Chapter 5
Debug Support

This chapter describes the Revision B enhanced hardware debug support in the MC5307. This revision of the ColdFire debug architecture encompasses the earlier revision.

5.1 Overview

The debug module is shown in Figure 5-1.

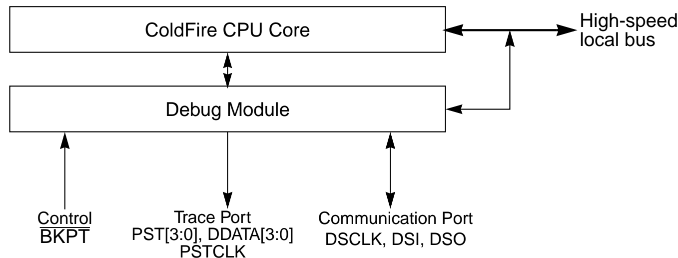


Figure 5-1. Processor/Debug Module Interface

Debug support is divided into three areas:

- Real-time trace support—The ability to determine the dynamic execution path through an application is fundamental for debugging. The ColdFire solution implements an 8-bit parallel output bus that reports processor execution status and data to an external emulator system. See Section 5.3, “Real-Time Trace Support.”
- Background debug mode (BDM)—Provides low-level debugging in the ColdFire processor complex. In BDM, the processor complex is halted and a variety of commands can be sent to the processor to access memory and registers. The external emulator uses a three-pin, serial, full-duplex channel. See Section 5.5, “Background Debug Mode (BDM),” and Section 5.4, “Programming Model.”
- Real-time debug support—BDM requires the processor to be halted, which many real-time embedded applications cannot do. Debug interrupts let real-time systems execute a unique service routine that can quickly save the contents of key registers and variables and return the system to normal operation. The emulator can access saved data because the hardware supports concurrent operation of the processor and BDM-initiated commands. See Section 5.6, “Real-Time Debug Support.”

The Version 2 ColdFire core implemented the original debug architecture, now called Revision A. Based on feedback from customers and third-party developers, enhancements have been added to succeeding generations of ColdFire cores. The Version 3 core implements Revision B of the debug architecture, providing more flexibility for configuring the hardware breakpoint trigger registers and removing the restrictions involving concurrent BDM processing while hardware breakpoint registers are active.

5.2 Signal Description

Table 5-1 describes debug module signals. All ColdFire debug signals are unidirectional and related to a rising edge of the processor core's clock signal. The standard 26-pin debug connector is shown in Section 5.7, "Motorola-Recommended BDM Pinout."

Table 5-1. Debug Module Signals

Signal	Description
Development Serial Clock (DSCLK)	Internally synchronized input. (The logic level on DSCLK is validated if it has the same value on two consecutive rising CLKIN edges.) Clocks the serial communication port to the debug module. Maximum frequency is 1/5 the processor CLK speed. At the synchronized rising edge of DSCLK, the data input on DSI is sampled and DSO changes state.
Development Serial Input (DSI)	Internally synchronized input that provides data input for the serial communication port to the debug module.
Development Serial Output (DSO)	Provides serial output communication for debug module responses. DSO is registered internally.
Breakpoint ($\overline{\text{BKPT}}$)	Input used to request a manual breakpoint. Assertion of $\overline{\text{BKPT}}$ puts the processor into a halted state after the current instruction completes. Halt status is reflected on processor status/debug data signals (PST[3:0]) as the value 0xF. If CSR[BKD] is set (disabling normal $\overline{\text{BKPT}}$ functionality), asserting $\overline{\text{BKPT}}$ generates a debug interrupt exception in the processor.
Processor Status Clock (PSTCLK)	Delayed version of the processor clock. Its rising edge appears in the center of valid PST and DDATA output. See Figure 5-2. PSTCLK indicates when the development system should sample PST and DDATA values. If real-time trace is not used, setting CSR[PCD] keeps PSTCLK, PST and DDATA outputs from toggling without disabling triggers. Non-quiescent operation can be reenabled by clearing CSR[PCD], although the emulator must resynchronize with the PST and DDATA outputs. PSTCLK starts clocking only when the first non-zero PST value (0xC, 0xD, or 0xF) occurs during system reset exception processing. Table 5-2 describes PST values. Chapter 7, "Phase-Locked Loop (PLL)," describes PSTCLK generation.
Debug Data (DDATA[3:0])	These output signals display the hardware register breakpoint status as a default, or optionally, captured address and operand values. The capturing of data values is controlled by the setting of the CSR. Additionally, execution of the WDDATA instruction by the processor captures operands which are displayed on DDATA. These signals are updated each processor cycle.
Processor Status (PST[3:0])	These output signals report the processor status. Table 5-2 shows the encoding of these signals. These outputs indicate the current status of the processor pipeline and, as a result, are not related to the current bus transfer. The PST value is updated each processor cycle.

Figure 5-2 shows PSTCLK timing with respect to PST and DDATA.

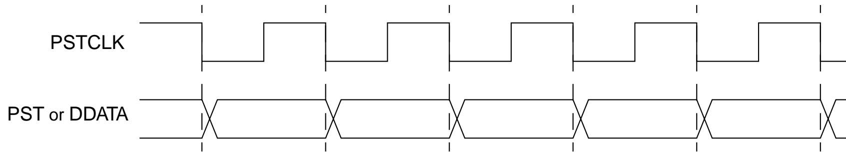


Figure 5-2. PSTCLK Timing

5.3 Real-Time Trace Support

Real-time trace, which defines the dynamic execution path, is a fundamental debug function. The ColdFire solution is to include a parallel output port providing encoded processor status and data to an external development system. This port is partitioned into two 4-bit nibbles: one nibble allows the processor to transmit processor status, (PST), and the other allows operand data to be displayed (debug data, DDATA). The processor status may not be related to the current bus transfer.

External development systems can use PST outputs with an external image of the program to completely track the dynamic execution path. This tracking is complicated by any change in flow, especially when branch target address calculation is based on the contents of a program-visible register (variant addressing). DDATA outputs can be configured to display the target address of such instructions in sequential nibble increments across multiple processor clock cycles, as described in Section 5.3.1, “Begin Execution of Taken Branch (PST = 0x5).” Two 32-bit storage elements form a FIFO buffer connecting the processor’s high-speed local bus to the external development system through PST[3:0] and DDATA[3:0]. The buffer captures branch target addresses and certain data values for eventual display on the DDATA port, one nibble at a time starting with the lsb.

Execution speed is affected only when both storage elements contain valid data to be dumped to the DDATA port. The core stalls until one FIFO entry is available.

Table 5-2 shows the encoding of these signals.

Table 5-2. Processor Status Encoding

PST[3:0]		Definition
Hex	Binary	
0x0	0000	Continue execution. Many instructions execute in one processor cycle. If an instruction requires more clock cycles, subsequent clock cycles are indicated by driving PST outputs with this encoding.
0x1	0001	Begin execution of one instruction. For most instructions, this encoding signals the first clock cycle of an instruction's execution. Certain change-of-flow opcodes, plus the PULSE and WDDATA instructions, generate different encodings.
0x2	0010	Reserved
0x3	0011	Entry into user-mode. Signaled after execution of the instruction that caused the ColdFire processor to enter user mode.
0x4	0100	Begin execution of PULSE and WDDATA instructions. PULSE defines logic analyzer triggers for debug and/or performance analysis. WDDATA lets the core write any operand (byte, word, or longword) directly to the DDATA port, independent of debug module configuration. When WDDATA is executed, a value of 0x4 is signaled on the PST port, followed by the appropriate marker, and then the data transfer on the DDATA port. Transfer length depends on the WDDATA operand size.
0x5	0101	Begin execution of taken branch. For some opcodes, a branch target address may be displayed on DDATA depending on the CSR settings. CSR also controls the number of address bytes displayed, indicated by the PST marker value preceding the DDATA nibble that begins the data output. See Section 5.3.1, "Begin Execution of Taken Branch (PST = 0x5)."
0x6	0110	Reserved
0x7	0111	Begin execution of return from exception (RTE) instruction.
0x8–0xB	1000–1011	Indicates the number of bytes to be displayed on the DDATA port on subsequent clock cycles. The value is driven onto the PST port one clock PSTCLK cycle before the data is displayed on DDATA. 0x8 Begin 1-byte transfer on DDATA. 0x9 Begin 2-byte transfer on DDATA. 0xA Begin 3-byte transfer on DDATA. 0xB Begin 4-byte transfer on DDATA.
0xC	1100	Exception processing. Exceptions that enter emulation mode (debug interrupt or optionally trace) generate a different encoding, as described below. Because the 0xC encoding defines a multiple-cycle mode, PST outputs are driven with 0xC until exception processing completes.
0xD	1101	Entry into emulator mode. Displayed during emulation mode (debug interrupt or optionally trace). Because this encoding defines a multiple-cycle mode, PSToutputs are driven with 0xD until exception processing completes.
0xE	1110	Processor is stopped. Appears in multiple-cycle format when the MCF5307 executes a STOP instruction. The ColdFire processor remains stopped until an interrupt occurs, thus PST outputs display 0xE until the stopped mode is exited.
0xF	1111	Processor is halted. Because this encoding defines a multiple-cycle mode, the PST outputs display 0xF until the processor is restarted or reset. (see Section 5.5.1, "CPU Halt")

5.3.1 Begin Execution of Taken Branch (PST = 0x5)

PST is 0x5 when a taken branch is executed. For some opcodes, a branch target address may be displayed on DDATA depending on the CSR settings. CSR also controls the number of address bytes displayed, which is indicated by the PST marker value immediately preceding the DDATA nibble that begins the data output.

Bytes are displayed in least-to-most-significant order. The processor captures only those target addresses associated with taken branches which use a variant addressing mode, that is, RTE and RTS instructions, JMP and JSR instructions using address register indirect or indexed addressing modes, and all exception vectors.

The simplest example of a branch instruction using a variant address is the compiled code for a C language case statement. Typically, the evaluation of this statement uses the variable of an expression as an index into a table of offsets, where each offset points to a unique case within the structure. For such change-of-flow operations, the MCF5307 uses the debug pins to output the following sequence of information on successive processor clock cycles:

1. Use PST (0x5) to identify that a taken branch was executed.
2. Using the PST pins, optionally signal the target address to be displayed sequentially on the DDATA pins. Encodings 0x9–0xB identify the number of bytes displayed.
3. The new target address is optionally available on subsequent cycles using the DDATA port. The number of bytes of the target address displayed on this port is configurable (2, 3, or 4 bytes).

Another example of a variant branch instruction would be a JMP (A0) instruction. Figure 5-3 shows when the PST and DDATA outputs that indicate when a JMP (A0) executed, assuming the CSR was programmed to display the lower 2 bytes of an address.

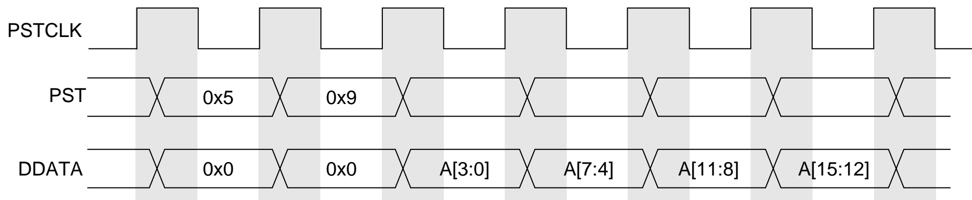


Figure 5-3. Example JMP Instruction Output on PST/DDATA

PST is driven with a 0x5 in the first cycle and 0x9 in the second. The 0x5 indicates a taken branch and the marker value 0x9 indicates a 2-byte address. Thus, the 4 subsequent DDATA nibbles display the lower 2 bytes of address register A0 in least-to-most-significant nibble order. The PST output after the JMP instruction completes depends on the target instruction. The PST can continue with the next instruction before the address has completely displayed on DDATA because of the DDATA FIFO. If the FIFO is full and the next instruction has captured values to display on DDATA, the pipeline stalls (PST = 0x0) until space is available in the FIFO.

