MSC8101 USER'S GUIDE

MSC8101UG
Rev. 3, December 2005

freescale semiconductor

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations not listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GMBH
Technical Information Center
Schatzbogen 7
81829 München, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
+800 2666 8080

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Order Number: MSC8101UG
Rev. 3
12/2005

freescale™
semiconductor

Contents

1 MSC8101 Overview

2 Reset Configuration and Boot

5 Balancing Between the System and Local Buses

6 DMA Channels

7 Interrupts and Interrupt Priorities

8 Host Interface (HDI16)

9 Enhanced Filter Coprocessor (EFCOP)

MSC8101 Overview 1

The Freescale MSC8101 is a versatile, one-chip integration of a high-performance SC140 core, large internal memory (0.5 MB), a communications processor module (CPM), a very flexible system interface unit (SIU), and a 16-channel DMA controller. The SC140 core contains four ALUs and features high performance, low cost, low power, and superscalar architecture. It performs at 1200 DSP MIPS using an internal 300 MHz clock at 1.6 V core voltage. The large internal memory (0.5 MB) minimizes the penalties caused by accessing external program and data memory by accessing memory at full speed using a 128-bit wide program bus and two 64-bit wide data buses. The SIU user-defined memory controller interfaces with almost any memory system and external peripherals. The CPM is based on the PowerQUICC II™ CPM. It supports a wide variety of serial interfaces and protocols including 155-Mbps Asynchronous Transfer Mode (ATM) and 100-Mbps Fast Ethernet. The MSC8101 targets third-generation wireless infrastructure systems as well as wireline multi-channel applications, such as media over packet.

1.1 Target Markets

The MSC8101 target markets include:

- *Wireless infrastructure systems*. The MSC8101 can be used in both 2.5G (EDGE) and 3G (3GPP) systems. In a base station (BTS), the MSC8101 performs functions such as channel coding. In a base station controller and transcoder unit, the MSC8101 performs speech coding and echo cancellation. Many wireless infrastructure applications use a packetized network, so network connectivity is a key attribute. The MSC8101 performs the traditional digital signal processing tasks and network interface tasks, such as linking to the packetized ATM AAL2 network.
- *Media (Voice/Fax/Data) over Packet gateways*. In these systems, media streams (voice, fax, or data) are packetized and transmitted over a packetized network (ATM, Ethernet, IP). The MSC8101 performs digital signal processing tasks such as speech compression, echo cancellation, fax or modem data pump, error correction/data compression (ECDC), or even real-time protocol (RTP). Furthermore, the MSC8101 performs the network interface task (using the CPM), thus removing bottlenecks in these systems.

1.2 Features

The following sections give an overview of MSC8101 features.

1.2.1 High-performance SC140 Core

- Up to 1200 true DSP MIPS/3000 RISC MIPS at 300 MHz and 1.6 V core voltage; DSP MIPS measure multiply-accumulate (MAC) operations and associated MOVEs and pointer updates
- Four 16-bit arithmetic logic units (ALUs), each with a 40-bit parallel barrel shifter
- Two address arithmetic units (AAUs) with integer arithmetic capabilities and unique DSP addressing modes
- 32-bit data and program address space
- Sixteen 40-bit wide data registers
- Eight 32-bit wide address pointer registers, eight 32-bit wide bus address registers, four 32-bit wide offset registers, and four 32-bit wide modifier registers
- Two 32-bit wide stack pointers: user stack pointer and supervisor stack pointer
- Hardware support for fractional and integer data types
- Very rich 16-bit wide orthogonal instruction set
- Up to 6 instructions executed in a single clock cycle
- Variable-Length Execution Set (VLES) execution model, optimized for performance and code density
- Zero overhead hardware DO loops
- Single unified memory space with byte addressability
- Position independent code (PIC) support
- IEEE 1149.1-compatible JTAG port
- Enhanced On-Chip Emulation (EOnCE) module with real-time debugging capability

1.2.2 Internal Memories

- Total of 512 KB (256 K × 16-bit words) unified internal RAM
- 2 KB bootstrap ROM

1.2.3 100 MHz System Bus

- 64/32-bit data and 32-bit address 60x-compatible bus
- Data width selectable at reset: 64-bit mode without the host interface (HDI16) or 32-bit mode, with the HDI16
- Bus supports multiple master designs
- Four-beat burst transfers (eight beat in 32-bit wide mode)
- Port size of 64, 32, 16, and 8 bits wide controlled by internal memory controller
- Support for data parity
- Bus can access internal memory expansion and external peripherals or enable an external host device to access internal resources

1.2.4 Eight-bank Memory Controller

- Glueless interface to SRAM, 100 MHz page mode SDRAM, DRAM, EPROM, FLASH and other user-definable peripherals
- Byte write enables and selectable parity generation
- 32-bit address decodes with programmable bank size
- User-programmable machines (UPMs); general-purpose chip-select machine (GPCM); and a page-mode, pipelined SDRAM machine
- Byte selects for 64-bit bus width and byte selects for 32-bit bus width

1.2.5 System Interface Unit

- Clock synthesizer
- Reset controller
- Two interrupt controllers
- Real-time clock register
- Periodic interrupt timer
- Hardware bus monitor and software watchdog timer

1.2.6 Internal Peripherals

- Enhanced 16-bit parallel host interface (HDI16) supports multiple buses and provides glueless connection to a number of industry-standard microcomputers, microprocessors, and DSPs
- Enhanced filter coprocessor (EFCOP) is a 300 MHz 32 × 32 bit filtering and echo-cancellation coprocessor that runs in parallel to the SC140

1.2.7 DMA Engine

- 16 independent unidirectional channels (each one is either read or write)
- Priority based time multiplexing between channels using 16 internal priority levels between channels
- Support for four external peripherals
- Support for misaligned addresses in source and destination
- Efficient bus usage via bursts and packing/unpacking
- Support for either dual address or single address (flyby) transfers
- Flexible buffer configuration; support for simple buffer, cyclic buffers, single address buffers (I/O devices), multi buffers and chained buffers

1.2.8 Communications Processor Module (CPM)

- Embedded 32-bit RISC controller architecture for flexible communication peripherals
- Interface to the SC140 through dual-port RAM and serial DMA controllers

- Two serial DMA (SDMA) channels for receive and transmit on all serial channels
- Parallel I/O registers with open-drain and interrupt capability
- Three full-duplex fast serial communication controllers (FCCs) support IEEE 802.3 and Fast Ethernet protocols, HDLC up to E3 rates (45 Mbps), and totally transparent operation. Each FCC can be configured to transmit in fully transparent mode and receive in HDLC mode or *vice versa*. FCC1 can also support the ATM (155 Mbps) protocol through the UTOPIA2 interface. Two FCCs have dedicated pins; the third one operates only in TDM mode.
- Two multi-channel controllers (MCCs)
  - Two 128-serial full-duplex data channels (a total of 256 64-Kbps channels); each MCC can be split into four subgroups of 32 channels each
  - Almost any combination of subgroups can be multiplexed to single or multiple TDM interfaces
- Four full-duplex serial communication controllers (SCCs) supporting IEEE 802.3/Ethernet, high-level synchronous data link control, HDLC, local talk, UART, Synchronous UART, BISYNC, and transparent operations. Two SCCs have dedicated pins; the other two can operate only in TDM mode.
- Two full-duplex serial management controllers (SMCs) supporting GCI, UART, and transparent operation
- One serial peripheral interface (SPI)
- One inter-integrated circuit (I^2C) controller (microwire-compatible) with multimaster, master, and slave modes
- Up to four time-division multiplex (TDM) interfaces (one of which can be T3/E3)
  - Support for two groups of TDM channels for a total of four TDMs
  - Support for T1, CEPT, T1/E1,T3/E3, pulse code modulation highway, ISDN basic rate, ISDN primary rate, Freescale interchip digital link (IDL), general circuit interface (GCI), and user-defined TDM serial interfaces
  - Up to 256 entries in SI RAM for each TDM interface
  - Bit or byte resolution
  - Independent transmit and receive routing, frame synchronization
- Eight independent baud-rate generators and 10 input clock pins for supplying clocks to FCC, SCC, and SMC serial channels
- Four independent 16-bit timers that can interconnect as two 32-bit timers

1.2.9 Separate PLLs for SC140 Core, Bus, and CPM

- SC140, CPM, and bus can run at different frequencies for power/performance optimization
- Phase-lock loop (PLL) values at reset are based on configuration pin values

1.2.10 Packaging

- 332-pin 0.8 mm pitch
- 17 × 17 mm flip chip plastic ball grid array (FC-PBGA)

1.2.11 Software Development Tools

- Highly efficient C and C++ compilers, which enable development of DSP algorithms and control-oriented code in a high-level language
- Debug environments, which support non-intrusive real-time tracing and profiling
- Device simulation models integrated into leading system simulation environments enable design and simulation of systems around SC140-based devices

1.2.12 Hardware Development Tools

- MSC8101 application development system (ADS)
- MSC8101 evaluation module (EVM)

1.3 Architecture

The MSC8101 is composed of the following major functional blocks:

- SC140 core
- SRAM block
- System interface unit (SIU)
- DMA controller
- Host Interface (HDI16)
- Enhanced filter coprocessor (EFCOP)
- Communications processor module (CPM)

Figure 1-1 shows a functional block diagram of the MSC8101.

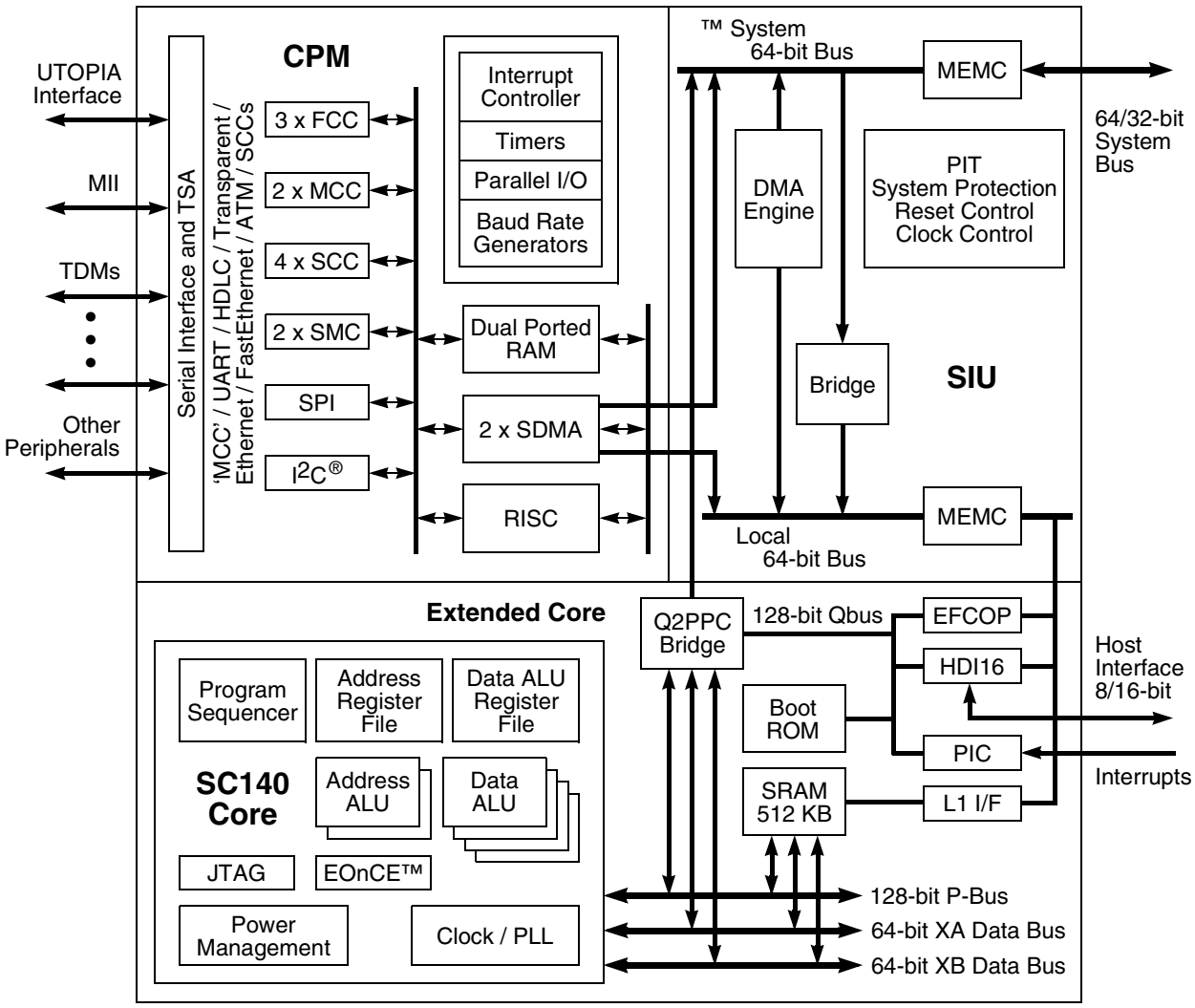


Figure 1-1. MSC8101 Block Diagram

1.3.1 SC140 Core

The 16-bit SC140 core packs four data arithmetic-logic execution units (ALUs), each consisting of a MAC unit, a logic unit, and a bit field unit (BFU), which also serves as a barrel shifter. This number of MAC units delivers very high performance in such essential DSP tasks as finite impulse response (FIR) and infinite impulse response (IIR) filters and fast Fourier transforms (FFTs). In addition to the four data execution units, the core contains two address arithmetic units (AAUs), one bit mask unit (BMU), and one branch unit. Overall, the SC140 can issue and execute up to six instructions per clock—for example, four independent arithmetic instructions and two pointer-related instructions (such as moves or other operations on addresses).

At its initial clock speed of 300 MHz, the SC140 can therefore execute 1200 true DSP MIPS—1.2 billion MAC operations per second, together with associated data movement functions and pointer updates. Note that one such DSP MIPS is the equivalent of several RISC

MIPS, the performance measure used by some other DSPs. For purposes of comparison, the SC140 can be said to perform 3000 RISC MIPS—ten RISC operations per cycle at 300 MHz.

The SC140 core can sustain this high level of performance over time because its four data execution units can operate simultaneously in any combination. For example, the SC140 could execute four MAC operations in a single clock, or one MAC, two arithmetic/logical operations, and one bit field operation. All four data ALUs are identical. This permits great flexibility in the assignment and execution of instructions, increasing the likelihood that four execution units can be kept busy on any given cycle and enabling programs to take better advantage of the core's parallel architecture. For details on the SC140, consult the *SC140 Core Reference Manual*.

1.3.2 Internal SRAM

The 512 KB of SRAM is arranged as a 256K × 16 bit unified memory. The internal memory provides zero-wait-state access to as many as 256 bits (128 program bits and 128 data bits) per 300 MHz cycle. It also provides access of 64 bit per clock cycle on the local bus. At the same time, the DMA engine can perform its own 64-bit wide access. The SRAM has sufficient storage capacity to hold all the program code and data the MSC8101 needs for many target applications, thus eliminating the cost, space, and performance penalties of external memory. When memory requirements exceed the internal storage capacity, the MSC8101 can address up to 4 GB of external memory via the bus interface.

1.3.3 System Interface Unit (SIU)

The SIU consists of the following:

- A 60x-compatible parallel system bus configurable at reset to either 64-bit or 32-bit data width. Port sizes can be 64, 32, 16, and 8 bits wide. The MSC8101 internal arbiter can support the external bus. External bus-request pins allows external masters to acquire the bus. The MSC8101 internal arbiter arbitrates between internal masters (SC140, CPM, DMA, and three external masters). You can disable this arbiter and use an external arbiter if necessary.
- An internal local 64-bit data, 32-bit address bus. The local bus is synchronous to the system bus and runs at the same frequency.
- A memory controller supporting eight external memory banks. The memory controller, which is based on the MPC8260 memory controller, supports UPMs as well as an SDRAM machine with page mode and address data pipeline.
- A bus monitor that prevents system bus lock-ups, a real-time clock, a periodic interrupt timer, and other system functions useful in embedded applications.

1.3.4 DMA Controller

The 16 independent unidirectional channels of the DMA controller move data among internal and external memories and internal and external peripherals without the involvement of the SC140 core. Transfers occur on the internal (local) bus, the external bus, or between the two buses. The DMA engine handles full 64-bit transfers and bursting to take maximum advantage of available bus bandwidth. Non-aligned transfers are also handled. Dual address transfers require two DMA channels.

Bus usage is further improved by the DMA unit's first-in/first-out buffers (FIFOs), which store data temporarily between read and write operations. For example, if the DMA is transferring data from an external memory to internal memory, it need not wait until the local bus is free to begin reading the data. Instead, it fetches the data over the system bus, storing it temporarily in a FIFO until the local bus is available, and then it completes the transfer by writing the data to internal memory. The DMA also runs in a "flyby" mode in which data is transferred directly from a source to a destination in a single cycle (from an internal peripheral to internal memory or from an external peripheral to external memory). In flyby mode, the source and destination are of equal width and aligned. The DMA controller supports complex addressing such as circular buffers, dual buffers, and multiple buffers. Buffer type is configured in the DMA parameter RAM.

1.3.5 Host Interface (HDI16)

In addition to its bus interface, the MSC8101 features an enhanced HDI16, supporting a variety of standard buses and providing glueless connection to industry-standard microcontrollers, microprocessors and DSPs. The host interface can be used concurrently with the interface operating in 32-bit mode. The combination of bus interfaces provides a great deal of system design flexibility. For example, in systems where a large amount of data is passed between a host processor (for example, a or PowerQUICC II) and a bank of MSC8101s, the DSPs can communicate with the host via the bus interface in 64-bit mode. In systems with a smaller amount of host-DSP traffic, the MSC8101s could communicate via their 16-bit HDI16 interfaces, while simultaneously connecting to private or shared memory via 32-bit buses.

1.3.6 Enhanced Filter Coprocessor (EFCOP)

The EFCOP performs filtering operations vital to such DSP tasks as echo cancellation. These filtering operations include both adaptive and non-adaptive FIR and IIR filtering with 32-bit precision (the EFCOP contains a 32-bit × 32-bit multiply unit and 72-bit accumulator). The EFCOP's hardwired circuitry performs one FIR filter tap per clock cycle, drawing very little power. The EFCOP can also update coefficients in an adaptive filter. The coprocessor operates independently and in parallel with the core processor. This allows the MSC8101 to perform such operations as echo cancellation in parallel with such operations as voice compression, boosting overall performance in applications such as Internet telephony. At a 70-percent usage rate—a

typical average for EFCOP utilization in DSP applications—the coprocessor provides 210 MIPS above the SC140 core's 1,200-MIPS performance.

1.3.7 Communications Processor Module (CPM)

The CPM allows the MSC8101 to excel in a variety of applications mainly targeted for the networking and the telecommunication markets. The CPM, based on the PowerQUICC II, is a supersede of the MPC860 PowerQUICC CPM. It is enhanced with RISC performance and additional hardware and microcode routines that handle high bit rate protocols, such as ATM (up to 155-Mbps full-duplex) and Fast Ethernet (up to 100-Mbps full-duplex). The CPM consists of the following functional blocks:

- An embedded 32-bit RISC controller, also called the communications processor (CP), that handles the lower-layer tasks and serial DMA control activities, freeing the SC140 to handle higher-layer activities as well as DSP tasks. The RISC controller has an instruction set optimized for communication, but it can also handle general-purpose applications, relieving the SC140 of small, often repeated tasks.
- Two serial DMA (SDMA) controllers that can perform simultaneous transfers, optimized for burst transfers to the bus and to the local bus.
- Three full-duplex fast serial communication controllers (FCCs) support IEEE 802.3 and Fast Ethernet protocols, HDLC up to E3 rates (45Mbps), and totally transparent operation. Each FCC can be configured to transmit in transparent mode and receive in HDLC mode or *vice versa*. FCC1 can also support the ATM (155 Mbps) protocol through the UTOPIA2 interface. Two FCCs have dedicated pins; the third one operates only in TDM mode.
- Two multi-channel controllers (MCCs) that can handle an aggregate of 256 × 64-Kbps HDLC or transparent channels, multiplexed on up to four TDM interfaces. The MCC also supports super channels of rates higher than 64 Kbps and subchanneling of the 64 Kbps channels.
- Four full-duplex serial communication controllers (SCCs) supporting IEEE802.3/Ethernet, synchronous data link control, high-level data link control protocol (HDLC), local talk, UART, Synchronous UART, BISYNC, and transparent mode. Two SCCs have dedicated pins; the other two can operate only in TDM mode.
- Two full-duplex SMCs for general circuit interface (GCI), UART, and transparent operation.
- SPI and I^2C bus controllers.
- Two serial interfaces (SIs) with time-slot assigners (TSAs) that support multiplexing of data from any of the three FCCs, two MCCs, four SCCs, and two SMCs.
- The time-slot assigner (TSA) supports data multiplexing of data from any of the four SCCs, three FCCS, and two SMCs.

