b) Write-Clearing with Higher Priority Service Request Pending



(c) Write-Clearing with Lower Priority Service Request Pending



(d) Write-Clearing with Last Transfer of the Block

DMA CONTROLLED BUS CYCLE   CPU CONTROLLED BUS CYCLE

Figure 8. Write-Clearing Interrupts at 100% Bandwidth

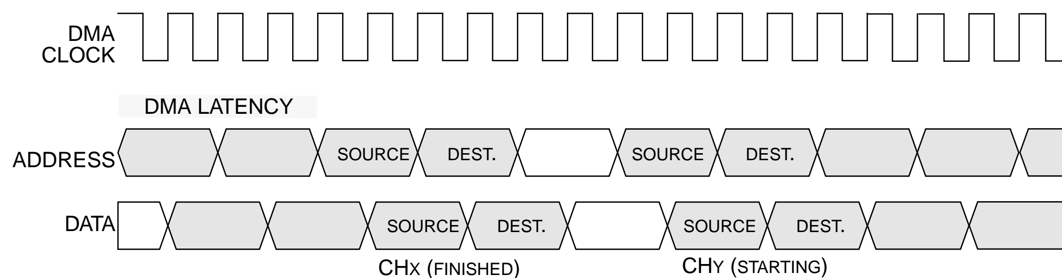


Figure 9. Software-Initiated Transfers/Read-Clearing Service Request Flags Behavior with 100% Bandwidth

Cycle-By-Cycle Operation

When the DMA transfers data, it takes control of the address bus, data bus and read/$\overline{\text{write}}$ (R/$\overline{\text{W}}$) line. During the transfer, the DMA instructs the system integration module (SIM) to suspend the CPU clock; the state of the CPU, therefore, remains unchanged until the end of the transfer when the DMA relinquishes control of the buses and R/$\overline{\text{W}}$ line to the CPU. The CPU then resumes operation as though nothing had happened.

The DMA uses two bus cycles to transfer a byte and four cycles to transfer a 16-bit word. A two cycle latency allows the DMA to respond to an interrupt. During these two cycles normal CPU operation continues. Selection of the transfer mode is via the byte-word control (BWC) bit in the channel control register. The actions of the DMA for byte and word transfer modes, respectively, are discussed in the following paragraphs.

NOTE: *During a DMA transfer, any CPU interrupts can be recognized at the end of a byte/word transfer. See* *DMA Module Registers* on page 52.

Byte Transfers

Figure 10 shows the timing of a single-byte DMA transfer, with reference to the DMA clock, which is twice the bus frequency.

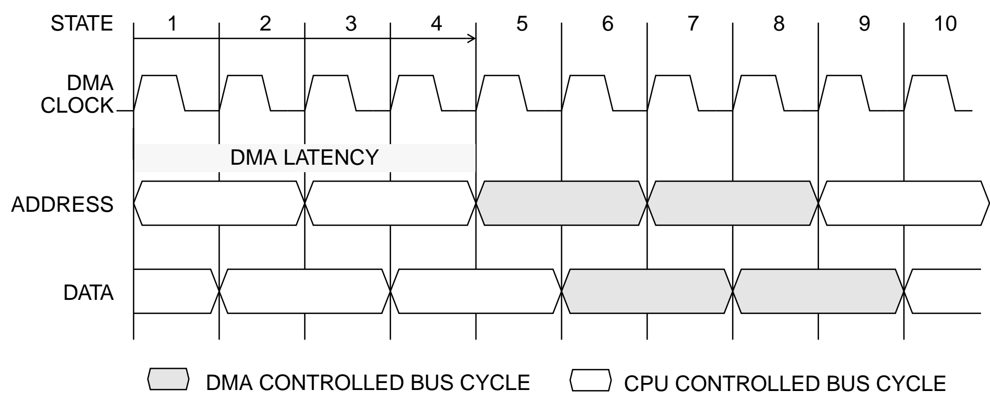


Figure 10. MCU Bus Activity During a DMA Byte Transfer

DMA Activity During a Byte Transfer

An explanation of principal DMA actions during each state is given in **Table 1**. The numbers refer to the state numbers shown in **Figure 10**.

Table 1. Byte Transfer Activity

State	Operation
1	Service request occurs (software or hardware)
2	Arbitrates channel priorities and activates a channel
3	Generates internal control signals
4	Calculates the source address
	Stores the source address in a temporary register
5	Takes control of the address bus
	Drives the source address onto the address bus
	Takes control of the $R/\overline{W}$ line and drives it high
	Calculates the destination address
	Stores the destination address in a temporary register
6	Takes control of the data bus
	Latches source data in a temporary register
	Increments the byte count register
7	Drives the destination address onto the address bus
	Drives the $R/\overline{W}$ line low
	Subtracts the byte count register from the block length register. If the result is zero, the channel enable bit is cleared. Interrupts the CPU if the interrupt enable (IE) bit is set
8	Drives the source data onto the data bus
9	Releases the address bus and $R/\overline{W}$ to the CPU
10	Relinquishes the data bus to the CPU

Word Transfers

A word transfer occurs when the source and/or destination is set to static and the relevant BWC bit is set. Figure 12 shows the timing of a 16-bit word DMA transfer, with reference to the DMA clock, which is at twice the bus frequency.

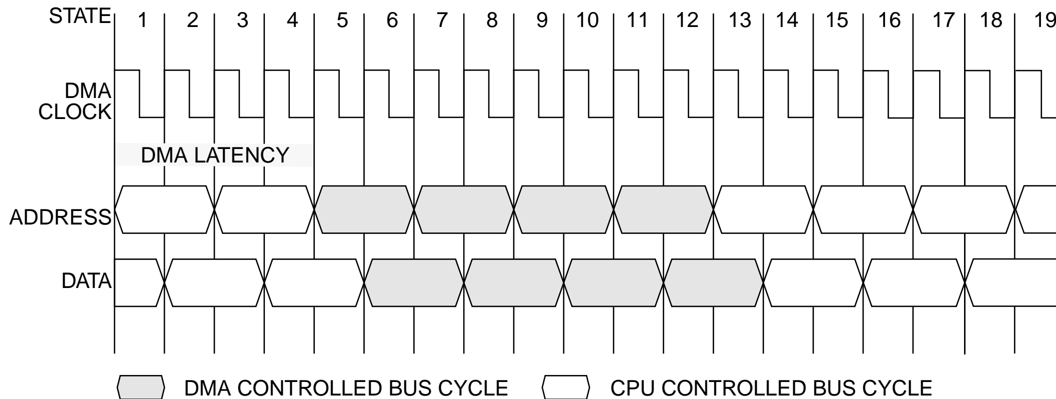


Figure 11. MCU Bus Activity During a DMA Word Transfer (100% Bandwidth)

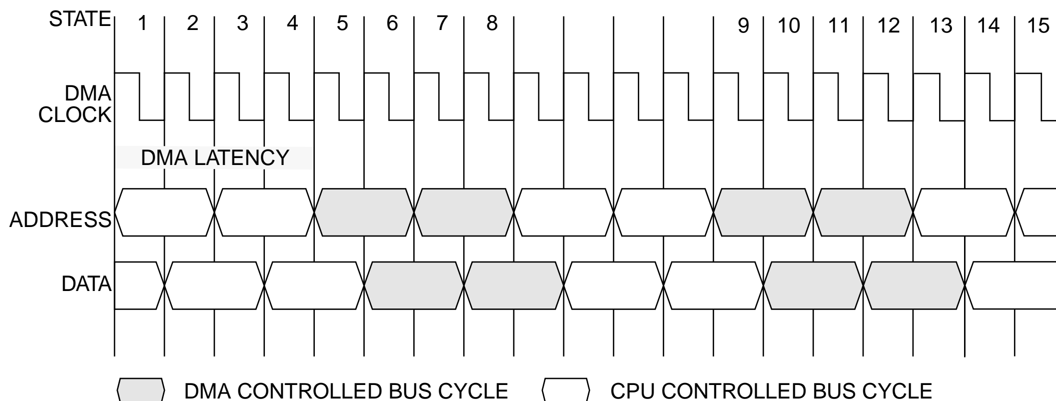


Figure 12. MCU Bus Activity During a DMA Word Transfer (50% Bandwidth)

DMA Activity During a Word Transfer

An explanation of principal DMA actions during each state is given in Table 2. The numbers refer to the state numbers shown in Figure 12, that is, at 100% bandwidth.

Table 2. Word Transfer Activity

State	Operation
1	Service request occurs (software or hardware)
2	Arbitrates channel priorities and activates a channel
3	Generates internal control signals
4	Calculates the source address and stores it in a temporary register
5	Takes control of the address bus
	Drives the source address onto the address bus
	Takes control of the $R/\overline{W}$ line and drives it high
	Calculates the destination address
	Stores the destination address in a temporary register
6	Takes control of the data bus
	Latches source data in a temporary register
	Increments the byte count register
7	Drives the destination address onto the address bus
	Drives the $R/\overline{W}$ line low
	Subtracts the byte count register from the block length register
8	Drives the source data onto the data bus
	Calculates the source address for the second byte
	Stores the source address in a temporary register
9	Drives the source address for the second byte onto the address bus
	Drives the $R/\overline{W}$ line high
	Calculates the destination address for the second byte
	Stores the destination address in a temporary register
10	Latches source data into a temporary register
	Increments the byte count register
11	Drives the destination address for the second byte onto the address bus
	Drives the $R/\overline{W}$ line low
	Subtracts the byte count register from the block length register. If the result is zero, the channel enable bit is cleared. Interrupts the CPU if the interrupt enable (IE) bit set.
12	Drives the source data for the second byte onto the data bus
13	Releases the address bus and $R/\overline{W}$ to the CPU
14	Relinquishes the data bus to the CPU

DMA Transfer Latency

DMA transfer latency is usually two bus cycles, but it can be extended to three if a read/write to a DMA channel register is taking place during the last of these two cycles. The access does not have to be to the active channel to increase latency. Applications that use multiple DMA channels may have to allow for a 3-cycle latency in performance calculations.

For comparison, the interrupt latency of the CPU is nine bus cycles.

Example of the DMA Transfer Programming Procedure

The following procedure illustrates the required sequence of actions to program a DMA transfer. In all DMA channels, the operation and the register structure are identical, so only channel 0 will be described here:

1. Turn off channel if previously enabled.
2. Write the source base address to source base address registers (D0SH and D0SL).
3. Write the destination base address to destination base address registers (D0DH and D0DL).
4. In the DMA channel control register (D0C):
   a. Select increment/decrement/remain static for the source base addresses and for the destination base addresses.
   b. Select byte/word (takes effect only if source and/or destination base address calculation is set to static).
   c. Assign the DMA channel to a DMA transfer source input.
5. In the DMA channel block length register (D0BL), enter the number of bytes to be transferred.

NOTE: *Because each word equals two bytes, the block length should always be an even number for word transfers.*

6. In the DMA status and control register (DSC):
   a. Enable or disable looping of the source and destination addresses for the channel.
   b. Select DMA transfers priority.
   c. Enable or disable DMA transfer operation during wait mode.
7. To control the DMA transfer with software, set the SWIx bit in DMA control register 2 (DC2) that corresponds to the selected transfer source input. The transfer will begin two cycles after the channel is enabled.
8. For a hardware interrupt-driven transfer, the transfer begins two cycles after the selected peripheral generates a DMA service request, providing the DMA channel is enabled.
   d. In DMA control register (DC1):
   a. Enable or disable CPU interrupt request generation on completion of DMA transfer.
   b. Select the DMA bandwidth.
   c. Enable the DMA channel.

Address Calculation

The arithmetic logic unit (ALU) is a 16-bit subsystem that can calculate the 16-bit source and destination addresses in one cycle. During a DMA transfer, the ALU performs the following actions:

- Calculates the transfer source and transfer destination addresses
- Increments the byte count register for each byte transferred
- Determines when a block or loop transfer is complete by comparing the content of the byte count register with the value programmed into the block length register

The DMA source base address registers and destination base address registers contain the base addresses for a DMA transfer. The ALU uses these address registers as base pointers when it starts the transfer. The byte count register contains the number of bytes transferred to this point in the current DMA operation. The ALU uses the base address registers and the byte count register to calculate the actual source and destination addresses in the following manner:

- When an address is configured to increment, the ALU adds the contents of the byte count register to the base address.
- When an address is configured to decrement, the ALU subtracts the contents of the byte count register from the base address.
- When an address is configured to remain static, the ALU simply uses the base address (for example, it adds $0000 to the base address).

The DMA module can be programmed to stop after a number of bytes is transferred (block mode) or to loop back to the base addresses and continue the transfer (loop mode).

Figure 7 shows schematically how the DMA module calculates source and destination addresses.

Bandwidth Control

The bandwidth control bits in DMA control register 1 are used to apportion the available MCU bus cycles between the DMA and the CPU. By setting these two bits, it is possible to assign the DMA either 25, 50, 67, or 100% of the bus cycles. See DMA Control Register 1 on page 52 for information on setting the bus bandwidth bits. Together with the DMAP bit, the bus bandwidth bits control the relative priorities of the DMA and the CPU.

Table 3. DMA/CPU Bus Bandwidth Sharing

BB1:BB0	DMA/CPU Bus Bandwidth Sharing	
	Ratio (DMA/CPU)	DMA:CPU cycles
00	25/75%	2:6
01	50/50%	2:2
10	67/33%	2:1
11	100/0%	The DMA controls the bus for as long as required.

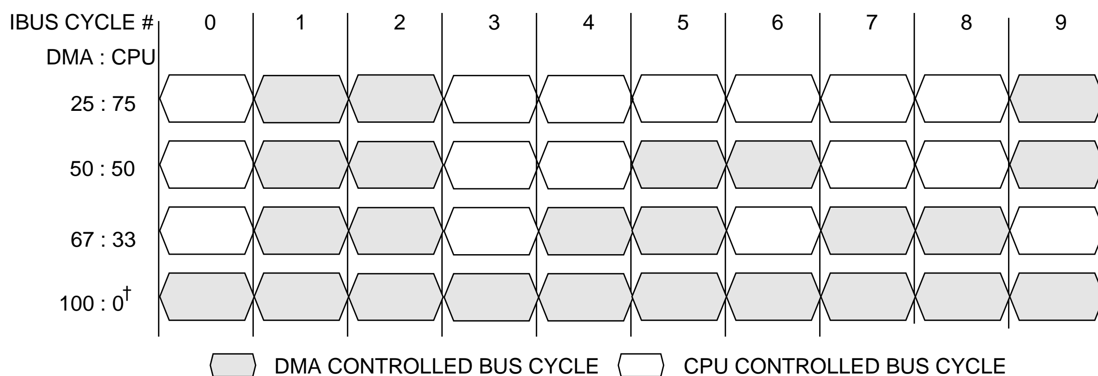


Figure 13. DMA and CPU Use of the IBUS

NOTE: *Bus activity looks the same for both byte and word modes.*

NOTE: *Regardless of the bandwidth setting, the DMA only consumes bus cycles during a transfer.*

When the bus bandwidth is set to 25%, two out of every eight bus cycles are available for DMA transfers. Similarly, for 50% and 67%, the ratios are two out of four and two out of three, respectively. This also applies in word transfer mode. (See DMA Control Register 1 on page 52 for more information.) When 100% is selected, the DMA is allocated every available bus cycle.