5.4 Programming Model

In addition to the existing BDM commands that provide access to the processor’s registers and the memory subsystem, the debug module contains nine registers to support the required functionality. These registers are also accessible from the processor’s supervisor

programming model by executing the WDEBUG instruction. Thus, the breakpoint hardware in the debug module can be accessed by the external development system using the debug serial interface or by the operating system running on the processor core. Software is responsible for guaranteeing that accesses to these resources are serialized and logically consistent. Hardware provides a locking mechanism in the CSR to allow the external development system to disable any attempted writes by the processor to the breakpoint registers (setting CSR[IPW]). BDM commands must not be issued if the MCF5307 is using the WDEBUG instruction to access debug module registers or the resulting behavior is undefined.

These registers, shown in Figure 5-4, are treated as 32-bit quantities, regardless of the number of implemented bits.

Note: Each debug register is accessed as a 32-bit register; shaded fields above are not used (don’t care).
All debug control registers are writable from the external development system or the CPU via the WDEBUG instruction.
CSR is write-only from the programming model. It can be read or written through the BDM port using the RDMREG and WDMREG commands.

Figure 5-4. Debug Programming Model

These registers are accessed through the BDM port by new BDM commands, WDMREG and RDMREG, described in Section 5.5.3.3, “Command Set Descriptions.” These commands contain a 5-bit field, DRc, that specifies the register, as shown in Table 5-3.

Table 5-3. BDM/Breakpoint Registers

DRc[4–0]	Register Name	Abbreviation	Initial State	Page
0x00	Configuration/status register	CSR	0x0010_0000	p. 5-10
0x01–0x04	Reserved	—	—	—
0x05	BDM address attribute register	BAAR	0x0000_0005	p. 5-9
0x06	Address attribute trigger register	AATR	0x0000_0005	p. 5-7
0x07	Trigger definition register	TDR	0x0000_0000	p. 5-14
0x08	Program counter breakpoint register	PBR	—	p. 5-13
0x09	Program counter breakpoint mask register	PBMR	—	p. 5-13
0x0A–0x0B	Reserved	—	—	—
0x0C	Address breakpoint high register	ABHR	—	p. 5-8
0x0D	Address breakpoint low register	ABLR	—	p. 5-8
0x0E	Data breakpoint register	DBR	—	p. 5-12
0x0F	Data breakpoint mask register	DBMR	—	p. 5-12

NOTE:

Debug control registers can be written by the external development system or the CPU through the WDEBUG instruction.

CSR is write-only from the programming model. It can be read or written through the BDM port using the RDMREG and WDMREG commands.

5.4.1 Address Attribute Trigger Register (AATR)

The address attribute trigger register (AATR) defines address attributes and a mask to be matched in the trigger. The register value is compared with address attribute signals from the processor's local high-speed bus, as defined by the setting of the trigger definition register (TDR).

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	RM	SZM		TTM		TMM			R	SZ		TT		TM		
Reset	0000_0000_0000_0101															
R/W	Write only. AATR is accessible in supervisor mode as debug control register 0x06 using the WDEBUG instruction and through the BDM port using the WDMREG command.															
DRc[4–0]	0x06															

Figure 5-5. Address Attribute Trigger Register (AATR)

Table 5-4 describes AATR fields.

Table 5-4. AATR Field Descriptions

Bits	Name	Description
15	RM	Read/write mask. Setting RM masks R in address comparisons.
14–13	SZM	Size mask. Setting an SZM bit masks the corresponding SZ bit in address comparisons.
12–11	TTM	Transfer type mask. Setting a TTM bit masks the corresponding TT bit in address comparisons.
10–8	TMM	Transfer modifier mask. Setting a TMM bit masks the corresponding TM bit in address comparisons.
7	R	Read/write. R is compared with the R/$\overline{W}$ signal of the processor's local bus.
6–5	SZ	Size. Compared to the processor's local bus size signals. 00 Longword 01 Byte 10 Word 11 Reserved
4–3	TT	Transfer type. Compared with the local bus transfer type signals. 00 Normal processor access 01 Reserved 10 Emulator mode access 11 Acknowledge/CPU space access These bits also define the TT encoding for BDM memory commands. In this case, the 01 encoding indicates an external or DMA access (for backward compatibility). These bits affect the TM bits.
2–0	TM	Transfer modifier. Compared with the local bus transfer modifier signals, which give supplemental information for each transfer type. TT = 00 (normal mode): 000 Explicit cache line push 001 User data access 010 User code access 011 Reserved 100 Reserved 101 Supervisor data access 110 Supervisor code access 111 Reserved TT = 10 (emulator mode): 0xx–100 Reserved 101 Emulator mode data access 110 Emulator mode code access 111 Reserved TT = 11 (acknowledge/CPU space transfers): 000 CPU space access 001–111 Interrupt acknowledge levels 1–7 These bits also define the TM encoding for BDM memory commands (for backward compatibility).

5.4.2 Address Breakpoint Registers (ABLR, ABHR)

The address breakpoint low and high registers (ABLR, ABHR), Figure 5-6, define regions in the processor's data address space that can be used as part of the trigger. These register values are compared with the address for each transfer on the processor's high-speed local bus. The trigger definition register (TDR) identifies the trigger as one of three cases:

1. identically the value in ABLR
2. inside the range bound by ABLR and ABHR inclusive
3. outside that same range

	31 — 0
Field	Address
Reset	—
R/W	Write only. ABHR is accessible in supervisor mode as debug control register 0x0C using the WDEBUG instruction and via the BDM port using the RDMREG and WDMREG commands. ABLR is accessible in supervisor mode as debug control register 0x0D using the WDEBUG instruction and via the BDM port using the WDMREG command.
DRc[4–0]	0x0D (ABLR); 0x0C (ABHR)

Figure 5-6. Address Breakpoint Registers (ABLR, ABHR)

Table 5-5 describes ABLR fields.

Table 5-5. ABLR Field Description

Bits	Name	Description
31–0	Address	Low address. Holds the 32-bit address marking the lower bound of the address breakpoint range. Breakpoints for specific addresses are programmed into ABLR.

Table 5-6 describes ABHR fields.

Table 5-6. ABHR Field Description

Bits	Name	Description
31–0	Address	High address. Holds the 32-bit address marking the upper bound of the address breakpoint range.

5.4.3 BDM Address Attribute Register (BAAR)

The BAAR defines the address space for memory-referencing BDM commands. See Figure 5-7. The reset value of 0x5 sets supervisor data as the default address space.

	7	6	5	4	3	2	1	0
Field	R	SZ		TT		TM		
Reset	0000_0101							
R/W	Write only. BAAR[R,SZ] are loaded directly from the BDM command; BAAR[TT,TM] can be programmed as debug control register 0x05 from the external development system. For compatibility with Rev. A, BAAR is loaded each time AATR is written.							
DRc[4–0]	0x05							

Figure 5-7. BDM Address Attribute Register (BAAR)

Table 5-7 describes BAAR fields.

Table 5-7. BAAR Field Descriptions

Bits	Name	Description
7	R	Read/write 0 Write 1 Read
6–5	SZ	Size 00 Longword 01 Byte 10 Word 11 Reserved
4–3	TT	Transfer type. See the TT definition in Table 5-4.
2–0	TM	Transfer modifier. See the TM definition in Table 5-4.

5.4.4 Configuration/Status Register (CSR)

The configuration/status register (CSR) defines the debug configuration for the processor and memory subsystem and contains status information from the breakpoint logic.

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Field	BSTAT				FOF	TRG	HALT	BKPT	HRL				—	BKD	—	IPW
Reset	0000				0	0	0	0	0001				—	—	—	0
R/W[1]	R				R	R	R	R	R				—	—	—	R/W

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	MAP	TRC	EMU	DDC		UHE	BTB		—[2]	NPL	IPI	SSM	—			
Reset	0	0	0	00		0	00		0	0	—	0	—			
R/W	R/W	R/W	R/W	R/W		R/W	R/W		R	R/W	—	R/W	—			
DRc[4–0]	0x00															

1 CSR is write-only from the programming model. It can be read from and written to through the BDM port. CSR is accessible in supervisor mode as debug control register 0x00 using the WDEBUG instruction and through the BDM port using the RDMREG and WDMREG commands.

2 Bit 7 is reserved for Motorola use and must be written as a zero.

Figure 5-8. Configuration/Status Register (CSR)

Table 5-8 describes CSR fields.

Table 5-8. CSR Field Descriptions

Bit	Name	Description
31–28	BSTAT	Breakpoint status. Provides read-only status information concerning hardware breakpoints. BSTAT is cleared by a TDR write or by a CSR read when either a level-2 breakpoint is triggered or a level-1 breakpoint is triggered and the level-2 breakpoint is disabled. 0000 No breakpoints enabled 0001 Waiting for level-1 breakpoint 0010 Level-1 breakpoint triggered 0101 Waiting for level-2 breakpoint 0110 Level-2 breakpoint triggered
27	FOF	Fault-on-fault. If FOF is set, a catastrophic halt occurred and forced entry into BDM.
26	TRG	Hardware breakpoint trigger. If TRG is set, a hardware breakpoint halted the processor core and forced entry into BDM. Reset and the debug GO command clear TRG.
25	HALT	Processor halt. If HALT is set, the processor executed a HALT and forced entry into BDM. Reset and the debug GO command reset HALT.
24	BKPT	Breakpoint assert. If BKPT is set, $\overline{\text{BKPT}}$ was asserted, forcing the processor into BDM. Reset and the debug GO command clears this bit.
23–20	HRL	Hardware revision level. Indicates the level of debug module functionality. An emulator could use this information to identify the level of functionality supported. 0000 Initial debug functionality (Revision A) 0001 Revision B (this is the only valid value for the MCF5307)
19	—	Reserved, should be cleared.
18	BKD	Breakpoint disable. Used to disable the normal $\overline{\text{BKPT}}$ input functionality and to allow the assertion of $\overline{\text{BKPT}}$ to generate a debug interrupt. 0 Normal operation 1 $\overline{\text{BKPT}}$ is edge-sensitive: a high-to-low edge on $\overline{\text{BKPT}}$ signals a debug interrupt to the processor. The processor makes this interrupt request pending until the next sample point, when the exception is initiated. In the ColdFire architecture, the interrupt sample point occurs once per instruction. There is no support for nesting debug interrupts.
17	PCD	PSTCLK disable. Setting PCD disables generation of PSTCLK, PST and DDATA outputs and forces them to remain quiescent.
16	IPW	Inhibit processor writes. Setting IPW inhibits processor-initiated writes to the debug module's programming model registers. IPW can be modified only by commands from the external development system.
15	MAP	Force processor references in emulator mode. 0 All emulator-mode references are mapped into supervisor code and data spaces. 1 The processor maps all references while in emulator mode to a special address space, TT = 10, TM = 101 or 110.
14	TRC	Force emulation mode on trace exception. If TRC = 1, the processor enters emulator mode when a trace exception occurs.
13	EMU	Force emulation mode. If EMU = 1, the processor begins executing in emulator mode. See Section 5.6.1.1, "Emulator Mode."
12–11	DDC	Debug data control. Controls operand data capture for DDATA, which displays the number of bytes defined by the operand reference size before the actual data; byte displays 8 bits, word displays 16 bits, and long displays 32 bits (one nibble at a time across multiple clock cycles). See Table 5-2. 00 No operand data is displayed. 01 Capture all write data. 10 Capture all read data. 11 Capture all read and write data.