The CPM includes many other functions; the preceding list is only an overview of major features.

1.3.7.1 Serial Protocol

Table 1-1 summarizes the available protocols for each serial port.

Table 1-1. MSC8101 Serial Protocols

Available Protocol	FCC	SCC	MCC	SMC
ATM (UTOPIA)	+			
ATM (serial)	+			
100BaseT	+			
10BaseT	+	+		
HDLC	+	+		
HDLC_BUS		+		
TRANSPARENT	+	+		+
UART		+		+
DPLL		+		
Multiple channel			+	

1.3.7.2 CPM Configurations

The CPM comprises many different functional blocks and offers flexibility in configuring the device for specific applications. The functions described in the preceding sections are all available in the device, but not all of them can be used at the same time. This does not mean that the device is not fully activated in any given implementation. The CPM architecture uses common hardware resources for many different protocols and applications. Two physical factors limit the functionality in any given system: pinout and performance. To fit into a small pin count package, some pins have multiple functions. In some cases, choosing a function may preclude the use of another function. The CPM handles an aggregate rate of 750 Mbps on the communication channels at 150 MHz CPM clock and 100 MHz system bus clock. Performance depends on the following factors:

- Serial rate versus CPM clock frequency for adequate sampling on serial channels
- Serial rate and protocol versus CPM clock frequency for CPM RISC protocol handling
- Serial rate and protocol versus bus bandwidth
- Serial rate and protocol versus core clock for adequate protocol handling

Table 1-2 describes a few options to configure the fast communication channels on the MSC8101. The frequency specified is the minimum CPM frequency necessary to run the mentioned protocols concurrently at full-duplex.

Table 1-2. MSC8101 Serial Performance Example

FCC 1	FCC 2	MCC	CPM Clock	System Bus Clock
155-Mbps ATM	100 BaseT		150 MHz	100 MHz
100-BaseT	100 BaseT		150 MHz	100 MHz
155-Mbps ATM		128 64-Kbps channels	150 MHz	100 MHz
100-BaseT	100 BaseT	128 64-Kbps channels	150 MHz	100 MHz
155-Mbps ATM		256 64-Kbps channels	150 MHz	100 MHz
100-BaseT		256 64-Kbps channels	150 MHz	100 MHz
45-Mbps HDLC		256 64-Kbps	150 MHz	100 MHz
45-hbps HDLC	100 BaseT	256 64-Kbps	150 MHz	100 MHz
100 BaseT		16 576-Kbps	150 MHz	100 MHz

The FCCs run not only in high-speed mode but also in slower modes, such as HDLC or 10BaseT. The CPM RISC architecture has the advantage of using common hardware resources for all FCCs.

1.3.7.3 Buffer Descriptors

If you are programming the CPM serial controllers, you need to know how the serial controllers use buffer descriptors to define buffer allocation. A buffer descriptor (BD) contains the essential information about each buffer in memory. Each buffer is referenced by a BD that can reside anywhere in dual-port RAM. These BDs are shared among all serial controllers, including:

- SCCs in UART, HDLC, BISYNC, Transparent, Ethernet, and AppleTalk modes
- FCCs in HDLC, Fast Ethernet, and Transparent modes
- SMCs in UART, Transparent, and GCI modes
- MCCs in HDLC and Transparent modes
- SPI
- I^2C

Each 64-bit BD has the structure shown in **Figure 1-2**. This structure is common to all communications controllers. A receive buffer descriptor (RxBD) table and a transmit buffer descriptor (TxBD) table are associated with each serial controller. Each table can have multiple BDs.

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0x0	Status and Control															
0x2	Data Length															
0x4	High-Order Buffer Pointer															
0x6	Low-Order Buffer Pointer															

Figure 1-2. Buffer Descriptor Structure

In this discussion, the BD and field values use the following convention:

```
BD.field
```

Table 1-3 shows the possible BD and field naming conventions. Bit names in `RxBD.bd_cstat` and `TxBD.bd_cstat` use the following convention:

```
BD.bd_cstat.bit
```

Table 1-3. Buffer Descriptor Naming Conventions

BD	Field	Example
RxBD/TxBD	bd_cstat	TxBD.bd_cstat. R refers to the ready bit in the TxBD's status and control field. Refer to the *MSC8101 Reference Manual* for the protocol's status and control field bit definition.
	bd_length	RxBD.bd_length refers to RxBD's data length field.
	bd_addr	RxBD.bd_addr refers to RxBD's buffer pointer field.

The structural elements of a buffer descriptor are defined as follows:

- *Status and control*. The 16-bit value at `offset+0x0`, which contains status and control bits that control and report status information on the data transfer. The CPM updates the status bits after the buffer is sent or received. Only this field differs for each protocol. Refer to the *MSC8101 Reference Manual* for each protocol's `RxBD.bd_cstat` and `TxBD.bd_cstat` bit description.
- *Data length*. The 16-bit value at `offset+0x2`, which contains the number of bytes sent or received.
- *RxBD data length*. The number of bytes the CP writes into the RxBD buffer once the BD closes. The communications processor (CP) updates this field after the received data is placed into the buffer and the buffer is closed. You do not need to initialize this field. In frame-based protocols, except for the SCC transparent mode, `RxBD.bd_length` contains the total frame length including CRC bytes. If a received frame's length, including CRC, is an exact multiple of the parameter RAM maximum receive buffer length MRBLR, the last buffer holds no actual data but the associated BD contains the total frame length.

- *TxBD data length*. The number of data bytes the controller needs to transmit from its buffer. The CP never modifies this field, which is initialized by the user.
- *Buffer pointer*. The 32-bit data at `offset+0x4`, which points to the beginning of the buffer in internal or external memory.
- *RxBD buffer pointer*. The buffer pointer value must be a multiple of four to be word-aligned.
- *TxBD buffer pointer*. The buffer pointer value can be even or odd.

1.3.7.4 Parameter RAM

In the dual-port RAM memory map (see **Table 1-4**), Banks 9–10 (`IMM+$8000:IMM+$8FFF`) store parameters associated with the SCCs, FCCs, MCCs, SMCs, SPI, and I^2C controllers. The parameter RAM contains parameters for operating these channels. The exact definition of the parameter RAM, which differs for each protocol, is provided in the *MSC8101 Reference Manual*. **Table 1-5** shows the MSC8101 parameter RAM structure. The parameters for the SCCs, FCCs, and MCCs are stored in the parameter RAM. The parameters for the SMCs, SPI, and I^2C are stored in locations to which the user-programmable values in the parameter RAM point. For example, `IMM+$8200` contains the SCC3 parameters. However, `IMM+$8AFC` contains a pointer to the I^2C parameters, which can be placed in Banks 1–8 in the dual-port RAM.

Table 1-4. Dual-Port RAM Memory Map

Memory Location	Bank	Content	Size
IMM+0x0000	Bank 1	BD/Data/Code	2 KB
IMM+0x0800	Bank 2	BD/Data/Code	2 KB
IMM+0x1000	Bank 3	BD/Data/Code	2 KB
IMM+0x1800	Bank 4	BD/Data/Code	2 KB
IMM+0x2000	Bank 5	BD/Data/Code	2 KB
IMM+0x2800	Bank 6	BD/Data/Code	2 KB
IMM+0x3000	Bank 7	BD/Data/Code	2 KB
IMM+0x3800	Bank 8	BD/Data/Code	2 KB
IMM+0x4000	Reserved		16 KB
IMM+0x8000	Bank 9	Parameter RAM	2 KB
IMM+0x8800	Bank 10	Parameter RAM	2 KB
IMM+0x9000	Reserved		8 KB
IMM+0xB000	Bank 11	FCC Data	2 KB
IMM+0xB800	Bank 12	FCC Data	2 KB

Table 1-5. MSC8101 Parameter RAM Structure

Offset from IMM	Peripheral	Size (Bytes)	Offset from IMM	Peripheral	Size (Bytes)
0x8000	SCC1	256	0x8900	Reserved	224
0x8100	SCC2	256	0x89FC	SPI_BASE	2
0x8200	SCC3	256	0x89FE	Reserved	2
0x8300	SCC4	256	0x8A00	Reserved	224
0x8400	FCC1	256	0x8AE0	RISC Timers	16
0x8500	FCC2	256	0x8AF0	REV_NUM	2
0x8600	FCC3	256	0x8AF2	Reserved	2
0x8700	MCC1	128	0x8AF4	Reserved	4
0x8780	Reserved	124	0x8AF8	RAND	4
0x87FC	SMC1_BASE	2	0x8AFC	I^2C_BASE	2
0x87FE	Reserved	2	0x8AFE	Reserved	2
0x8800	MCC2	128	0x8B00	Reserved	1280
0x8880	Reserved	124			
0x88FC	SMC2_BASE	2			
0x88FE	Reserved	2			

Table 1-6 shows the parameter RAM for all SCC protocols. You must initialize entries with boldfaced names before the SCC can be enabled. Refer to the *MSC8101 Reference Manual* for the protocol-specific parameters.

Table 1-6. SCC Parameter RAM

Offset from SCC Base[1]	Name	Width	Description
0x00	**RBASE**	16-bits	**RxBD/TxBD table base address**. Offset from the beginning of dual-port RAM. The BD tables can be placed in any unused portion of Banks 1–8. The CP starts BD processing at the top of the table. These values must be initialized before the corresponding channels are enabled. RBASE and TBASE values should be multiples of 8.
0x02	**TBASE**	16-bits	
0x04	**RFCR**	8-bits	**Rx/Tx function code**. Contains the transaction specification associated with SDMA channel accesses to external memory.
0x05	**TFCR**	8-bits	
0x06	**MRBLR**	16-bits	**Maximum receive buffer length**. Defines the maximum number of bytes the MSC8101 writes to a receive buffer before it goes to the next buffer. The MSC8101 can write fewer bytes than MRBLR if an error or an end-of-frame occurs. It never writes more bytes than the MRBLR value. MRBLR should be changed only while the receiver is disabled.
0x08	RSTATE	32-bits	**Rx internal state**. For CP use only.

Table 1-6. SCC Parameter RAM (Continued)

Offset from SCC Base[1]	Name	Width	Description
0x0C	—	32-bits	**Rx internal buffer pointer**. Updated by the SDMA channels to show the next address in the buffer to be accessed.
0x10	RBPTR	16-bits	**Current RxBD pointer**. Points to the current BD being processed or to the next BD the receiver uses when it is idling. After reset or when the end of the BD table is reached, the CP initializes RBPTR to the value in RBASE.
0x12	—	16-bits	**Rx internal byte count**. Down-count value initialized with MRBLR and decremented with each byte written by the supporting SDMA channel.
0x14	—	32-bits	**Rx temp**. For CP use only.
0x18	TSTATE	32-bits	**Tx internal state**. For CP use only.
0x1C	—	32-bits	**Tx internal buffer pointer**. Updated by the SDMA channels to show the next address in the buffer to be accessed.
0x20	TBPTR	16-bits	**Current TxBD pointer**.
0x22	—	16-bits	**Tx internal byte count**. Down-count value initialized with TxBD.bd_length and decremented with each byte read by the supporting SDMA channel.
0x24	—	32-bits	**Tx temp**. For CP use only.
0x28	RCRC	32-bits	**Temp receive/transmit cyclic redundancy check (CRC)**. Does not need to be accessed for normal operation but may be helpful for debugging.
0x2C	TCRC	32-bits	
0x30	—		**Protocol-specific area**.

Notes: 1. SCC base is IMM+0x8000. Refer to **Table 1-5**

1.3.7.5 BD and Buffer Memory Structure

The BDs of all protocols can point to data buffers that are located in the internal dual-port RAM. Banks 1–8 (`IMM+0x0:IMM+0x4FFF`) are available for storing BDs and their buffers. However, if the data buffers are large, they can be located in external memory. In the SCC2 example shown in **Figure 1-3**, the SCC2 RxBD and the TxBD parameters are located in the dual-port RAM, and the buffers are located in external memory. `RxBD.bd_addr` contains a pointer to the receive buffer in external memory, and `TxBD.bd_addr` contains a pointer to the transmit buffer in external memory. The 256-byte SCC2 parameter RAM is located at `IMM+0x8100`.

In the SPI example shown in **Figure 1-4**, the SPI RxBD and TxBD tables are located in the dual-port RAM, and the buffers are located in external memory. `RxBD.bd_addr` contains a pointer to the receive buffer in external memory, and `TxBD.bd_addr` contains a pointer to the transmit buffer in external memory. The two-byte SPI_BASE parameter RAM is located at `IMM+0x89FC`, which contains a pointer to the SPI parameter table. The SPI parameter table can be placed at any 64-byte aligned address in the dual-port RAM's general-purpose area or in Banks 1–8.

1.3.7.6 RxBD Processing Example

Figure 1-5 shows how the RxBD is processed in SCC UART mode. This example assumes that the maximum receive buffer length (MRBLR) is 80 bytes. The MRBLR is the number of bytes the MSC8101 writes to a receive buffer before it moves to the next buffer. However, the MSC8101 can write fewer bytes than the MRBLR value if an error or end-of-frame (for frame-based protocols) occurs. It never writes more bytes than the MRBLR value, so the receive buffers cannot be smaller than the MRBLR.

When data arrives, the CP moves the data to the buffer to which the first RxBD in the table is pointing. The CP continues to move data until the buffer is full or an error occurs. Then the buffer is closed. Subsequent data uses the next BD.