NOTE: *The CPU always executes at least one cycle before the next DMA loop begins, even if the DMA has been allocated 100% of the bus bandwidth. This ensures that it is impossible to lock out the CPU completely through inadvertently programming an endless DMA transfer.*

Clearly, since there is only one system bus for addressing memory, sharing it with the DMA means that the performance of the CPU will be affected. When deciding on the bus allocation, care needs to be taken to understand the size and frequency of the expected DMA transfers, so that an appropriate share of the common bus resource is chosen. For a small and relatively infrequent transfer, it may be appropriate to give the DMA 100% of the bus to ensure that the transfers occur as quickly as possible. However, where large or frequent transfers are concerned, due consideration must be given to the needs of the main CPU routines. Further caution is required when DMA transfers are given priority over CPU interrupts (by setting the DMAP bit) and the DMA is given 100% of the bus bandwidth. For example:

1. When CPU interrupts have priority (DMAP bit = 0):
   a. The CPU will have access to those bus cycles not allocated to DMA.
   b. Any CPU interrupt will suspend any current DMA transfer at the end of the byte or word transfer in progress. All channel enable bits are cleared. DMA operation will resume only when a channel is specifically re-enabled.
2. When DMA transfers have priority:
   a. The CPU will have access to those bus cycles not allocated to DMA.
   b. CPU interrupts will not be recognized until all current DMA activity is complete.

DMA Registers

Contents

Introduction 50

DMA Register Latency 51

DMA Module Registers 52
- DMA Control Register 1 52
- DMA Status and Control Register 54
- DMA Control Register 2 57

Individual DMA Channel Registers 59
- DMA Channel Control Register 59
- DMA Source Base Address Registers 62
- DMA Destination Base Address Registers 63
- DMA Block Length Register 64
- DMA Byte Count Register 65

Introduction

The M68HC08 DMA module may be implemented on silicon with one to seven independent DMA channels. Since the operation of each channel is identical, only channel 0 is described here.

In addition to the general control registers that control the operation of the entire DMA module, each channel requires seven 8-bit registers to define the details of its own operation. Four of these registers are concatenated into the two 16-bit source base and destination base address registers; the others are used for individual channel control, for specifying the size of the transfer block and for counting the number of bytes transferred.

Register name	Adress	Bit 7	6	5	4	3	2	1	Bit 0
DMA control 1 (DC1)	$xxxx	BB1	BB0					TEC0	IEC0
DMA status and control (DSC)	$xxxx	DMAP			L0	DMAWE			IFC0
DMA control 2 (DC2)	$xxxx	SWI7	SWI6	SWI5	SWI4	SWI3	SWI2	SWI1	SWI0
Channel 0 register name	**Address**	**Bit 7**	**6**	**5**	**4**	**3**	**2**	**1**	**Bit 0**
Source base address (D0SH)	$xxxx	AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
Source base address (D0SL)	$xxxx+1	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
Destination base address (D0DH)	$xxxx	AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
Destination base address (D0DL)	$xxxx+1	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
Control (D0C)	$xxxx	SDC3	SDC2	SDC1	SDC0	BWC	DTS2	DTS1	DTS0
Block length (D0BL)	$xxxx+1	BL7	BL6	BL5	BL4	BL3	BL2	BL1	BL0
Byte count (D0BC)	$xxxx	BC7	BC6	BC5	BC4	BC3	BC2	BC1	BC0

NOTE: *Shaded bits are reserved for additional DMA channels. For the maximum number of channels, registers DC1A and DSCA are required. The extra TECx, IECx, Lx, and IFCx bits for each channel are added.*

Reference should be made to the specific device data sheet for details of the number of channels, register addresses, and bit names.

DMA Register Latency

The following registers control and monitor operation of the channel of the DMA module:

- DMA control register 1 (DC1)
- DMA status and control register (DSC)
- DMA control register 2 (DC2)

NOTE: *The operation of a single channel is described throughout this manual for consistency.*

DC1, DSC, and DC2 can be read/written during a DMA transfer with no affect on DMA latency.

The following registers control operation of an individual DMA channel. (Each channel has an identical set of these registers.)

- DMA channel 0 source base address register, high and low byte (D0SH:D0SL)
- DMA channel 0 destination base address register, high and low byte (D0DH:D0DL)
- DMA channel 0 control register (D0C)
- DMA channel 0 block length register (D0BL)
- DMA channel 0 byte count register (D0BC)

Writing to any of these registers during the cycle prior to a transfer can add one cycle to DMA latency.

DMA Module Registers

The following registers affect all the DMA channels.

DMA Control Register 1

DMA control register 1 (DC1) is used to enable individual channels to transfer data, to enable DMA interrupts to the CPU, and to allocate the amount of the total bus bandwidth that the DMA can use.

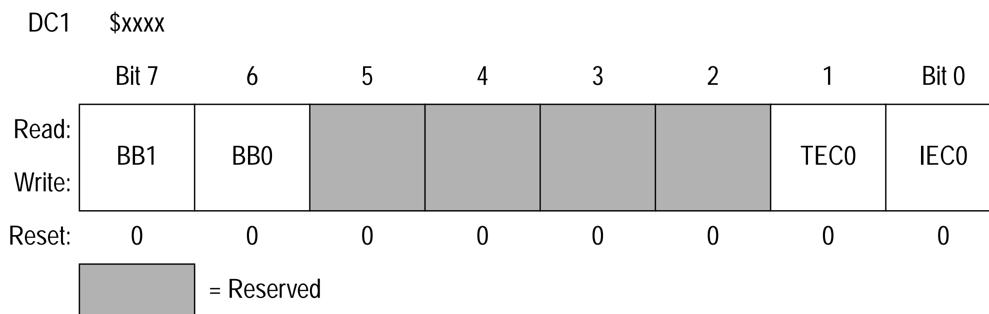


Figure 14. DMA Control Register (DC1)

BB1:BB0 — Bus bandwidth control bits

These read/write bits control the ratio of DMA to CPU bus activity. The DMA can be allocated 25%, 50%, 67%, or 100% of the total bus bandwidth, with the CPU taking the remaining cycles. After a DMA transfer, the next transfer is inhibited until the required number of cycles has been executed by the CPU. See Figure 7 on page 36 and Figure 13 on page 47.

For DMA transfers of a few bytes, giving the DMA module 100% of the bus bandwidth may be appropriate. However, for large, software-initiated transfers, limiting the bus bandwidth of the DMA module may be necessary to maintain an acceptable level of CPU activity.

TEC0 — Transfer enable (channel 0)

This read/write bit enables channel 0 to take control of the M68HC08 data and address buses two cycles after a valid request for DMA transfer occurs.

1 = DMA channel 0 enabled.

0 = DMA channel 0 disabled.

This bit is cleared on completion of a block transfer if looping is disabled. (See DMA Status and Control Register on page 54.)

NOTE: *A CPU interrupt request (if recognized) will disable all DMA channels by clearing the transfer enable control (TECx) bit for each channel if DMAP = 0.*

IEC0 — CPU interrupt enable (channel 0)

This read/write bit enables channel 0 to generate a CPU interrupt when the interrupt flag IFC0 becomes set.

1 = DMA interrupt to the CPU enabled (channel 0)

0 = DMA interrupt to the CPU disabled (channel 0)

DMA Status and Control Register

The DMA status and control register contains a flag that indicates the completion of DMA block transfer. It also controls the DMA loopback facility and the priority of DMA transfers with respect to CPU interrupts. One bit is used to enable DMA operation during wait mode.

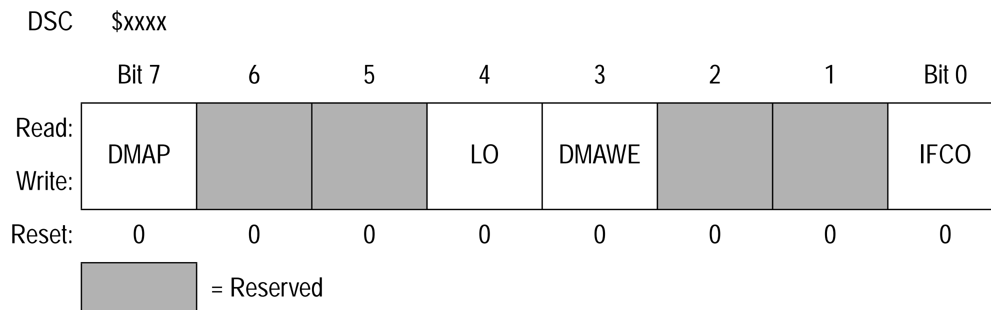


Figure 15. DMA Status and Control Register (DSC)

DMAP — DMA priority

This read/write bit controls the priority of DMA module transfers with respect to CPU interrupt requests.

1 = DMA has priority.
DMA activity cannot be interrupted by the CPU. CPU interrupts are recognized only after the DMA releases the internal bus.

0 = CPU has priority.
CPU interrupts are recognized at the end of the current byte or word transfer.

When DMAP = 1, CPU interrupts are recognized only after the DMA module releases the internal bus. In this case, the increase in CPU interrupt latency depends on the size of the block and the state of the bus bandwidth bits, BB0:BB1. For example, for a 256-byte block transfer with a 100% bandwidth, the worst case latency is 512 cycles. This is only true if the DMA request is permanently asserted, as is the case with a software-initiated transfer. A hardware-initiated transfer will continue to transfer bytes or words until the request bit is cleared. This is normally done by reading or writing to a data register of the module that made the request. In situations where the DMA does not have 100% of the bus bandwidth, CPU interrupts will be serviced during the CPU-controlled bus cycles.

When DMAP = 0, CPU interrupts are recognized after the DMA completes the byte or word transfer in progress; the CPU then disables the DMA by clearing the transfer enable bit(s) for all channels. Therefore, the DMA can increase CPU interrupt latency by up to three cycles, or five in the case of a word transfer. Software must re-enable a DMA channel, if required, by setting the channel enable bit (TEC0). When this is done, the DMA transfer can continue.

L0 — Loop enable bit

This read/write bit enables DMA transfer looping. On completion of a block transfer, the DMA restarts the transfer from the addresses contained in the source base address and destination base address registers. In this way, a circular buffer can be set up or serviced. Reset clears the L0 bit.

1 = Looping is enabled (loop mode).
0 = Looping is disabled (block mode).

When looping is enabled (L0 = 1), the DMA module takes the following actions after it has completed transferring the number of bytes specified in the block length register:

- Sets the channel interrupt flag (IFC0)
- Generates an interrupt request, if enabled (IEC0 = 1)
- Clears the byte count register
- Continues transfer from the base address.

When looping is disabled (L0 = 0), the DMA module takes the following actions after it has completed transferring the number of bytes specified in the block length register:

- Sets the channel interrupt flag (IFC0)
- Generates an interrupt request, if enabled (IEC0 = 1)
- Clears the byte count register
- Disables the channel by clearing the TEC0 bit

NOTE: *The CPU always executes at least one cycle before the next DMA loop begins, even if the DMA has been allocated 100% of the bus bandwidth. This ensures that it is impossible to lock out the CPU completely through inadvertently programming an endless DMA transfer. Care must be*

taken when using the settings DMAP =1, BB1:BB0 = 11, and L0 = 1 during a software-initiated transfer.

DMAWE — DMA wait enable

This read/write bit is used to enable DMA operation while the CPU is in wait mode. Reset clears the DMAWE bit.

1 = DMA transfer possible during wait mode.

0 = DMA transfers suspended in wait mode.

IFC0 — Interrupt flag

This read/write bit becomes set when a DMA transfer is complete or at the end of each transfer loop. The interrupt flag (IFC0) becomes set when the content of the byte count register equals that of the block length register. IFC0 = 1 will generate a CPU interrupt request if the corresponding IEC0 bit is set in the DMA status and control register. If interrupts are disabled, IFC0 can be polled by software to see when a transfer or loop is complete. But care must be taken not to inadvertently clear the IFCx bits of other channels in a multi-channel implementation. IFC0 is cleared by reading it and then writing a zero to it. Reset clears this bit.

1 = DMA transfer complete

0 = DMA transfer not complete

DMA Control Register 2

DMA control register 2 is used to initiate a DMA transfer.

DC2 $xxxx

	Bit 7	6	5	4	3	2	1	Bit 0
Read: / Write:	SWI7	SWI6	SWI5	SWI4	SWI3	SWI2	SWI1	SWI0
Reset:	0	0	0	0	0	0	0	0

Figure 16. DMA Control Register 2 (DC2)

SWI7–SWI0 — Software initiate 7–0

Each of these read/write bits is used to initiate a DMA request from one of the eight DMA transfer sources (see **Table 4**). For the transfer to begin, the channel first must be enabled by the TEC0 bit in the DMA control register. The channel also must be assigned, by its DMA channel control register, to the relevant DMA transfer input.

1 = Start software-initiated DMA transfer
0 = No software-initiated DMA transfer

The DMA responds to the setting of an SWIx bit in the same way that it responds to a hardware service request. The DC2 bit position corresponds to the DTS [2:0] assignment in the channel. The SWIx bit is ORed with the hardware service request. The SWIx bits are cleared only by a write to the DC2 registers.

Table 4. DMA Transfer Source Selection

DTS[2:0]	Interrupt-Driven Transfer Source	Software-Driven
000	DMA service request input 0	SWI0
001	DMA service request input 1	SWI1
010	DMA service request input 2	SWI2
011	DMA service request input 3	SWI3
100	DMA service request input 4	SWI4
101	DMA service request input 5	SWI5
110	DMA service request input 6	SWI6
111	DMA service request input 7	SWI7

NOTE: *Because the SWIx bits are ORed with the hardware service requests, take care when choosing which SWIx bits to use. Choose a bit that corresponds to a hardware service request that is not being used. The DMA service request enable bit in the corresponding peripheral should be turned off.*

A software-initiated DMA transfer differs from a normal hardware-initiated transfer in that the entire block of data can be transferred before any CPU interrupts are recognized. In the case of loop mode, the CPU may have only one cycle in every 512 cycles to execute its code. SWIx remains asserted after a transfer is complete, but TEC will be cleared. To repeat the same transfer, it is only necessary to reassert TEC.

Individual DMA Channel Registers

Each channel in the DMA module has the following set of registers.

DMA Channel Control Register

The DMA channel control register (D0C) contains bits to control the calculation of the source base and destination base addresses throughout a transfer, selects byte or word transfer mode, and assigns the channel to one of the eight possible DMA transfer sources. The state of the bits in the DMA channel control register is undefined after reset.

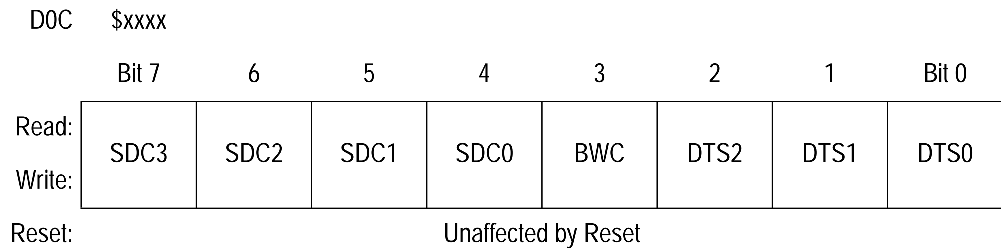


Figure 17. DMA Channel Control Register (D0C)

SDC3–SDC0 — Source/destination base address control bits 3–0

These read/write bits control calculation of the source base and destination base addresses as shown in **Table 5**.

Table 5. DMA Channel Control Register

SDC[3:0]	Source Address	Destination Address
1010	Increment	Increment
1001	Increment	Decrement
1000	Increment	Static
0110	Decrement	Increment
0101	Decrement	Decrement
0100	Decrement	Static
0010	Static	Increment
0001	Static	Decrement
0000	Static	Static

The DMA calculates an incremented address by adding the contents of the byte count register to the base address contained in the source base (or destination base) address register. Similarly, to calculate a decremented address, the DMA subtracts the byte count register from the base address. For static addressing modes, the DMA simply adds zero or one to the base address, depending on whether the DMA is configured for byte or word transfers.

BWC — Byte/word control

This read/write bit determines whether the DMA channel transfers 8-bit bytes or 16-bit words. A DMA byte transfer takes two bus cycles and a word transfer takes four bus cycles. CPU interrupts can be recognized only after the completion of a byte or word transfer.