Table 5-8. CSR Field Descriptions (Continued)

Bit	Name	Description
10	UHE	User halt enable. Selects the CPU privilege level required to execute the HALT instruction. 0 HALT is a supervisor-only instruction. 1 HALT is a supervisor/user instruction.
9–8	BTB	Branch target bytes. Defines the number of bytes of branch target address DDATA displays. 00 0 bytes 01 Lower 2 bytes of the target address 10 Lower 3 bytes of the target address 11 Entire 4-byte target address See Section 5.3.1, "Begin Execution of Taken Branch (PST = 0x5)."
7	—	Reserved, should be cleared.
6	NPL	Non-pipelined mode. Determines whether the core operates in pipelined or mode or not. 0 Pipelined mode 1 Nonpipelined mode. The processor effectively executes one instruction at a time with no overlap. This adds at least 5 cycles to the execution time of each instruction. Instruction folding is disabled. Given an average execution latency of 1.6, throughput in non-pipeline mode would be 6.6, approximately 25% or less of pipelined performance. Regardless of the NPL state, a triggered PC breakpoint is always reported before the triggering instruction executes. In normal pipeline operation, the occurrence of an address and/or data breakpoint trigger is imprecise. In non-pipeline mode, triggers are always reported before the next instruction begins execution and trigger reporting can be considered precise. An address or data breakpoint should always occur before the next instruction begins execution. Therefore the occurrence of the address/data breakpoints should be guaranteed.
5	IPI	Ignore pending interrupts. 1 Core ignores any pending interrupt requests signalled while in single-instruction-step mode. 0 Core services any pending interrupt requests that were signalled while in single-step mode.
4	SSM	Single-step mode. Setting SSM puts the processor in single-step mode. 0 Normal mode. 1 Single-step mode. The processor halts after execution of each instruction. While halted, any BDM command can be executed. On receipt of the GO command, the processor executes the next instruction and halts again. This process continues until SSM is cleared.
3–0	—	Reserved, should be cleared.

5.4.5 Data Breakpoint/Mask Registers (DBR, DBMR)

The data breakpoint registers, Figure 5-9, specify data patterns used as part of the trigger into debug mode. Only DBR bits not masked with a corresponding zero in DBMR are compared with the data from the processor's local bus, as defined in TDR.

	31 … 0
Field	Data (DBR); Mask (DBMR)
Reset	Uninitialized
R/W	DBR is accessible in supervisor mode as debug control register 0x0E, using the WDEBUG instruction and through the BDM port using the RDMREG and WDMREG commands. DBMR is accessible in supervisor mode as debug control register 0x0F using the WDEBUG instruction and via the BDM port using the WDMREG command.
DRc[4–0]	0x0E (DBR), 0x0F (DBMR)

Figure 5-9. Data Breakpoint/Mask Registers (DBR and DBMR)

Table 5-9 describes DBR fields.

Table 5-9. DBR Field Descriptions

Bits	Name	Description
31–0	Data	Data breakpoint value. Contains the value to be compared with the data value from the processor's local bus as a breakpoint trigger.

Table 5-10 describes DBMR fields.

Table 5-10. DBMR Field Descriptions

Bits	Name	Description
31–0	Mask	Data breakpoint mask. The 32-bit mask for the data breakpoint trigger. Clearing a DBR bit allows the corresponding DBR bit to be compared to the appropriate bit of the processor's local data bus. Setting a DBMR bit causes that bit to be ignored.

The DBR supports both aligned and misaligned references. Table 5-11 shows relationships between processor address, access size, and location within the 32-bit data bus.

Table 5-11. Access Size and Operand Data Location

A[1:0]	Access Size	Operand Location
00	Byte	D[31:24]
01	Byte	D[23:16]
10	Byte	D[15:8]
11	Byte	D[7:0]
0x	Word	D[31:16]
1x	Word	D[15:0]
xx	Longword	D[31:0]

5.4.6 Program Counter Breakpoint/Mask Registers (PBR, PBMR)

The PC breakpoint register (PBR) defines an instruction address for use as part of the trigger. This register's contents are compared with the processor's program counter register when TDR is configured appropriately. PBR bits are masked by clearing corresponding PBMR bits. Results are compared with the processor's program counter register, as defined in TDR. Figure 5-10 shows the PC breakpoint register.

	31	1 0
Field	Program Counter	
Reset	—	
R/W	Write. PC breakpoint register is accessible in supervisor mode using the WDEBUG instruction and through the BDM port using the RDMREG and WDMREG commands using values shown in Section 5.5.3.3, “Command Set Descriptions.”	
DRc[4–0]	0x08	

Figure 5-10. Program Counter Breakpoint Register (PBR)

Table 5-12 describes PBR fields.

Table 5-12. PBR Field Descriptions

Bits	Name	Description
31–0	Address	PC breakpoint address. The 32-bit address to be compared with the PC as a breakpoint trigger.

Figure 5-11 shows PBMR.

	31 0
Field	Mask
Reset	—
R/W	Write. PBMR is accessible in supervisor mode as debug control register 0x09 using the WDEBUG instruction and via the BDM port using the wdmreg command.
DRc[4–0]	0x09

Figure 5-11. Program Counter Breakpoint Mask Register (PBMR)

Table 5-13 describes PBMR fields.

Table 5-13. PBMR Field Descriptions

Bits	Name	Description
31–0	Mask	PC breakpoint mask. A zero in a bit position causes the corresponding PBR bit to be compared to the appropriate PC bit. Set PBMR bits cause PBR bits to be ignored.

5.4.7 Trigger Definition Register (TDR)

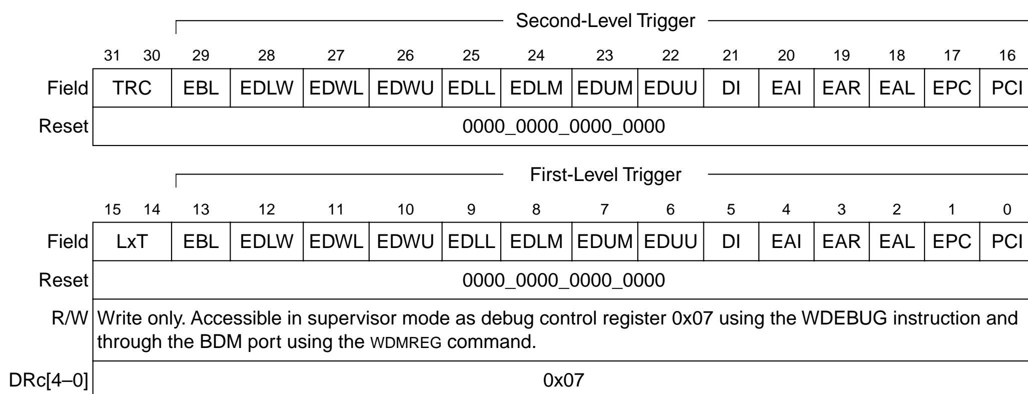
The TDR, shown in Table 5-12, configures the operation of the hardware breakpoint logic that corresponds with the ABHR/ABLR/AATR, PBR/PBMR, and DBR/DBMR registers within the debug module. The TDR controls the actions taken under the defined conditions. Breakpoint logic may be configured as a one- or two-level trigger. TDR[31–16] define the second-level trigger and bits 15–0 define the first-level trigger.

NOTE:

The debug module has no hardware interlocks, so to prevent spurious breakpoint triggers while the breakpoint registers are being loaded, disable TDR (by clearing TDR[29,13] before defining triggers.

A write to TDR clears the CSR trigger status bits, CSR[BSTAT].

Section Table 5-14., "TDR Field Descriptions," describes how to handle multiple breakpoint conditions.



	Second-Level Trigger													
Bits	31 30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Field	TRC	EBL	EDLW	EDWL	EDWU	EDLL	EDLM	EDUM	EDUU	DI	EAI	EAR	EAL	EPC	PCI
Reset	0000_0000_0000_0000														

	First-Level Trigger														
Bits	15 14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	LxT	EBL	EDLW	EDWL	EDWU	EDLL	EDLM	EDUM	EDUU	DI	EAI	EAR	EAL	EPC	PCI
Reset	0000_0000_0000_0000														
R/W	Write only. Accessible in supervisor mode as debug control register 0x07 using the WDEBUG instruction and through the BDM port using the WDMREG command.														
DRc[4–0]	0x07														

Figure 5-12. Trigger Definition Register (TDR)

Table 5-14 describes TDR fields.

Table 5-14. TDR Field Descriptions

Bits	Name	Description
31–30	TRC	Trigger response control. Determines how the processor responds to a completed trigger condition. The trigger response is always displayed on DDATA. 00 Display on DDATA only 01 Processor halt 10 Debug interrupt 11 Reserved
15:14	LxT	Level-x trigger. This is a Rev. B function. The Level-x Trigger bit determines the logic operation for the trigger between the PC_condition and the (Address_range & Data_condition) where the inclusion of a Data condition is optional. The ColdFire debug architecture supports the creation of single or double-level triggers. TDR[15] 0 Level-2 trigger = PC_condition & Address_range & Data_condition 1 Level-2 trigger = PC_condition | (Address_range & Data_condition) TDR[14] 0 Level-1 trigger = PC_condition & Address_range & Data_condition 1 Level-1 trigger = PC_condition | (Address_range & Data_condition)
29/13	EBL	Enable breakpoint. Global enable for the breakpoint trigger. Setting TDR[EBL] enables a breakpoint trigger. Clearing it disables all breakpoints.

Table 5-14. TDR Field Descriptions (Continued)

Bits	Name	Description	
28–22 12–6	EDx	Setting an EDx bit enables the corresponding data breakpoint condition based on the size and placement on the processor's local data bus. Clearing all EDx bits disables data breakpoints.	
28/12		EDLW	Data longword. Entire processor's local data bus.
27/11		EDWL	Lower data word.
26/10		EDWU	Upper data word.
25/9		EDLL	Lower lower data byte. Low-order byte of the low-order word.
24/8		EDLM	Lower middle data byte. High-order byte of the low-order word.
23/7		EDUM	Upper middle data byte. Low-order byte of the high-order word.
22/6		EDUU	Upper upper data byte. High-order byte of the high-order word.
21/5	DI	Data breakpoint invert. Provides a way to invert the logical sense of all the data breakpoint comparators. This can develop a trigger based on the occurrence of a data value other than the DBR contents.	
20–18/ 4–2	EAx	Enable address bits. Setting an EA bit enables the corresponding address breakpoint. Clearing all three bits disables the breakpoint.	
20/4		EAI	Enable address breakpoint inverted. Breakpoint is based outside the range between ABLR and ABHR.
19/3		EAR	Enable address breakpoint range. The breakpoint is based on the inclusive range defined by ABLR and ABHR.
18/2		EAL	Enable address breakpoint low. The breakpoint is based on the address in the ABLR.
17/1	EPC	Enable PC breakpoint. If set, this bit enables the PC breakpoint.	
16/0	PCI	Breakpoint invert. If set, this bit allows execution outside a given region as defined by PBR and PBMR to enable a trigger. If cleared, the PC breakpoint is defined within the region defined by PBR and PBMR.	

5.5 Background Debug Mode (BDM)

The ColdFire Family implements a low-level system debugger in the microprocessor hardware. Communication with the development system is handled through a dedicated, high-speed serial command interface. The ColdFire architecture implements the BDM controller in a dedicated hardware module. Although some BDM operations, such as CPU register accesses, require the CPU to be halted, other BDM commands, such as memory accesses, can be executed while the processor is running.

5.5.1 CPU Halt

Although many BDM operations can occur in parallel with CPU operations, unrestricted BDM operation requires the CPU to be halted. The sources that can cause the CPU to halt are listed below in order of priority:

1. A catastrophic fault-on-fault condition automatically halts the processor.

2. A hardware breakpoint can be configured to generate a pending halt condition similar to the assertion of BKPT. This type of halt is always first made pending in the processor. Next, the processor samples for pending halt and interrupt conditions once per instruction. When a pending condition is asserted, the processor halts execution at the next sample point. See Section 5.6.1, "Theory of Operation."
3. The execution of a HALT instruction immediately suspends execution. Attempting to execute HALT in user mode while CSR[UHE] = 0 generates a privilege violation exception. If CSR[UHE] = 1, HALT can be executed in user mode. After HALT executes, the processor can be restarted by serial shifting a GO command into the debug module. Execution continues at the instruction after HALT.
4. The assertion of the BKPT input is treated as a pseudo-interrupt; that is, the halt condition is postponed until the processor core samples for halts/interrupts. The processor samples for these conditions once during the execution of each instruction. If there is a pending halt condition at the sample time, the processor suspends execution and enters the halted state.