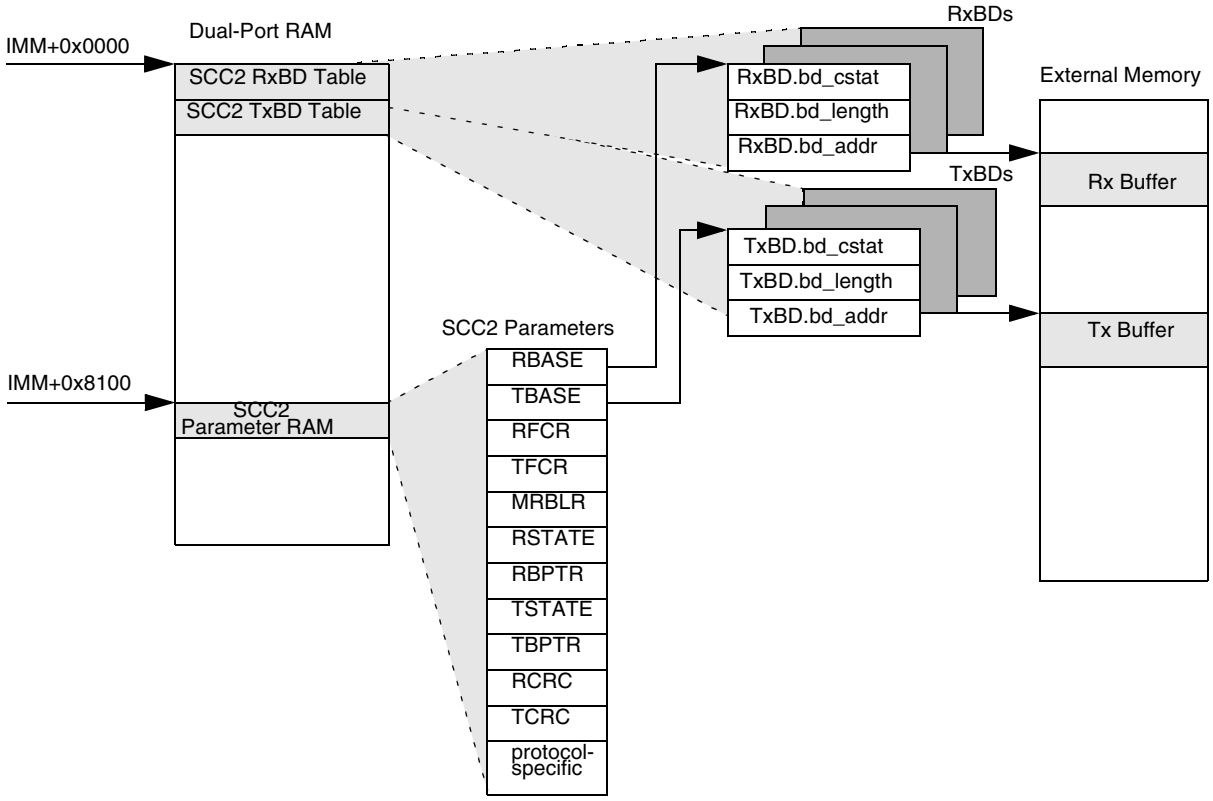


Figure 1-3. Example SCC2 BD and Buffer Memory Structure

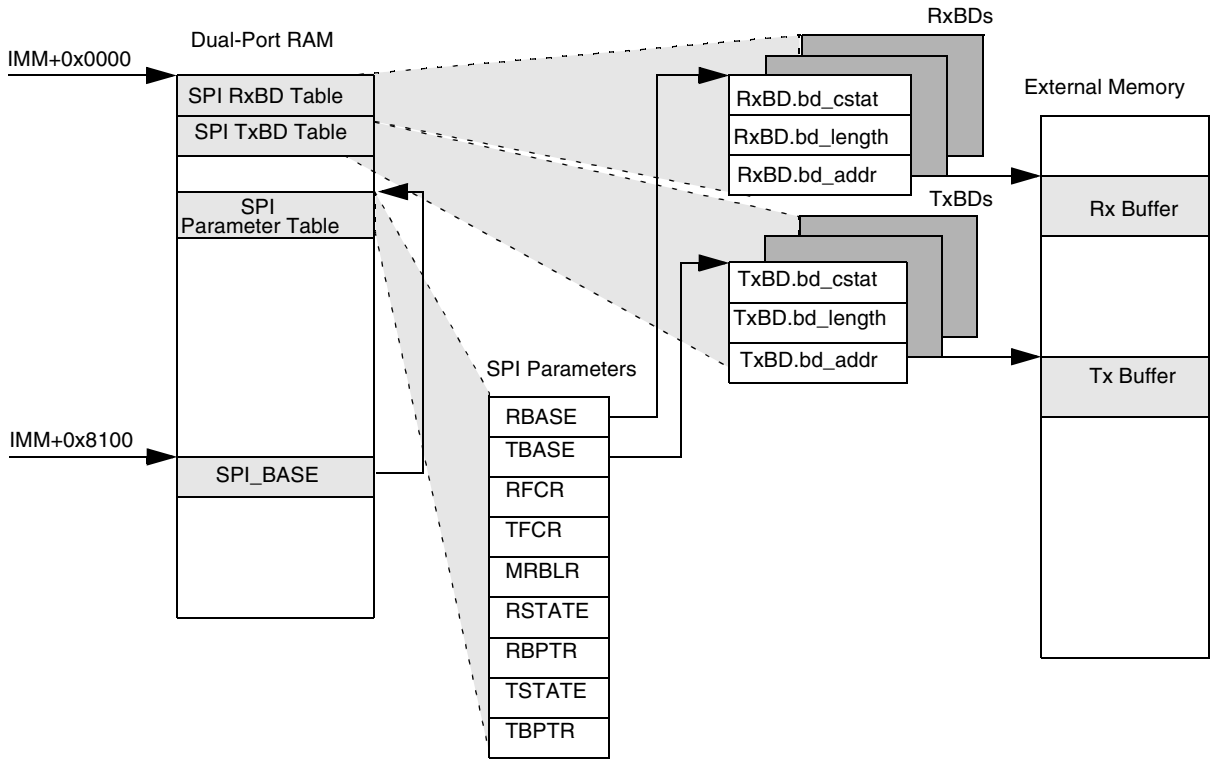


Figure 1-4. Example SPI BD and Buffer Memory Structure

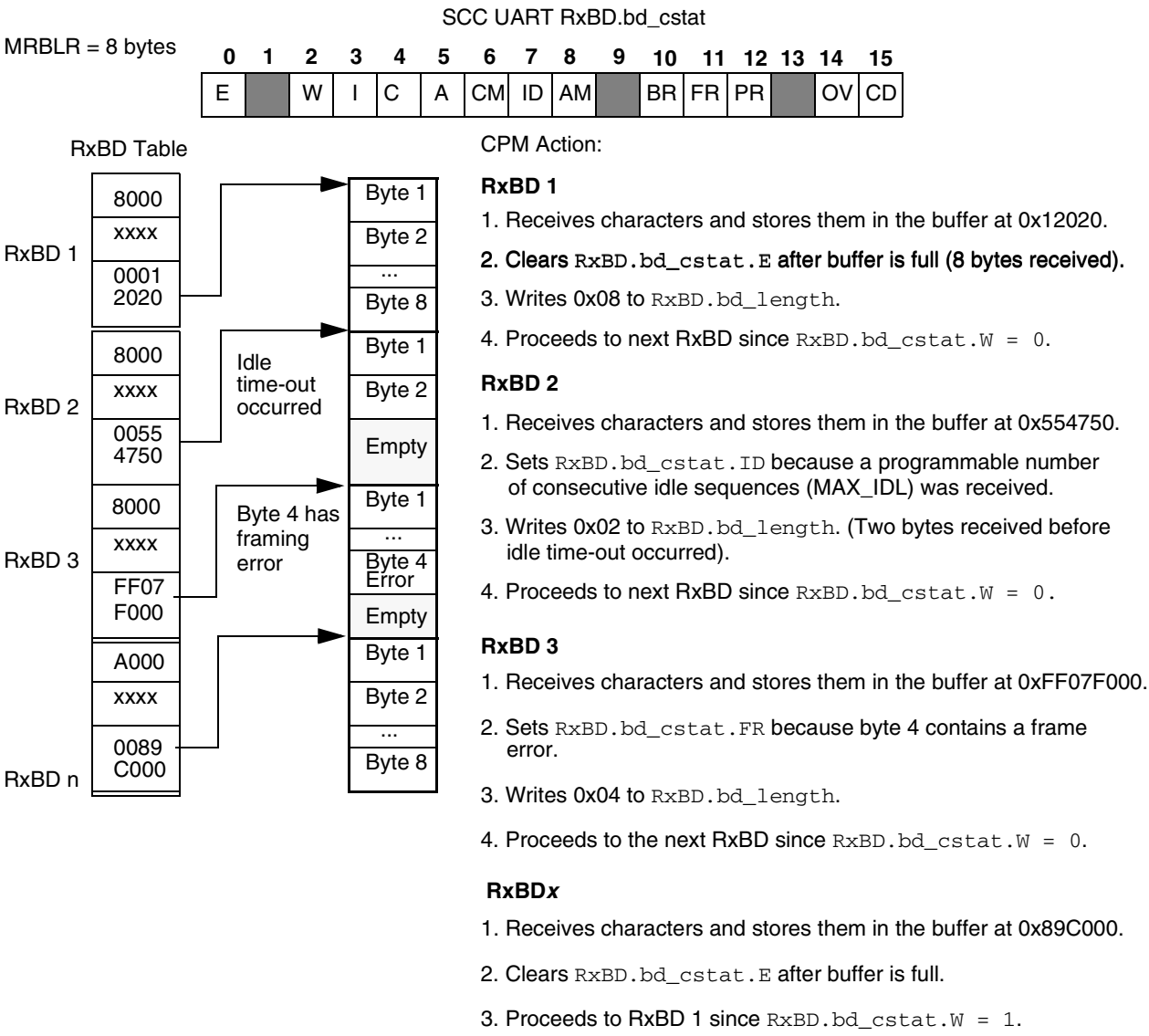


Figure 1-5. Example SCC UART RxBD Processing

If `RxBD.bd_cstat.E` is cleared, the current buffer is not empty, and it reports a busy error. The CP does not move from the current BD until the SC140 core sets `RxBD.bd_cstat.E` to indicate that the buffer is empty. After using a descriptor, the CP clears `RxBD.bd_cstat.E` and does not reuse a BD until the core processes it. However, in continuous mode when `RxBD.bd_cstat.CM` is set, `RxBD.bd_cstat.E` remains set so the buffer can be overwritten when the CP accesses this BD again. When the CP discovers a descriptor's `RxBD.bd_cstat.W` (wrap) is set, which indicates that it is the last BD in the circular BD table, it returns to the beginning of the table when it is time to move to the next BD.

When the UART receives idle characters (all ones), the channel begins counting consecutive idle characters received. If the maximum idle characters (MAX_IDL) is reached, `RxBD.bd_cstat.ID` is set, the buffer is closed, and an interrupt is generated if not masked. When the UART receives no stop bit, it reports framing errors. The channel writes the received

character to the buffer, closes it, sets RxBD.bd_cstat.FR, generates an interrupt if not masked, and increments the received characters with the framing error counter (FRMEC). A new receive buffer receives subsequent data.

1.3.7.7 TxBD Processing Example

Figure 1-6 shows how the TxBD is processed in SCC UART mode. When the CP detects that the TxBD.bd_cstat.R (ready) is set, it starts transmitting the buffer. After the buffer is transmitted, the CP waits for the next descriptor's TxBD.bd_cstat.R to be set before proceeding. When the CP detects that a descriptor's TxBD.bd_cstat.W (wrap) is set, indicating that this BD is last in the BD table, it returns to the start of the BD table after this last BD is processed. The CP clears TxBD.bd_cstat.R (not ready) after using a TxBD, which keeps it from being retransmitted before the SC140 core confirms it. However, some protocols support a continuous mode for which TxBD.bd_cstat.R remains set after the buffer is closed to allow the buffer to be resent next time the CP accesses this BD. Continuous mode is enabled by setting TxBD.bd_cstat.CM.

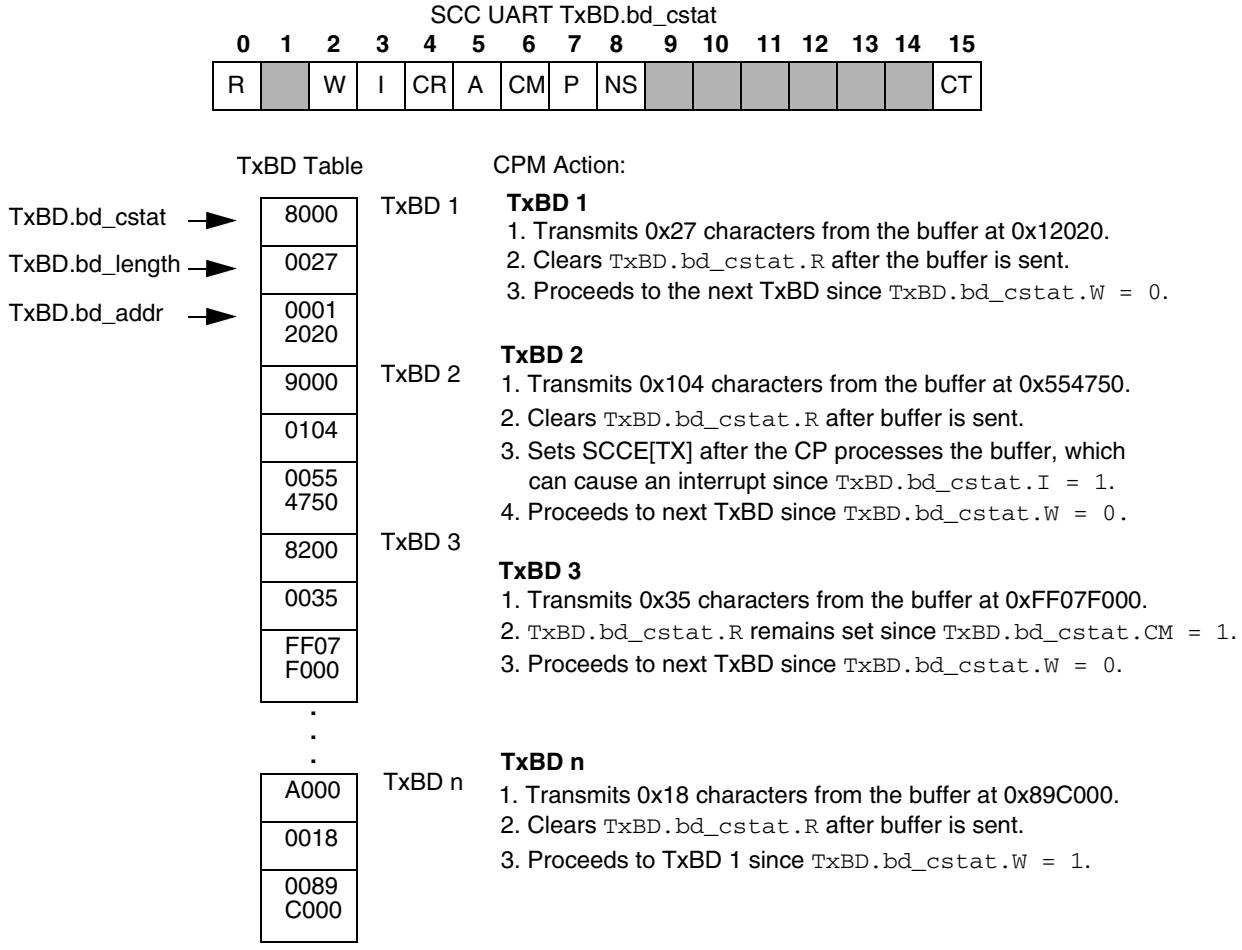


Figure 1-6. Example SCC UART TxBD Processing

1.4 MSC8101 Application Examples

The MSC8101 can be configured to meet many system application needs. Example applications include:

- Media (voice/fax/data) over packet gateway
- 3G infrastructure BTS
- Centralized DSP architecture
- Distributed DSP architecture

In all the examples, the SCCs, SMCs, I^2C, SPI ports can be used for management.

1.4.1 Media (Voice/Fax/Data) Over Packet Gateway (ATM/FR/IP)

Figure 1-7 shows the media (voice/fax/data) over packet gateway (ATM/FR/IP) configuration.

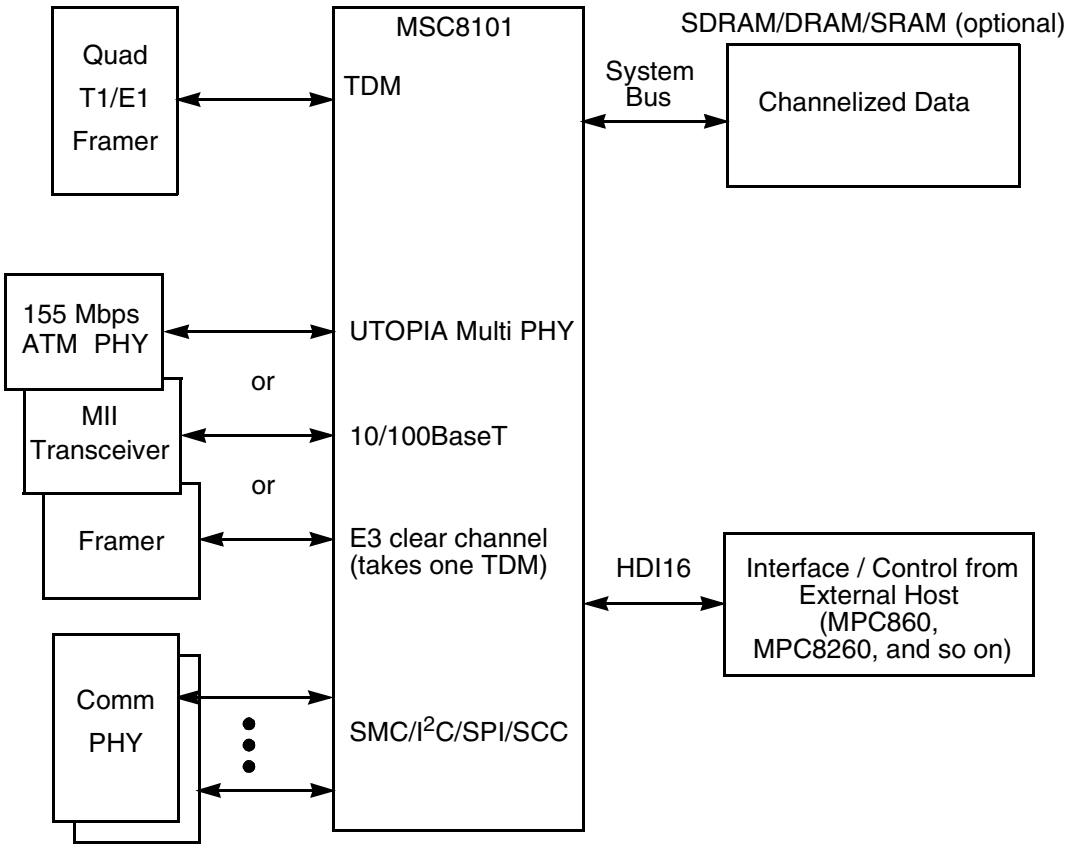


Figure 1-7. Media (Voice/Fax/Data) Over Packet (ATM/FR/IP)

In this application, a single TDM port connects to an external framer. Up to four MSC8101 TDM interfaces can handle an aggregate of 256 channels (one of which can be T3/E3). One TDM interface can support 32–128 channels. The MSC8101 receives and transmits data in transparent or HDLC mode and stores or retrieves the channelized data from memory. The data is stored

either in memory residing internally in the device or externally via the system bus (optional). The main trunk can be configured as 155 Mbps full-duplex ATM using the UTOPIA interface, as 100 Mbps 10/100 Base Fast Ethernet with the MII interface, or as a high-speed serial channel (up to 45 Mbps).

The HDI16 can interface with/control a bank of MSC8101s via a host controller. Through the HDI16, the host controller downloads the code via a bootstrap routine at reset and handles scheduling and overall control of the MSC8101. The MSC8101 memory controller supports many types of memories, including EDO DRAM and page-mode as well as pipeline SDRAM for efficient burst transfers.

1.4.2 3G Infrastructure Cellular BTS

Figure 1-8 shows a 3G infrastructure cellular BTS configuration.

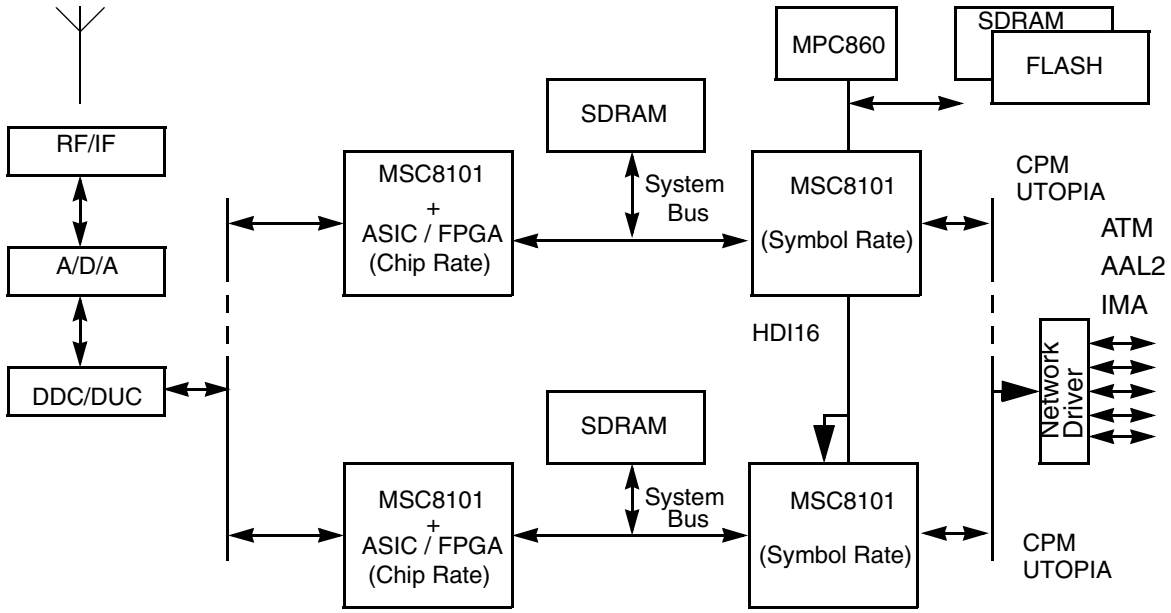


Figure 1-8. 3G Infrastructure Cellular BTS

In this application, the MSC8101 connects by the system bus to SDRAM and an external ASIC/FPGA. These external accesses are controlled by the programmable memory controller. The main trunk in this configuration is configured as 155 Mbps full-duplex ATM UTOPIA Multi-PHY interface.

1.4.3 Centralized DSP Architecture

Figure 1-9 shows a centralized DSP architecture configuration. The protocol processing is in one place. The host—in this example, the MPC8260—terminates the protocol stack, distributes the data payload to the slave devices, and coordinates the resource allocation. The MPC8260 local bus can control a bank of DSPs. Data to and from the DSPs transfer through the parallel bus with

the internal virtual IDMA. The slow communication ports (SCCs, SMCs, I^2C, and SPI) perform management and debug functions. Each MSC8101 is memory-mapped, through its HDI16 port, on the MPC8260 local bus. In addition, each MSC8101 can connect to a private local memory through the 32-bit wide bus.

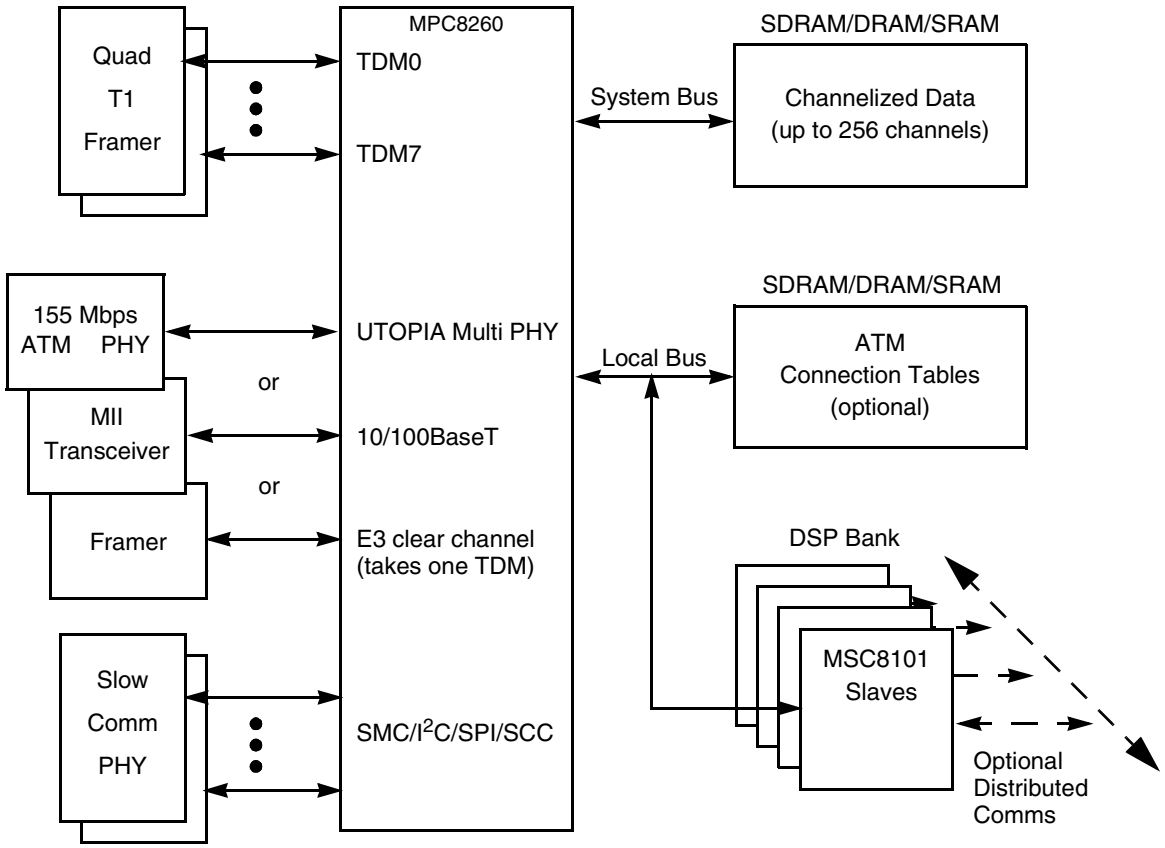


Figure 1-9. Centralized DSP Architecture

1.4.4 Distributed DSP Architecture

Figure 1-10 shows a distributed DSP architecture connected through the HDI16 port. In this example, the protocol processing is distributed, spreading the task of terminating the protocol stack to both the host and slave devices. The host still coordinates the DSP resource allocation. Either an MSC8101 or MPC8260 controls the bank of MSC8101s. If an MPC8260 is the host, then the bus system memory is split from the local bus. Each slave DSP completely manages its own subsystem resources, that is, HDI16 and TDM accesses are managed via the internal DMA controller. Also, each slave has an optional 32-bit bus to connect to either a private or shared memory sub-system. You have the option of whether to connect the host to slaves via either to HDI16 or directly through a shared bus.