1 = 1 = DMA transfers 16-bit words
0 = 0 = DMA transfers 8-bit bytes

The BWC bit has no effect unless either the source base or destination base address is static or both are static. **Table 6** shows how the DMA calculates addresses in word transfers. When both the source base and destination base addresses are static, the first byte of the word transfers from the source base address to the destination base address. The second byte transfers from the source base address plus one to the destination base address plus one. When either the source base or destination base address increments or decrements, the DMA module transfers bytes from or to incrementing or decrementing addresses.

Table 6. DMA Word Transfer

		STATIC SOURCE	STATIC DESTINATION	INCREMENTED SOURCE	STATIC DESTINATION	STATIC SOURCE	INCREMENTED DESTINATION	DECREMENTED SOURCE	STATIC DESTINATION	STATIC SOURCE	DECREMENTED DESTINATION
Word	**Byte**	**From**	**To**	**From**	**To**	**From**	**To**	**From**	**To**	**From**	**To**
1	1	SBA	DBA	SBA	DBA	SBA	DBA	SBA	DBA	SBA	DBA
	2	SBA+1	DBA+1	SBA+1	DBA+1	SBA+1	DBA+1	SBA–1	DBA+1	SBA+1	DBA–1
2	3	SBA	DBA	SBA+2	DBA	SBA	DBA+2	SBA–2	DBA	SBA	DBA–2
	4	SBA+1	DBA+1	SBA+3	DBA+1	SBA+1	DBA+3	SBA–3	DBA+1	SBA+1	DBA–3
3	5	SBA	DBA	SBA+4	DBA	SBA	DBA+4	SBA–4	DBA	SBA	DBA–4
	6	SBA+1	DBA+1	SBA+5	DBA+1	SBA+1	DBA+5	SBA–5	DBA+1	SBA+1	DBA–5
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
n	2n–1	SBA	DBA	SBA+ 2n–2	DBA	SBA	DBA+ 2n–2	SBA– (2n–2)	DBA	SBA	DBA– (2n–2)
	2n	SBA+1	DBA+1	SBA+ 2n–1	DBA+1	SBA+1	DBA+ 2n–1	SBA– (2n–1)	DBA+1	SBA+1	DBA– (2n–1)

Note: SBA = Source Base Address, DBA = Destination Base Address

DTS2–DTS0 — DMA transfer source bits 2–0

These read/write bits assign the individual DMA channel to one of the eight transfer source inputs, as shown in **Table 4**. This information is hardwired for each M68HC08 device. For details about peripheral module interrupts designated as service request inputs, refer to the specific device data sheet.

DMA Source Base Address Registers

The DMA channel reads its data from the source base address defined in the 16-bit source base address register. During a block transfer, the DMA determines successive source base addresses by adding to (increment) or subtracting from (decrement) the base address. In static address transfers, the DMA finds the source address by simply adding zero or one to the address, depending on whether the DMA is configured for byte or word transfers.

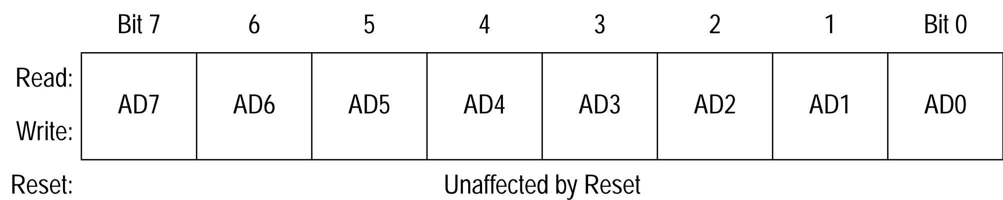


Figure 18. DMA Source Base Address Registers (D0SH and D0SL)

DMA Destination Base Address Registers

The DMA channel writes data to the destination address defined by the 16-bit destination base address register. During a block transfer, the DMA determines successive destination base addresses by adding to (increment) or subtracting from (decrement) the base address. In static address transfers, the DMA finds the destination base address by simply adding zero or one to the base address, depending on whether the DMA is configured for byte or word transfers. The state of the destination base address registers is undefined after reset.

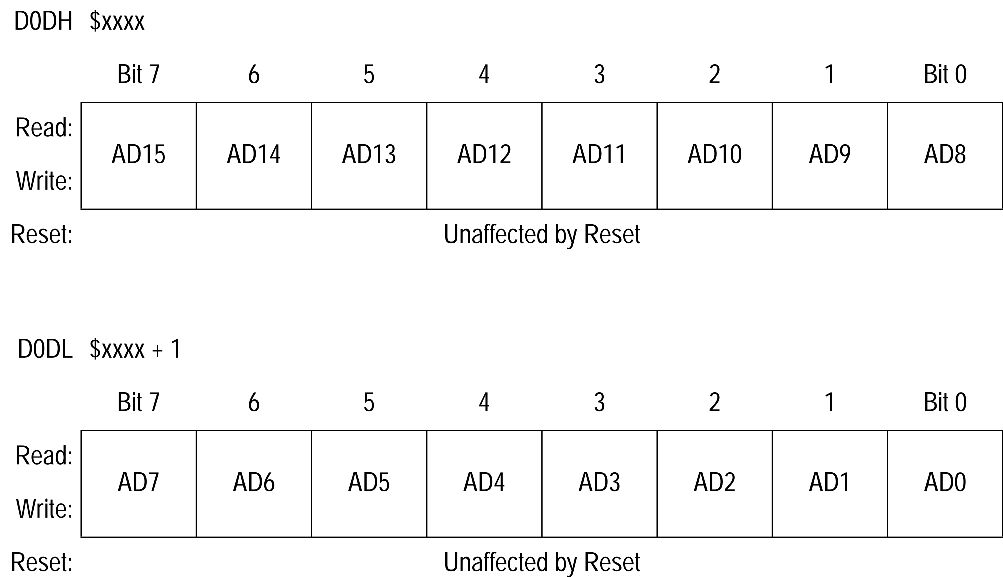


Figure 19. DMA Destination Base Address Registers (D0DH and D0DL)

NOTE: *Support for the optional address extension module is built into the DMA. In this case, the source base and destination base address registers are effectively 24 bits. See* *Address Extension Module* on page 28*.*

DMA Block Length Register

The read/write block length register controls the number of bytes transferred. In word mode, the block length register must be written with the number of words times two. During a block transfer, the DMA compares the number programmed into the DMA block length register to the number in the DMA byte count register. When the byte count reaches the value in the block length register, the DMA does the following:

- Sets the channel interrupt flag (IFC0)
- Generates CPU interrupt request, if enabled (IEC0 = 1)
- Clears the byte count register

If looping is disabled (L0 = 0), the DMA then stops the transfer by clearing the TEC0 bit in DMA control register 1, thus disabling the channel. If looping is enabled (L0 = 1), the DMA continues the transfer from the base address.

A value of $00 in the register designates the maximum block length of 256 bytes. After reset, the state of the DMA block length registers is undefined.

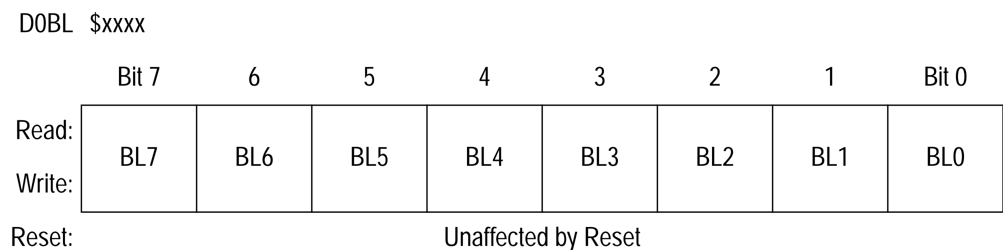


Figure 20. DMA Block Length Register (D0BL)

DMA Byte Count Register

The read/write DMA byte count register contains the number of bytes transferred on the channel during the current DMA transfer.

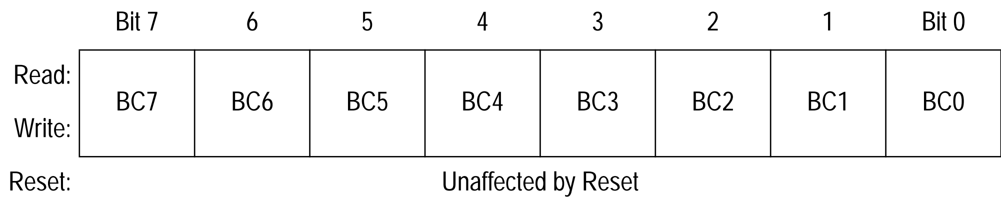


Figure 21. DMA Byte Count Register (D0BC)

Writing to the source base address or destination base address register clears the byte count register. The byte count register also is cleared when its count reaches the value in the block length register. Reset clears the byte count register.

DMA Application Examples

Contents

Introduction 68
Software-Initiated Block Transfer 69
A – Simple, Small Block Transfer 69
B – Flexible, Large Block Transfer 71
Summary 73
DMA Service of Serial Communications 73
A – Transmitting a Buffered Message Using the CPU 74
B – Servicing the SCI Transmitter Using the DMA 75
Summary 76
DMA Timer Servicing 77
A – Generating a Pseudo Buffered PWM 77
B – Buffering Input Captures for Period Calculation 81
Summary 83
Full Assembler Listings 84
Listing 1 – Fixed Block Length Transfer .84
Listing 2 – Variable Block Length Transfer 86
Listing 3 – SCI Transmitter 90
Listing 4 – SCI Transmitter 92
Listing 5 – Timer Output Compare 94
Listing 6 – PWM Generation 96
Listing 7 – Timer Input Capture 98
Listing 8 – Period Measurement 101

Introduction

The examples provided in the following sections are intended to provide a greater understanding of the operation of the DMA08 module, together with ideas on how to apply the DMA to various applications.

Due to the modular nature of the MC68HC705 Family and the DMA module in particular, the addresses of the DMA registers and various memory areas used in the examples may change in different devices. All of the examples in this manual are based on the memory and register map of the CPU08, which has three DMA channels. Each of the examples, except the multichannel one, is based on DMA channel 0 for consistency.

The user of these examples must take into account any other DMA channels that are active. For simplicity, except where indicated otherwise, the examples assume that no other DMA channels are active except those being used in the example. Where this is not the case in a real application, some of the software may have to be changed to avoid interfering with other active DMA channels — for example, change MOV instructions to multiple BSET/BCLRs.

Software-Initiated Block Transfer

This example is designed to show the DMA's efficiency in moving blocks of data around the system. Two examples are given: one with minimum setup and fixed operation, the other with flexible boundaries and assuming that the DMA channel has previously been used. In both cases, the setup overhead and time taken to transfer different sized blocks of memory are compared with the traditional CPU software method.

A – Simple, Small Block Transfer

This example shows the simplest form of software-initiated block transfer. It involves moving a known number of data bytes (up to 256) from one fixed address to another fixed address. The transfer is performed once (that is, no looping) and does not generate any CPU interrupts. A typical use of this type of transfer would be to move the contents of an input buffer to another area of memory for subsequent CPU processing, thus freeing the input buffer for further activity.

This example shows how to transfer a block of code using the DMA. (See Listing 1 – Fixed Block Length Transfer on page 84 for a full assembler listing.)

This code assumes that the DMA module is still in the reset condition with no other channels active and all registers in their reset states.

The software should configure all DMA registers relating to the operation of channel 0 that need to be changed from their reset condition. Note that the enable bit for the channel is written last. This is good programming practice and ensures that no spurious transfers can occur due to a partially configured channel.

The various DMA registers are configured as follows:

- DC1 = $C2: DMA has 100% of the bus bandwidth, DMA interrupt requests disabled, channel 0 enabled
- DSC = $00: (reset state) No looping, DMA disabled in WAIT mode, CPU interrupts higher priority than DMA

- DC2 = $01: Force a DMA request on channels configured to service transfer source 0
- D0C = $A0: Increment source base and destination base, transfer bytes, service transfer source 0

The number of bus cycles taken by each instruction in the example code is shown in parenthesis in the comment field. The total setup overhead for the transfer is 30 cycles. The DMA takes two bus cycles to transfer each byte. On completion of the transfer, the transfer enable bit for channel 0 (TEC0) will be cleared and the interrupt flag (ICF0) will be set. The software-initiate bit in DC2 remains set. This means that to repeat the same transfer, it is only necessary to re-enable the channel by setting TEC0 in DC1.

To show the relative performance of the DMA compared with a full software transfer of the data, the corresponding code for a software-driven transfer is given. The overhead of this method is just two cycles, but it takes 11 cycles to transfer each byte.

In this case, the software method uses fewer bytes of program memory than using the DMA. But it is obvious that once more than a few bytes are to be transferred then the DMA has a significant performance advantage. **Table 7** compares the total number of bus cycles required to transfer various sized blocks of data using the two methods.

Table 7. Relative Performance in Two Block Transfer Methods

Number of Bytes in Block		4	8	16	32	64	128	256
Number of Bus Cycles	Software Method	46	90	178	354	706	1410	2818
	DMA Method	38	46	62	94	158	286	542
Relative Performance		1.21	1.96	2.87	3.77	4.47	4.93	5.20

B – Flexible, Large Block Transfer

This example shows a further development of the software-initiated block transfer. It involves moving a variable number of data bytes (up to 64 K) from one address to another address. The code, shown in full in Listing 2 – Variable Block Length Transfer on page 86, is structured as a general-purpose subroutine. The source base address, destination base address, and number of bytes to transfer are all specified by 16-bit parameters in RAM and can, therefore, be different each time the routine is called. The transfer is performed once (that is, no looping) and does not generate any CPU interrupts. As an example, this type of transfer would be used to move around pages of on-screen display data in a TV application.

Since the DMA can transfer up to 256 bytes at a time, and the number of bytes here could be up to 64 K, a loop structure is used to transfer the whole block successfully. The DMA is first configured to transfer 256 bytes at a time until the upper byte of the block size is zero, when the DMA is configured to transfer the remaining number of bytes. The loop to transfer multiple blocks of 256 bytes can be very simple since only the upper bytes of the source base and destination base addresses need to be incremented and the channel re-enabled to transfer the next 256 bytes.

A further difference from example A is that this code does not assume the reset state of the DMA module. For this reason, the DMA channel is first turned off before configuration begins and the DSC register must be written.

The following shows how various DMA control registers are configured before the first transfer:

- DC1 = $C2: DMA has 100% of the bus bandwidth, DMA interrupt requests disabled, channel 0 enabled.
- DSC = $00: No looping, DMA disabled in wait mode, CPU interrupts higher priority than DMA.
- DC2 = $01: Force a DMA request on channels configured to service transfer source 0.
- D0C = $A0: Increment source base and destination base, transfer bytes, service transfer source 0.

The number of bus cycles taken by each instruction in the code is shown in parenthesis in the comment field. Due to the possibility of multiple transfers, the setup time is dependent on the number of bytes to be transferred. The total setup overhead can be estimated using the formula below. In each transfer, the DMA takes two bus cycles to transfer each byte.

$$\text{Setup time (bus cycles)} = \left\{23 \times \left\lceil \frac{\text{SRCESIZE}}{256} \right\rceil \right\} + 58$$

It follows that

$$\text{Total transfer time} = 58 + \left\{23 \times \left\lceil \frac{\text{SRCESIZE}}{256} \right\rceil \right\} + \{\text{SRCESIZE} \times 2\}$$

To show the relative performance of the DMA compared with a full software transfer of the data, the corresponding code for a software-driven transfer is also given:

The transfer time for this method is:

$$\text{Total transfer time} = 44 + \left\{10 \times \left\lceil \frac{\text{SRCESIZE}}{256} \right\rceil \right\} + \{\text{SRCESIZE} \times 30\}$$

Table 8 compares the total number of bus cycles required to transfer various blocks of data using the two methods:

Table 8. Relative Performance in Two Block Transfer Methods

Number of Bytes in Block		32	64	128	256	512	1024	2048
Number of Bus Cycles	Software Method	1004	1964	3884	7734	15424	30804	61564
	DMA Method	122	186	314	593	1128	2198	4338
Relative Performance		8.23	10.56	12.37	13.04	13.67	14.01	14.19

Summary

As can be seen from the two examples, the DMA module provides major performance improvements in moving blocks of data around the system. Even if only a small amount of data is to be moved, and time is critical, the DMA can increase performance. The above examples were performed with the bus 100% dedicated to the DMA module for the duration of the transfer. However, because of the performance improvement caused by the DMA, in many cases the DMA could be assigned only 67% or 50% of the bus bandwidth and still provide a significant performance increase while allowing the CPU to continue other tasks. In this case, the DMA could be configured to interrupt the CPU when the entire block has been transferred, or alternately, the software could poll the TEC bit to see if the transfer was complete.