The assertion of BKPT should be considered in the following two special cases:

- After the system reset signal is negated, the processor waits for 16 processor clock cycles before beginning reset exception processing. If the BKPT input is asserted within eight cycles after RSTI is negated, the processor enters the halt state, signaling halt status (0xF) on the PST outputs. While the processor is in this state, all resources accessible through the debug module can be referenced. This is the only chance to force the processor into emulation mode through CSR[EMU].

  After system initialization, the processor's response to the GO command depends on the set of BDM commands performed while it is halted for a breakpoint. Specifically, if the PC register was loaded, the GO command causes the processor to exit halted state and pass control to the instruction address in the PC, bypassing normal reset exception processing. If the PC was not loaded, the GO command causes the processor to exit halted state and continue reset exception processing.
- The ColdFire architecture also handles a special case of BKPT being asserted while the processor is stopped by execution of the STOP instruction. For this case, the processor exits the stopped mode and enters the halted state, at which point, all BDM commands may be exercised. When restarted, the processor continues by executing the next sequential instruction, that is, the instruction following the STOP opcode.

CSR[27–24] indicates the halt source, showing the highest priority source for multiple halt conditions.

5.5.2 BDM Serial Interface

When the CPU is halted and PST reflects the halt status, the development system can send unrestricted commands to the debug module. The debug module implements a synchronous protocol using two inputs (DSCLK and DSI) and one output (DSO), where DSCLK and

DSI must meet the required input setup and hold timings and the DSO is specified as a delay relative to the rising edge of the processor clock. See Table 5-1. The development system serves as the serial communication channel master and must generate DSCLK.

The serial channel operates at a frequency from DC to 1/5 of the processor frequency. The channel uses full-duplex mode, where data is sent and received simultaneously by both master and slave devices. The transmission consists of 17-bit packets composed of a status/control bit and a 16-bit data word. As shown in Figure 5-13, all state transitions are enabled on a rising edge of the processor clock when DSCLK is high; that is, DSI is sampled and DSO is driven.

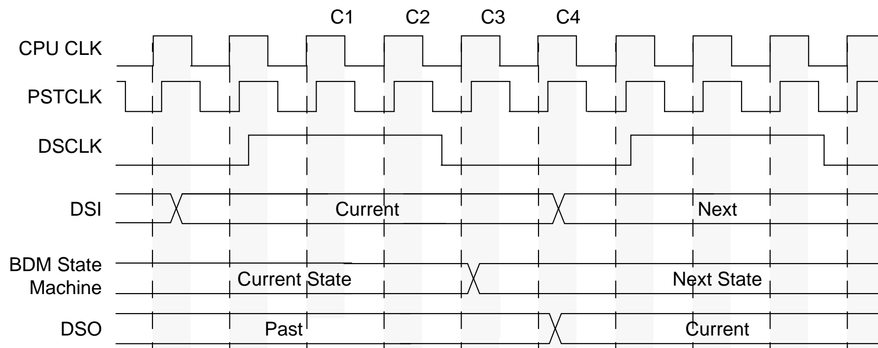


Figure 5-13. BDM Serial Interface Timing

DSCLK and DSI are synchronized inputs. DSCLK acts as a pseudo clock enable and is sampled on the rising edge of the processor CLK as well as the DSI. DSO is delayed from the DSCLK-enabled CLK rising edge (registered after a BDM state machine state change). All events in the debug module's serial state machine are based on the processor clock rising edge. DSCLK must also be sampled low (on a positive edge of CLK) between each bit exchange. The MSB is transferred first. Because DSO changes state based on an internally-recognized rising edge of DSCLK, DSDO cannot be used to indicate the start of a serial transfer. The development system must count clock cycles in a given transfer. C1–C4 are described as follows:

- C1—First synchronization cycle for DSI (DSCLK is high).
- C2—Second synchronization cycle for DSI (DSCLK is high).
- C3—BDM state machine changes state depending upon DSI and whether the entire input data transfer has been transmitted.
- C4—DSO changes to next value.

NOTE:

A not-ready response can be ignored except during a memory-referencing cycle. Otherwise, the debug module can accept a new serial transfer after 32 processor clock periods.

5.5.2.1 Receive Packet Format

The basic receive packet, Figure 5-14, consists of 16 data bits and 1 status bit.

Figure 5-14. Receive BDM Packet

Table 5-15 describes receive BDM packet fields.

Table 5-15. Receive BDM Packet Field Description

Bits	Name	Description
16	S	Status. Indicates the status of CPU-generated messages listed below. The not-ready response can be ignored unless a memory-referencing cycle is in progress. Otherwise, the debug module can accept a new serial transfer after 32 processor clock periods. S Data Message 0 xxxx Valid data transfer 0 0xFFFF Status OK 1 0x0000 Not ready with response; come again 1 0x0001 Error—Terminated bus cycle; data invalid 1 0xFFFF Illegal command
15–0	Data	Data. Contains the message to be sent from the debug module to the development system. The response message is always a single word, with the data field encoded as shown above.

5.5.2.2 Transmit Packet Format

The basic transmit packet, Figure 5-15, consists of 16 data bits and 1 control bit.

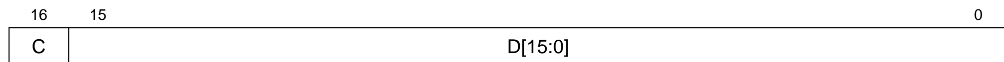


Figure 5-15. Transmit BDM Packet

Table 5-16 describes transmit BDM packet fields.

Table 5-16. Transmit BDM Packet Field Description

Bits	Name	Description
16	C	Control. This bit is reserved. Command and data transfers initiated by the development system should clear C.
15–0	Data	Contains the data to be sent from the development system to the debug module.

5.5.3 BDM Command Set

Table 5-17 summarizes the BDM command set. Subsequent paragraphs contain detailed descriptions of each command. Issuing a BDM command when the processor is accessing debug module registers using the WDEBUG instruction causes undefined behavior.

Table 5-17. BDM Command Summary

Command	Mnemonic	Description	CPU State[1]	Section	Command (Hex)
Read A/D register	RAREG/ RDREG	Read the selected address or data register and return the results through the serial interface.	Halted	5.5.3.3.1	0x218 {A/D, Reg[2:0]}
Write A/D register	WAREG/ WDREG	Write the data operand to the specified address or data register.	Halted	5.5.3.3.2	0x208 {A/D, Reg[2:0]}
Read memory location	READ	Read the data at the memory location specified by the longword address.	Steal	5.5.3.3.3	0x1900—byte 0x1940—word 0x1980—lword
Write memory location	WRITE	Write the operand data to the memory location specified by the longword address.	Steal	5.5.3.3.4	0x1800—byte 0x1840—word 0x1880—lword
Dump memory block	DUMP	Used with READ to dump large blocks of memory. An initial READ is executed to set up the starting address of the block and to retrieve the first result. A DUMP command retrieves subsequent operands.	Steal	5.5.3.3.5	0x1D00—byte 0x1D40—word 0x1D80—lword
Fill memory block	FILL	Used with WRITE to fill large blocks of memory. An initial WRITE is executed to set up the starting address of the block and to supply the first operand. A FILL command writes subsequent operands.	Steal	5.5.3.3.6	0x1C00—byte 0x1C40—word 0x1C80—lword
Resume execution	GO	The pipeline is flushed and refilled before resuming instruction execution at the current PC.	Halted	5.5.3.3.7	0x0C00
No operation	NOP	Perform no operation; may be used as a null command.	Parallel	5.5.3.3.8	0x0000
Output the current PC	SYNC_PC	Capture the current PC and display it on the PST/DDATA output pins.	Parallel	5.5.3.3.9	0x0001
Read control register	RCREG	Read the system control register.	Halted	5.5.3.3.10	0x2980
Write control register	WCREG	Write the operand data to the system control register.	Halted	5.5.3.3.11	0x2880
Read debug module register	RDMREG	Read the debug module register.	Parallel	5.5.3.3.12	0x2D {0x4[2] DRc[4:0]}
Write debug module register	WDMREG	Write the operand data to the debug module register.	Parallel	5.5.3.3.13	0x2C {0x4[2] Drc[4:0]}

[1] General command effect and/or requirements on CPU operation:
- Halted. The CPU must be halted to perform this command.
- Steal. Command generates bus cycles that can be interleaved with bus accesses.
- Parallel. Command is executed in parallel with CPU activity.

[2] 0x4 is a three-bit field.

Unassigned command opcodes are reserved by Motorola. All unused command formats within any revision level perform a NOP and return the illegal command response.

5.5.3.1 ColdFire BDM Command Format

All ColdFire Family BDM commands include a 16-bit operation word followed by an optional set of one or more extension words, as shown in Figure 5-16.

15–10	9	8	7–6	5	4	3	2–0
Operation	0	R/W	Op Size	0	0	A/D	Register
Extension Word(s)							

Figure 5-16. BDM Command Format

Table 5-18 describes BDM fields.

Table 5-18. BDM Field Descriptions

Bit	Name	Description
15–10	Operation	Specifies the command. These values are listed in Table 5-17.
9	0	Reserved
8	R/W	Direction of operand transfer. 0 Data is written to the CPU or to memory from the development system. 1 The transfer is from the CPU to the development system.
7–6	Operand Size	Operand data size for sized operations. Addresses are expressed as 32-bit absolute values. Note that a command performing a byte-sized memory read leaves the upper 8 bits of the response data undefined. Referenced data is returned in the lower 8 bits of the response. Operand Size Bit Values 00 Byte 8 bits 01 Word 16 bits 10 Longword 32 bits 11 Reserved —
5–4	00	Reserved
3	A/D	Address/data. Determines whether the register field specifies a data or address register. 0 Indicates a data register. 1 Indicates an address register.
2–0	Register	Contains the register number in commands that operate on processor registers.

5.5.3.1.1 Extension Words as Required

Some commands require extension words for addresses and/or immediate data. Addresses require two extension words because only absolute long addressing is permitted. Longword accesses are forcibly longword-aligned and word accesses are forcibly word-aligned. Immediate data can be 1 or 2 words long. Byte and word data each requires a single extension word and longword data requires two extension words.

Operands and addresses are transferred most-significant word first. In the following descriptions of the BDM command set, the optional set of extension words is defined as address, data, or operand data.

5.5.3.2 Command Sequence Diagrams

The command sequence diagram in Figure 5-17 shows serial bus traffic for commands. Each bubble represents a 17-bit bus transfer. The top half of each bubble indicates the data the development system sends to the debug module; the bottom half indicates the debug module's response to the previous development system commands. Command and result transactions overlap to minimize latency.