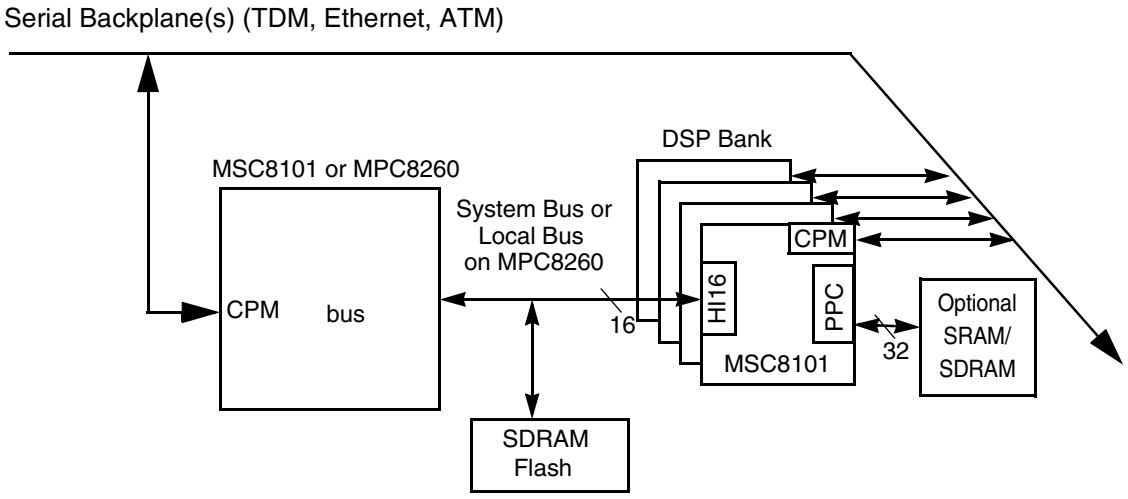


Figure 1-10. Distributed DSP Architecture Connected Through the HDI16 Port

In the configuration depicted in **Figure 1-11**, the bus port can be used in both master and slave mode, eliminating the need for a separate HDI16 connection. The bus operates in either 32- or 64-bit wide mode, and each of the DSPs can access the shared SDRAM, eliminating the need for private memories. The connection to the serial backplane is a combination of TDMs, Ethernet, or ATM. When connected to an ATM backbone, all devices connect to a shared UTOPIA bus. One of the device is the UTOPIA master, and all others (including the PHY device) are UTOPIA slaves.

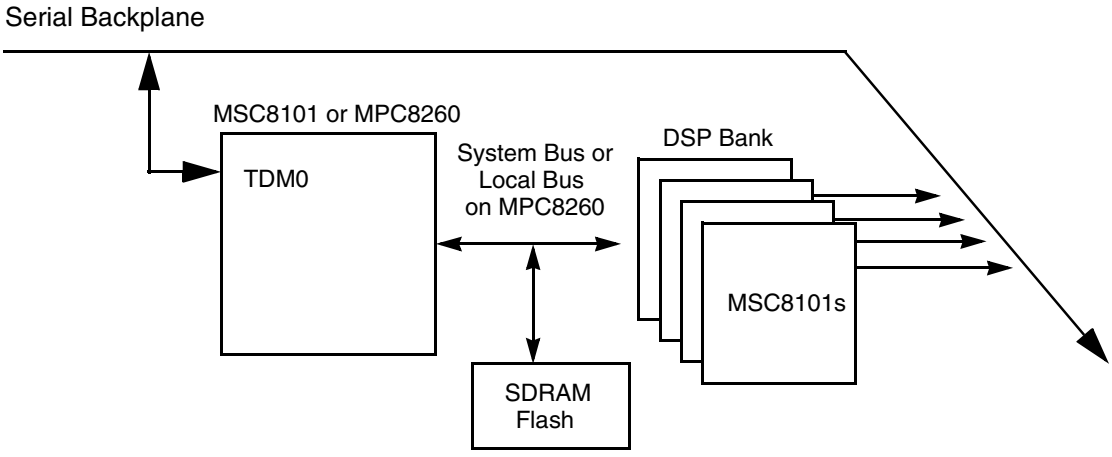


Figure 1-11. Distributed DSP Architecture Connected Through a Shared Bus

1.5 Software Development

Figure 1-12 shows the typical software development flow for the MSC8101.

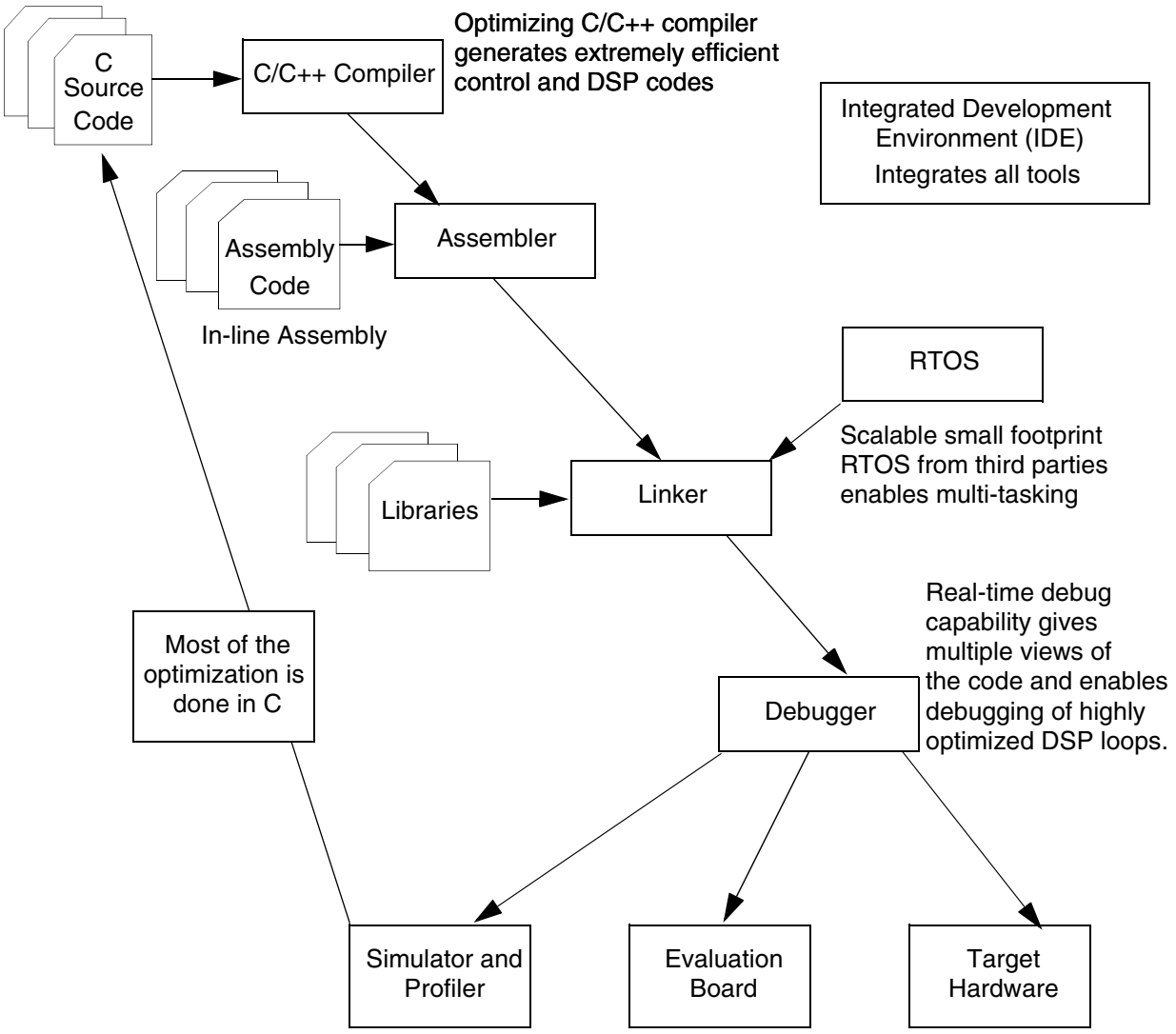


Figure 1-12. Software Development Flow

Software development starts at the C level. The architecture of the SC140 DSP core is very C friendly, so much of the code is written, optimized, and debugged in C. Most code development time is spent optimizing the C code. The software development flow also supports code optimization in assembly, either by linking assembly modules or by in-line assembly embedded within the C code. The linker links C files, assembly files, application software libraries, and real-time operating system (RTOS) files. The code generated can then be debugged either on the software simulator or on the hardware. The debugger is window-based and provides multiple views of the code. You can use it to debug C source code, assembly code, or mixed code. It provides effective debug capabilities even within tight DSP loops that have undergone significant optimization by the compiler. The MSC8101 hardware provides non-intrusive real-time tracing for use by the profiler. Another key element of the tool chain, the profiler provides detailed information on deficiencies and hot spots in the code. The programmer can focus optimization

efforts on these critical code sections, either by modifying the C code or by optimizing code segments in assembly.

A variety of optimization techniques can be implemented either in C code or assembly code. Optimization focuses mainly on exposing the parallelism that is embedded in the algorithm so that the compiler can place as many instructions as possible into the same execution set. This is mainly true for the DSP routines in which the parallelism is not always visible to the compiler. It is less true for the control code routines in which the compiler can expose the potential parallelism (where it exists) by itself. Example optimization techniques are split summation, multisample, loop unrolling, and loop merging. These techniques are described in detail in an application note entitled *Introduction to the StarCore Tools: An Approach in Nine Exercises* (AN2009/D), which is available with accompanying code at the Website listed on the back cover of this manual.

All tools are integrated under a comprehensive integrated development environment (IDE), which increases developer productivity by tying together code generation and system debug tools with project management and editing tools.

Freescale also provides an MSC8101 application development module (ADM). This board has an MSC8101 device surrounded by hardware that enables the customer to test most of the device networking and connectivity capabilities. The board includes a memory sub-system (SDRAM and Flash memory), various network transceivers (155 Mbps ATM, 100 Mbps Ethernet, Quad E1/T1, RS232), Stereo CODEC, and a JTAG debug interface to the MSC8101. It links to the host via a variety of interfaces (parallel, serial, and PCI).

Reset Configuration and Boot 2

This chapter describes the MSC8101 reset and boot process illustrated with examples of different system configurations. It also describes the device clocking system, as it pertains to the reset and boot process. The MSC8101 communicates with other devices in a system either through the system bus or the host port (HDI16). The chosen communication mode defines the reset configuration and the boot method.

2.1 Reset Configuration and Boot Basics

Reset configuration sets the basic mode of operation for the MSC8101, including the operating frequency, arbitration, boot port size, memory controller functionality, and bus behavior. These are the minimal parameters that must be set for correct operation. The system configuration (memory controller, system protection logic, interrupt controller, parallel I/O, and clocks) do not change. There are three kinds of reset in the MSC8101 system:

- *Soft reset*. Invoked externally by asserting the $\overline{\text{SRESET}}$ signal or internally by events. The soft reset initializes the core and internal logic, but system configuration and clocks do not change.
- *Hard reset*. Invoked externally by asserting the $\overline{\text{HRESET}}$ signal or internally by events. The hard reset initializes the core internal logic and system configuration. The phased-lock loop (PLL) and delay-lock loop (DLL) are not affected.
- *Power-on reset*. Invoked only externally by asserting the $\overline{\text{PORESET}}$ signal. Power-on reset initializes the core internal logic, the system configuration, and the clocks. Asserting $\overline{\text{PORESET}}$ input also asserts the $\overline{\text{HRESET}}$ and $\overline{\text{SRESET}}$ signals internally. During power-on reset, several configuration pins are sampled to set the boot mode, the source of the reset configuration values, and the basic clocking mode. The configuration pins are sampled at the rising edge of $\overline{\text{PORESET}}$, and then the reset configuration process starts. During reset configuration, a reset configuration word that determines basic parameter settings is written to the MSC8101, and a PLL and DLL locking process starts. The $\overline{\text{HRESET}}$ and $\overline{\text{SRESET}}$ pins remain asserted for a time after the PLL and DLL locking.

2.1.1 Bootloader Program

The bootloader program loads and executes source code that initializes the MSC8101 after the MSC8101 completes a reset sequence, and the MSC8101 programs its registers for the required mode of operation. The bootloader program, which is provided in the on-chip ROM of the

MSC8101 and presented in **Appendix C** of the *MSC8101 Reference Manual* (MSC8101RM), loads and executes source programs received from a host processor, external memory (an EPROM or a standard memory device on the system bus), or a serial EPROM using the I^2C protocol. The bootloader code starts at location 0xF80000 in the on-chip memory. The bootloader operation mode is set by configuring the BTM[0–1]/EE[4–5] external pins. These pins are sampled on the rising edge of $\overline{\text{PORESET}}$ and their value is stored. In the case of a hard or soft reset, the stored value defines the boot mode. After power-on reset, these pins are available for other uses. **Table 2-1** shows the mode options selected by BTM[0–1]/EE[4–5].

Table 2-1. Boot Mode Selection

External Pin		Source Program Location
BTM0/EE4	BTM1/EE5	
0	0	External Memory
0	1	Host
1	0	Serial EPROM
1	1	Reserved

Note: The contents of the stack memory location can be corrupted when a reset occurs during normal operation. The first instruction of the boot code sets the starting stack address to 0x68000. If a hard or soft reset occurs, the boot code overwrites any user data at this location. Ensure that the user program reinitializes the stack contents after a reset occurs before executing code that uses the stack contents. Never load any source code to 0x68000 during the boot process, because it will corrupt the stack during execution of the bootloader program.

The system does not support interrupt handling during the bootloader operation in any mode. The user must plan for any interrupts that occur while the boot procedure is in progress. Always load interrupt handling code and change the location of the interrupt handler table as soon as possible during the boot procedure. Make sure that no non-maskable interrupt (NMI) occurs before the interrupt handler loading is complete.

2.1.2 Clocks

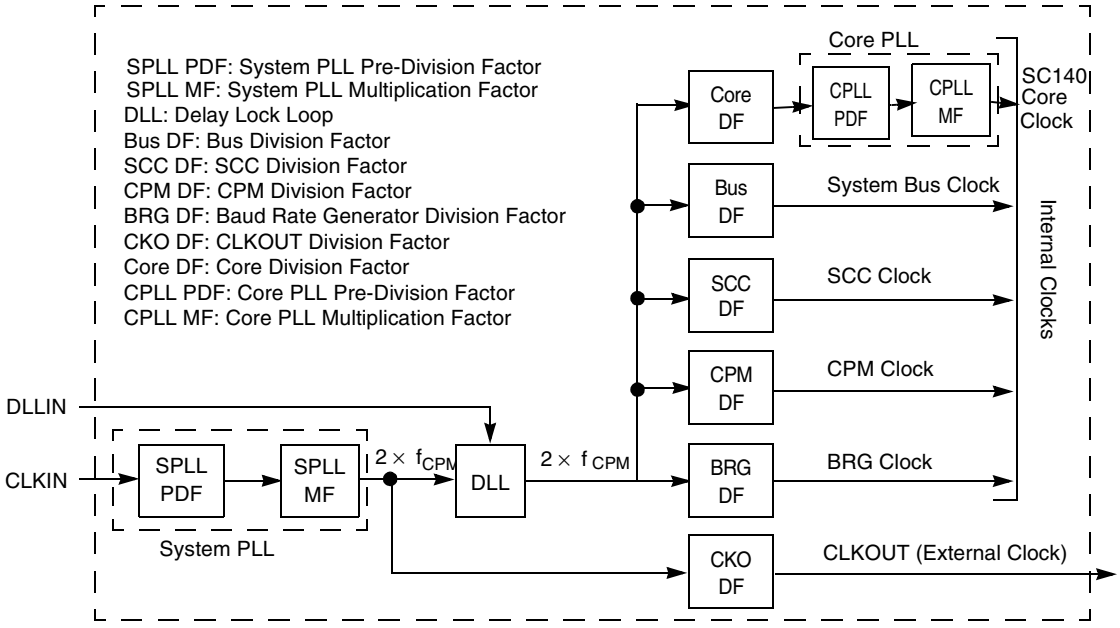
The MSC8101 clocking architecture includes two PLLs: the system PLL (SPLL) and the core PLL (CPLL). The SPLL is the source clock for the following:

- Internal clocks for all blocks in the device, including the CPLL.
- The external system bus clock.
- A DLL used to eliminate possible clock skews in the system, thus allowing multiple MSC8101s to be connected on the same board synchronously, as well as to SRAM.

Each CPLL has the following:

- A predivider that can divide by any integer number between 2 and 5.
- The capability to multiply the input frequency by any even integer number between 10 and 14, with a phase-locking mechanism for skew elimination.

Refer to the *MSC8101 Technical Data sheet* and *AN2306/D Clock Mode Selection for MSC8101 and MSC8103 Mask Set 2K87M* for details on the clocking structure. **Figure 2-1** shows the block diagram of the MSC8101 clocking structure.



Note: SPLL PDF, SPLL MF, CPLL PDF, and CPLL MF are determined by the clock configuration mode; Bus DF = CLKOUT DF and is 4 or 5, determined by the clock configuration mode; SCC DF is always 4; CPM DF is always 2; BRG DF is set by the System Clock Control Register (SCCR) and is 4, 16 (default), 64, or 256.

Figure 2-1. MSC8101 Clocking Structure

The MSC8101 supports the following set of frequency ratios:

- Ratios between the system bus clock and the CPM clock—limited to 1:2, 1:2.5, 1:3, and 1:4.
- Ratios between the system bus clock and the SC140 clock—limited to 1:3, 1:3.5, 1:4, 1:5, and 1:6.

Six bits map the MSC8101 clocks to one of 64 possible configuration mode options, 27 of which are valid modes. Each option determines the CLKIN, system bus, SC140, and CPM frequency ratios. The six bits comprise three dedicated pins (MOSCK[1-3]) and three bits from the reset configuration word (MODCK_H). For information on clock configuration modes and examples, see *AN2306/D Clock Mode Selection for MSC8101 and MSC8103 Mask Set 2K87M*.

2.2 Configuring a Single MSC8101

This section describes the configuration for a system that consists of a single MSC8101 and external memories. The input clock operates at 20 MHz, and the required system clocks are the CPM clock at 150 MHz, the system bus clock at 100 MHz, and the SC140 clock at 300 MHz. There are three possible ways to apply the hard reset configuration word in such a system.

- The MSC8101 is a reset configuration master and reads its own reset configuration word from EPROM.
- The MSC8101 is a reset configuration slave and the system has no boot EPROM.
- The default configuration is used, and the system does not access the boot EPROM.

2.2.1 Master Mode With EPROM

In the configuration described in **Figure 2-2** and **Table 2-2**, the MSC8101 works as a reset configuration master and reads its own reset configuration word from external EPROM. The MSC8101 performs the first part of the reset configuration process described in **Section 2.3.1**, *Reset Configuration Sequence*.