DMA Service of Serial Communications

This example shows the benefit, in terms of overhead, of using the DMA module to service the asynchronous serial communications interface (SCI). An overhead comparison is made between CPU and DMA methods of servicing the SCI.

A – Transmitting a Buffered Message Using the CPU

In this example, a multi-character message contained in a buffer has to be transmitted using the SCI. The code to initialize the SCI and transmit a message using CPU interrupts is shown in Listing 3 – SCI Transmitter on page 90. The transmit buffer is contained in RAM starting at an address pointed to by TXADDR; the end of the message is indicated by a pointer TXEND. For simplicity, it is assumed that another routine has placed a message in the transmit buffer and has initialized the TXADDR and TXEND pointers to point to the start of the message and the byte following the end of the message respectively.

The code begins with the steps needed to initialize the SCI, including the transfer format and the baud rate to enable the SCI and CPU interrupts.

The transmit interrupt service routine (ISR) checks the TXADDR pointer and compares it with the TXEND pointer. If they are not equal, the byte pointed to by TXADDR is moved to the SCI data register (SCD), the pointer is incremented by one, and the interrupt flag is cleared before the routine exits. If they are equal, the complete message has been transmitted and the routine ends by disabling the SCI transmitter.

This CPU routine can be analyzed to predict the number of CPU cycles required to transmit a message of arbitrary length:

1. Initializing the SCI takes 14 cycles.
2. For each character that is transmitted, 33 cycles will be executed plus a minimum of nine cycles to enter the ISR.
3. While the last character is being transmitted, one final ISR will be generated which will take the branch to TCEND. This takes 30 cycles plus the nine overhead cycles.

From these figures, the total number of CPU cycles to transmit a message of length N is

$$\begin{aligned} \text{TcyclesIRQ} &= 14 + N(33 + 9) + (30 + 9) \\ &= (42N + 53) \end{aligned}$$

This is the best case figure; the actual number will be somewhat larger because interrupts can be taken only on instruction boundaries. Therefore, it is likely that there will be one or more additional cycles of latency for each interrupt service entry. Note that the H register is pushed onto the stack for the duration of the ISR. This is normal practice on the HC08 in any ISR that uses the H register.

B – Servicing the SCI Transmitter Using the DMA

The DMA module can also be configured to service the SCI transmitter. The required code is shown in Listing 4 – SCI Transmitter on page 92. The code assumes operation from reset conditions. First, the SCI is initialized as in the CPU service case, except this time the DMA is assigned to handle the TDRE interrupts. The DMA is then configured.This involves initializing the DMA source base and destination base addresses (TXADDR and SCD respectively), setting the block length (number of bytes to be transferred) and transfer type, and assigning the channel to service the SCI TX interrupt. A CPU interrupt also is enabled after the last DMA transfer is complete. The total initialization process takes 46 cycles and the various DMA control registers are configured as follows:

- D0C = $87: Increment source base, static destination base, transfer bytes, service transfer source 7.
- D0BL = $0D: Block length = 13 bytes (characters).
- DSC = $00 (reset state): No looping, DMA disabled in wait mode, CPU interrupts have priority over DMA transfers.
- DC2 = $00 (reset state): No software-initiated transfer.
- DC1 = $03: DMA has 25% of bus, enable transfers, and CPU interrupt from DMA channel 0.

NOTE: *Since overhead and not performance is the criteria in this example and the DMA will only transfer one byte for each request, the bus bandwidth value is not important. The overhead is not affected by this assignment.*

The mapping of the DMA transfer source base bits in the DMA channel control registers to specific peripheral interrupt sources varies from device to device, depending on the peripherals present and their relevant importance. On the 708XL36, the SCI transmitter is selected when DTS2:0 = 111, hence $87 is written to D0C in the example. Refer to the applicable data book for details of DTS bit assignments for devices other than the 708XL36.

Once the SCI transmitter is enabled, the DMA takes two bus cycles to transfer each byte to the transmit register. After the last DMA transfer is completed, a CPU DMA interrupt occurs. The ISR checks to see if DMA channel 0 made the interrupt request. If so, the SCI transmitter is

disabled and the DMA interrupt flag is cleared. This takes 24 cycles plus nine to enter the ISR. (See A – Transmitting a Buffered Message Using the CPU on page 74 for information regarding ISR entry delays.)

From the above analysis, the total number of cycles for the DMA to transmit a message of N characters is:

$$\text{TcyclesDMA} = 46 + N(2) + (24 + 9)$$

$$= (79 + 2N)$$

Table 9 shows the number of cycles needed by the CPU and DMA methods for several sizes of messages.

Table 9. Relative Performance in DMA and CPU Transfer Methods

Message Size	# Clocks CPU	# Clocks DMA	Relative Performance
1	95	81	1.17
2	137	83	1.65
4	221	87	2.54
6	305	91	3.35
8	389	95	4.09
10	473	99	4.78
20	893	119	7.50
50	2153	179	12.03
256	10805	591	18.28

Summary

From **Table 9** it can be seen that when servicing the SCI, the DMA provides significant savings in overhead compared with the CPU. In this case, the number of cycles needed to transmit even one byte is less with the DMA method. A small code size penalty is paid for using the DMA (14 bytes in this example), but in most cases the throughput improvement will easily justify the additional bytes of code.

The DMA can be used in a similar manner to service input or output on the serial peripheral interface (SPI), resulting in overhead benefits comparable to those proven for the SCI in this example. Latency is also improved.

DMA Timer Servicing

This example shows the possible performance benefits from servicing the timer module with the DMA module instead of the CPU. The timer is a flexible peripheral, and it is not possible to cover all uses of the DMA in conjunction with the timer. Two examples are given, one using an output compare to generate a pulse-width modulator (PWM) and the other buffering an input capture for period calculation by the CPU. In both cases, the performance is compared with CPU control.

A – Generating a Pseudo Buffered PWM

The timer interface module offers two types of PWM capability, buffered and unbuffered. In buffered PWM the pulse width (duty cycle) can be changed in software at any time and the timer hardware synchronizes the duty cycle change to the beginning of the next period. In this way, a smooth transition from one duty cycle to another is achieved with no glitches. However, this function requires two timer channels per buffered PWM output.

Unbuffered PWM operation requires only one channel, but an unsynchronized change in pulse width can cause improper operation for up to two periods. Refer to the *Timer Interface Module Reference Manual* for more details on the operation of buffered and unbuffered PWMs. CPU output compare interrupts can be used to buffer the PWM generation, but the minimum duty cycle is limited by the latency and service time of the interrupt. For example, when switching from a high duty cycle to a low duty cycle, the CPU has very little time to respond to the previous output compare and write the new output compare value for the low duty cycle. If the CPU is late, no output compare will take place in the next cycle, resulting in improper operation.

This example shows that by using the DMA to service the timer, a single timer channel can create a pseudo buffered PWM with pulse widths as short as four bus cycles. Suitable code for such a function is shown in Listing 5 – Timer Output Compare on page 94.

First, timer channel 0 is configured for unbuffered PWM operation. The timer is reset and stopped. An 8-bit PWM is selected by writing $FF to the timer counter modulo register. The channel is configured as an

output compare, set to toggle on timer overflow and clear on compare. The high time value, contained in the RAM location PWMHI, is copied into the timer channel 0 register. The timer channel is configured for DMA service.

DMA channel 0 is then set up to service the compare interrupt from timer channel 0. The various DMA control registers are configured as follows:

- D0C = $00: Static source base, static destination base, transfer bytes, service transfer source 0.
- D0BL = $01: Block length = 1 byte.
- DSC = $10: Loop on channel 0, DMA disabled in wait mode, CPU interrupts have priority over DMA transfers.
- DC2 = $00 (reset state): No software initiated transfer.
- DC1 = $C2: DMA has 100% of bus, enable transfers on DMA channel 0, no DMA interrupts to CPU.

NOTE: *The mapping of the DMA transfer source base bits in the DMA channel control registers to specific peripheral interrupt sources varies from device to device, depending on the peripherals present and their relevant importance. On the 708XL36, timer channel 0 is selected when DTS2:0 = 000, hence $28 is written to D0C in the example. Refer to the applicable data book for details of DTS bit assignments for devices other than the 708XL36.*

Finally, the TSTOP bit is cleared to enable the timer and start PWM generation. Each output compare event generates a DMA request and in response the DMA moves the latest high time value from PWMHI into the channel compare register. In this way, PWMHI can be changed at any time by the CPU, but the transfer to the timer channel is synchronized by the DMA, resulting in a pseudo buffered PWM.

Assuming only one DMA channel is active, with 100% of the bus, the channel will always receive the required bus cycles when the compare DMA request occurs. The DMA requires two cycles to transfer a byte plus an initial two cycles for address calculation so that the minimum time between output compares will be:

$$2 + 2 = 4 \text{ cycles}$$

This corresponds to the minimum PWM high time that can usually be achieved.

If more than one DMA channel is running, the possibility of two additional cycles per active channel must be taken into account if channel interrupts coincide.

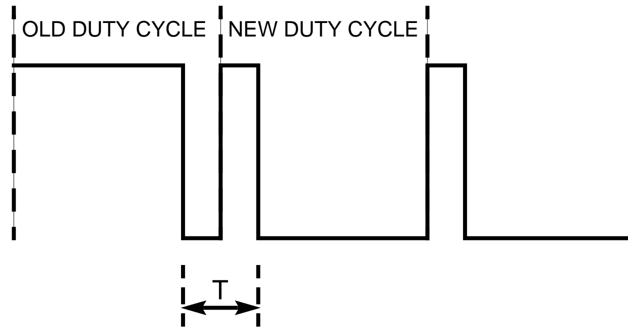
For comparison, Listing 6 – PWM Generation on page 96 shows code used to perform the same operation without the aid of the DMA module. Timer setup is identical to the DMA version except that the TDMA register is left in its reset state ($00) so that the timer will be serviced by the CPU. The timer channel 0 interrupt service routine (ISR) moves the latest PWM high time value from PWMHI to the lower byte of the channel register. The output compare flag is then cleared before the routine exits. The minimum reliable pulse width is dependant on the latency of the ISR and the actual execution time of the ISR, since one interrupt must be serviced fully before the next can occur. The execution time of the ISR is 20 cycles. Assuming that timer channel 0 is the highest priority CPU interrupt enabled (best case), then the latency is nominally nine cycles. However, since CPU interrupts can only be taken on instruction boundaries, the cycle count of the longest instruction used in the program must also be added for reliable performance. The longest instruction that is likely to be used in a user application is the DIV instruction which takes seven cycles.

The minimum period between interrupt requests is therefore:

$$20 + 9 + 7 = 36 \text{ cycles}$$

This corresponds to the minimum PWM high time that can be guaranteed under all circumstances. Smaller high times can be generated, but they may result in the previously described glitch of incorrect operation when written while the PWM is generating a very high duty cycle (see **Figure 22**).

This example shows that using the DMA to service the timer allows reliable generation of PWM pulse widths almost 10 times smaller than with CPU servicing.



If T < 36 cycles, then glitches of incorrect operation will occur, as shown below.

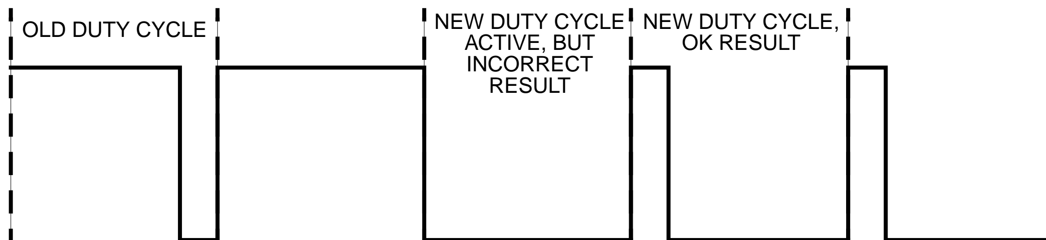


Figure 22. Minimum PWM High Time

B – Buffering Input Captures for Period Calculation

A common use of the timer module's input capture function is input signal period measurement. To perform this function, the timer pin is configured to capture the timer value on one edge type (rising or falling). By storing the captured time in a buffer and subtracting from the next captured value, the period of the input signal can be deduced. Sampling more than two edges and then averaging the result in software to get a more accurate and reliable result is normal. The example provided captures two periods (three edges) and calculates the average period which is stored in a 16-bit result register.

This operation can be performed entirely by the CPU, responding to each input capture interrupt, but the minimum period that can be reliably measured is limited by the latency and service time of the interrupt service routine. The CPU must respond to and store the results from one input capture before the next can be detected. Using the DMA to store captured values in a buffer and then calling the CPU once three values have been received allows much smaller periods to be resolved.

The code required to perform this period measuring task using the DMA is shown in Listing 7 – Timer Input Capture on page 98. First, timer channel 0 is initialized as a rising edge input capture channel and configured to be serviced by the DMA. Next, DMA channel 0 is configured to service timer channel 0; the transfer source base address is set up as the timer channel 0 register and the destination base as a buffer in RAM starting at address ICBUFFER. By programming the DMA to have a static source base and incrementing destination base, subsequent input capture values will be stored in the buffer. The DMA is configured to interrupt the CPU once the 6-byte buffer is full. Since the timer channel registers are 16 bits in length, the DMA is configured to transfer a word of data for each request. The code assumes operation from reset conditions. The DMA control registers are configured as follows:

- D0C = $28: Static source base, increment destination base, transfer words, service transfer source 0.
- D0BL = $06: Block length = 6 bytes (3 words)
- DSC = $80: No looping, DMA disabled in wait mode, DMA has priority over CPU interrupts.

- DC2 = $00 (reset state): No software-initiated transfer.
- DC1 = $C3: DMA has 100% of bus, enable transfers and CPU interrupt from DMA channel 0.

NOTE: *Since the DMA will transfer only one word for each timer input capture, it is safe to assign the DMA a higher priority than CPU interrupts (DSC = $80) and 100% of the bus (DC1 = $C3) without danger of locking out the CPU. However, if another channel were to be used for a software-initiated block transfer, then the bus bandwidth allocation may need to be changed.*

The mapping of the DMA transfer source base bits in the DMA channel control registers to specific peripheral interrupt sources varies from device to device depending on the peripherals present and their relevant importance. On the 708XL36, timer channel 0 is selected when DTS2:0 = 000, hence $28 is written to D0C in the example. Refer to the relevant technical summary for details of DTS bit assignments for other M68HC05 Microcontroller Applications Guide Family devices.

Once configured, the DMA will transfer the contents of timer channel 0 register to the buffer in RAM in response to each rising edge on the device pin associated with timer channel 0. After three words have been transferred (three edges -> two signal periods), the DMA interrupts the CPU which calculates the mean of the two periods and stores the result in PERIOD before reinitializing DMA channel 0 to repeat the process.

The DMA takes four cycles to transfer a word of data and an additional two cycles to set up and calculate addresses. This means that the minimum time between input edges on the capture pin is six CPU cycles.

Therefore, with the timer clocked at a bus frequency of 8 MHz, the maximum frequency input signal that can be measured is:

$$1/(0.125\ \mu S \times 6) = 1.33\ MHz$$

The code required to perform the same operation without the DMA module is found in Listing 8 – Period Measurement on page 101. The timer is again initialized, but this time it is configured for CPU servicing. In this case, every input capture results in a CPU interrupt and the interrupt service routine (ISR) must store the captured timer value

manually in the buffer and keep track of the number of edges via a counter. When the counter indicates that the third edge has been captured, the ISR calculates the mean period in the same manner as the DMA example before restarting the measurement process.