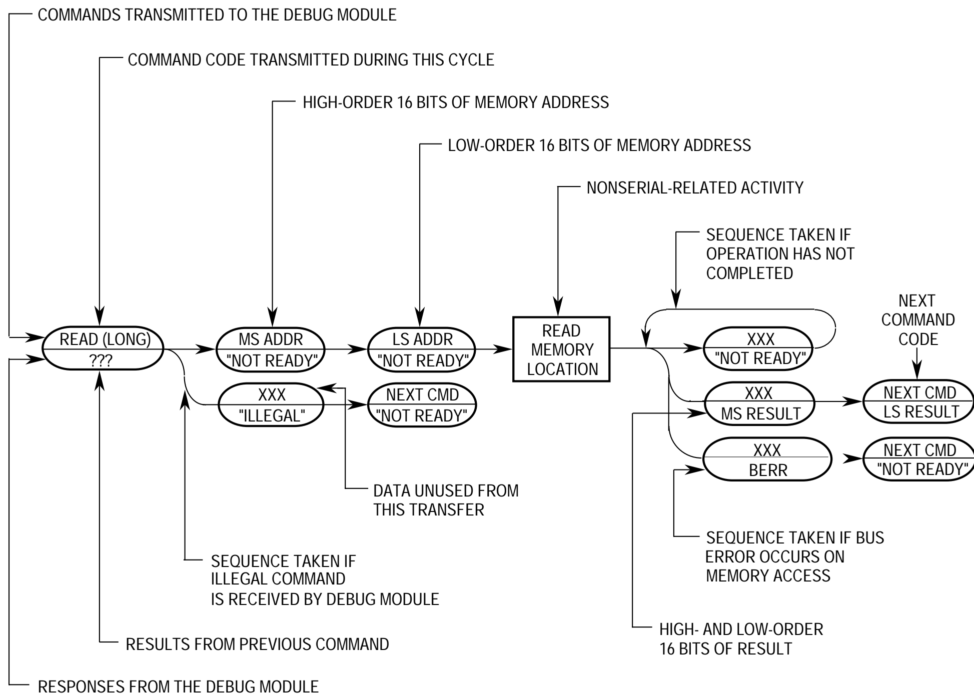


Figure 5-17. Command Sequence Diagram

The sequence is as follows:

- In cycle 1, the development system command is issued (READ in this example). The debug module responds with either the low-order results of the previous command or a command complete status of the previous command, if no results are required.
- In cycle 2, the development system supplies the high-order 16 address bits. The debug module returns a not-ready response unless the received command is decoded as unimplemented, which is indicated by the illegal command encoding. If this occurs, the development system should retransmit the command.

NOTE:

A not-ready response can be ignored except during a memory-referencing cycle. Otherwise, the debug module can accept a new serial transfer after 32 processor clock periods.

- In cycle 3, the development system supplies the low-order 16 address bits. The debug module always returns a not-ready response.
- At the completion of cycle 3, the debug module initiates a memory read operation. Any serial transfers that begin during a memory access return a not-ready response.

- Results are returned in the two serial transfer cycles after the memory access completes. For any command performing a byte-sized memory read operation, the upper 8 bits of the response data are undefined and the referenced data is returned in the lower 8 bits. The next command's opcode is sent to the debug module during the final transfer. If a memory or register access is terminated with a bus error, the error status (S = 1, DATA = 0x0001) is returned instead of result data.

5.5.3.3 Command Set Descriptions

The following sections describe the commands summarized in Table 5-17.

NOTE:

The BDM status bit (S) is 0 for normally completed commands; S = 1 for illegal commands, not-ready responses, and transfers with bus-errors. Section 5.5.2, "BDM Serial Interface," describes the receive packet format.

Motorola reserves unassigned command opcodes for future expansion. Unused command formats in any revision level perform a NOP and return an illegal command response.

5.5.3.3.1 Read A/D Register (RAREG/RDREG)

Read the selected address or data register and return the 32-bit result. A bus error response is returned if the CPU core is not halted.

Command/Result Formats:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Command	0x2				0x1				0x8				A/D	Register		
Result	D[31:16]															
	D[15:0]															

Figure 5-18. RAREG/RDREG Command Format

Command Sequence:

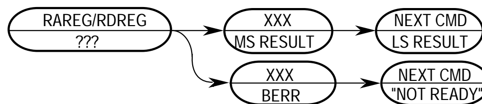


Figure 5-19. RAREG/RDREG Command Sequence

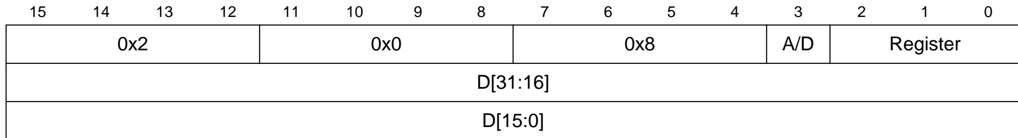
Operand Data: None

Result Data: The contents of the selected register are returned as a longword value, most-significant word first.

5.5.3.3.2 Write A/D Register (WAREG/WDREG)

The operand longword data is written to the specified address or data register. A write alters all 32 register bits. A bus error response is returned if the CPU core is not halted.

Command Format:



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x2				0x0				0x8				A/D	Register		
D[31:16]															
D[15:0]															

Figure 5-20. WAREG/WDREG Command Format

Command Sequence

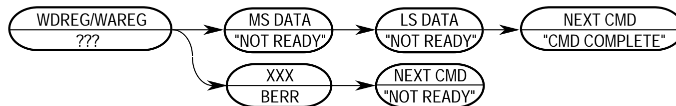


Figure 5-21. WAREG/WDREG Command Sequence

Operand Data — Longword data is written into the specified address or data register. The data is supplied most-significant word first.

Result Data — Command complete status is indicated by returning 0xFFFF (with S cleared) when the register write is complete.

5.5.3.3.3 Read Memory Location (READ)

Read data at the longword address. Address space is defined by BAAR[TT,TM]. Hardware forces low-order address bits to zeros for word and longword accesses to ensure that word addresses are word-aligned and longword addresses are longword-aligned.

Command/Result Formats:

<table>
<tr><th></th><th></th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td rowspan="4">Byte</td><td rowspan="3">Command</td><td colspan="4">0x1</td><td colspan="4">0x9</td><td colspan="4">0x0</td><td colspan="4">0x0</td></tr>
<tr><td colspan="16">A[31:16]</td></tr>
<tr><td colspan="16">A[15:0]</td></tr>
<tr><td>Result</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td colspan="8">D[7:0]</td></tr>
<tr><td rowspan="4">Word</td><td rowspan="3">Command</td><td colspan="4">0x1</td><td colspan="4">0x9</td><td colspan="4">0x4</td><td colspan="4">0x0</td></tr>
<tr><td colspan="16">A[31:16]</td></tr>
<tr><td colspan="16">A[15:0]</td></tr>
<tr><td>Result</td><td colspan="16">D[15:0]</td></tr>
<tr><td rowspan="5">Longword</td><td rowspan="3">Command</td><td colspan="4">0x1</td><td colspan="4">0x9</td><td colspan="4">0x8</td><td colspan="4">0x0</td></tr>
<tr><td colspan="16">A[31:16]</td></tr>
<tr><td colspan="16">A[15:0]</td></tr>
<tr><td rowspan="2">Result</td><td colspan="16">D[31:16]</td></tr>
<tr><td colspan="16">D[15:0]</td></tr>
</table>

Figure 5-22. READ Command/Result Formats

Command Sequence:

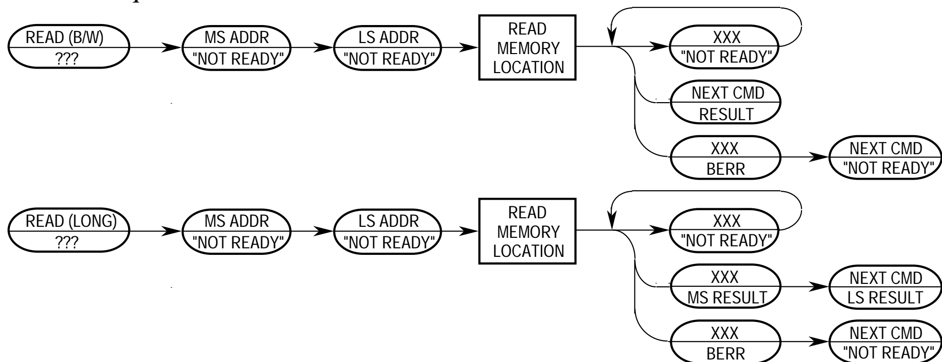


Figure 5-23. READ Command Sequence

Operand Data	The only operand is the longword address of the requested location.
Result Data	Word results return 16 bits of data; longword results return 32. Bytes are returned in the LSB of a word result, the upper byte is undefined. 0x0001 (S = 1) is returned if a bus error occurs.

5.5.3.3.4 Write Memory Location (WRITE)

Write data to the memory location specified by the longword address. The address space is defined by BAAR[TT,TM]. Hardware forces low-order address bits to zeros for word and longword accesses to ensure that word addresses are word-aligned and longword addresses are longword-aligned.

Command Formats:

<table>
<tr><th></th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th></tr>
<tr><td rowspan="4">Byte</td><td colspan="4">0x1</td><td colspan="4">0x8</td><td colspan="4">0x0</td><td colspan="3">0x0</td></tr>
<tr><td colspan="15">A[31:16]</td></tr>
<tr><td colspan="15">A[15:0]</td></tr>
<tr><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td colspan="7">D[7:0]</td></tr>
<tr><td rowspan="4">Word</td><td colspan="4">0x1</td><td colspan="4">0x8</td><td colspan="4">0x4</td><td colspan="3">0x0</td></tr>
<tr><td colspan="15">A[31:16]</td></tr>
<tr><td colspan="15">A[15:0]</td></tr>
<tr><td colspan="15">D[15:0]</td></tr>
<tr><td rowspan="5">Longword</td><td colspan="4">0x1</td><td colspan="4">0x8</td><td colspan="4">0x8</td><td colspan="3">0x0</td></tr>
<tr><td colspan="15">A[31:16]</td></tr>
<tr><td colspan="15">A[15:0]</td></tr>
<tr><td colspan="15">D[31:16]</td></tr>
<tr><td colspan="15">D[15:0]</td></tr>
</table>

Figure 5-24. WRITE Command Format

Command Sequence:

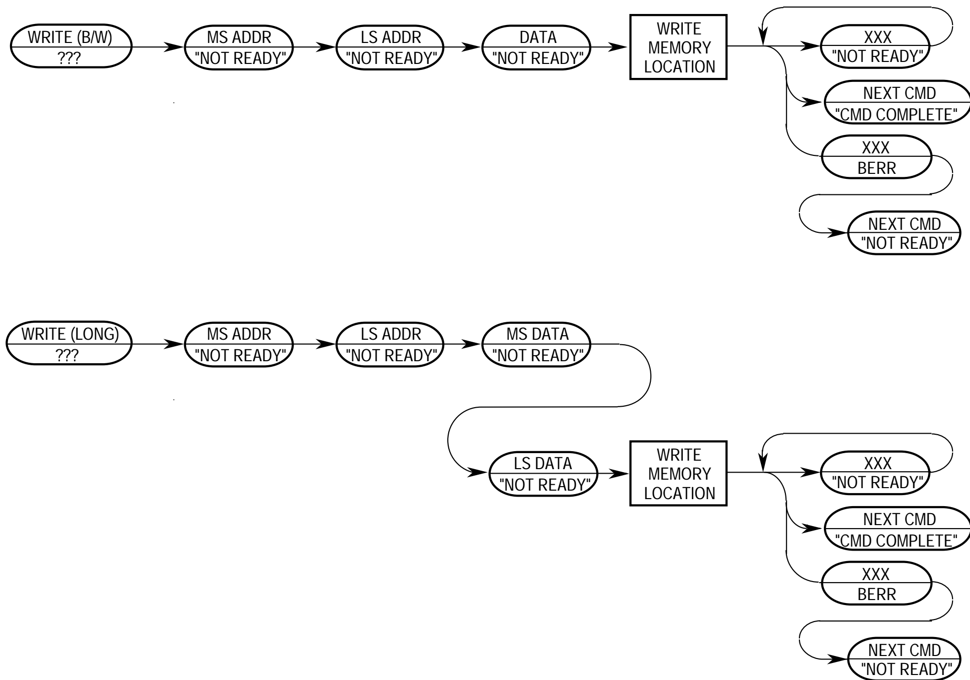


Figure 5-25. WRITE Command Sequence

Operand Data — This two-operand instruction requires a longword absolute address that specifies a location to which the data operand is to be written. Byte data is sent as a 16-bit word, justified in the LSB; 16- and 32-bit operands are sent as 16 and 32 bits, respectively

Result Data — Command complete status is indicated by returning 0xFFFF (with S cleared) when the register write is complete. A value of 0x0001 (with S set) is returned if a bus error occurs.