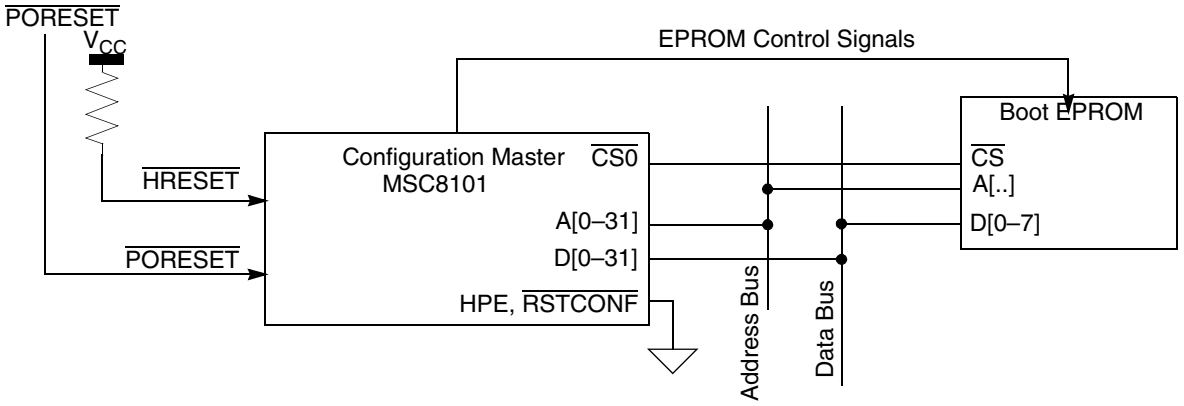


Figure 2-2. Configuring a Single Device From EPROM

Table 2-2. Pin Connectivity for a Reset Configuration From Boot EPROM

Pin/Function	Connection
PORESET	External reset
HRESET	Pulled up
DBREQ/EE0	To GND for normal operation of the SC140 core
HPE/EE1	To GND to disable the host port
BTM[0–1]/EE[4–5]	To GND to enable boot from external memory
MODCK[1–3]	As required to enable the desired clock frequency

Table 2-2. Pin Connectivity for a Reset Configuration From Boot EPROM

Pin/Function	Connection
$\overline{\text{RSTCONF}}$	To GND
Boot EPROM	To the address (A[0–31]) and data (D[0–63]) buses

2.2.2 Slave Mode With No EPROM

For a system with no boot EPROM, you can configure the MSC8101 as a configuration slave by deasserting $\overline{\text{RSTCONF}}$ during $\overline{\text{PORESET}}$ assertion and then asserting $\overline{\text{RSTCONF}}$ while driving an appropriate configuration word on D[0–31] (see**Table 2-3**). In such a system, asserting $\overline{\text{HRESET}}$ in the middle of an operation causes the MSC8101 to return to the configuration programmed after $\overline{\text{PORESET}}$ assertion. The reset configuration word should be applied to D[0–31] using pull-ups and pull-downs on each signal of the bus.

Table 2-3. Pin Connectivity for a Reset Configuration With No Boot EPROM

Pin/Function	Connection
External reset	To $\overline{\text{PORESET}}$
$\overline{\text{HRESET}}$	Pulled up
DBREQ/EE0	To GND for normal operation of the SC140 core
HPE/EE1	To GND to disable the host port
BTM[0–1]/EE[4–5]	To GND to enable boot from external memory
MODCK[1–3]	As required to enable the desired clock frequency
$\overline{\text{RSTCONF}}$	To V_{CC} during $\overline{\text{PORESET}}$; is driven to GND when the configuration word is applied
Boot EPROM	Any external memory connected to address and data bus is not accessed for reset configuration, but it can be accessed for boot and other purposes.
D[0–31]	Pull each line up to V_{CC} or down to ground as appropriate to generate the correct reset configuration word

2.2.3 Default Configuration With No EPROM

The default MSC8101 reset configuration is the simplest configuration scenario (see **Figure 2-3** and **Table 2-4**). The MSC8101 does not access the boot EPROM; it is assumed that the default configuration is used when exiting hard reset.

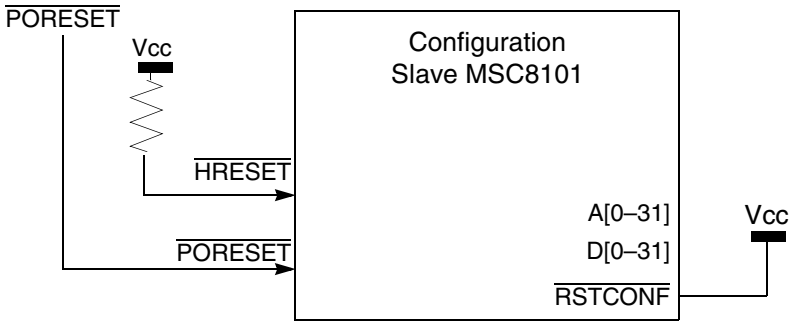


Figure 2-3. Configuring a Single Chip With the Default Configuration

Table 2-4. Pin Connectivity for the Default Reset Configuration

Pin/Function	Connection
$\overline{\text{PORESET}}$	External reset
$\overline{\text{HRESET}}$	Pulled up
DBREQ/EE0	To GND for normal operation of the core
HPE/EE[1]	To GND to disable the host port
BTM[0–1]/EE[4–5]	To GND to enable boot from external memory
MODCK[1–3]	As required to enable the desired clock frequency
$\overline{\text{RSTCONF}}$	To V_{CC} during $\overline{\text{PORESET}}$
Boot EPROM	Any external memory connected to address and data bus is not accessed for reset configuration, but it can be accessed for boot and other purposes.

After exiting reset, the MSC8101 accesses an address table that starts from address 0xFE000110. This address table is a jump table that contains the address for the boot routine. The address table has eight entries, which are accessed according to the MSC8101 internal space base (ISB) of the SIU Internal Memory Map Register (IMMR), which are configured during reset. Each entry is four bytes wide. For example, an MSC8101 with an ISB of 000 accesses the first entry, which resides at address 0xFE000110. An MSC8101 with an ISB of 010 accesses the third entry, which resides at address 0xFE000118. After getting the boot code location, the MSC8101 starts executing the boot code. The boot memory connects to $\overline{\text{CS0}}$, which gates access to the memory.

2.3 Configuring a Multi-MSC8101 System, Bus Connected

This section describes a system of up to eight MSC8101s that connect to a bus. The reset configuration and boot process occur via the bus. In such a system, an EPROM or other standard memory device usually serves the reset and the boot process. This memory device also connects to the bus. The input clock operates at 20 MHz, and the required system clocks are the CPM clock at 150 MHz, the bus clock at 100 MHz, and the SC140 clock at 300 MHz.

One of the MSC8101 devices acts as a reset configuration master for the reset configuration process, and the rest are slaves. A reset configuration word for each MSC8101 in the system is stored at a known address in the memory device. The reset configuration master reads the first reset configuration word, configures itself, and then reads the other configuration words to configure the rest of MSC8101s. This process is described in detail in the following sections.

After exiting reset, each MSC8101 boots from a different address in order to provide maximum system flexibility. After reset, the MSC8101 accesses the external memory device to perform the boot routine. Each MSC8101 can access a unique address to perform its own boot routine. **Table 2-5** and **Figure 2-4** describe the MSC8101 pins and system connectivity.

Table 2-5. Pin Connectivity for a Multi-MSC8101 System, Bus Connected

Pin/Function	Connection
All $\overline{\text{PORESET}}$	External reset
$\overline{\text{HRESET}}$	Connected among themselves and pulled up, if a simultaneous out of reset is required. As long as one of the chips is still in the reset condition, $\overline{\text{HRESET}}$ is asserted and does not allow the others to exit reset.
DBREQ/EE0	To GND for normal operation of the core
HPE/EE1	To GND to disable the host port
BTM[0–1]/EE[4–5]	To GND to enable boot from external memory or to V_{CC} to enable boot from the HDI16
$\overline{\text{RSTCONF}}$	Configuration master MSC8101: To GND Configuration slave MSC8101s: To one of A[0–6]

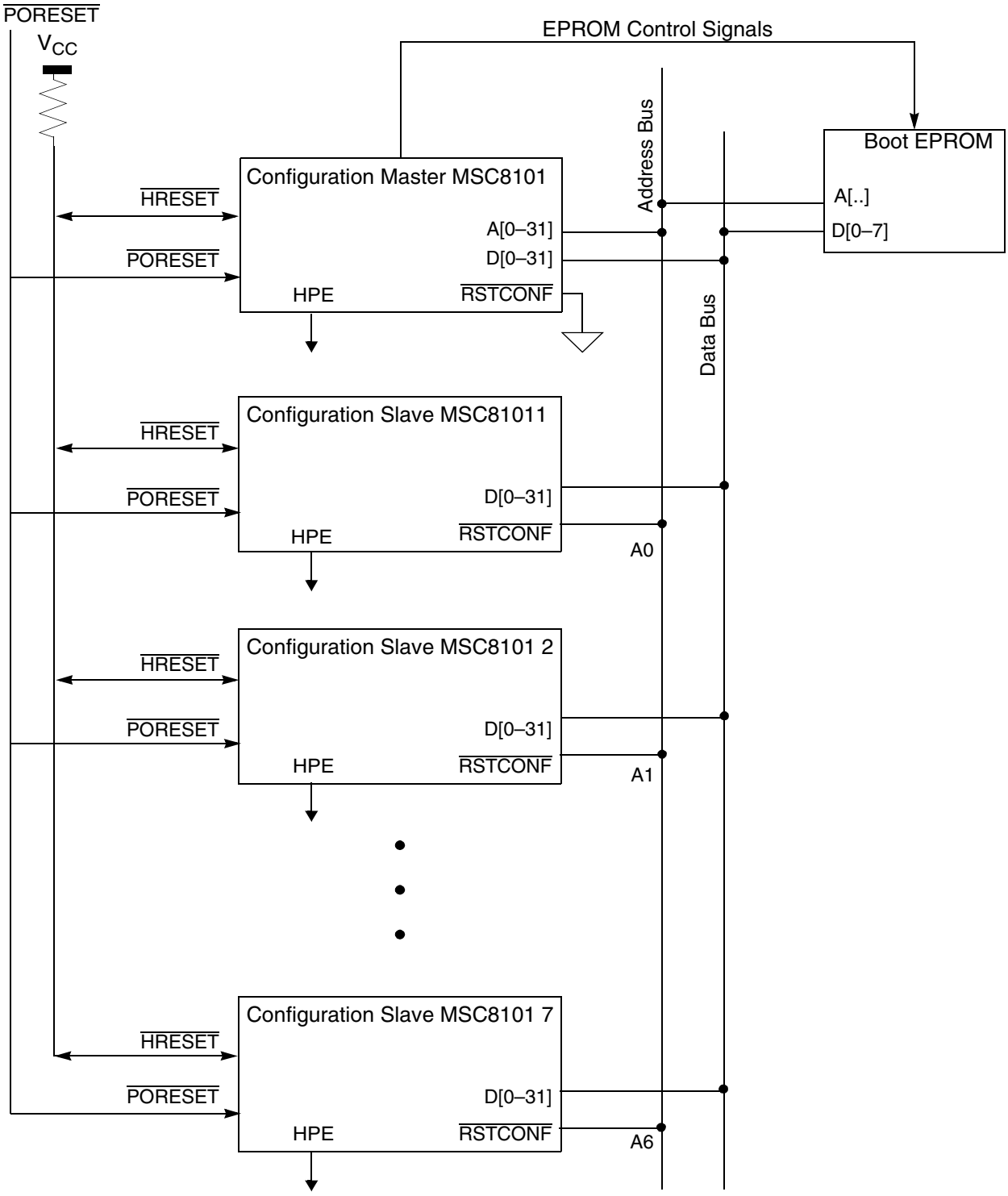


Figure 2-4. Multi-MSC8101 Bus System

2.3.1 Reset Configuration Sequence

The reset configuration sequence supports a system with up to eight MSC8101 devices, each configured differently. It needs no additional glue logic for reset configuration. In a typical multi-MSC8101 system, one MSC8101 acts as the configuration master while all other MSC8101s act as configuration slaves. The configuration master reads eight configuration words from EPROM in the system and uses them to configure itself as well as the configuration slaves. The way that the MSC8101 acts during hardware reset configuration is determined by the value of the $\overline{\text{RSTCONF}}$ input during the period in which $\overline{\text{PORESET}}$ is asserted and deasserted. If $\overline{\text{RSTCONF}}$ is asserted (0) while $\overline{\text{PORESET}}$ changes state, the MSC8101 is a configuration master; otherwise, it is a slave.

In a typical multiple-MSC8101 system, the $\overline{\text{RSTCONF}}$ input to the configuration master is hardwired to ground, while the $\overline{\text{RSTCONF}}$ inputs of other devices connect to the high-order address bits of the configuration master, as described in **Table 2-6**.

Table 2-6. RSTCONF Connections in a Multiple-MSC8101 System

Configured Device	$\overline{\text{RSTCONF}}$ Connection
Configuration master	GND
First configuration slave	A0
Second configuration slave	A1
Third configuration slave	A2
Fourth configuration slave	A3
Fifth configuration slave	A4
Sixth configuration slave	A5
Seventh configuration slave	A6

The configuration words for all MSC8101s reside in an EPROM connected to $\overline{\text{CS0}}$ of the configuration master. Because the port size of this EPROM is not known to the configuration master, the configuration master must read all the hard reset configuration words byte-by-byte from locations that are port size independent before reading the configuration words. **Table 2-7** shows the addresses used to configure the various MSC8101s. Byte addresses that are not listed in this table have no effect on the configuration of the MSC8101 devices. The values of the bytes in **Table 2-7** are always read on byte lane D[0–7], regardless of the port size.

Table 2-7. Configuration EPROM Addresses

Configured Device	Byte 0 Address	Byte 1 Address	Byte 2 Address	Byte 3 Address
Configuration master	0x00	0x08	0x10	0x18
First configuration slave	0x20	0x28	0x30	0x38
Second configuration slave	0x40	0x48	0x50	0x58

Table 2-7. Configuration EPROM Addresses (Continued)

Configured Device	Byte 0 Address	Byte 1 Address	Byte 2 Address	Byte 3 Address
Third configuration slave	0x60	0x68	0x70	0x78
Fourth configuration slave	0x80	0x88	0x90	0x98
Fifth configuration slave	0xA0	0xA8	0xB0	0xB8
Sixth configuration slave	0xC0	0xC8	0xD0	0xD8
Seventh configuration slave	0xE0	0xE8	0xF0	0xF8

The configuration master first reads a value from address 0x00 and then reads a value from addresses 0x08, 0x10, and 0x18. These four bytes form the configuration word of the configuration master, which then proceeds reading the bytes that form the configuration word of the first slave device. The configuration master drives the whole configuration word on D[0–31] and toggles its A0 address line. Each configuration slave uses its $\overline{\text{RSTCONF}}$ input as a strobe for latching the hard reset configuration word during $\overline{\text{HRESET}}$ assertion time. Thus, the first configuration slave whose $\overline{\text{RSTCONF}}$ input connects to the configuration masters A0 output latches the word driven on D[0–31] as its configuration word. The configuration master continues to configure all MSC8101 devices in the system. The configuration master always reads eight configuration words, regardless of the number of MSC8101 devices in the system.

2.3.2 Reset Configuration Word Values

In the system described in **Figure 2-4**, one arbiter handles the system bus arbitration and one memory controller controls the signals and attributes of all memory accesses. Any MSC8101 in the system can handle the roles of system arbiter and memory controller. The MSC8101 that serves as a system arbiter uses its internal arbiter, and the rest of the MSC8101s are configured to work in an external arbitration mode. The MSC8101 that serves as the memory controller for the system uses its memory controller, and the remaining MSC8101s are configured to work with an external memory controller. The arbitration mode and memory controller mode are set in the hard reset configuration word.

Each MSC8101 in the system must have a unique value in the IMMR. In the reset configuration word, there can be up to eight different values for the first three bits of the IMMR (ISB[0–2]). Each MSC8101 in the system must be configured to one of these values. These values can be changed during boot. MODCK is also configured in reset configuration to choose the desired clock frequency.

Table 2-8. Hard Reset Configuration Word Values

Master MSC8101		Slave MSC8101 Devices	
Value	**Description**	**Value**	**Description**
EARB = 0	Internal arbitration	EARB = 1	External arbitration
EXMC = 0	Internal memory controller	EXMC = 1	External memory controller
EBM = 1	system bus-compatible mode	EBM = 1	system bus-compatible mode
BPS = 01	8-bit boot port size according to the example in **Figure 2-4**	BPS = 01	8-bit boot port size according to the example in **Figure 2-4**
SCDIS = 0	SC140 core enabled	SCDIS = 0	SC140 core enabled
DLLDIS = 0	No DLL bypass for normal operation	DLLDIS = 0	No DLL bypass for normal operation
ISB = 000	IMMR value is 0xf000_0000	ISB = xxx	IMMR value is different for each MSC8101
MODCK_H = xxx	As required to enable the desired clock	MODCK_H = xxx	As required to enable the desired clock
The rest of the fields should be configured according to system requirements: IRQ7INT, ISPS, IRPC, DPPC, NMI OUT, BBD, TCPC, BC1PC. Assume they are all equal to zero.			

2.3.3 Boot in a Multi-MSC8101 Bus System

The MSC8101 executes commands directly from external memory when BTM[0–1]/EE[4–5] are both pulled low. If required, a boot sequence can be loaded into internal RAM as part of the boot routine. There are no restrictions on the format of the boot sequence. The MSC8101 internal memory controller supports specific boot functionality. The boot chip-select operation allows address decoding prior to system initialization for the external memory boot operation. $\overline{\text{CS0}}$ is the boot chip-select output, and the external boot memory should connect to it. The MSC8101 boot chip-select operation also provides a programmable port size during system reset, if the BPS bits in the reset configuration word are written.

The bootloader program accesses an address table that resides at address 0xFE000110. This table holds the address of the boot routine as a 32-bit entry. This address is user-programmable, and the routine can be placed at any address in the space controlled by the chip-select. The MSC8101 retrieves the boot address from the table according to the ISBs in first three IMMR[0–2], which are configured during reset. Each entry is four bytes wide. For example, a device with an ISB cleared to 000 accesses the first entry, which resides at address 0xFE000110. A device with an ISB of 010 accesses the third entry, which resides at address 0xFE000118. After getting the boot code location, the MSC8101 begins executing the boot. Therefore, each device can have its own boot routine.

2.4 Configuring a Multi-MSC8101 System Connected Via the Host Port

This section presents an example of a system with three MSC8101s controlled by one MSC8101 that serves as a host. The host exits reset and boots, and then it writes the reset configuration word to each of the other two MSC8101 devices. After the last MSC8101 hard reset configuration word is written, the MSC8101s exit reset and the boot process starts. The host loads the hard reset configuration word into each MSC8101, which then executes its boot code. The host MSC8101 can be configured in any of the ways described for a single device, and it exits reset as a single device. In the example discussed here, it is configured by reading a hard reset configuration word from external memory. Then it executes its boot code, which resides in the external EPROM. The host MSC8101 is ready to configure the rest of the system. It executes code from the EPROM or other external memory, or it downloads the code to its internal SRAM. This code contains the hard reset configuration words and the boot code for each MSC8101 device in the system. **Table 2-9** and **Table 2-10** show the pin connectivity for the host and for the multi-MSC8101 system. **Figure 2-5** describes the MSC8101 pins and basic system connectivity.

Table 2-9. Host MSC8101 of a Multi-MSC8101 System Connected Via Host Port

Pin/Function	Connection
External reset	To all $\overline{\text{PORESET}}$
$\overline{\text{HRESET}}$	Does *not* connect to the $\overline{\text{HRESET}}$ of the other MSC8101s
DBREQ/EE0	To GND for normal operation of the SC140 core.
HPE/EE1	Connected to GND to disable the host port
BTM[0–1]/EE[4–5]	Connected to GND to enable boot from external memory
$\overline{\text{RSTCONF}}$	Connected to GND

Table 2-10. Multi-MSC8101 System Connected Via Host Port

Pin/Function	Connection
External reset	To all $\overline{\text{PORESET}}$
$\overline{\text{HRESET}}$	Connected among themselves and pulled up, if a simultaneous out of reset is required. As long as one of the devices is still in the reset condition, $\overline{\text{HRESET}}$ is asserted and does not allow the others to exit reset.
DBREQ/EE0	To GND for normal operation of the SC140 core.
HPE/EE1]	To V_{CC} to enable the host port
BTM[0–1]/EE[4–5]	All BTM0s connect to GND and BTM1s connect to V_{CC} to enable boot loading through host port
$\overline{\text{RSTCONF}}$	Connects to V_{CC}
NOTE: Ensure that the ISPS bit 7 of the hard reset configuration word is set (1) when the Host Port (HDI16) is in use. This changes the data bus from 64 to 32 bits wide. Failure to set this bit results in data bus conflicts and errors.	