The execution time of the ISR (excluding the third edge/calculation case) is 46 cycles. To calculate performance, the latency of entering the ISR must also be taken into account. Assuming that timer channel 0 is the highest priority CPU interrupt enabled (best case), then the latency is nominally nine cycles. However, since CPU interrupts can be taken only on instruction boundaries, the cycle count of the longest instruction used in the program must also be added for reliable performance. The longest instruction that is likely to be used in a user application is the seven-cycle DIV instruction.

The minimum period between interrupt requests is, therefore, 46 + 9 + 7 = 62 cycles which with an 8-MHz bus speed corresponds to a maximum measurable frequency of:

1(0.125µS x 62) = 129 KHz

The DMA, therefore, provides a factor of 10x improvement in measurement performance in this example.

Summary

Both of the examples using the timer in conjunction with the DMA show a major benefit in performance over traditional CPU servicing. Not only is the timer performance enhanced, but the CPU is released to perform its other tasks more efficiently.

Full Assembler Listings

The following assembler listings give full details of the setup and execution for each of the examples discussed in this chapter.

Listing 1 – Fixed Block Length Transfer

```
*********************************************************************
*                                    COPYRIGHT (c) MOTOROLA 1994
* FILE NAME: BLOKMV1.S08
*
* PURPOSE: To demonstrate the block move capabilities of DMA08 and
* compare with traditional software methods.
*
* ASSEMBLER: IASM08                        VERSION: 3.03
*
* DESCRIPTION:
* Transfer assuming no DMA channels have been used since reset
* Fixed source base and destination base addresses and fixed block
* length - transfer $80 bytes starting at $100 to address starting at
* $200
*
*********************************************************************
* Register definitions for DMA channel 0 as mapped on the 68HC708XL36

D0SH     EQU        $0034       ; DMA Channel 0 Source High
D0SL     EQU        $0035       ; DMA Channel 0 Source Low
D0DH     EQU        $0036       ; DMA Channel 0 Destination High
D0DL     EQU        $0037       ; DMA Channel 0 Destination Low
D0C      EQU        $0038       ; DMA Channel 0 Control
D0BL     EQU        $0039       ; DMA Channel 0 Block Length
D0BC     EQU        $003B       ; DMA Channel 0 Byte Count

DC1      EQU        $004C       ; DMA Control 1
DSC      EQU        $004D       ; DMA Status/Control
DC2      EQU        $004E       ; DMA Control 2
```

```
* Program equates for example code.

SOURCE   EQU         $100         ; Start address of block to be
                                  ; transferred.
DESTIN   EQU         $200         ; Start address of destination of
                                  ; block.
BLSIZE   EQU         $80          ; Transfer 128 bytes.

         ORG         $7000        ; Code area

FIRSTDMA
         LDHX        #SOURCE      ; (3) Get start addr of source block.
         STHX        D0SH         ; (4) Store in channel0 source
                                  ; address.

         LDHX        #DESTIN      ; (3) Get start addr of
                                  ; destination
                                  ; block.
         STHX        D0DH         ; (4) store in channel0
                                  ; destination address
         MOV         #BLSIZE,D0BL; (4) No. of bytes into block
                                  ; length register.
         MOV         #$A0,D0C     ; (4) Inc source & dest, xfer
                                  ; bytes,
                                  ; DTS=0
         MOV         #$01,DC2     ; (4) Set software initiate bit
                                  ; for DTS=0.
DOIT
         MOV         #$C2,DC1     ; (4) 100% bus for DMA, enable
                                  ; chan0 to start xfer.
         BRA         *            ; End of example
*********************************************************************
* Below is the code required for the CPU08 to perform the same task
* as above but without the aid of the DMA module

SWEXAMPA
         LDX         #BLSIZE      ; (2) Use X as byte counter &
                                  ; pointer.
XFERLOOP
         LDA         SOURCE,X     ; (4) Get source byte.
         STA         DESTIN,X     ; (4) Transfer to destination
         DBNZX       XFERLOOP     ; (3) Move pointer and check for
                                  ; end of block
DONE
         BRA         *            ; End of example
```

Listing 2 – Variable Block Length Transfer

```
*******************************************************************
*                                       COPYRIGHT (c) MOTOROLA 1994
* FILE NAME: BLOKMV2.S08
*
* PURPOSE: To Demonstrate the block move capabilities of DMA08 and
* compare with traditional software methods.
*
* ASSEMBLER: IASM08                         VERSION: 3.03
*
* DESCRIPTION:
* Variable sized transfer, structured as a subroutine with transfer
* size and addresses passed as RAM variables. Does not assume reset
* state
*
*******************************************************************

* Register definitions for DMA channel 0 as mapped on the 68HC708XL36

D0SH  EQU          $0034       ; DMA Channel 0 Source High
D0SL  EQU          $0035       ; DMA Channel 0 Source Low
D0DH  EQU          $0036       ; DMA Channel 0 Destination High
D0DL  EQU          $0037       ; DMA Channel 0 Destination Low
D0C   EQU          $0038       ; DMA Channel 0 Control
D0BL  EQU          $0039       ; DMA Channel 0 Block Length
D0BC  EQU          $003B       ; DMA Channel 0 Byte Count

DC1   EQU          $004C       ; DMA Control 1
DSC   EQU          $004D       ; DMA Status/Control
DC2   EQU          $004E       ; DMA Control 2

* Bit definitions

SWI0  EQU          0
TEC0  EQU          1
IEC0  EQU          0

      ORG          $60         ; RAM data area

*variables required for example.
SRCEADDRRMB        2           ; Start address of source block to be
                               ; moved
DESTADDRRMB        2           ; Destination address of first byte
                               ; in block.
SRCESIZERMB        2           ; Length in bytes of block to be moved
SADDTEMPRMB        2           ; Temporary variables for software
                               ; version
DADDTEMPRMB        2
SIZETEMPRMB        1

      ORG          $7000       ; Code area

NEWDMA
      JSR          COPY
      BRA          *
```

```
********************************************************************
*
* NAME: COPY
*
* PURPOSE: Uses DMA to transfer a variable size block of memory
* and A to B
*
* ENTRY CONDITIONS: SRCEADDR = 16bit start address of block to be
* copied
*
*      DESTADDR = 16bit start address of destination of copy
*      SRCESIZE = Number of bytes in the block - 16bits.
*
* EXIT CONDITIONS: H, X destroyed, SRCEADDR, DESTADDR, SRCESIZE, Acc
* unchanged.
*
*      DMA channel 0 disables, but configuration changed.
*
*      STACK SPACE USED (Bytes): 1           RAM USAGE (Bytes): 0
*      EXTERNALVARIABLESUSED:SRCEADDR,DESTADDR,SRCESIZE
*
********************************************************************
COPY
      MOV         #0,DC1       ; (4) Turn off DMA channels and
                               ; interrupts.
      LDHX        SRCEADDR     ; (4) Get start addr of source block.
      STHX        D0SH         ; (4) Store in channel0 source
                               ; address.
      LDHX        DESTADDR     ; (4) Get start addr of destination
                               ; block.
      STHX        D0DH         ; (4) store in channel0 destination
                               ; address
      MOV         #$A0,D0C     ; (4) Inc source & dest, xfer bytes,
                               ; DTS=0
      BSET        SWI0,DC2     ; (4) Set software initiate bit for
                               ; DTS=0.
      MOV         #0,DSC       ; (4) Disable looping and DMA in wait
                               ; mode.
      LDX         SRCESIZE     ; (3) get upper byte of block length.
      PSHX                     ; (2) Store upper half of byte count
                               ; on stack.
      BEQ         DOLOWER      ; (3) If upper = 0 then just do lower
                               ; byte
      CLR         D0BL         ; (3) Transfer 256 bytes.

DOXFER
      MOV         #$C2,DC1     ; (4) Assign DMA 100% of bus & enable
                               ; chan0.
      BRSET       TEC0,DC1,*   ; (5) Wait for this xfer to complete.
      INC         D0SH         ; (4) 256 bytes done; update source
                               ; addr.
      INC         D0DH         ; (4) - and update destination addr.
      DBNZ        1,SP,DOXFER  ; (6) Loop until all 256 byte blocks
                               ; done.
```

```
DOLOWER
      PULX                    ; (2) Clean up stack.
      LDX         SRCESIZE+1  ; (3) Get lower byte of block length.
      BEQ         ALLDONE     ; (3) Test for zero.
      STX         D0BL        ; (3) Set up last transfer.
      MOV         #$C2,DC1    ; (4) Assign DMA 100% of bus & enable
                              ; chan0.
ALLDONE
      RTS

*********************************************************************
* Below is the code required for the CPU08 to perform the same task as
* above without the aid of the DMA module

SWEXAMPB
      JSR         COPYSW
      BRA         *

*********************************************************************
* NAME: COPYSW
*
* PURPOSE: Uses software to transfer variable size data block from A
* to B
*
* ENTRY CONDITIONS: SRCEADDR = 16bit start address of block to be copied
*
* DESTADDR = 16bit start address of destination of copy
* SRCESIZE = Number of bytes in the block - 16bits.
*
* EXIT CONDITIONS: H, X, Acc destroyed,
* SRCEADDR,DESTADDR,SRCESIZE
* DMA channel 0 disables, but configuration changed.
*
* STACK SPACE USED (Bytes): 2           RAM USAGE (Bytes): 4
*
* EXTERNAL VARIABLES USED: SRCEADDR, DESTADDR,
* SRCESIZE
*
********************************************************************

COPYSW
      LDHX        SRCEADDR    ; (4) Store transfer addresses in
                              ; temporary.
      STHX        SADDTEMP    ; (4) variables.
      LDHX        DESTADDR    ; (4)
      STHX        DADDTEMP    ; (4)
      LDX         SRCESIZE    ; (3) Get high byte of block size and
                              ; inc it.
      INCX                    ; (1) - to compensate for later test.
      PSHX                    ; (2) Store modified high byte on
                              ; stack.
      LDX         SRCESIZE+1  ; (3) Get low byte.
      PSHX                    ; (2) - store it on the stack.
      BEQ         DOHIGH      ; (3) Skip first loop if low byte = 0.
```

```
XFERATOB
      LDHX        SADDTEMP    ; (4) Get source address of byte.
      LDA         ,X          ; (2) Get the data.
      AIX         #1          ; (2) Increment source addr for next
                              ; byte.
      STHX        SADDTEMP    ; (4)
      LDHX        DADDTEMP    ; (4) Get destination address.
      STA         ,X          ; (2) Transfer byte
      AIX         #1          ; (2) Increment destination for next
                              ; byte.
      STHX        DADDTEMP    ; (4)
      DBNZ        1,SP,XFERATOB; (6) Check for end of loop.
DOHIGH:
      CLR         1,SP        ; (4)
      DBNZ        2,SP,XFERATOB; (6) Decrement & test block length
                              ; high byte
DONEB
      PULX                    ; (2) Clean up stack and return.
      PULX                    ; (2)
      RTS
```

Listing 3 – SCI Transmitter

```
********************************************************************
*                               COPYRIGHT (c) MOTOROLA 1994
* FILENAME:SCISW.S08
*
* PURPOSE: To show a CPU driven SCI transmitter for comparison with DMA
*
* ASSEMBLER: IASM08                         VERSION: 3.03
*
* DESCRIPTION:
* This routine uses a CPU SCI TX IRQ to send a buffered message. The
* message has been placed in the buffer by a separate routine along
* with the start address of the message and an end pointer that is 1
* plus the address of the last byte to be sent.
*
********************************************************************
* Register definitions for SCI as mapped on the 68HC708XL36
SCC1  EQU         $0013       ; SCI Control register 1
SCC2  EQU         $0014       ; SCI Control register 2
SCC3  EQU         $0015       ; SCI Control register 3
SCS1  EQU         $0016       ; SCI Status register 1
SCS2  EQU         $0017       ; SCI Status register 2
SCD   EQU         $0018       ; SCI Data Register
SCBR  EQU         $0019       ; SCI Baud Rate Register
*
********************************************************************
Memory
      ORG         $60
MESSAGEDB         'HELLO WORLD! '
TXSTARTFDB        MESSAGE
TXEND FDB         MESSAGE+13T
*
*******************************************************************
      ORG         $6E00       ; beginning of program area
START EQU         *
*******************************************************************
* Init Routine

* Initialize the SCI

      mov         #$40,SCC1   ; (4) ensci
      mov         #$03,SCBR   ; (4) 9600 baud
      mov         #$88,SCC2   ; (4) enable transmitter and TDRE
                              ; IRQ
      cli                     ; (2) enable CPU interrupts
      bra         *           ; (3) wait for irq
```

```
* SCI Transmit Interrupt Service Routine

TXISR pshh                    ; (2) Stack H.
      ldhx        TXSTART     ; (4) Point to next byte.
      cphx        TXEND       ; (4) Have we sent the last byte?
      beq         TCEND       ; (3) Yes, go to end of message
                              ; shutdown.
      lda         SCS1        ; (3) Dummy read to clear the flag.
      mov         x+, scd     ; (4) No, send another character.
      sthx        TXSTART     ; (4) Save the pointer for next time.
      pulh                    ; (2) Restore H.
      rti                     ; (7)

TCEND bclr        7,SCC2      ; (4) disable transmitter
      pulh                    ; (2) Restore H
      rti                     ; (7)

      ORG         $FFE2
SCITXVDW          TXISR       ; SCI TX ISR VECTOR

      ORG         $FFFE
RESETVDW          $6e00       ; RESET VECTOR
```

Listing 4 – SCI Transmitter

```
*******************************************************************
*                                        COPYRIGHT (c) MOTOROLA 1994
* FILE NAME: SCIEX.S08
*
* PURPOSE: To show the use of the DMA module with an SCI transmitter
*
* ASSEMBLER: IASM08                        VERSION: 3.03
*
* DESCRIPTION:
* This routine uses the DMA on the 708XL36 to service the SCI
* transmitter and send a buffered message. The message has been placed
* in the buffer by a separate routine along with the start address of
* the message and a byte count.
*
*******************************************************************

* Register definitions for DMA channel 0 as mapped on the 68HC708XL36
D0SH  EQU         $0034       ; DMA Channel 0 Source High
D0SL  EQU         $0035       ; DMA Channel 0 Source Low
D0DH  EQU         $0036       ; DMA Channel 0 Destination High
D0DL  EQU         $0037       ; DMA Channel 0 Destination Low
D0C   EQU         $0038       ; DMA Channel 0 Control
D0BL  EQU         $0039       ; DMA Channel 0 Block Length
D0BC  EQU         $003B       ; DMA Channel 0 Byte Count

DC1   EQU         $004C       ; DMA Control 1
DSC   EQU         $004D       ; DMA Status/Control
DC2   EQU         $004E       ; DMA Control 2

* Register definitions for SCI as mapped on the 68HC708XL36
SCC1  EQU         $0013       ; SCI Control register 1
SCC2  EQU         $0014       ; SCI Control register 2
SCC3  EQU         $0015       ; SCI Control register 3
SCS1  EQU         $0016       ; SCI Status register 1
SCS2  EQU         $0017       ; SCI Status register 2
SCD   EQU         $0018       ; SCI Data Register
SCBR  EQU         $0019       ; SCI Baud Rate Register

*******************************************************************
* Memory
*
      ORG         $60
MESSAGEDB         'HELLO WORLD! '
TXSTARTFDB        MESSAGE
COUNT DB          $D

*******************************************************************

      ORG         $6E00       ; beginning of program area
START EQU         *

*******************************************************************
```

```
* Init Routine

* Initialize the SCI

      MOV         #$40,SCC1   ; (4) ENSCI, 8bits, idle line, no
                              ; parity
      MOV         #$10,SCC3   ; (4) DMATE enabled.
      MOV         #$03,SCBR   ; (4) 9600 baud

* Initialize the DMA

      MOV         #$87,D0C    ; (4) Inc/static, byte, sci tx
      MOV         COUNT,D0BL  ; (5) block length = 13
      LDHX        TXSTART     ; (4) init channel 0 source address
      STHX        D0SH        ; (4)
      LDHX        #SCD        ; (3) init channel 0 destination
                              ; address
      STHX        D0DH        ; (4)
      MOV         #$03,DC1    ; (4) TEC0 and IEC0 enabled

* Begin transmission

      MOV         #$08,SCC2   ; (4) Start transmission.
      CLI                     ; (2) Enable CPU interrupts.
      BRA         *           ; (3)
DMAISRbrclr       0,DSC,DMAEND; (5) Check for interrupt source.
      bclr        7,SCC2      ; (4) Disable TDRE IRQs.
      bclr        3,SCC2      ; (4) Disable transmitter.
      bclr        0,DSC       ; (4) Clear the interrupt flag.
DMAENDrti         ;(7)


       ORG        $FFF6
DMAV   DW         DMAISR      ; DMA ISR VECTOR

       ORG        $FFFE
RESETV DW         $6e00       ; RESET VECTOR
```