5.5.3.3.5 Dump Memory Block (DUMP)

DUMP is used with the READ command to access large blocks of memory. An initial READ is executed to set up the starting address of the block and to retrieve the first result. If an initial READ is not executed before the first DUMP, an illegal command response is returned. The DUMP command retrieves subsequent operands. The initial address is incremented by the operand size (1, 2, or 4) and saved in a temporary register. Subsequent DUMP commands use this address, perform the memory read, increment it by the current operand size, and store the updated address in the temporary register.

NOTE:

> DUMP does not check for a valid address; it is a valid command only when preceded by NOP, READ, or another DUMP command. Otherwise, an illegal command response is returned. NOP can be used for intercommand padding without corrupting the address pointer.

The size field is examined each time a DUMP command is processed, allowing the operand size to be dynamically altered.

Command/Result Formats:

<table>
<tr><th></th><th></th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td rowspan="2">Byte</td><td>Command</td><td colspan="4">0x1</td><td colspan="4">0xD</td><td colspan="4">0x0</td><td colspan="4">0x0</td></tr>
<tr><td>Result</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td colspan="8">D[7:0]</td></tr>
<tr><td rowspan="2">Word</td><td>Command</td><td colspan="4">0x1</td><td colspan="4">0xD</td><td colspan="4">0x4</td><td colspan="4">0x0</td></tr>
<tr><td>Result</td><td colspan="16">D[15:0]</td></tr>
<tr><td rowspan="3">Longword</td><td>Command</td><td colspan="4">0x1</td><td colspan="4">0xD</td><td colspan="4">0x8</td><td colspan="4">0x0</td></tr>
<tr><td rowspan="2">Result</td><td colspan="16">D[31:16]</td></tr>
<tr><td colspan="16">D[15:0]</td></tr>
</table>

Figure 5-26. DUMP Command/Result Formats

Command Sequence:

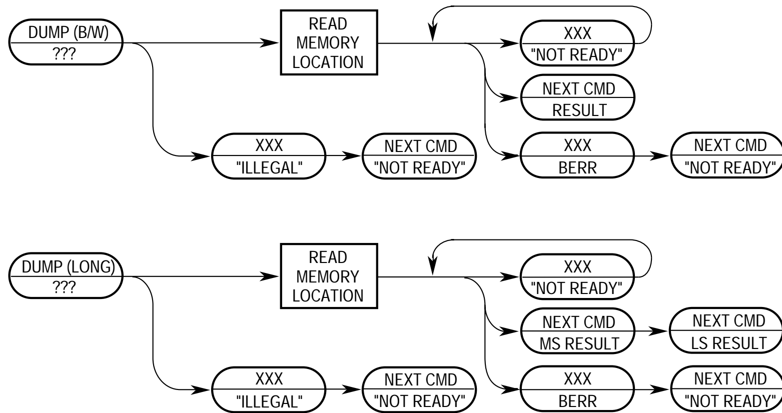


Figure 5-27. DUMP Command Sequence

Operand Data: None

Result Data: Requested data is returned as either a word or longword. Byte data is returned in the least-significant byte of a word result. Word results return 16 bits of significant data; longword results return 32 bits. A value of 0x0001 (with S set) is returned if a bus error occurs.

5.5.3.3.6 Fill Memory Block (FILL)

A FILL command is used with the WRITE command to access large blocks of memory. An initial WRITE is executed to set up the starting address of the block and to supply the first operand. The FILL command writes subsequent operands. The initial address is incremented by the operand size (1, 2, or 4) and saved in a temporary register after the memory write. Subsequent FILL commands use this address, perform the write, increment it by the current operand size, and store the updated address in the temporary register.

If an initial WRITE is not executed preceding the first FILL command, the illegal command response is returned.

NOTE:

The FILL command does not check for a valid address—FILL is a valid command only when preceded by another FILL, a NOP, or a WRITE command. Otherwise, an illegal command response is returned. The NOP command can be used for intercommand padding without corrupting the address pointer.

The size field is examined each time a FILL command is processed, allowing the operand size to be altered dynamically.

Command Formats:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Byte	0x1				0xC				0x0				0x0			
	X	X	X	X	X	X	X	X	D[7:0]							
Word	0x1				0xC				0x4				0x0			
	D[15:0]															
Longword	0x1				0xC				0x8				0x0			
	D[31:16]															
	D[15:0]															

Figure 5-28. FILL Command Format

Command Sequence:

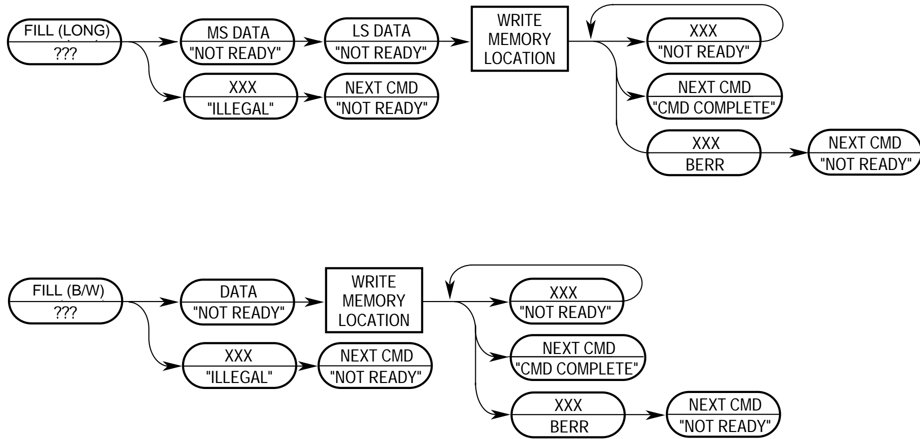


Figure 5-29. FILL Command Sequence

Operand Data: A single operand is data to be written to the memory location. Byte data is sent as a 16-bit word, justified in the least-significant byte; 16- and 32-bit operands are sent as 16 and 32 bits, respectively.

Result Data: Command complete status (0xFFFF) is returned when the register write is complete. A value of 0x0001 (with S set) is returned if a bus error occurs.

5.5.3.3.7 Resume Execution (GO)

The pipeline is flushed and refilled before normal instruction execution resumes. Prefetching begins at the current address in the PC and at the current privilege level. If any register (such as the PC or SR) is altered by a BDM command while the processor is halted, the updated value is used when prefetching resumes. If a GO command is issued and the CPU is not halted, the command is ignored.

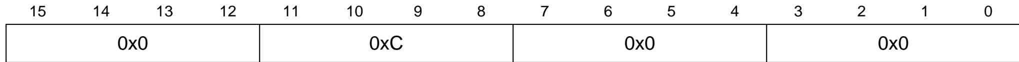


Figure 5-30. GO Command Format

Command Sequence:

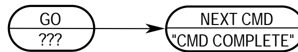


Figure 5-31. GO Command Sequence

Operand Data: None

Result Data: The command-complete response (0xFFFF) is returned during the next shift operation.

5.5.3.3.8 No Operation (NOP)

NOP performs no operation and may be used as a null command where required.

Command Formats:

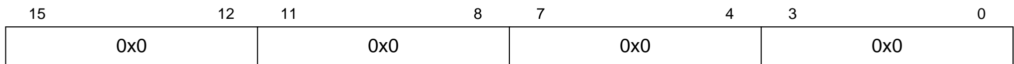


Figure 5-32. NOP Command Format

Command Sequence:

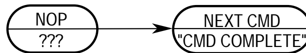


Figure 5-33. NOP Command Sequence

Operand Data: None

Result Data: The command-complete response, 0xFFFF (with S cleared), is returned during the next shift operation.

5.5.3.3.9 Synchronize PC to the PST/DDATA Lines (SYNC_PC)

The SYNC_PC command captures the current PC and displays it on the PST/DDATA outputs. After the debug module receives the command, it sends a signal to the ColdFire processor that the current PC must be displayed. The processor then forces an instruction fetch at the next PC with the address being captured in the DDATA logic under control of CSR[BTB]. The specific sequence of PST and DDATA values is as follows:

1. Debug signals a SYNC_PC command is pending.
2. CPU completes the current instruction.
3. CPU forces an instruction fetch to the next PC, generates a PST = 0x5 value indicating a taken branch and signals the capture of DDATA.
4. The instruction address corresponding to the PC is captured.
5. The PST marker (0x9–0xB) is generated and displayed as defined by CSR[BTB] followed by the captured PC address.

The SYNC_PC command can be used to dynamically access the PC for performance monitoring. The execution of this command is considerably less obtrusive to the real-time operation of an application than a HALT-CPU/READ-PC/RESUME command sequence.

Command Formats:

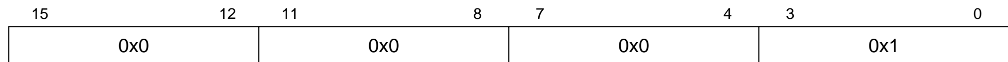


Figure 5-34. SYNC_PC Command Format

Command Sequence:

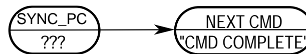


Figure 5-35. SYNC_PC Command Sequence

Operand Data: None

Result Data: Command complete status (0xFFFF) is returned when the register write is complete.

5.5.3.3.10 Read Control Register (RCREG)

Read the selected control register and return the 32-bit result. Accesses to the processor/memory control registers are always 32 bits wide, regardless of register width. The second and third words of the command form a 32-bit address, which the debug module uses to generate a special bus cycle to access the specified control register. The 12-bit Rc field is the same as that used by the MOVEC instruction.

Command/Result Formats:

	15–12	11–8	7–4	3–0
Command	0x2	0x9	0x8	0x0
	0x0	0x0	0x0	0x0
	0x0	Rc		
Result	D[31:16]			
	D[15:0]			

Figure 5-36. RCREG Command/Result Formats

Rc encoding:

Table 5-19. Control Register Map

Rc	Register Definition	Rc	Register Definition
0x002	Cache control register (CACR)	0x805	MAC mask register (MASK)[1]
0x004	Access control register 0 (ACR0)	0x806	MAC accumulator (ACC)[1]
0x005	Access control register 1 (ACR1)	0x80E	Status register (SR)
0x801	Vector base register (VBR)	0x80F	Program register (PC)
0x804	MAC status register (MACSR)[1]	0xC04	RAM base address register (RAMBAR)

[1] Available if the optional MAC unit is present.

Command Sequence:

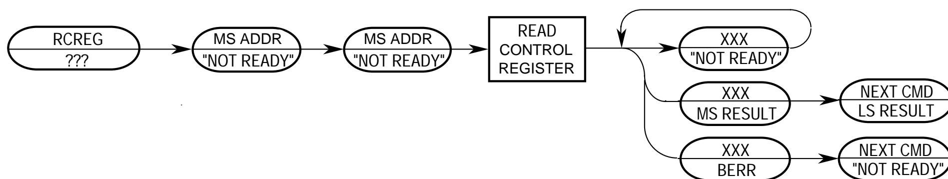


Figure 5-37. RCREG Command Sequence

Operand Data: The only operand is the 32-bit Rc control register select field.

Result Data: Control register contents are returned as a longword, most-significant word first. The implemented portion of registers smaller than 32 bits is guaranteed correct; other bits are undefined.

5.5.3.3.11 Write Control Register (WCREG)

The operand (longword) data is written to the specified control register. The write alters all 32 register bits.