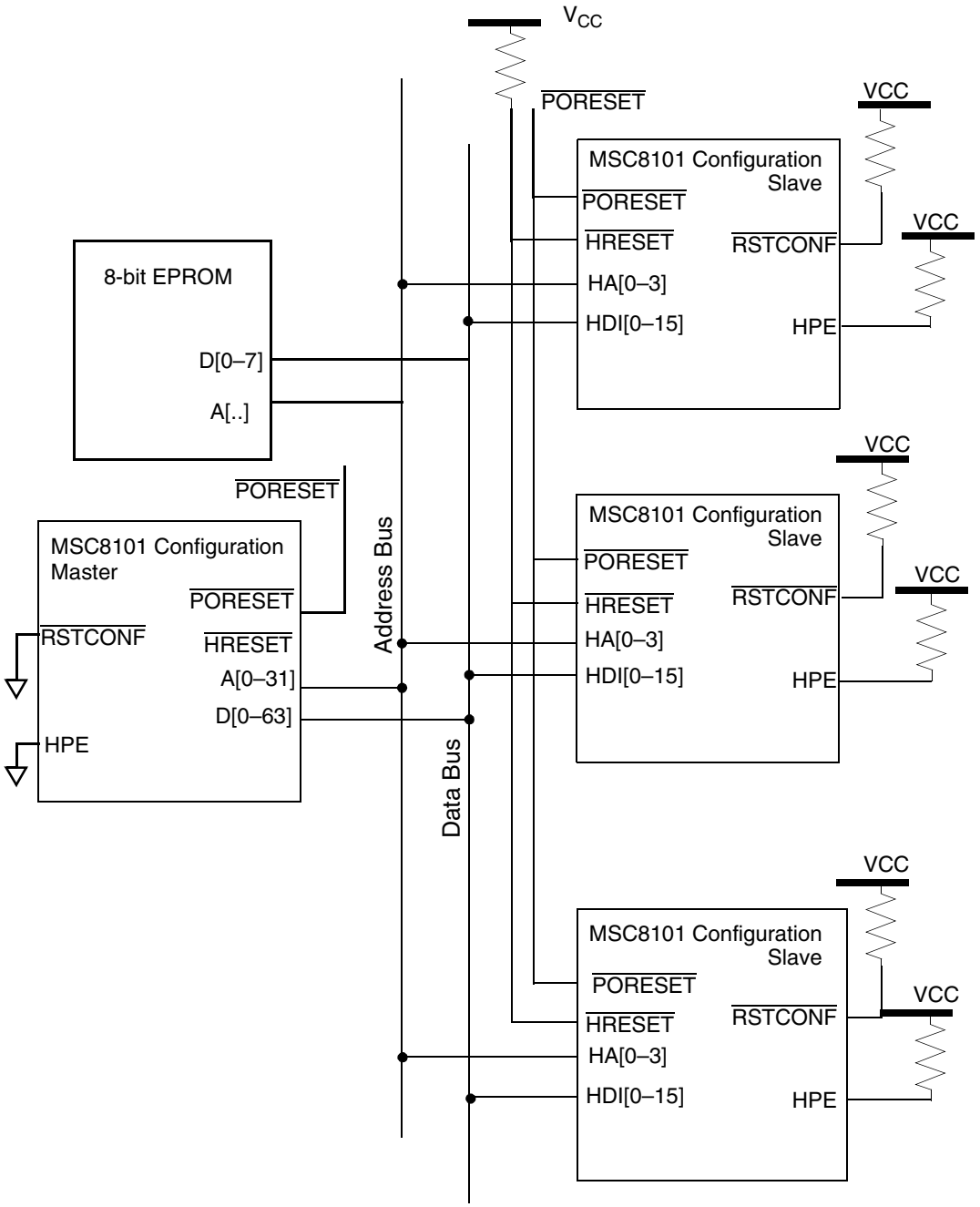


Figure 2-5. Multiple MSC8101s Connected Via the Host Port

2.4.1 Host Reset Configuration Sequence

This section describes how the reset configuration word is applied to a host-controlled MSC8101. Host reset configuration allows the host to program the reset configuration word via the host port after $\overline{\text{PORESET}}$ is deasserted. If HPE is sampled high at the rising edge of $\overline{\text{PORESET}}$, the host port is enabled. In this mode the $\overline{\text{RSTCONF}}$ pin *must* be pulled up deasserted. The device extends the internal $\overline{\text{PORESET}}$ until the host programs the reset configuration word register. The host must write four 8-bit half-words to the host reset configuration register address to program the reset

configuration word, which is 32-bits wide.[1] This register is programmed before the internal PLL and DLL in the MSC8101 are locked. The host must program this register after the rising edge of $\overline{\text{PORESET}}$ input. The host has its own clock and does not depend on the MSC8101 clock. After the PLL and DLL are locked, $\overline{\text{HRESET}}$ remains asserted for another 512 bus clocks and is then released. The $\overline{\text{SRESET}}$ is released three bus clocks later.

2.4.1.1 Reset Configuration Word Value

In the system depicted in **Figure 2-5**, the host MSC8101 transfers data and control to/from the other MSC8101s through the host port. The host uses its address and data buses for data transfers. Only the host can initiate transactions, so there is no need for a system arbiter. The host accesses external memory via its system bus. The other MSC8101s connect to their own external memory, a memory for each MSC8101, so there is no need for a system memory controller.

Table 2-11. Reset Configuration Word Values for Host Reset Configuration

Master MSC8101		Slave MSC8101 Devices	
Value	**Description**	**Value**	**Description**
EARB = 0	Internal arbitration	EARB = 0	Internal arbitration
EXMC = 0	Internal memory controller	EXMC = 0	Internal memory controller
EBM = 0	Single-chip mode	EBM = 0	Single-chip mode
BPS = 01	8-bit boot port size according to the example in **Figure 2-5**	BPS = 00	Boot occurs via the host port; this value has no effect
SCDIS = 0	SC140 core enabled	SCDIS = 0	SC140 core enabled
DLLDIS = 0	No DLL bypass for normal operation	DLLDIS = 0	No DLL bypass for normal operation
ISB = 000	IMMR value is 0xF000_0000	ISB = 010	Each MSC8101 can have any IMMR value; can be changed by boot
ISPS = 1	Select 32-bit data bus	ISPS = 1	Selects 32-bit data bus
The rest of the fields should be configured according to system requirements: IRQ7INT, ISPS, IRPC, DPPC, NMI OUT, BBD, TCPC, BC1PC. Assume they are all equal to zero.			

2.4.2 Boot Through Host Port

The MSC8101 host interface supports bootloading from hosts with 8-bit or 16-bit ports. The system in the example discussed here uses a 16-bit port. The MSC8101 host treats all accesses as address accesses. Single-data strobe or dual-data strobe access and data strobe polarity are configured via external pins. The MSC8101 host interacts with the host port of the MSC8101 slave in polling mode. The MSC8101 host accesses the host port registers to determine the status of the host port—for example, whether the host port can receive more data.

1. For details on the host port registers, refer to the HDI16 chapter in the *MSC8101 Reference Manual.*

2.4.3 Source Program Data Stream Structure

The source program can be organized into several blocks. Each block is either a data block or an instruction block, and it can be loaded to a different specified destination. The checksum method ensures correct data loading. Each block specifies its size and destination address, and it ends with a checksum and a $\overline{\text{CHECKSUM}}$. **Figure 2-6** summarizes the structure of the source program data stream.

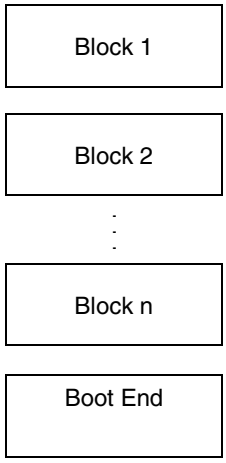


Figure 2-6. Boot Code Stream Structure

The data stream source programs must be structured in the format shown in **Table 2-12**.

Table 2-12. Data Stream Source Program Block Structure

Word	Description
1	Block size in 16 bits of the first program block to be loaded, most significant part
2	Block size in 16 bits of the first program block to be loaded, least significant part
3	Address where the first block of the source program is to be loaded, most significant part
4	Address where the first block of the source program is to be loaded, least significant part
5	0x0000
6	0x0000
7	Checksum—xor
8	$\overline{\text{Checksum}}$—$\overline{\text{xor}}$

The bootloader routine expects at least one code block. When more than one block is included in the source program data stream, word n+5 contains the address of the second block, as shown in **Table 2-12**. The sequence repeats for subsequent blocks until the final block in the data stream is reached. A special boot end block indicates the end of the boot code stream (see **Table 2-13**).

Table 2-13. Structure of the Boot End Block

Word	Description
1	0x0000
2	0x0000
3	Boot start address, most significant part
4	Boot start address, least significant part
5	Checksum, XOR
6	Checksum, $\overline{\text{XOR}}$

The first two words indicate the end of the source blocks. At least one block of source code is loaded when the bootloader is invoked.

2.4.4 Host Interface Load Procedure

The host interface load procedure includes:

1. Loading the source code blocks
2. Storing the blocks in a given address
3. Performing checks

The host MSC8101 writes the source code word after word to the host interface registers of the MSC8101 slave. For each code word that is loaded, the routine calculates a checksum. The checksum is calculated by XORing the current word bit by bit with the result of XORing previous words. The value of bit *i* of the current result is equal to XORing bit *i* of the current word with bit *i* of the previous result. After the entire block is loaded, the calculated checksum is compared to the loaded checksum to verify that the code loading completed correctly. Note that the checksum is calculated on all the block words, starting from the address.

A handshaking mechanism between the host and the slave indicates the status of the code loading. The handshake mechanism uses one flag of the Interface Control Register (ICR) and two flags in the Host Control Register (HCR). The host can set a flag in the ICR to ignore the checksum comparison. If the flag is set, the checksum should be compared.

The SC140 core of the slave MSC8101 sets a flag in the HCR to indicate the completion of code loading. It sets a different flag to indicate the occurrence of an error during the checksum checks. The host MSC8101 ignores this flag if it does not need the checksum checks.

2.5 Related Reading

- *MSC8101 Reference Manual*
- *MSC8101 Technical Data*

Optimizing Memory on the SC140 Core 3

This chapter describes the memory mechanism of the SC140 core and explains how to allocate the memory efficiently for an application running on the MSC8101. The recommended SC140 application development methodology involves significant use of a C compiler, which reduces development time and results in high-performance code. Most of the memory allocation tasks are relegated to the compiler, and the recommendations presented here complement the default memory allocation it performs.

3.1 Memory Requirements

Any memory used by the SC140 core must conform to the following system requirements:

- The MSC8101 core supports only unified memory, meaning that memory is regarded as a single space. There is no distinction between program memory locations and data memory locations, and each memory location possesses a unique address. For example, memory address 0x1000 can hold either data or program information.
- Data must be byte-addressable and accessible to both memory data buses.
- The memory must support all data width accesses used by the SC140 core, namely half-word (8 bits), word (16 bits), double-word (32 bits), or quad-word (64 bits).
- Memory must be accessed cycle by cycle so that all accesses on a given cycle are identified and resolved before proceeding to accesses in the next cycle.
- Multiple access rules in a given cycle are as follows:
  - Multiple read or write accesses to different memory locations execute without any predetermined sequence.
  - When multiple accesses to the same memory location occur, the access sequence is as follows: program fetch, data read, and data write.
  - If two write operations access overlapping bytes in memory in the same cycle, there is a memory contention. The memory subsystem detects these cases and issues an interrupt to the SC140 core.
- The SC140 core does not support accesses to a non-existent memory location. If required, the memory subsystem detects these occurrences and generates non-maskable interrupts (NMIs) to the SC140 core.

3.2 Partitioning Memory

The SC140 core is flexible in its support for various memory structures, including different division into submemories. The example in **Figure 3-1** presents a general structure that partitions the memory as follows:

- The memory is made up of a number of 32 KB groups.
- Each group consists of eight 4 KB modules.
- Each module includes 128 rows.

Addresses are interleaved over the modules within a group, in row boundaries. This organization enables consecutive addressing across more than one module in a group.

3.3 Allocating Memory

The compiler allocates program memory and data RAM. By default, the program memory is allocated to the subroutines sequentially. In the data memory, a sophisticated algorithm efficiently allocates the various arrays and variables. For each variable or array, a time period analysis is performed in which the living time of the array or variable is examined, and the option to share other variables on the same physical memory locations is tested. If there is no overlap in the living times of two variables, they can share the same memory location. The algorithm generally finds the most efficient way to allocate the variables. Therefore, it is recommended that you allow the compiler to allocate the data RAM. However, for flexibility, the option to allocate memory manually still exists. In memory-critical applications, a more optimized allocation may be achieved by manual allocation.

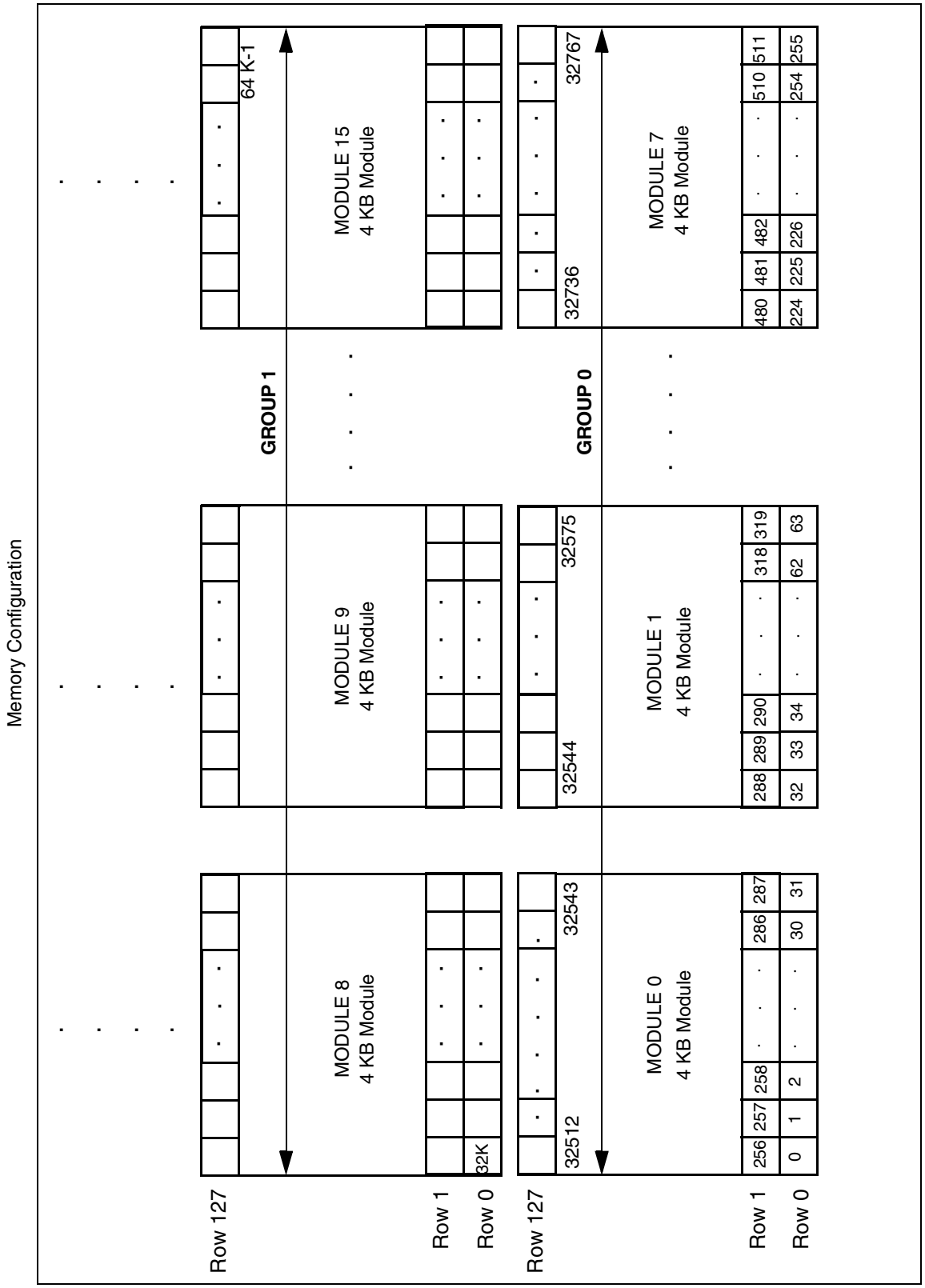


Figure 3-1. Memory Organization

3.4 Avoiding Memory Contentions

Contentions occur when there are multiple requests for access to a memory group in the same cycle. A contention event is followed by a stall cycle per conflict in which the contention is resolved. The following guidelines apply:

- There can be up to three core memory accesses per clock cycle:
  - Two data accesses
  - One program access
- Only one program access, or one or two data accesses, can refer to a specific group in the same clock cycle.
- Two conflicting accesses to a group (program and data) stall the core for one cycle.
- Two data accesses cause a one-cycle stall if the accesses are to two different rows in the same module.
- Two data accesses to the same group do not cause a stall in the following cases:
  - Accesses to different modules
  - Accesses to the same row of the same module
- The address interleaving structure, in which consecutive addresses are interleaved over the eight modules within a group, implies that two data accesses to different memory locations do not cause a stall if the addresses of these locations fall within a range of 224 bytes (7 rows).

As **Figure 3-2** shows, the SC140 memory space is divided into 32 KB groups, each divided into eight 4 KB modules. The modules are divided into 32-byte lines.

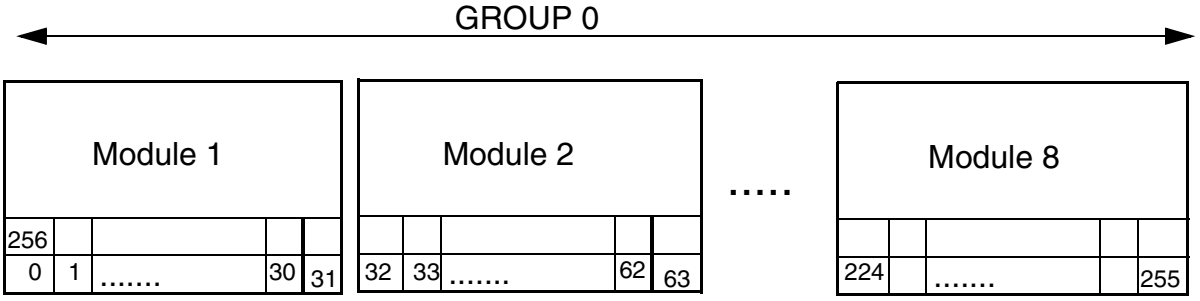


Figure 3-2. Memory Structure

Memory contentions between program memory and data memory occur when the program bus and the data bus attempt to access the memory within the same group. To prevent memory contentions, keep the data and program memory in different groups. For example use the addresses 0..0x7FFF (group0) for data storage and addresses 0x8000..0xFFF for program memory.

A memory contention is also caused if the DMA and data bus attempt to access the memory within the same group. For information on how to avoid this type of contention, see **Section 6.5**, *Avoiding DMA and SC140 Core Contentions,* on page 1-25.

Data memory contentions are caused when the two address generation unit (AGU) instructions in the execution set attempt to access two different lines in the same memory module. This causes the execution set to take one more cycle. To avoid data memory contentions, you can do the following:

- If possible, write each memory access in a different execution set.
- If not, analyze the code to find what combination of memory transfers may cause a contention, and then separate them.
- If possible, change the start addresses to avoid contention.

The analysis and contention checks can be done using the simulator, through the `display on stall` option.

3.5 Related Reading

Table 2-1.

MSC8101 Reference Manual

Chapter 8, Memory Map

Chapter 9, *Internal Memory System and Reservation Operation*

Chapter 10, *Memory Controller*

Chapter 11, *QBus*

Connecting External Memories and Memory-Mapped Devices 4

This chapter illustrates several memory interconnection options for the MSC8101 bus and memory controller. It outlines the hardware connections and memory register settings for the MSC8101 when the system bus connects to Flash memory, Synchronous DRAM (SDRAM) or an MSC8101 HDI16 slave.

4.1 Memory Controller Basics

The MSC8101 has three integrated memory controllers tailored to suit a variety of bus control profiles. This chapter discusses all of these memory controllers and illustrates them with examples:

- *General-Purpose Chip Select Machine (GPCM)*. A baseline controller for simple non-multiplexed interfaces such as EPROM, Flash EPROM, and SRAM. A GPCM-derived chip select interfaces simple, non-bursting devices over a selection of port sizes (8-, 16-, 32- and 64-bit) and a wide range of speed grades. To illustrate its use, the chapter discusses the interface to a Flash EEPROM device.
- *Dedicated SDRAM controller*. Gluelessly connects to JEDEC-compliant SDRAM devices. The SDRAM controller generates the row address strobe ($\overline{\text{RASn}}$), column address strobe ($\overline{\text{CASn}}$), chip select ($\overline{\text{CSn}}$), and control signal combinations. The programmable address multiplexing and timing characteristics enables configuration of the SDRAM parameters including, row to column address latch timing, page-mode burst operation, and interleaving. To illustrate SDRAM controller operation, this chapter discusses the interface to a SDRAM device on the 100 MHz System Bus.
- *User-Programmable Machine (UPM)*. A flexible alternative controller by which users can define a fully programmable bus cycle profile for a range of such standard or proprietary interfaces as SRAM, Slave Interfaces (HDI16), and ASICs. The UPM offers much more flexibility in timing to target a broader range of system devices than the GPCM. Through the UPM-controlled interface, software can define the chip selects and control strobes on each bus clock to one quarter clock granularity. Developers commonly use this flexibility for user-defined interfaces to ASICs or DSPs. Any or all of the eight external chip selects can use the same UPM timing. To illustrate the UPM capabilities, this chapter discusses a

UPM-defined interface that gives a programmable port size and strobe generation matching that of the MSC8101 HDI16 host port.