Listing 5 – Timer Output Compare

```
******************************************************************
*                                        COPYRIGHT (c) MOTOROLA 1994
* FILENAME:   TIMPWMEX.S08
*
* PURPOSE: To show the use of the DMA module with timer output compare      *
* ASSEMBLER: IASM08                        VERSION: 3.03
*
* DESCRIPTION:
* This routine uses the DMA on the 708XL36 to service the timer.
* It does a single transfer loop from a memory location to the TCH0L
* register at each output compare time. A CPU routine can write a new
* value to this register to change the PWM high time. This allows the
* CPU to change the PWM high time without regard to the position of any
* PWM edges.
*
******************************************************************

* Register definitions for DMA channel 0 as mapped on the 68HC708XL36
D0SH  EQU         $0034       ; DMA Channel 0 Source High
D0SL  EQU         $0035       ; DMA Channel 0 Source Low
D0DH  EQU         $0036       ; DMA Channel 0 Destination High
D0DL  EQU         $0037       ; DMA Channel 0 Destination Low
D0C   EQU         $0038       ; DMA Channel 0 Control
D0BL  EQU         $0039       ; DMA Channel 0 Block Length
D0BC  EQU         $003B       ; DMA Channel 0 Byte Count

DC1   EQU         $004C       ; DMA Control 1
DSC   EQU         $004D       ; DMA Status/Control
DC2   EQU         $004E       ; DMA Control 2

* Register definitions for Timer channel 0 as mapped on the
* 708XL36
TSC   EQU         $0020       ; Timer Status/Control
TDMA  EQU         $0021       ; Timer DMA select
TCNTH EQU         $0022       ; Timer Counter High
TCNTL EQU         $0023       ; Timer Counter Low
TMODH EQU         $0024       ; Timer Counter Modulo High
TMODL EQU         $0025       ; Timer Counter Modulo Low

TSC0  EQU         $0026       ; Timer Channel 0 Status/Control
TCH0H EQU         $0027       ; Timer Channel 0 High
TCH0L EQU         $0028       ; Timer Channel 0 Low


      ORG         $60         ; RAM data area
* variables required for example.

PWMHI DB          $80
*
```

```
*******************************************************************

      ORG          $6E00       ; beginning of program area
START EQU          *

*******************************************************************

* Init Routine

*Initialize the Timer
      MOV          #$30,TSC    ; (4) Stop & reset timer.
      MOV          #$01,TDMA   ; (4) Service timer with DMA.
      MOV          #$00,TMODH  ; (4) Set to 8-bit PWM.
      MOV          #$FF,TMODL  ; (4)
      MOV          #$00,TCH0H  ; (4) Set initial PWMHI.
      MOV          PWMHI,TCH0L ; (5)
      MOV          #$5A,TSC0   ; (4) unbuff, TOV, low compare,
                               ; en IRQ
*Initialize the DMA for Timer
      MOV          #$10,DSC    ; (4) channel 0 loop enable
      MOV          #$00,D0C    ; (4) static/static,byte,timer0
      MOV          #$01,D0BL   ; (4) block length = 1
      LDHX         #PWMHI      ; (3) init channel 0 source address
      STHX         D0SH        ; (4)
      LDHX         #TCH0L      ; (3) init channel 0 destn. address
      STHX         D0DH        ; (4)
      MOV          #$C2,DC1    ; (4) 100% bandwidth, TEC0 enabled
      LDA          TSC         ; (3) Clear TSTOP to begin.
      AND          #$DF        ; (2)
      STA          TSC         ; (3)

DONE  BRA          *           ; (3)

      ORG          $FFFE
RESETVDW           $6e00
```

Listing 6 – PWM Generation

```
*******************************************************************
*                                    COPYRIGHT (c) MOTOROLA 1994
* FILE NAME: TIMPWMSW.S08
*
* PURPOSE: To generate a PWM using CPU servicing for comparison with DMA
*
* ASSEMBLER: IASM08                       VERSION: 3.03
*
* DESCRIPTION:
* Timer chan 0 is configured to generate an unbuffered PWM. Each output * compare
is serviced by the CPU which copies the latest high time from
* RAM thus creating a semi or pseudo buffered PWM.
*******************************************************************

* Register definitions for Timer channel 0 as mapped on the
* 708XL36
TSC   EQU         $0020       ; Timer Status/Control
TDMA  EQU         $0021       ; Timer DMA select
TCNTH EQU         $0022       ; Timer Counter High
TCNTL EQU         $0023       ; Timer Counter Low
TMODH EQU         $0024       ; Timer Counter Modulo High
TMODL EQU         $0025       ; Timer Counter Modulo Low

TSC0  EQU         $0026       ; Timer Channel 0 Status/Control
TCH0H EQU         $0027       ; Timer Channel 0 High
TCH0L EQU         $0028       ; Timer Channel 0 Low


      ORG         $60         ; RAM data area
* variables required for example.

PWMHI DB          $80
*
*******************************************************************

      ORG         $6E00       ; beginning of program area
START EQU         *

*******************************************************************
```

```
* Init Routine

* Initialize the Timer
      MOV         #$30,TSC     ; (4) Stop & reset timer.
      MOV         #$00,TMODH   ; (4) Set to 8-bit PWM.
      MOV         #$FF,TMODL   ; (4)
      MOV         #$00,TCH0H   ; (4) Set initial PWMHI.
      MOV         PWMHI,TCH0L  ; (5)
      MOV         #$5A,TSC0    ; (4) unbuff, TOV, low compare,
                               ; en IRQ
      LDA         TSC          ; (3) Clear TSTOP to begin.
      AND         #$DF         ; (2)
      STA         TSC          ; (3)
      CLI                      ; (2) Enable interrupts.

DONE  BRA         *            ; (3)

TIMOCIRQEQU       *            ; Service timer output compare.
      MOV         PWMHI,TCH0L  ; (5) Update high time.
      LDA         TSC0         ; (3)
      AND         #$7F         ; (2) Clear channel flag
      STA         TSC0         ; (3)
      RTI                      ; (7)

      ORG         $FFF4
TIM0V DW          TIMOCIRQ     ; Timer channel 0 vector

      ORG         $FFFE
RESETVDW          START        ; Reset vector
```

Listing 7 – Timer Input Capture

```
********************************************************************
*                                     COPYRIGHT (c) MOTOROLA 1994
*
* FILENAME: TIMICEX.S08
*
* PURPOSE: To demonstrate the use of the HC08 DMA with the timer module.
*
* ASSEMBLER: IASM08                        VERSION: 3.03
*
* DESCRIPTION:
* The DMA is used to store timer values corresponding to input capture
* edges. After three edges have been captured, a CPU interrupt is
* generated and the average of the two periods calculated and stored
* in a result register.
*
********************************************************************

* Register definitions for DMA channel 0 as mapped on the 68HC708XL36
D0SH  EQU         $0034       ; DMA Channel 0 Source High
D0SL  EQU         $0035       ; DMA Channel 0 Source Low
D0DH  EQU         $0036       ; DMA Channel 0 Destination High
D0DL  EQU         $0037       ; DMA Channel 0 Destination Low
D0C   EQU         $0038       ; DMA Channel 0 Control
D0BL  EQU         $0039       ; DMA Channel 0 Block Length
D0BC  EQU         $003B       ; DMA Channel 0 Byte Count

DC1   EQU         $004C       ; DMA Control 1
DSC   EQU         $004D       ; DMA Status/Control
DC2   EQU         $004E       ; DMA Control 2

* Register definitions for Timer channel 0 as mapped on the
* 708XL36
TSC   EQU         $0020       ; Timer Status/Control
TDMA  EQU         $0021       ; Timer DMA select
TCNTH EQU         $0022       ; Timer Counter High
TCNTL EQU         $0023       ; Timer Counter Low
TMODH EQU         $0024       ; Timer Counter Modulo High
TMODL EQU         $0025       ; Timer Counter Modulo Low

TSC0  EQU         $0026       ; Timer Channel 0 Status/Control
TCH0H EQU         $0027       ; Timer Channel 0 High
TCH0L EQU         $0028       ; Timer Channel 0 Low


      ORG         $60         ; RAM data area
* variables required for example.

PERIODRMB2                    ; Result calculated in CPU interrupt ; routine.
ICBUFFERRMB       6           ; Buffer for captured Timer values.


      ORG         $7000       ; Code area
```

```
TIMICDMA
* Setup Timer channel 0
      MOV         #$00,TSC    ; (4) Disable Overflow IRQ, clock at
                              ; bus speed.
      MOV         #$01,TDMA   ; (4) Service channel 0 with the DMA
      MOV         #$44,TSC0   ; (4) Enable chan0 IRQs, capture on
                              ; rising edge

* Setup DMA channel 0
      LDHX        #TCH0H      ; (3) Get addr of timer channel0
                              ; register.
      STHX        D0SH        ; (4) Store in channel0 source
                              ; address.
      LDHX        #ICBUFFER   ; (3) Get start addr of destination
                              ; buffer.
      STHX        D0DH        ; (4) Store in channel0 destination
                              ; address.
      MOV         #$28,D0C    ; (4) Static srce, inc dest, xfer
                              ; words-Tim Ch0
      MOV         #$06,D0BL   ; (4) Buffer size - 3 edges->3 words-
                              ; >6 bytes
      MOV         #$80,DSC    ; (4) Disable looping, DMA has
                              ; priority over CPU.
DOIT
      CLI                     ; (2) Enable interrupts.
      MOV         #$C3,DC1    ; (4) 100% for DMA, enable chan0 xfer
                              ; & IRQ.
      BRA         *           ; (3)

DMA0IRQ
* Service the DMA interrupt and calculate and store period.
      LDA         ICBUFFER+3  ; (3) Calculate and store the two
                              ; periods.
      SUB         ICBUFFER+1  ; (3)
      STA         ICBUFFER+1  ; (3)
      LDA         ICBUFFER+2  ; (3)
      SBC         ICBUFFER    ; (3)
      STA         ICBUFFER    ; (3)
      LDA         ICBUFFER+5  ; (3)
      SUB         ICBUFFER+3  ; (3)
      STA         ICBUFFER+3  ; (3)
      LDA         ICBUFFER+4  ; (3)
      SBC         ICBUFFER+2  ; (3)
      STA         ICBUFFER+2  ; (3)
```

```
AVERAGE
      LDA         ICBUFFER+3  ; (3) Average period by adding and
      ADD         ICBUFFER+1  ; (3) - shifting right.
      STA         ICBUUFER+3  ; (3)
      LDA         ICBUFFER+2  ; (3)
      ADC         ICBUFFER    ; (3)
      RORA                    ; (1) Divide by 2.
      STA         PERIOD      ; (3)
      LDA         ICBUFFER+3  ; (3)
      RORA                    ; (1)
      STA         PERIOD+1    ; (3) PERIOD now contains 16bit
                              ; average.
      LDA         DSC         ; (3)
      AND         #$FE        ; (2)
      STA         DSC         ; (3) Clear DMA chan0 interrupt flag.
      MOV         #$C3,DC1    ; (4) Re-enable channel0 transfers.
      RTI                     ; (7)

      ORG         $FFF6       ; Vectors

      FDB         DMA0IRQ     ; Setup DMA interrupt vector.

      ORG         $FFFE

      FDB         TIMICDMA    ; Setup Reset vector.
```

Listing 8 – Period Measurement

```
*******************************************************************
*                                       COPYRIGHT (c) MOTOROLA 1994
*
* FILE NAME: TIMICSW.S08
*
* PURPOSE: To demonstrate period measurement in s/w for comparison
* with DMA
*
* ASSEMBLER: IASM08                        VERSION:   3.03
*
* DESCRIPTION:
* Signal is fed to an input capture channel. Input capture interrupt
* service routine stores captured value in buffer and calculates
* average period every third edge.
*
******************************************************************

* Register definitions for DMA channel 0 as mapped on the 68HC708XL36
D0SH  EQU         $0034       ; DMA Channel 0 Source High
D0SL  EQU         $0035       ; DMA Channel 0 Source Low
D0DH  EQU         $0036       ; DMA Channel 0 Destination High
D0DL  EQU         $0037       ; DMA Channel 0 Destination Low
D0C   EQU         $0038       ; DMA Channel 0 Control
D0BL  EQU         $0039       ; DMA Channel 0 Block Length
D0BC  EQU         $003B       ; DMA Channel 0 Byte Count

DC1   EQU         $004C       ; DMA Control 1
DSC   EQU         $004D       ; DMA Status/Control
DC2   EQU         $004E       ; DMA Control 2

* Register definitions for Timer channel 0 as mapped on the
* 708XL36
TSC   EQU         $0020       ; Timer Status/Control
TDMA  EQU         $0021       ; Timer DMA select
TCNTH EQU         $0022       ; Timer Counter High
TCNTL EQU         $0023       ; Timer Counter Low
TMODH EQU         $0024       ; Timer Counter Modulo High
TMODL EQU         $0025       ; Timer Counter Modulo Low

TSC0  EQU         $0026       ; Timer Channel 0 Status/Control
TCH0H EQU         $0027       ; Timer Channel 0 High
TCH0L EQU         $0028       ; Timer Channel 0 Low


      ORG         $60         ; RAM data area
* variables required for example.

PERIODRMB         2           ; Result calculated in CPU interrupt
                              ; routine.
ICBUFFERRMB       4           ; Buffer for captured Timer values
TEMP  RMB         1


      ORG         $7000       ; Code area
```

```
TIMICSW
* Setup Timer channel 0
      MOV         #$00,TSC    ; (4) Disable Overflow IRQ, clock at
                              ; bus speed.
      MOV         #$00,TDMA   ; (4) Service all timer channels with
                              ; the CPU.
      MOV         #$44,TSC0   ; (4) Enable chan0 IRQs, capture on
                              ; rising edge.
      CLR         TEMP        ; (3) Clear edge counter.
DOIT
      CLI                     ; (2) Enable interrupts.
      BRA         *           ; (3)

TIC0IRQ
* Service the TIC interrupt, calculate and store period every 3rd edge.
      PSHH                    ; (2) Stack H
      LDA         TEMP        ; (3) Check edge counter.
      CMP         #2          ; (2) 3rd edge?
      BEQ         DO_CALC     ; (3) - Yes, so go calculate period.
      LDHX        TCH0H       ; (4) - No, so get captured value
      INC         TEMP        ; (4) Increment edge counter.
      TSTA                    ; (1)
      BNE         STORE_2ND   ; (3) 1st or 2nd edge?
      STHX        ICBUFFER    ; (4) Store captured time in buffer.
      BRA         CLRIRQ      ; (3)
STORE_2ND
      STHX        ICBUFFER+2  ; (4)
      BRA         CLRIRQ      ; (3)
DO_CALC
      LDA         ICBUFFER+3  ; (3) Calculate and store the 2
                              ; periods.
      SUB         ICBUFFER+1  ; (3)
      STA         ICBUFFER+1  ; (3)
      LDA         ICBUFFER+2  ; (3)
      SBC         ICBUFFER    ; (3)
      STA         ICBUFFER    ; (3)
      LDA         TCH0L       ; (3)
      SUB         ICBUFFER+3  ; (3)
      STA         ICBUFFER+3  ; (3)
      LDA         TCH0H       ; (3)
      SBC         ICBUFFER+2  ; (3)
      STA         ICBUFFER+2  ; (3)
```

```
AVERAGE
      LDA         ICBUFFER+3  ; (3) Average period by adding and
      ADD         ICBUFFER+1  ; (3) - shifting right.
      STA         ICBUUFER+3  ; (3)
      LDA         ICBUFFER+2  ; (3)
      ADC         ICBUFFER    ; (3)
      RORA                    ; (1) Divide by 2.
      STA         PERIOD      ; (3)
      LDA         ICBUFFER+3  ; (3)
      RORA                    ; (1)
      STA         PERIOD+1    ; (3) PERIOD now contains 16bit
                              ; average.
      CLR         TEMP        ; (3) Reinitialize edge counter.
CLRIRQ
      LDA         TSC0        ; (3)
      AND         #$7F        ; (2) Clear channel flag
      STA         TSC0        ; (3)
      PULH                    ; (2) Restore H.
      RTI                     ; (7)

      ORG         $FFF4       ; Vectors

      FDB         TIC0IRQ     ; Setup DMA interrupt vector.

      ORG         $FFFE

      FDB         TIMICSW     ; Setup Reset vector.
```

DMA Module

Contents

Introduction 106
708XL36 DMA Registers 107
708XL36 DMA Transfer Source Mapping 108
708XL36 Peripheral Interrupt Prioritization 108

Introduction

As an example of an actual implementation of the DMA module, this section contains register, bit and channel assignment details for the 708XL36 device. Refer to the applicable data book for full details of this MCU.