Command/Result Formats:

	15–12	11–8	7–4	3–0
Command	0x2	0x8	0x8	0x0
	0x0	0x0	0x0	0x0
	0x0	Rc		
Result	D[31:16]			
	D[15:0]			

Figure 5-38. WCREG Command/Result Formats

Command Sequence:

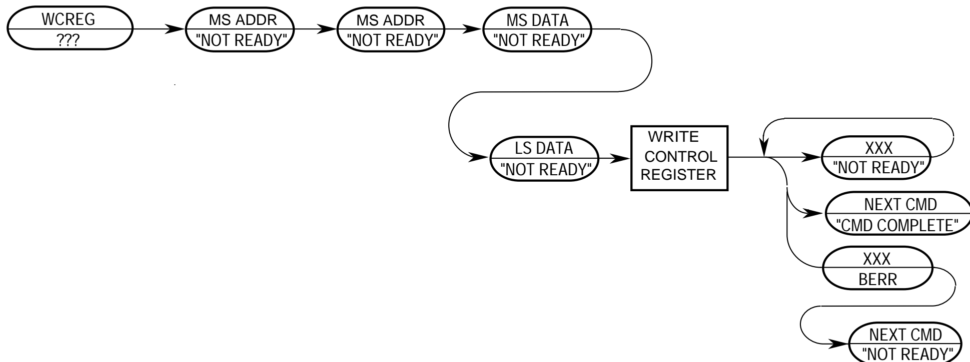


Figure 5-39. WCREG Command Sequence

Operand Data: This instruction requires two longword operands. The first selects the register to which the operand data is to be written; the second contains the data.

Result Data: Successful write operations return 0xFFFF. Bus errors on the write cycle are indicated by the setting of bit 16 in the status message and by a data pattern of 0x0001.

5.5.3.3.12 Read Debug Module Register (RDMREG)

Read the selected debug module register and return the 32-bit result. The only valid register selection for the RDMREG command is CSR (DRc = 0x00). Note that this read of the CSR clears the trigger status bits (CSR[BSTAT]) if either a level-2 breakpoint has been triggered or a level-1 breakpoint has been triggered and no level-2 breakpoint has been enabled.

Command/Result Formats:

<table>
<tr><td></td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>Command</td><td colspan="4">0x2</td><td colspan="4">0xD</td><td colspan="3">0x4[1]</td><td colspan="5">DRc</td></tr>
<tr><td rowspan="2">Result</td><td colspan="16">D[31:16]</td></tr>
<tr><td colspan="16">D[15:0]</td></tr>
</table>

Figure 5-40. RDMREG BDM Command/Result Formats

[1] Note 0x4 is a 3-bit field

Table 5-20 shows the definition of DRc encoding.

Table 5-20. Definition of DRc Encoding—Read

DRc[4:0]	Debug Register Definition	Mnemonic	Initial State	Page
0x00	Configuration/Status	CSR	0x0	p. 5-10
0x01–0x1F	Reserved	—	—	—

Command Sequence:

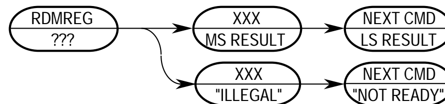


Figure 5-41. RDMREG Command Sequence

Operand Data: None

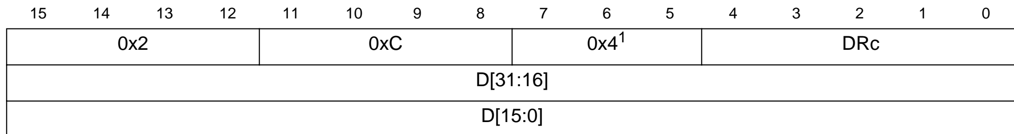
Result Data: The contents of the selected debug register are returned as a longword value. The data is returned most-significant word first.

5.5.3.3.13 Write Debug Module Register (WDMREG)

The operand (longword) data is written to the specified debug module register. All 32 bits of the register are altered by the write. DSCLK must be inactive while the debug module register writes from the CPU accesses are performed using the WDEBUG instruction.

Command Format:

Figure 5-42. WDMREG BDM Command Format



[1] Note: 0x4 is a three-bit field

Table 5-3 shows the definition of the DRc write encoding.

Command Sequence:

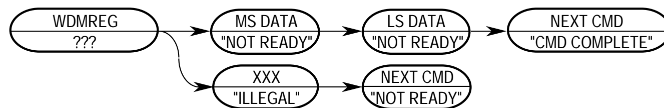


Figure 5-43. WDMREG Command Sequence

Operand Data: Longword data is written into the specified debug register. The data is supplied most-significant word first.

Result Data: Command complete status (0xFFFF) is returned when register write is complete.

5.6 Real-Time Debug Support

The ColdFire Family provides support debugging real-time applications. For these types of embedded systems, the processor must continue to operate during debug. The foundation of this area of debug support is that while the processor cannot be halted to allow debugging, the system can generally tolerate small intrusions into the real-time operation.

The debug module provides three types of breakpoints—PC with mask, operand address range, and data with mask. These breakpoints can be configured into one- or two-level triggers with the exact trigger response also programmable. The debug module programming model can be written from either the external development system using the debug serial interface or from the processor's supervisor programming model using the WDEBUG instruction. Only CSR is readable using the external development system.

5.6.1 Theory of Operation

Breakpoint hardware can be configured to respond to triggers in several ways. The response desired is programmed into TDR. As shown in Table 5-21, when a breakpoint is triggered, an indication (CSR[BSTAT]) is provided on the DDATA output port when it is not displaying captured processor status, operands, or branch addresses.

Table 5-21. DDATA[3:0]/CSR[BSTAT] Breakpoint Response

DDATA[3:0]/CSR[BSTAT] [1]	Breakpoint Status
0000/0000	No breakpoints enabled
0010/0001	Waiting for level-1 breakpoint
0100/0010	Level-1 breakpoint triggered
1010/0101	Waiting for level-2 breakpoint
1100/0110	Level-2 breakpoint triggered

[1] Encodings not shown are reserved for future use.

The breakpoint status is also posted in CSR. Note that CSR[BSTAT] is cleared by a CSR read when either a level-2 breakpoint is triggered or a level-1 breakpoint is triggered and a level-2 breakpoint is not enabled. Status is also cleared by writing to TDR.

BDM instructions use the appropriate registers to load and configure breakpoints. As the system operates, a breakpoint trigger generates the response defined in TDR.

PC breakpoints are treated in a precise manner—exception recognition and processing are initiated before the excepting instruction is executed. All other breakpoint events are recognized on the processor's local bus, but are made pending to the processor and sampled like other interrupt conditions. As a result, these interrupts are imprecise.

In systems that tolerate the processor being halted, a BDM-entry can be used. With TDR[TRC] = 01, a breakpoint trigger causes the core to halt (PST = 0xF).

If the processor core cannot be halted, the debug interrupt can be used. With this configuration, TDR[TRC] = 10, the breakpoint trigger becomes a debug interrupt to the processor, which is treated higher than the nonmaskable level-7 interrupt request. As with all interrupts, it is made pending until the processor reaches a sample point, which occurs once per instruction. Again, the hardware forces the PC breakpoint to occur before the targeted instruction executes. This is possible because the PC breakpoint is enabled when interrupt sampling occurs. For address and data breakpoints, reporting is considered imprecise because several instructions may execute after the triggering address or data is detected.

As soon as the debug interrupt is recognized, the processor aborts execution and initiates exception processing. This event is signaled externally by the assertion of a unique PST value (PST = 0xD) for multiple cycles. The core enters emulator mode when exception processing begins. After the standard 8-byte exception stack is created, the processor

fetches a unique exception vector, 12, from the vector table.

Execution continues at the instruction address in the vector corresponding to the breakpoint triggered. All interrupts are ignored while the processor is in emulator mode. The debug interrupt handler can use supervisor instructions to save the necessary context such as the state of all program-visible registers into a reserved memory area.

When debug interrupt operations complete, the RTE instruction executes and the processor exits emulator mode. After the debug interrupt handler completes execution, the external development system can use BDM commands to read the reserved memory locations.

The generation of another debug interrupt during the first instruction after the RTE exits emulator mode is inhibited. This behavior is consistent with the existing logic involving trace mode where the first instruction executes before another trace exception is generated. Thus, all hardware breakpoints are disabled until the first instruction after the RTE completes execution, regardless of the programmed trigger response.

5.6.1.1 Emulator Mode

Emulator mode is used to facilitate non-intrusive emulator functionality. This mode can be entered in three different ways:

- Setting CSR[EMU] forces the processor into emulator mode. EMU is examined only if $\overline{\text{RSTI}}$ is negated and the processor begins reset exception processing. It can be set while the processor is halted before reset exception processing begins. See Section 5.5.1, “CPU Halt.”
- A debug interrupt always puts the processor in emulation mode when debug interrupt exception processing begins.
- Setting CSR[TRC] forces the processor into emulation mode when trace exception processing begins.

While operating in emulation mode, the processor exhibits the following properties:

- All interrupts are ignored, including level-7 interrupts.
- If CSR[MAP] = 1, all caching of memory and the SRAM module are disabled. All memory accesses are forced into a specially mapped address space signaled by TT = 0x2, TM = 0x5 or 0x6. This includes stack frame writes and the vector fetch for the exception that forced entry into this mode.

The RTE instruction exits emulation mode. The processor status output port provides a unique encoding for emulator mode entry (0xD) and exit (0x7).

5.6.2 Concurrent BDM and Processor Operation

The debug module supports concurrent operation of both the processor and most BDM commands. BDM commands may be executed while the processor is running, except those following operations that access processor/memory registers:

- Read/write address and data registers
- Read/write control registers

For BDM commands that access memory, the debug module requests the processor's local bus. The processor responds by stalling the instruction fetch pipeline and waiting for current bus activity to complete before freeing the local bus for the debug module to perform its access. After the debug module bus cycle, the processor reclaims the bus.

Breakpoint registers must be carefully configured in a development system if the processor is executing. The debug module contains no hardware interlocks, so TDR should be disabled while breakpoint registers are loaded, after which TDR can be written to define the exact trigger. This prevents spurious breakpoint triggers.

Because there are no hardware interlocks in the debug unit, no BDM operations are allowed while the CPU is writing the debug's registers (DSCLK must be inactive).

5.7 Motorola-Recommended BDM Pinout

The ColdFire BDM connector, Figure 5-44, is a 26-pin Berg connector arranged 2 x 13.

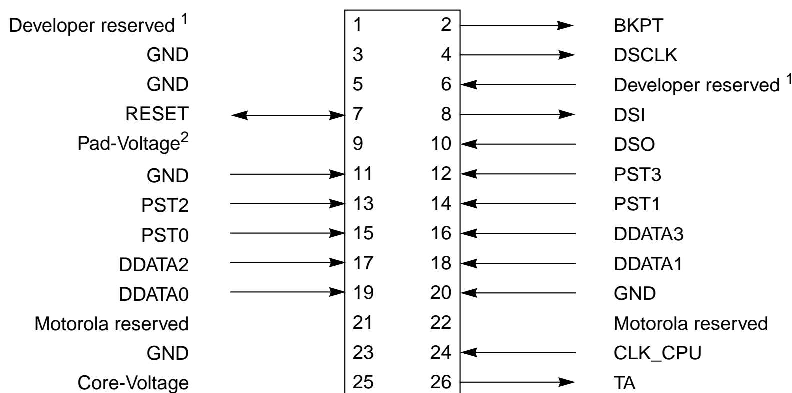


Figure 5-44. Recommended BDM Connector

5.8 Processor Status, DDATA Definition

This section specifies the ColdFire processor and debug module's generation of the processor status (PST) and debug data (DDATA) output on an instruction basis. In general, the PST/DDATA output for an instruction is defined as follows:

PST = 0x1, {PST = [0x89B], DDATA= operand}

where the {...} definition is optional operand information defined by the setting of the CSR.

The CSR provides capabilities to display operands based on reference type (read, write, or both). Additionally, for certain change-of-flow branch instructions, another CSR field provides the capability to display {0x2, 0x3, 0x4} bytes of the target instruction address. For both situations, an optional PST value {0x8, 0x9, 0xB} provides the marker identifying the size and presence of valid data on the DDATA output.