For each chip select ($\overline{\text{CSn[0-7]}}$, $\overline{\text{CSn[10-11]}}$) and associated memory bank, the associated memory controller (that is: GPCM, SDRAM, or UPM) must be programmed in the Machine Select field of the respective Base Register (BRx[MS]). Two of the total chip selects ($\overline{\text{CSn[10-11]}}$) are allocated internally on the local bus (for peripherals and memory control). The remaining eight chip selects are available for the external system bus. For these chip selects, one set of SDRAM settings and up to two separate UPM settings are possible, although multiple chip selects can use the same configurations for mapping different memory regions. Any remaining chip selects can be programmed in the General-Purpose Chip select mode with individual timing settings per chip select.

Whatever memory controller mode is selected, the following parameters must be programmed:

- Base address
- Addressable memory window size
- Port width (8, 16, 32, 64 bits)

On the basis of the address decoded for an internal transaction, the bus operation is mapped onto the selected bus. If the data is larger than the corresponding port size, the memory controller automatically splits the data into multiple bus cycles of a data width up to the programmed maximum: for example, a 64-bit transaction to a 16-bit port generates four bus cycles back-to-back).

4.2 External Bus Basics

All memory controller types use exactly the same external 64-bit (or 32-bit) system bus, with the distinction that each memory controller offers differing additional strobe signals. This system bus has two completely separate configurable bus modes, Single-Master MSC8101 Bus mode and Multi-Master Bus mode. The Single-Master MSC8101 Bus mode is selected when the Bus Configuration Register BCR[EBM] bit is cleared, and multi-master mode selected when this bit is set. The Flash memory, SDRAM, and HDI16 examples in this chapter focus on the Single-Master MSC8101 Bus mode.

The Single-Master MSC8101 Bus mode gives the simplest interconnect to external peripherals. In this mode, there is no external arbitration, so the MSC8101 is the only master on the bus. The MSC8101 drives the address and data for the duration of every bus access, so that external memories can connect directly to it, if the capacitive load of the connected devices allows. This direct interconnect method is used for the three memory controller examples, GPCM-controlled Flash memory, UPM-controlled HDI16 interfaces, and direct SDRAM interfacing in which all the signalling, including row and column address multiplexing is handled internally.

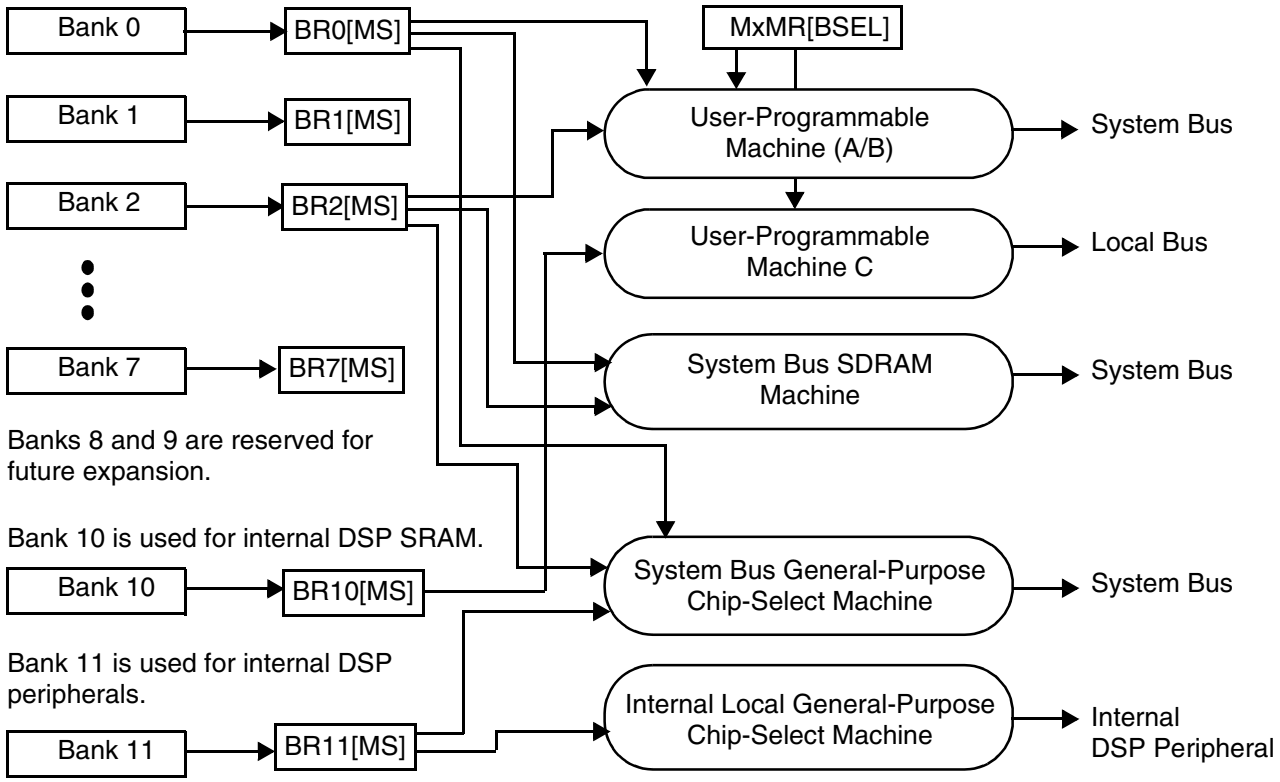


Figure 4-1. Memory Controller Machine Selection

The System bus operating in multi-master mode has one or more bus masters arbitrating for access to the shared bus resource. The advantage of multi-master mode is that several compatible bus masters can interconnect directly and make full use of the bus pipelining potential. To achieve full bandwidth performance, the separate address and data tenures are pipelined. Therefore, the address and associated address controls lines can be ready for the next access before the data phase for the current access is complete. The interface to memory devices requires the use of external address latches to register the address and maintain it to the memory for the full duration of the memory access. Meanwhile, the processor can overlap the next arbitration and address phase in readiness for a subsequent access. The MSC8101 simplifies external address latching through an address latch signal (ALE), and for SDRAM usage, it also provides an address multiplex signal for row and column multiplexing (PSDAMUX).

4.3 Connecting the Bus to the Flash Memory Device

In most embedded systems, Flash memory is the standard way to store the non-volatile bootstrap code for the system at power-up. In a DSP environment, at least one device typically connects to the Flash memory to configure the system and bootload real-time firmware code into other devices from a power-up or reset. The MSC8101 supports such a Flash memory boot operation, as follows:

- $\overline{\text{PORESET}}$ is released and a number of registers are programmed automatically.

- Chip Select 0 (CSn0) is programmed by default as a GPCM (BR0[MSEL] = 000) with base address 0xFE000000 (BR0[BA] = 1111_1111_1110_0000_0), and an address mask of 32MB (OR[SDAM] = 1111_1110_0000_0000_0), with conservative timings (relaxed timing (OR[TRLX:EHTR] = 10) and 15 wait states (OR[SCY] = 1111)).
- The MSC8101 will access Flash memory to determine the 32-bit reset configuration word. To access Flash memory from reset the external Reset Configuration signals CNFGS and RSTCONF should be pulled low to enable the Hardware Reset Configuration Sequence.
- $\overline{\text{HRESET}}$ is released and code is fetched from memory as determined by the Reset Vector pointer address.

Refer to the *Reset Configuration and Boot* chapter for details on the reset configuration and booting procedures. The following subsections present a typical MSC8101-to-Flash-memory interconnection example, together with the primary GPCM register configuration and timing assumptions.

4.3.1 GPCM Hardware Interconnect

The GPCM timing can interface with any industry-standard Flash memory device directly. The example discussed here illustrates the use of an AMD AM29LV320DB 90 ns 2 M × 16-bit Flash memory. The chip select (CSn0), Output Enable ($\overline{\text{OEn}}$), and Write Enable ($\overline{\text{WEn}}$) signals connect directly to the Flash memory; with the appropriate programmed register settings (see **Figure 4-2**).

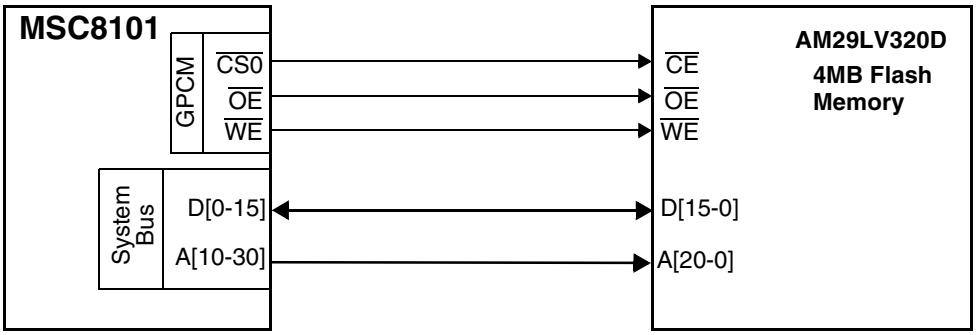


Figure 4-2. MSC8101-to-Flash-Memory Interconnect in Single Master Bus Mode

4.3.2 Single-Bus Mode GPCM-Based Timings

The timing characteristics of the MSC8101 chip select must meet the worst-case timing needs of the selected Flash memory device to assure operation over full temperature and voltage range. Valid data is driven from the Flash memory on a read access based on the combined, $\overline{\text{CEn}}$ and $\overline{\text{OEn}}$ timings. During a write the address is latched on the falling edge of $\overline{\text{WEn}}$ or $\overline{\text{CEn}}$**, whichever happens later. All data is latched into the memory on the rising edge of** $\overline{\text{WEn}}$ or $\overline{\text{CEn}}$, whichever happens first. The key timings to assure data transfer integrity are therefore the set-up and hold times around these two data latch points.

To maintain data integrity, it is very important that only one device drives the data bus at any given time, so data bus timing around the beginning and end of bus cycles is of great interest. Typically, on a write cycle, the data is driven on the first clock edge of the access. To prevent data bus contention, sufficient time should be allotted at the end of a read access to ensure that the data bus is appropriately tri-stated before the next cycle.

The example illustrated in **Figure 4-3** and **Figure 4-4** (10 ns CLK period) uses unbuffered data, so the situation is relatively simple. However, if data is buffered, the $\overline{\text{BCTLx}}$ signals that control the buffer direction ($\overline{\text{BCTL0}}$) and output enable ($\overline{\text{BCTL1}}$) timing are asserted on the first memory controller clock and negated on the clock of the $\overline{\text{TA}}$ assertion. Therefore, if data buffering is used for a GPCM Flash memory (or UPM) access, allot time at the end of read accesses to ensure that data is tri-stated. Also, allot time at the beginning of read accesses to ensure that $\overline{\text{OE}}$ timing does not contend with buffer write data.

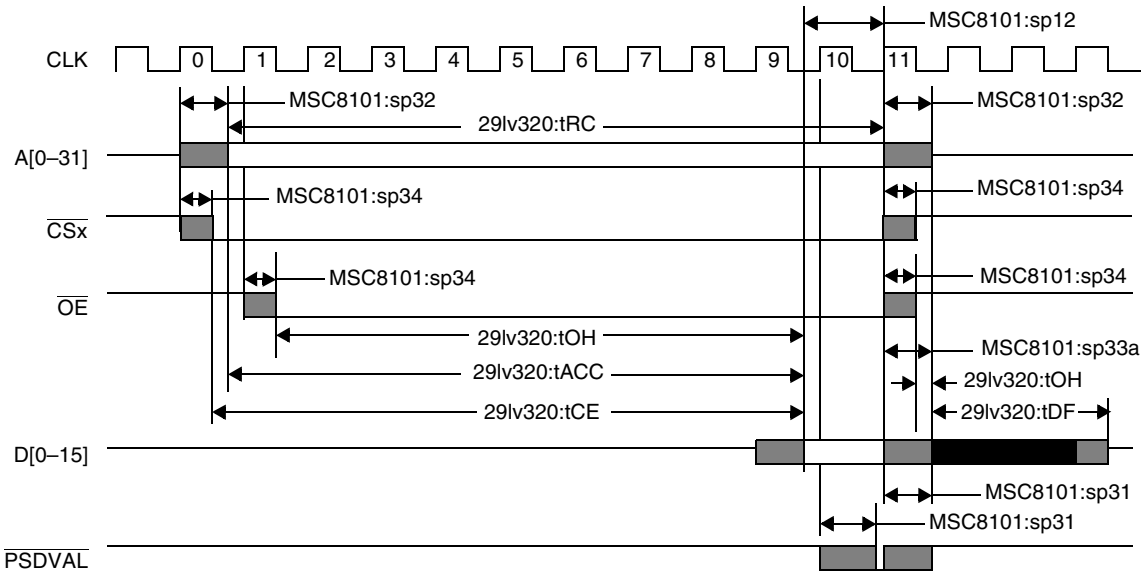


Figure 4-3. Flash Memory Read, Single Master

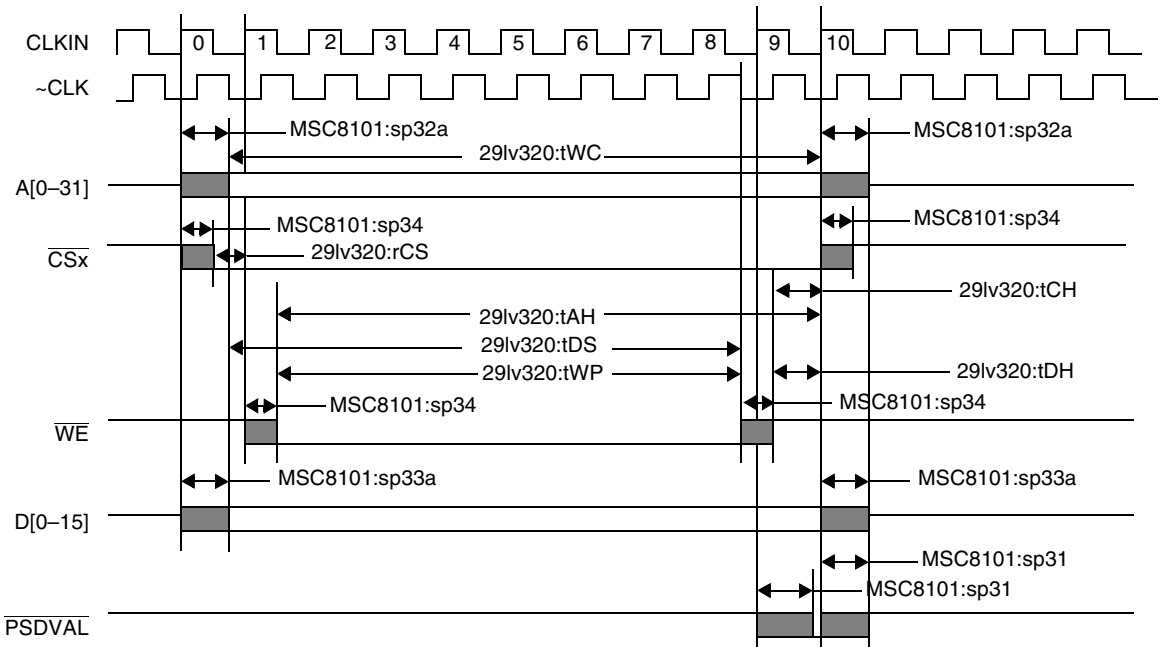


Figure 4-4. Flash Memory Write, Single Master

For the AMD AM29LV320D-90 ns device, the 90 ns access timing implies that around nine to ten 10 ns clock accesses should be possible. Taking into account the full timing requirements of the interdependent signals in the timing diagram example of **Figure 4-3** and **Figure 4-4**, both read and write accesses actually use eleven to twelve 100 MHz clock accesses. To achieve this, the Option Register (ORx) settings of ACS[0–1]) = 00, TRLX:EHTR = 10, SCY = 7 wait states and CSNT = 1 are used. See the Option Register settings in **Table 4-1**.

The memory controller timing options are set so that write accesses achieve the CSn hold time after deassertion of the $\overline{\text{WEn}}$ latches the data. The relaxed timing (OR[TRLX:EHTR] = 10) helps prevent data contention on the data bus when reads and writes are alternated. In particular, the extended hold time capability on reads ensures that the troublesome case of write data after a read does not cause data contention with the next cycle. The one disadvantage of these timings is that for back-to-back write accesses, the $\overline{\text{WEn}}$ deassertion time (tWPH) requirements of the Flash memory device are not met directly. This is not usually a problem because the Flash memory programming is infrequent and typically involves a read (poll) access between each write.

Close inspection of the timings reveals that it is not essential to set TRLX when no buffering is used. Therefore, the same timing settings could be altered so that TRLX = 0 and SCY = 7 wait states to get a more aggressive clock access with a 90 ns Flash memory. However, the case detailed here is more general because the relaxed timing allows more time at the beginning and end of cycles, so a buffered solution can use exactly the same settings.

Table 4-1. GPCM Option Register Settings

Register Setting	Description
OR[BCTLD] = 1	$\overline{\text{BCTL}}$[0–1] signals are not asserted upon access to the Flash memory
OR[CSNT] = 1	$\overline{\text{CSn}}$/$\overline{\text{WEn}}$ deasserted ¼ clock cycle earlier relative to address deassertion
OR[ACS] = 00	$\overline{\text{CSn}}$ asserted with new address
OR[SCY] = 0111	7 clock cycle wait states
OR[SETA] = 0	$\overline{\text{PSDVAL}}$ is generated internally
OR[TRLX:EHTR] = 10	4 idle cycles are inserted between read access from current bank and next access

4.4 SDRAM Memory Interface

SDRAMs are the most cost effective, high capacity read/write memories on the market, offering high-performance throughput with the cost benefits of a commodity item. The synchronous nature of the SDRAM allows precise access timing control, with data transactions possible on every clock cycle. SDRAMs are thus ideal for high-bandwidth memory systems.

The SDRAM machine within the MSC8101 memory controller provides all the necessary control functions and signals for interfacing to JEDEC-compliant SDRAMs.

The following subsections detail the hardware connection of a 32-bit SDRAM to the MSC8101 using the SDRAM controller in both Single-Master MSC8101 Bus and Multi-Master Bus modes. This is followed by a description of the Timing control settings and the SDRAM initialization procedure. Notice that the hardware timings differ for both modes. In Single-Master MSC8101 Bus mode, the memory controller uses memory timings and is completely glueless. In multi-master mode, the memory controller allows pipelined 60x-compatible accesses and requires an external address latch and multiplexor.

4.4.1 Single-Bus Mode SDRAM Hardware Interconnect

When operating in Single-Master MSC8101 Bus mode, the MSC8101 is the only master on the bus and typically connects directly to memory and/or slave peripherals. The MSC8101 SDRAM controller provides the address, data, and control signals for a direct, glueless interconnect to the SDRAM. All the address multiplexing is performed internally within the MSC8101, so there is no need for address latches or multiplexers typically required for SDRAM control. For the SDRAM control commands, the SDRAM needs a chip select ($\overline{\text{CSn}}$) together with Row and Column address strobes ($\overline{\text{RASn}}$ and $\overline{\text{CASn}}$), Write Enable ($\overline{\text{WEn}}$), byte lane selects ($\overline{\text{PSDDQMn}}$), and a multiplexed A10/AP bank select pin. **Figure 4-5** shows the connection between the MSC8101 memory controller and a 32-bit Micron MT48LC2M32B2TG using page-based interleaving.