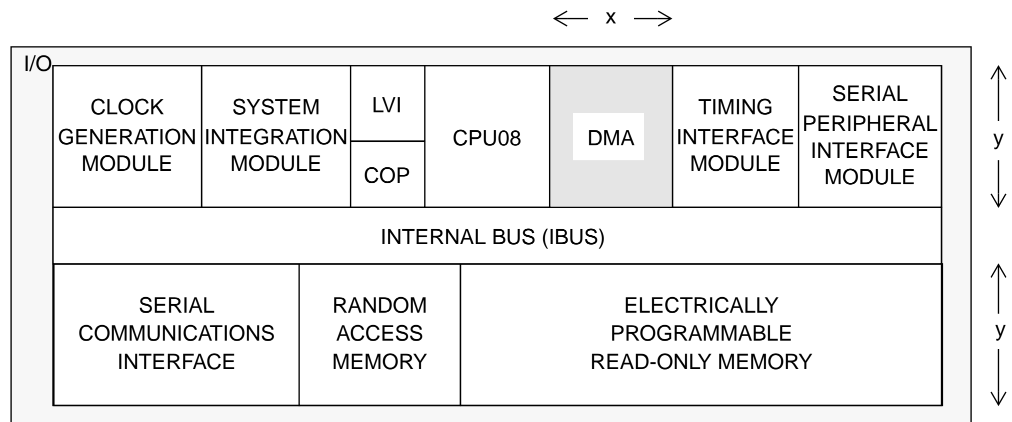


Figure 23. Diagram of the MC68HC708XL36 Layout

As can be seen from **Figure 23**, the 708XL36 contains a clock generation module (CGM), a system integration module (SIM), a timing interface module (TIM), a serial peripheral interface module (SPI), and a serial communications interface module (SCI) as well as a 3-channel DMA module. The TIM, SPI, and SCI modules can all generate requests for DMA transfers.

708XL36 DMA Registers

Table 10. MC68HC708XL36 DMA Registers

Register name	Address	Bit 7	6	5	4	3	2	1	Bit 0
SPI control (SPCR)	$0010	SPIE	DMAS	SPMSTR	CPOL	CPHA	SPWOM	SPE	
SCI control 3 (SCC3)	$0015	R8	T8	DMARE	DMATE	ORIE	NEIE	FEIE	PEIE
Timer DMA select (TDMA)	$0021					DMA3S	DMA2S	DMA1S	DMA0S
Ch. 0 source address (D0SH)	$0034	AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
Ch. 0 source address (D0SL)	$0035	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
Ch. 0 destination address (D0DH)	$0036	AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
Ch. 0 destination address (D0DL)	$0037	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
Ch. 0 control (D0C)	$0038	SDC3	SDC2	SDC1	SDC0	BWC	DTS2	DTS1	DTS0
Ch. 0 block length (D0BL)	$0039	BL7	BL6	BL5	BL4	BL3	BL2	BL1	BL0
Ch. 0 byte count (D0BC)	$003B	BC7	BC6	BC5	BC4	BC3	BC2	BC1	BC0
Ch. 1 source address (D1SH)	$003C	AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
Ch. 1 source address (D1SL)	$003D	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
Ch. 1 destination address (D1DH)	$003E	AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
Ch. 1 destination address (D1DL)	$003F	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
Ch. 1 control (D1C)	$0040	SDC3	SDC2	SDC1	SDC0	BWC	DTS2	DTS1	DTS0
Ch. 1 block length (D1BL)	$0041	BL7	BL6	BL5	BL4	BL3	BL2	BL1	BL0
Ch. 1 byte count (D1BC)	$0043	BC7	BC6	BC5	BC4	BC3	BC2	BC1	BC0
Ch. 2 source address (D2SH)	$0044	AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
Ch. 2 source address (D2SL)	$0045	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
Ch. 2 destination address (D2DH)	$0046	AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
Ch. 2 destination address (D2DL)	$0047	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
Ch. 2 control (D2C)	$0048	SDC3	SDC2	SDC1	SDC0	BWC	DTS2	DTS1	DTS0
Ch. 2 block length (D2BL)	$0049	BL7	BL6	BL5	BL4	BL3	BL2	BL1	BL0
Ch. 2 byte count (D2BC)	$004B	BC7	BC6	BC5	BC4	BC3	BC2	BC1	BC0
DMA control 1 (DC1)	$004C	BB1	BB0	TEC2	IEC2	TEC1	IEC1	TEC0	IEC0
DMA status and control (DSC)	$004D	DMAP	L2	L1	L0	DMAWE	IFC2	IFC1	IFC0
DMA control 2 (DC2)	$004E	SWI7	SWI6	SWI5	SWI4	SWI3	SWI2	SWI1	SWI0

NOTE: *Table 10 shows only those parts of the 708XL36 register block concerned with its three DMA channels and its modules' service request enable bits. (Refer to the applicable data book for details).*

708XL36 DMA Transfer Source Mapping

Bits DTS[2:0] in the channel control register assign the individual DMA channels to one of the eight transfer source inputs, as shown in **Table 11**. See DMA Channel Control Register on page 59 for further information on the DTS bits.

Table 11. DTS Bits

DTS[2:0]	Transfer Source
000	DMA service request input 0 = TIM channel 0
001	DMA service request input 1 = TIM channel 1
010	DMA service request input 2 = TIM channel 2
011	DMA service request input 3 = TIM channel 3
100	DMA service request input 4 = SPI receive
101	DMA service request input 5 = SPI transmit
110	DMA service request input 6 = SCI receive
111	DMA service request input 7 = SCI transmit

708XL36 Peripheral Interrupt Prioritization

Dependent on the state of the DMAP bit, DMA transfers can be either higher or lower in priority to the CPU interrupts shown. See DMA Status and Control Register on page 54 for further information on the DMAP bit.

Table 12. MC68HC708XL36 Peripheral Interrupt Prioritization

	Interrupt Source
Highest priority	Software interrupt
	External interrupt
	Peripheral interrupt 1 = Timer interrupt
	Peripheral interrupt 2 = SPI interrupt
Lowest priority	Peripheral interrupt 3 = SCI interrupt

DMA08 Version B

DMA Version B

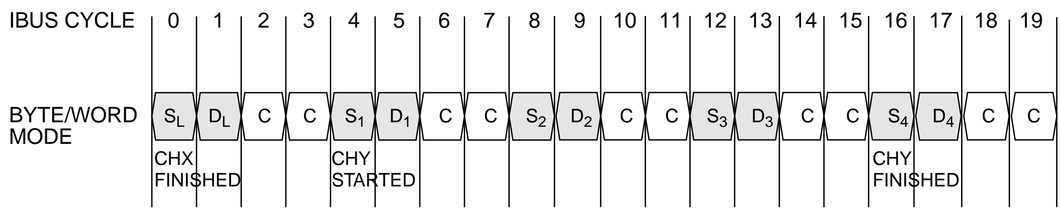
\Version B of the DMA08 also has three channels, but has a different word mode operation from version A. In version B all word transfers are allocated 100% of the total bandwidth irrespective of the bus bandwidth control bits in DMA Control Register 1. During word transfers there are no CPU cycles between the high and low bytes of the word or, in the case of block transfers, between words. This ensures that all words are transferred coherently, i.e., once a word transfer has started, the CPU cannot modify the source data values until the DMA releases the internal bus.

Whenever the active DMA channel changes or between block transfers when the DMA is configured for loop transfers, the number of CPU cycles is controlled by the bus bandwidth control bits according to **Table 13**.

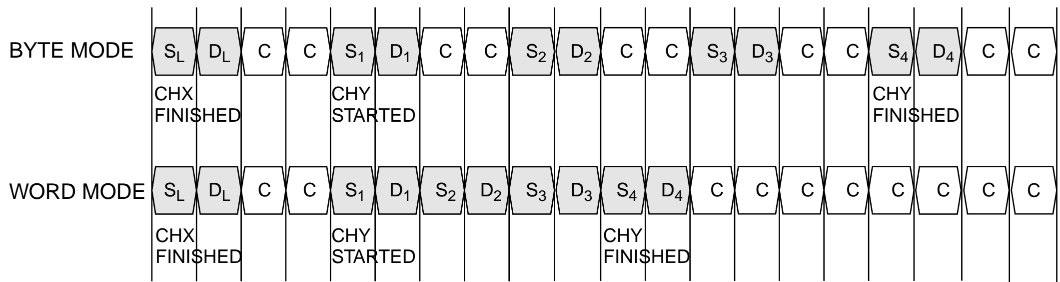
Table 13. DMA/CPU Bus Bandwidth Sharing (DMA08 Version B)

BB1:BB0	DMA/CPU Bus Bandwidth Sharing	
	Ratio (DMA/CPU)	Number of CPU cycles
00	25/75%	6
01	50/50%	2
10	67/33%	1
11	100/0%	1

Figure 24 highlights the difference between byte and word transfers for Versions A and B of the DMA. The particular case shown is Channel X finishing a transfer and Channel Y starting a 4-byte (2-word) transfer with a bandwidth of 50%. Note that on Version A of DMA08 the bus activity is the same for both byte and word transfers.



Version A of the DMA08



Version B of the DMA08

DMA CONTROLLED BUS CYCLE   CPU CONTROLLED BUS CYCLE

Figure 24. MCU Bus Activity During DMA Byte and Word Transfers (50% Bandwidth)

Glossary

$xxxx — The digits following the "$" are in hexadecimal format.

#xxxx — The digits following the "#" indicate an immediate operand.

A — Accumulator. See accumulator.

accumulator (A) — An 8-bit general-purpose register in the CPU08. The CPU08 uses the accumulator to hold operands and results of arithmetic and non-arithmetic operations.

address bus — The set of conductors used to select a specific memory location so that the CPU can write information into the memory location or read its contents.

addressing mode — The way that the CPU obtains (addresses) the information needed to complete an instruction. The M68HC08 CPU has 16 addressing modes.

algorithm — A set of specific procedures by which a solution is obtained in a finite number of steps, often used in numerical calculation.

ALU — Arithmetic logic unit. See arithmetic logic unit.

arithmetic logic unit (ALU) — The portion of the CPU of a computer where mathematical and logical operations take place. Other circuitry decodes each instruction and configures the ALU to perform the necessary arithmetic or logical operations at each step of an instruction.

assembly language — A method used by programmers for representing machine instructions (binary data) in a more convenient form. Each machine instruction is given a simple, short name, called a mnemonic (or memory aid), which has a

one-to-one correspondence with the machine instruction. The mnemonics are translated into an object code program which a microcontroller can use.

ASCII — American Standard Code for Information Interchange. A widely accepted correlation between alphabetic and numeric characters and specific 7-bit binary numbers.

asynchronous — Refers to circuitry and operations without common clock signals.

BCD — Binary-coded decimal. See binary-coded decimal.

binary — The binary number system using 2 as its base and using only the digits 0 and 1. Binary is the numbering system used by computers because any quantity can be represented by a series of ones and zeros. Electrically, these ones and zeros are represented by voltage levels of approximately V_{DD} (input) and V_{SS} (ground), respectively.

binary-coded decimal (BCD) — A notation that uses binary values to represent decimal quantities. Each BCD digit uses 4 binary bits. Six of the possible 16 binary combinations are considered illegal.

bit — A single binary digit. A bit can hold a single value of zero or one.

Boolean — A mathematical system of representing logic through a series of algebraic equations that can only be true or false, using operators such as AND, OR, and NOT.

branch instructions — Computer instructions that cause the CPU to continue processing at a memory location other than the next sequential address. Most branch instructions are conditional. That is, the CPU will continue to the next sequential address (no branch) if a condition is false, or continue to some other address (branch) if the condition is true.

bus — A collection of logic lines (conductor paths) used to transfer data.

byte — A set of exactly eight binary bits.

C — Abbreviation for carry/borrow in the condition code register of the CPU08. The CPU08 sets the carry/borrow flag when an addition operation produces a carry out of bit 7 of the accumulator or when a subtraction operation requires a borrow. Some logical operations and data manipulation instructions also clear or set the C flag (as in bit test and branch instructions and shifts and rotates).

CCR — Abbreviation for condition code register in the CPU08. See condition code register.

central processor unit (CPU) — The primary functioning unit of any computer system. The CPU controls the execution of instructions.

checksum — A value that results from adding a series of binary numbers. When exchanging information between computers, a checksum indicates the integrity of the data transfer. If values were transferred incorrectly, it is unlikely that the checksum would match the value that was expected.

clear — Establish logic zero state on a bit or bits; the opposite of set.

clock — A square wave signal used to sequence events in a computer.

condition code register (CCR) — An 8-bit register in the CPU08 that contains the interrupt mask bit and five bits (flags) that indicate the results of the instruction just executed.

control unit — One of two major units of the CPU. The control unit contains logic functions that synchronize the machine and direct various operations. The control unit decodes instructions and generates the internal control signals that perform the requested operations. The outputs of the control unit drive the execution unit, which contains the arithmetic logic unit (ALU), CPU registers, and bus interface.

CPU — Central processor unit. See central processor unit.

CPU08 — The central processor unit of the M68HC08 Family.

CPU cycles — A CPU clock cycle is one period of the internal bus-rate clock, normally derived by dividing a crystal oscillator source by two or more so the high and low times will be equal. The length of time required to execute an instruction is measured in CPU clock cycles.

CPU registers — Memory locations that are wired directly into the CPU logic instead of being part of the addressable memory map. The CPU always has direct access to the information in these registers. The CPU registers in an M68HC08 are:

— A (8-bit accumulator)

— H:X (16-bit accumulator)

— SP (16-bit stack pointer)

— PC (16-bit program counter)

— CCR (condition code register containing the V, H, I, N, Z, and C bits)

cycles — See CPU cycles.

data bus — A set of conductors used to convey binary information from a CPU to a memory location or from a memory location to a CPU.

decimal — Base 10 numbering system that uses the digits zero through nine.

direct address — Any address within the first 256 addresses of memory ($0000–$00FF). The high-order byte of these addresses is always $00. Special instructions allow these addresses to be accessed using only the low-order byte of their address. These instructions automatically fill in the assumed $00 value for the high-order byte of the address.

direct addressing mode — Direct addressing mode uses a program-supplied value for the low-order byte of the address of an operand. The high-order byte of the operand address is assumed to be $00 and so it does not have to be explicitly specified. Most direct addressing mode instructions can access any of the first 256 memory addresses.

direct memory access (DMA) — One of a number of modules that handle a variety of control functions in the modular M68HC08 Family. The DMA can perform interrupt-driven and software-initiated data transfers between any two CPU-addressable locations. Each DMA channel can independently transfer data between any addresses in the memory map. DMA transfers reduce CPU overhead required for data movement interrupts.

direct page — The first 256 bytes of memory ($0000–$00FF); also called page 0.