5.8.1 User Instruction Set

Table 5-22 shows the PST/DDATA specification for user-mode instructions. Rn represents any {Dn, An} register. In this definition, the 'y' suffix generally denotes the source and 'x' denotes the destination operand. For a given instruction, the optional operand data is displayed only for those effective addresses referencing memory. The 'DD' nomenclature refers to the DDATA outputs.

Table 5-22. PST/DDATA Specification for User-Mode Instructions

Instruction	Operand Syntax	PST/DDATA
add.l	<ea>y,Rx	PST = 0x1, {PST = 0xB, DD = source operand}
add.l	Dy,<ea>x	PST = 0x1, {PST = 0xB, DD = source}, {PST = 0xB, DD = destination}
addi.l	#imm,Dx	PST = 0x1
addq.l	#imm,<ea>x	PST = 0x1, {PST = 0xB, DD = source}, {PST = 0xB, DD = destination}
addx.l	Dy,Dx	PST = 0x1
and.l	<ea>y,Dx	PST = 0x1, {PST = 0xB, DD = source operand}
and.l	Dy,<ea>x	PST = 0x1, {PST = 0xB, DD = source}, {PST = 0xB, DD = destination}
andi.l	#imm,Dx	PST = 0x1
asl.l	{Dy,#imm},Dx	PST = 0x1
asr.l	{Dy,#imm},Dx	PST = 0x1
bcc.{b,w}		if taken, then PST = 0x5, else PST = 0x1
bchg	#imm,<ea>x	PST = 0x1, {PST = 0x8, DD = source}, {PST = 0x8, DD = destination}
bchg	Dy,<ea>x	PST = 0x1, {PST = 0x8, DD = source}, {PST = 0x8, DD = destination}
bclr	#imm,<ea>x	PST = 0x1, {PST = 0x8, DD = source}, {PST = 0x8, DD = destination}
bclr	Dy,<ea>x	PST = 0x1, {PST = 0x8, DD = source}, {PST = 0x8, DD = destination}
bra.{b,w}		PST = 0x5
bset	#imm,<ea>x	PST = 0x1, {PST = 0x8, DD = source}, {PST = 0x8, DD = destination}
bset	Dy,<ea>x	PST = 0x1, {PST = 0x8, DD = source}, {PST = 0x8, DD = destination}
bsr.{b,w}		PST = 0x5, {PST = 0xB, DD = destination operand}
btst	#imm,<ea>x	PST = 0x1, {PST = 0x8, DD = source operand}
btst	Dy,<ea>x	PST = 0x1, {PST = 0x8, DD = source operand}
clr.b	<ea>x	PST = 0x1, {PST = 0x8, DD = destination operand}
clr.l	<ea>x	PST = 0x1, {PST = 0xB, DD = destination operand}
clr.w	<ea>x	PST = 0x1, {PST = 0x9, DD = destination operand}

Table 5-22. PST/DDATA Specification for User-Mode Instructions (Continued)

Instruction	Operand Syntax	PST/DDATA
cmp.l	<ea>y,Rx	PST = 0x1, {PST = 0xB, DD = source operand}
cmpi.l	#imm,Dx	PST = 0x1
divs.l	<ea>y,Dx	PST = 0x1, {PST = 0xB, DD = source operand}
divs.w	<ea>y,Dx	PST = 0x1, {PST = 0x9, DD = source operand}
divu.l	<ea>y,Dx	PST = 0x1, {PST = 0xB, DD = source operand}
divu.w	<ea>y,Dx	PST = 0x1, {PST = 0x9, DD = source operand}
eor.l	Dy,<ea>x	PST = 0x1, {PST = 0xB, DD = source}, {PST = 0xB, DD = destination}
eori.l	#imm,Dx	PST = 0x1
ext.l	Dx	PST = 0x1
ext.w	Dx	PST = 0x1
extb.l	Dx	PST = 0x1
jmp	<ea>x	PST = 0x5, {PST = [0x9AB], DD = target address} [1]
jsr	<ea>x	PST = 0x5, {PST = [0x9AB], DD = target address}, {PST = 0xB , DD = destination operand}[1]
lea	<ea>y,Ax	PST = 0x1
link.w	Ay,#imm	PST = 0x1, {PST = 0xB, DD = destination operand}
lsl.l	{Dy,#imm},Dx	PST = 0x1
lsr.l	{Dy,#imm},Dx	PST = 0x1
mac.l		PST = 0x1
mac.l	Ry,Rx	PST = 0x1
mac.l	Ry,Rx,ea,Rw	PST = 0x1, {PST = 0xB, DD = source operand}
mac.w		PST = 0x1
mac.w	Ry,Rx	PST = 0x1
mac.w	Ry,Rx,ea,Rw	PST = 0x1, {PST = 0xB, DD = source operand}
move.b	<ea>y,<ea>x	PST = 0x1, {PST = 0x8, DD = source}, {PST = 0x8, DD = destination}
move.l	<ea>y,<ea>x	PST = 0x1, {PST = 0xB, DD = source}, {PST = 0xB, DD = destination}
move.l	<ea>y,ACC	PST = 0x1
move.l	<ea>y,MACSR	PST = 0x1
move.l	<ea>y,MASK	PST = 0x1
move.l	ACC,Rx	PST = 0x1
move.l	MACSR,CCR	PST = 0x1
move.l	MACSR,Rx	PST = 0x1
move.l	MASK,Rx	PST = 0x1
move.w	<ea>y,<ea>x	PST = 0x1, {PST = 0x9, DD = source}, {PST = 0x9, DD = destination}
move.w	CCR,Dx	PST = 0x1
move.w	{Dy,#imm},CCR	PST = 0x1

Table 5-22. PST/DDATA Specification for User-Mode Instructions (Continued)

Instruction	Operand Syntax	PST/DDATA
movem.l	#list,<ea>x	PST = 0x1, {PST = 0xB, DD = destination},... [2]
movem.l	<ea>y,#list	PST = 0x1, {PST = 0xB, DD = source},... [2]
moveq	#imm,Dx	PST = 0x1
msac.l	Ry,Rx	PST = 0x1
msac.l	Ry,Rx,ea,Rw	PST = 0x1, {PST = 0xB, DD = source}, {PST = 0xB, DD = destination}
msac.w	Ry,Rx	PST = 0x1
msac.w	Ry,Rx,ea,Rw	PST = 0x1, {PST = 0xB, DD = source}, {PST = 0xB, DD = destination}
muls.l	<ea>y,Dx	PST = 0x1, {PST = 0xB, DD = source operand}
muls.w	<ea>y,Dx	PST = 0x1, {PST = 0x9, DD = source operand}
mulu.l	<ea>y,Dx	PST = 0x1, {PST = 0xB, DD = source operand}
mulu.w	<ea>y,Dx	PST = 0x1, {PST = 0x9, DD = source operand}
neg.l	Dx	PST = 0x1
negx.l	Dx	PST = 0x1
nop		PST = 0x1
not.l	Dx	PST = 0x1
or.l	<ea>y,Dx	PST = 0x1, {PST = 0xB, DD = source operand}
or.l	Dy,<ea>x	PST = 0x1, {PST = 0xB, DD = source}, {PST = 0xB, DD = destination}
ori.l	#imm,Dx	PST = 0x1
pea	<ea>y	PST = 0x1, {PST = 0xB, DD = destination operand}
pulse		PST = 0x4
rems.l	<ea>y,Dx:Dw	PST = 0x1, {PST = 0xB, DD = source operand}
remu.l	<ea>y,Dx:Dw	PST = 0x1, {PST = 0xB, DD = source operand}
rts		PST = 0x1, {PST = 0xB, DD = source operand}, PST = 0x5, {PST = [0x9AB], DD = target address}
scc	Dx	PST = 0x1
sub.l	<ea>y,Rx	PST = 0x1, {PST = 0xB, DD = source operand}
sub.l	Dy,<ea>x	PST = 0x1, {PST = 0xB, DD = source}, {PST = 0xB, DD = destination}
subi.l	#imm,Dx	PST = 0x1
subq.l	#imm,<ea>x	PST = 0x1, {PST = 0xB, DD = source}, {PST = 0xB, DD = destination}
subx.l	Dy,Dx	PST = 0x1
swap	Dx	PST = 0x1
trap	#imm	PST = 0x1 [3]
trapf		PST = 0x1
tst.b	<ea>x	PST = 0x1, {PST = 0x8, DD = source operand}
tst.l	<ea>x	PST = 0x1, {PST = 0xB, DD = source operand}
tst.w	<ea>x	PST = 0x1, {PST = 0x9, DD = source operand}

Table 5-22. PST/DDATA Specification for User-Mode Instructions (Continued)

Instruction	Operand Syntax	PST/DDATA
unlk	Ax	PST = 0x1, {PST = 0xB, DD = destination operand}
wddata.b	<ea>y	PST = 0x4, {PST = 0x8, DD = source operand
wddata.l	<ea>y	PST = 0x4, {PST = 0xB, DD = source operand
wddata.w	<ea>y	PST = 0x4, {PST = 0x9, DD = source operand

[1] For JMP and JSR instructions, the optional target instruction address is displayed only for those effective address fields defining variant addressing modes. This includes the following <ea>x values: (An), (d16,An), (d8,An,Xi), (d8,PC,Xi).

[2] For Move Multiple instructions (MOVEM), the processor automatically generates line-sized transfers if the operand address reaches a 0-modulo-16 boundary and there are four or more registers to be transferred. For these line-sized transfers, the operand data is never captured nor displayed, regardless of the CSR value. The automatic line-sized burst transfers are provided to maximize performance during these sequential memory access operations.

[3] During normal exception processing, the PST output is driven to a 0xC indicating the exception processing state. The exception stack write operands, as well as the vector read and target address of the exception handler may also be displayed.

```
Exception Processing  PST = 0xC, {PST = 0xB, DD = destination},// stack frame
                                 {PST = 0xB, DD = destination},// stack frame
                                 {PST = 0xB, DD = source},// vector read
                      PST = 0x5, {PST = [0x9AB], DD = target}// PC of handler
```

The PST/DDATA specification for the reset exception is shown below:

```
Exception Processing PST = 0xC,
                     PST = 0x5,{PST = [0x9AB], DD = target}//  PC of handler
```

The initial references at address 0 and 4 are never captured nor displayed since these accesses are treated as instruction fetches.

For all types of exception processing, the PST = 0xC value is driven at all times, unless the PST output is needed for one of the optional marker values or for the taken branch indicator (0x5).

5.8.2 Supervisor Instruction Set

The supervisor instruction set has complete access to the user mode instructions plus the opcodes shown below. The PST/DDATA specification for these opcodes is shown in Table 5-23.

Table 5-23. PST/DDATA Specification for Supervisor-Mode Instructions

Instruction	Operand Syntax	PSTDDATA
cpushl		PST = 0x1
halt		PST = 0x1, PST = 0xF
move.w	SR,Dx	PST = 0x1
move.w	{Dy,#imm},SR	PST = 0x1, {PST = 3}

Table 5-23. PST/DDATA Specification for Supervisor-Mode Instructions

Instruction	Operand Syntax	PSTDDATA
movec	Ry,Rc	PST = 0x1
rte		PST = 0x7, {PST = 0xB, DD = source operand}, {PST = 3},{ PST =0xB, DD =source operand}, PST = 0x5, {[PST = 0x9AB], DD = target address}
stop	#imm	PST = 0x1, PST = 0xE
wdebug	<ea>y	PST = 0x1, {PST = 0xB, DD = source, PST = 0xB, DD = source}

The move-to-SR and RTE instructions include an optional PST = 0x3 value, indicating an entry into user mode. Additionally, if the execution of a RTE instruction returns the processor to emulator mode, a multiple-cycle status of 0xD is signaled.

Similar to the exception processing mode, the stopped state (PST = 0xE) and the halted state (PST = 0xF) display this status throughout the entire time the ColdFire processor is in the given mode.