DMA — Direct memory access. See direct memory access.

effective address (EA) — The address where an instruction operand is located. The addressing mode of an instruction determines how the CPU calculates the effective address of the operand.

EPROM — Erasable, programmable, read-only memory. A non-volatile type of memory that can be erased by exposure to an ultraviolet light source.

execution unit (EU) — One of the two major units of the CPU containing the arithmetic logic unit (ALU), CPU registers, and bus interface. The outputs of the control unit drive the execution unit.

extended addressing mode — In this addressing mode, the high-order byte of the address of the operand is located in the next memory location after the opcode. The low-order byte of the operand address is located in the second memory location after the opcode. Extended addressing mode instructions can access any address in a 64-Kbyte memory map.

H — Abbreviation for the upper byte of the 16-bit index register (H:X) in the CPU08.

H — Abbreviation for half-carry in the condition code register of the CPU08. This bit indicates a carry from the low-order four bits of the accumulator value to the high-order four bits. The half-carry bit is required for binary-coded decimal arithmetic operations.

The decimal adjust accumulator (DAA) instruction uses the state of the H and C flags to determine the appropriate correction factor.

hexadecimal — Base 16 numbering system that uses the digits 0 through 9 and the letters A through F. One hexadecimal digit can exactly represent a 4-bit binary value. Hexadecimal is used by people to represent binary values because a two-digit number is easier to use than the equivalent eight-digit number.

high order — The leftmost digit(s) of a number.

H:X — Abbreviation for the 16-bit index register in the CPU08. The upper byte of H:X is called H. The lower byte is called X. In the indexed addressing modes, the CPU uses the contents of H:X to determine the effective address of the operand. H:X can also serve as a temporary data storage location.

I — Abbreviation for interrupt mask bit in the condition code register of the CPU08. When I is set, all interrupts are disabled. When I is cleared, interrupts are enabled.

immediate addressing mode — In immediate addressing mode, the operand is located in the next memory location(s) after the opcode. The immediate value is one or two bytes, depending on the size of the register involved in the instruction.

index register (H:X) — A 16-bit register in the CPU08. The upper byte of H:X is called H. The lower byte is called X. In the indexed addressing modes, the CPU uses the contents of H:X to determine the effective address of the operand. H:X can also serve as a temporary data storage location.

indexed addressing mode — Indexed addressing mode instructions access data with variable addresses. The effective address of the operand is determined by the current value of the H:X register added to a 0-, 8-, or 16-bit value (offset) in the instruction. There are separate opcodes for 0-, 8-, and 16-bit variations of indexed mode instructions, and so the CPU knows how many additional memory locations to read after the opcode.

indexed, post increment addressing mode — In this addressing mode, the effective address of the operand is determined by the current value of the index register, added to a 0- or 8-bit value (offset) in the instruction, after which the index register is incremented. Operands with variable addresses can be addressed with the 8-bit offset instruction.

inherent addressing mode — The inherent addressing mode has no operand because the opcode contains all the information necessary to carry out the instruction. Most inherent instructions are one byte long.

input/output (I/O) — Input/output interfaces between a computer system and the external world. A CPU reads an input to sense the level of an external signal and writes to an output to change the level on an external signal.

instructions — Instructions are operations that a CPU can perform. Instructions are expressed by programmers as assembly language mnemonics. A CPU interprets an opcode and its associated operand(s) and instruction.

instruction set — The instruction set of a CPU is the set of all operations that the CPU can perform. An instruction set is often represented with a set of shorthand mnemonics, such as LDA, meaning load accumulator (A). Another representation of an instruction set is with a set of opcodes that is recognized by the CPU.

interrupt — Interrupts provide a means to temporarily suspend normal program execution so that the CPU is freed to service sets of instructions in response to requests (interrupts) from peripheral devices. Normal program execution can be resumed later from its original point of departure. The CPU08 can process up to 128 separate interrupt sources, including a software interrupt (SWI).

I/O — Input/output. I/O interfaces between a computer system and the external world. A CPU reads an input to sense the level of an external signal and writes to an output to change the level on an external signal.

$\overline{\text{IRQ}}$ — Interrupt request. The overline indicates an active-low signal.

least significant bit (LSB) — The rightmost digit of a binary value.

logic one — A voltage level approximately equal to the input power voltage (V_{DD}).

logic zero — A voltage level approximately equal to the ground voltage (V_{SS}).

low order — The rightmost digit(s) of a number.

LS — Least significant.

LSB — Least significant bit. The rightmost digit of a binary value.

M68HC08 — A Motorola family of 8-bit MCUs.

machine codes — The binary codes processed by the CPU as instructions. Machine code includes both opcodes and operand data.

MCU — Microcontroller unit. A complete computer system, including a CPU, memory, a clock oscillator, and input/output (I/O) on a single integrated circuit.

memory location — In the M68HC08, each memory location holds one byte of data and has a unique address. To store information into a memory location, the CPU places the address of the location on the address bus, the data information on the data bus, and asserts the write signal. To read information from a memory location, the CPU places the address of the location on the address bus and asserts the read signal. In response to the read signal, the selected memory location places its data onto the data bus.

memory map — A pictorial representation of all memory locations in a computer system.

memory-to-memory addressing mode — In this addressing mode, the accumulator has been eliminated from the data transfer process, thereby reducing execution cycles. This addressing mode, therefore, provides rapid data transfers because it does not require use of the accumulator and associated load and

store instructions. There are four memory-to-memory addressing mode instructions. Depending on the instruction, operands are found in the byte following the opcode, in a direct page location addressed by the byte immediately following the opcode, or in a location addressed by the index register.

microcontroller — Microcontroller unit (MCU). A complete computer system, including a CPU, memory, a clock oscillator, and input/output (I/O) on a single integrated circuit.

mnemonic — Three to five letters that represent a computer operation. For example, the mnemonic form of the load accumulator instruction is LDA.

most significant bit (MSB) — The leftmost digit of a binary value.

MS — Abbreviation for most significant.

MSB — Most significant bit. The leftmost digit of a binary value.

N — Abbreviation for negative, a bit in the condition code register of the CPU08. The CPU sets the negative flag when an arithmetic operation, logical operation, or data manipulation produces a negative result.

nibble — Half a byte; four bits.

object code — The output from an assembler or compiler that is itself executable machine code, or is suitable for processing to produce executable machine code.

one — A logic high level, a voltage level approximately equal to the input power voltage (V_{DD}).

one’s complement — An infrequently used form of signed binary numbers. Negative numbers are simply the complement of their positive counterparts. One’s complement is the result of a bit-by-bit complement of a binary word, all ones are changed to zeros and all zeros changed to ones. One’s complement is two’s complement without the increment.

opcode — A binary code that instructs the CPU to do a specific operation in a specific way.

operand — The fundamental quantity on which a mathematical operation is performed. Usually a statement consists of an operator and an operand. The operator may indicate an add instruction; the operand, therefore, will indicate what is to be added.

oscillator — A circuit that produces a constant frequency square wave that is used by the computer as a timing and sequencing reference.

page 0 — The first 256 bytes of memory ($0000–$00FF). Also called direct page.

PC — Program counter. A 16-bit register in the CPU08. See program counter.

pointer — Pointer register. An index register is sometimes called a pointer register because its contents are used in the calculation of the address of an operand and, therefore, points to the operand.

program — A set of computer instructions that causes a computer to perform a desired operation or operations.

programming model — The registers of a particular CPU.

program counter (PC) — A 16-bit register in the CPU08. The PC register holds the address of the next instruction or operand that the CPU will use.

pull — The act of reading a value from the stack. In the M68HC08, a value is pulled by the following sequence of operations. First, the stack pointer register is incremented so that it points to the last value saved on the stack. Next, the value at the address contained in the stack pointer register is read into the CPU.

push — The act of storing a value at the address contained in the stack pointer register and then decrementing the stack pointer so that it points to the next available stack location.

RAM — Random access memory. All RAM locations can be read or written by the CPU. The contents of a RAM memory location remain valid until the CPU writes a different value or until power is turned off.

read — To transfer the contents of a memory location to the CPU.

registers — Memory locations wired directly into the CPU logic instead of being part of the addressable memory map. The CPU always has direct access to the information in these registers. The CPU registers in an M68HC08 are:

— A (8-bit accumulator)

— (H:X) (16-bit accumulator)

— SP (16-bit stack pointer)

— PC (16-bit program counter)

— CCR (condition code register containing the V, H, I, N, Z, and C bits)

Memory locations that hold status and control information for on-chip peripherals are called input/output (I/O) and control registers.

relative addressing mode — Relative addressing mode is used to calculate the destination address for branch instructions. If the branch condition is true, the signed 8-bit value after the opcode is added to the current value of the program counter to get the address where the CPU will fetch the next instruction. If the branch condition is false, the effective address is the content of the program counter.

reset — Reset is used to force a computer system to a known starting point and to force on-chip peripherals to known starting conditions.

ROM — Read-only memory. A type of memory that can be read but cannot be changed (written). The contents of ROM must be specified before manufacturing the MCU.

set — To establish a logic one state on a bit or bits; opposite of clear.

signed — A form of binary number representation accommodating both positive and negative numbers. The most significant bit is used to indicate whether the number is positive or negative, normally zero for positive and one for negative, and the other seven bits indicate the magnitude.

SIM — System integration module. One of a number of modules that handle a variety of control functions in the modular M68HC08 Family. The SIM controls mode of operation, resets and interrupts, and system clock generation.

SP — Stack pointer. A 16-bit register in the CPU08. See stack pointer.

stack — A mechanism for temporarily saving CPU register values during interrupts and subroutines. The CPU maintains this structure with the stack pointer (SP) register, which contains the address of the next available (empty) storage location on the stack. When a subroutine is called, the CPU pushes (stores) the low-order and high-order bytes of the return address on the stack before starting the subroutine instructions. When the subroutine is done, a return from subroutine (RTS) instruction causes the CPU to recover the return address from the stack and continue processing where it left off before the subroutine. Interrupts work in the same way except that all CPU registers are saved on the stack instead of just the program counter.

stack pointer (SP) — A 16-bit register in the CPU08 containing the address of the next available (empty) storage on the stack.

stack pointer addressing mode — Stack pointer (SP) addressing mode instructions operate like indexed addressing mode instructions except that the offset is added to the stack pointer instead of the index register (H:X). The effective address of the operand is formed by adding the unsigned byte(s) in the stack pointer to the unsigned byte(s) following the opcode.

subroutine — A sequence of instructions to be used more than once in the course of a program. The last instruction in a subroutine is a return from subroutine (RTS) instruction. At each place in the main program where the subroutine instructions are needed, a jump or branch to subroutine (JSR or BSR) instruction is used to

call the subroutine. The CPU leaves the flow of the main program to execute the instructions in the subroutine. When the RTS instruction is executed, the CPU returns to the main program where it left off.

synchronous — Refers to two or more things made to happen simultaneously in a system by means of a common clock signal.

system integration module (SIM) — One of a number of modules that handle a variety of control functions in the modular M68HC08 Family. The SIM controls mode of operation, resets and interrupts, and system clock generation.

table — A collection or ordering of data (such as square root values) laid out in rows and columns and stored in a computer memory as an array.

two's complement — A means of performing binary subtraction using addition techniques. The most significant bit of a two's complement number indicates the sign of the number (1 indicates negative). The two's complement negative of a number is obtained by inverting each bit in the number and then adding one to the result.

unsigned — Refers to a binary number representation in which all numbers are assumed positive. With signed binary, the most significant bit is used to indicate whether the number is positive or negative, normally zero for positive and one for negative, and the other seven bits indicating the magnitude.

variable — A value that changes during the course of executing a program.

word — Two bytes or 16 bits, treated as a unit.

write — The transfer of a byte of data from the CPU to a memory location.

X — Abbreviation for the lower byte of the index register (H:X) in the CPU08.

Z — Abbreviation for zero, a bit in the condition code register of the CPU08. The CPU08 sets the zero flag when an arithmetic operation, logical operation, or data manipulation produces a result of $00.

zero — A logic low level, a voltage level approximately equal to the ground voltage (V_{SS}).

Index

A

address bus – see IBUS
address calculation 46, 59
address extension registers. 28
ADX 28
ALU 31, 46
assembler listing 84

B

bandwidth control 47
base addresses 27, 46, 59–63
BB1, BB0 bits in DC1 52
block diagrams
    DMA module 22, 26
    DMA operation 30, 36
    MC68HC708XL 36 20, 106
breakpoints 33
buses
    bandwidth. 47
    BB1, BB0 – bandwidth control bits. 52
    DMA/CPU use of IBUS 47
    during byte transfer 41
    during word transfer 42
IBUS 36, 41, 42
    priority 47
BWC — bit in D0C 60
byte transfers. 40

C

control registers 31, 52
CPU
    interrupt latency 20
    limitations 18

D

D0BC – byte count register 29, 65
D0BL – block length register 30, 64
D0C – channel control register 59
D0DH, D0DL – destination address registers 63
D0SH, D0SL – source address registers. 62
data bus – see IBUS
DC1 – control register 1. 52
DC2 – control register 2. 57
destination addresses – see base addresses
DMAP — bit in DSC 54
DMAWE-bit in DSC 56
DSC – control/status register. 54
DTS2-DTS0 bits in D0C 61

E

examples
    large block transfer. 71
    small block transfer 69

F

features list 23

I

IBUS 36, 41, 42
IEC0 — bit in DC1 53
IFC0 – interrupt flag. 56

interrupts
IFC0 56
transfer interrupt priority 54

L

L0 — bit in DSC 55
latency
CPU 20
DMA register latency 51
DMA transfer latency 20, 44
looping 55
low power modes
stop mode 32
wait mode 32

M

MC68HC708XL36 106
block diagram 20, 106
interrupt priority 108
register summary 107
transfer sources 108
memory stretch 31
modes of operation
expanded 33
normal 36
stop 32
wait 32

R

register summary 50

S

SDC3–SDC0 bits in D0C 59
source addresses – see base addresses
source selection 61
status registers 31, 54

stop mode 32
SW17-SW10 bits in DC2 57

T

TEC0 — bit in DC1 53
timing diagrams
    byte transfers 40
    DMA/CPU use of IBUS 47
    word transfers 42, 110
transfers
    bus cycles 40
    byte transfers 40
    hardware interrupt-driven 37
    looping 55
    priority 54
    programming procedure 44
    software-initiated 37
    word transfers 42

W

wait mode 32, 56
word transfers 42

How to reach us:

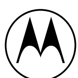
USA/EUROPE/Locations Not Listed: Motorola Literature Distribution; P.O. Box 20912; Phoenix, Arizona 85036.1-800-441-2447 or 602-303-5454

MFAX: RMFAX0@email.sps.mot.com – TOUCHTONE 602-244-6609

INTERNET: http://Design-NET.com

JAPAN: Nippon Motorola Ltd.; Tatsumi-SPD-JLDC, 6F Seibu-Butsuryu-Center, 3-14-2 Tatsumi Koto-Ku, Tokyo 135, Japan. 03-81-3521-8315

ASIA/PACIFIC: Motorola Semiconductors H.K. Ltd.; 8B Tai Ping Industrial Park, 51 Ting Kok Road, Tai Po, N.T., Hong Kong. 852-26629298



MOTOROLA