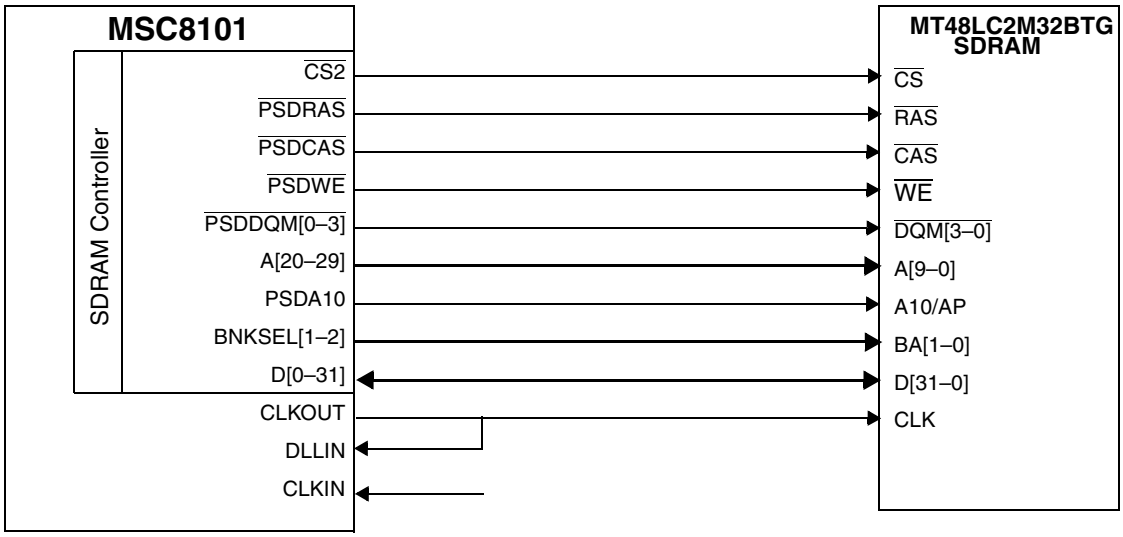


Figure 4-5. MSC8101-To-SDRAM Interconnect in Single-Bus Master Mode

4.4.2 Single-Bus Mode SDRAM Pin Control Settings

This section presents an example 32-bit Micron SDRAM with the following characteristics:

- 8 MByte size comprising internal 512 KB × 32-bit × 4 bank structure
- 8 column, 11 row addresses
- 2 bank address pins (BA[1–0]).

Figure 4-6 shows how the address is split into equivalent row and column addresses of a page-based interleaved configuration (PSDMR[PBI] = 1). In page interleaving, the least significant addresses (immediately below the row address lines) are used for bank select addresses so that interleaving is possible on every page boundary. The two least significant addresses from the MSC8101 are not connected to the SDRAM because of the 32-bit port size.

Two registers configure the main SDRAM. The SDRAM Mode Register (PSDMR) defines the timing and control related parameters, and the Option Register (OR) defines size parameters for the SDRAM. The SDRAM mask OR[SDAM] is set to 0xFF8 and OR[LSDAM] = 00000 to select 8 MB of addressable space for the chip select. The ORx fields are programmed as follows:

- OR[BPD] = 01 for four internal banks.
- OR[NUMR] = 010 for 11 row addresses.

For the page-based interleaving example discussed here, Page Mode Select OR[PMSEL] and OR[IBID] are both set to 0. **Figure 4-6** indicates that the appropriate row start address is A9 using OR[ROWST] = 0110.

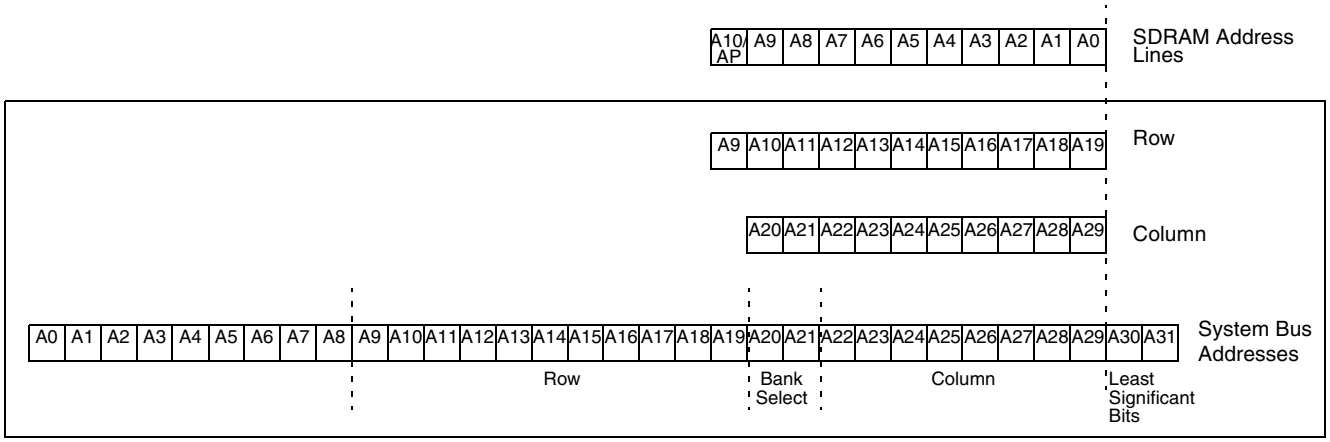


Figure 4-6. MSC8101 SDRAM Address Multiplexing

The PSDMR fields are set as follows:

- The address multiplexing control parameters of the PSDMR register indicate that row addresses A[9–19] are output on the physical address pins A[19–29] during the ACTIVATE command cycle using PSDMR[SDAM] = 010.
- The MSC8101 BNKSEL pins BNKSEL[1–2] are output on A[20-21], so PSDMR[BSMA] = 111. Note that the MSB BNKSEL[0] is not used.
- As **Figure 4-6** shows, the row address A9 must be output on the PSDA10 pin by programming PSDMR[SDA10] = 001.
- The A10/AP is a dual function pin, during an ACTIVATE it functions as an address signal while during a READ/WRITE it acts as an AUTOPRECHARGE control signal.

Table 4-2 summarizes the control settings of the MSC8101 SDRAM controller.

Table 4-2. SDRAM Controller Settings

Register Setting	Description
PSDMR[PBI] = 1	Page Based Interleaving (normal operation) is selected.
PSDMR[SDAM] = 010	Address signals A[5-21] are driven on external system bus pins A[15-31]
PSDMR[BSMA] = 111	Address A[19-21] are output on BNKSEL[0-2]
PSDMR[SDA10] = 001	Address A9 connects to SDA10
OR[SDAM] = 0xFF8 OR[LSDAM] = 0x0	SDRAM Address Mask (8 MB size of SDRAM)
OR[BPD] = 01	4 internal banks per device
OR[ROWST] = 0110	A9 is row start address line
OR[NUMR] = 010	11 row address lines
OR[PMSEL] = 0	Back to back page mode (normal operation)
OR[IBID] = 0	Enables bank interleaving

4.4.3 Single-Bus Mode SDRAM Timing Control Settings

The timing parameters for accessing an SDRAM device must be carefully selected based on a timing analysis between the MSC8101 and the SDRAM that includes any external glue logic required. For Single-Master MSC8101 Bus mode, you can connect the devices directly and verify one set of AC timing characteristics against another. Several programmable timing parameters are available within the MSC8101 SDRAM controller. Typically, these parameters vary according to the associated timing of the SDRAM. When an access misses an 8 ns MT48LC2M32B2 SDRAM page, the read access profile is 5-1-1-1 clocks (10 ns clock period). The write access profile is 3-1-1-1 clocks. The Single-Master MSC8101 Bus mode read access is shown in **Figure 4-7** and the write access in **Figure 4-8.**

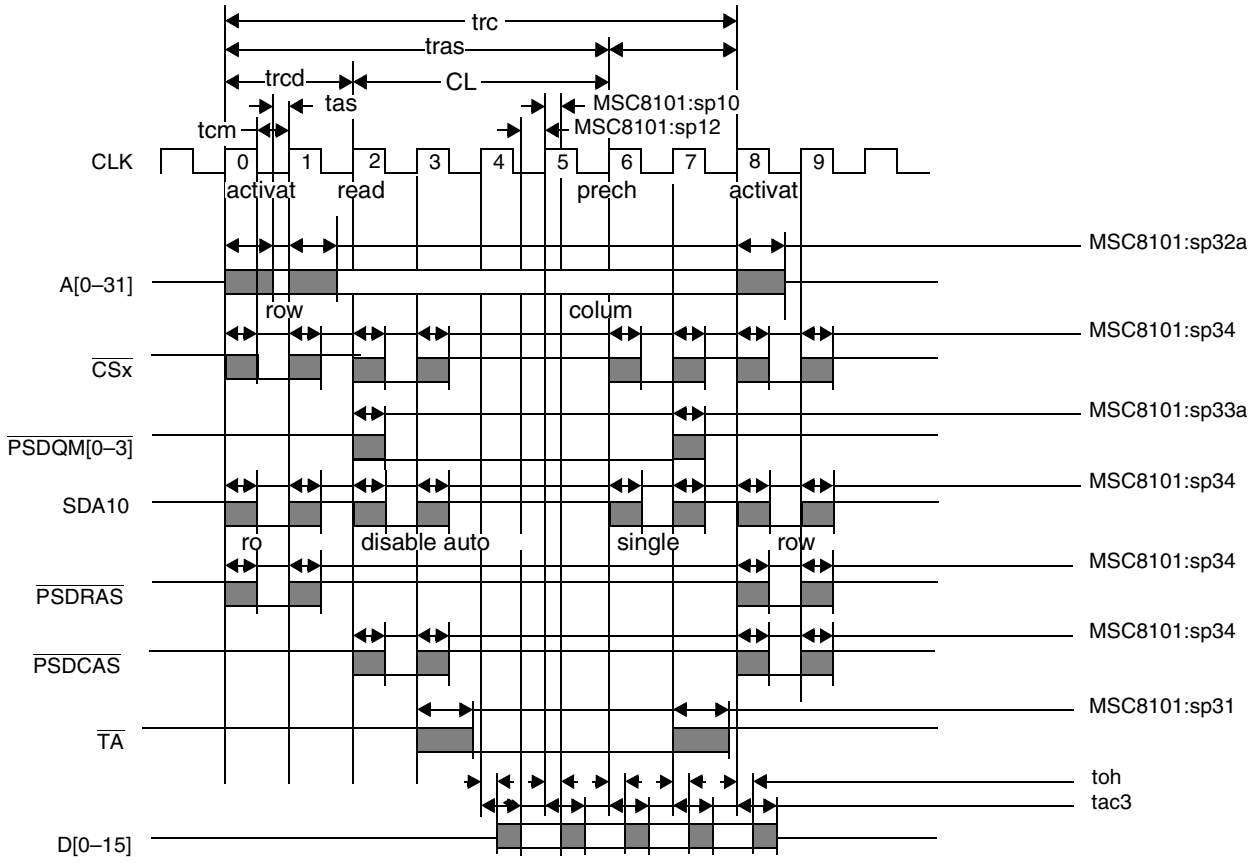


Figure 4-7. SDRAM Burst Read Page Miss, Single Master

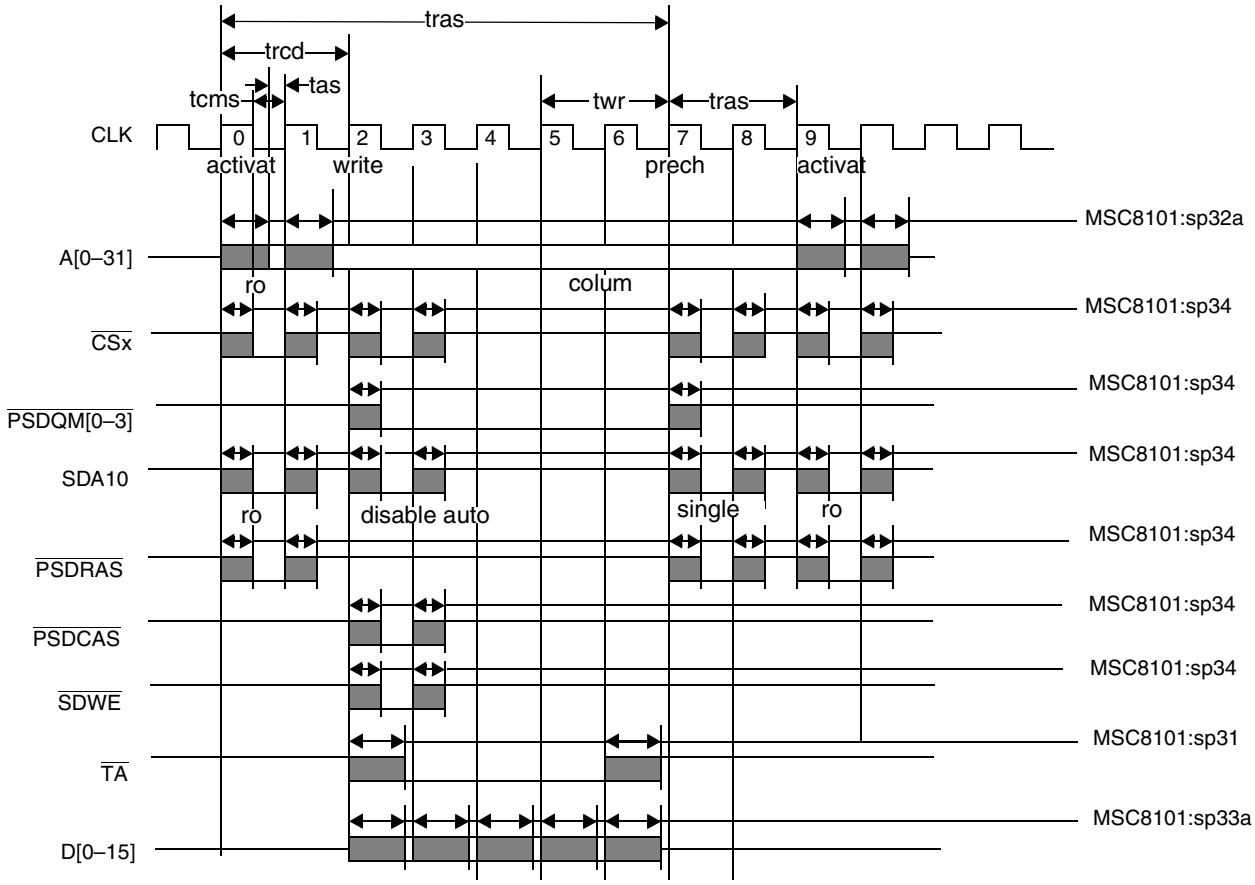


Figure 4-8. SDRAM Burst Write Page Miss, Single Master

For the read access, the SDRAM controller assumptions are $\overline{\text{CAS}}$ latency = 3, Last Data Out to Precharge = –2, and Precharge to Activate = 2 clocks. The underlying assumption here is that the 30pF output timing is used in Single MSC8101 Bus mode to meet the SDRAM address set-up time. Also, an 8 ns (125 MHz) SDRAM is used for aggressive set-up and hold times to meet the MSC8101 specifications.

For write accesses, the SDRAM controller Activate to R/W = 2 and Write Recovery = 2 clocks. The write recovery period defines the earliest time a Precharge can occur after the last data output in a write cycle. For a single-cycle access, the write recovery period must be 2 cycles to meet the overall SDRAM cycle time needs, assuming an Activate to R/W of 2. **Table 4-3** lists the SDRAM timing control values.

Table 4-3. SDRAM Timing Control Values

Register Setting	Description
PSDMR[RFRC] = 110	8-clock, Refresh to Activate minimum period.
PSDMR[PRETOACT] = 010	2-clock, Precharge to Activate/Refresh minimum period.
PSDMR[ACTTORW] = 010	2-clock, Activate to Read/Write minimum period.
PSDMR[BL] = 1	8-beat burst length (for 32-bit memory size).

Table 4-3. SDRAM Timing Control Values

PSDMR[LDOTOPRE] = 10	–2 clock, Last Data Read to Precharge minimum period.
PSDMR[WRC] = 10	2-clock, Last Data Written to Precharge minimum period.
OR[EAMUX] = 0	No external address multiplexing.
OR[BUFCMD] = 0	No external buffered control lines (Normal timing).
OR[CL] = 11	3-clock $\overline{\text{CAS}}$ latency. The minimum period between the SDRAM sampling a column address and the first data out

For operation in Single-Master MSC8101 Bus mode, the Burst Length (BL) of 8 and $\overline{\text{CAS}}$ latency of 3 cycles should be used for compatibility with the specified 32-bit, 8 ns device. The corresponding settings must also be made in the SDRAM itself during initialization.

The memory controller supplies auto-refreshes to the SDRAM according to the interval specified in the SDRAM refresh timer (PSRT) and Memory Refresh Timer Prescaler MPTPR[PTP] registers as follows:

$$RefreshRate = \frac{(PSRT + 1) \times (MPTPR[PTP] + 1)}{BusFrequency}$$

When the refresh timer expires, the memory controller requests the bus. If the request is granted, it issues an auto refresh request using the SDRAM Controller. The value of these registers depends on the specific SDRAM device used and the operating frequency of the MSC8101 system bus. The settings should allow for a potential collision between memory accesses and refresh cycles. The period of the refresh interval must be greater than the access time to ensure that read and write operations complete successfully.

The MT48LC2M32B2TG requires a refresh rate of 15.625 µs, assuming a 100 MHz system bus gives register values of PSRT = 0x31 and MPTPR[PTP] = 0x20.

4.4.4 SDRAM Mode Register Programming and Initialization

SDRAMs must be powered up and initialized in a predefined manner to prevent undefined behavior. Once power is applied and the clock is stable, a JEDEC standard initialization sequence is performed to configure the SDRAM. This is carried out in software utilizing the SDRAM controller Operation field PSDMR[OP]. The sequence required is detailed below:

1. Apply power and start clock.
2. Maintain stable power, stable clock and NOP input conditions for 100 us at the inputs (this time is device specific).
3. Issue PRECHARGE ALL BANKS command to the SDRAM.

⇒Program PSDMR[OP] bits to 101 and then perform a dummy access to the SDRAM.

4. Issue 8 CBR Refresh commands to the SDRAM.

⇒Program PSDMR[OP] bits to 001 and then perform 8 accesses to the SDRAM.

5. Issue Mode Register Set command to program the SDRAM Mode Register.

⇒Program PSDMR[OP] bits to 011 and then performing an access to the SDRAM bank at an address offset to 0x0CC.

6. Issue the Normal Operation command to the SDRAM.

⇒Program PSDMR[OP] bits to 000.

The Mode Register programmed in step 5 is used to define the specific mode of operation of the SDRAM. This definition includes the selection of a burst length, burst type, $\overline{\text{CAS}}$ latency, operating mode and write burst mode. It is programmed via address inputs A[10–0] as detailed in **Figure 4-9**.

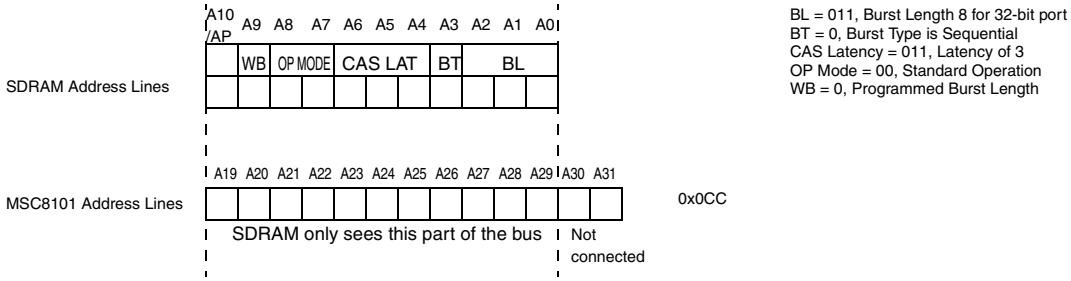


Figure 4-9. SDRAM Mode Register Programming

Note: In multi-master mode the bus master supplies the mode register data on the low bits of the address during the access, for example, the column strobe. The Mode register will power up in an unknown state so it is important that it is programmed prior to applying an operation command.

4.4.5 60x-Compatible Bus Mode SDRAM Hardware Interconnect

In Multi-Master Bus mode, there are separate address and data tenure phases in which the address is not driven for the entire bus transaction, so internal address multiplexing is not used. Therefore, external logic must latch the address and multiplex the column and row addresses to the SDRAM at the appropriate time. The MSC8101 memory controller provides an Address Latch Enable (ALE) and Select pin (PSDAMUX) to control these functions. **Figure 4-10** shows the interconnect between the MSC8101 and the 32-bit Micron MT48LC2M32B2 using page-based interleaving in Multi-Master Bus mode.

The interface signals are essentially the same as for Single-Master MSC8101 Bus mode, apart from the address portion. While separate latch and multiplexer devices could be used, this example uses a 74LVT16260, which has an integrated latch and 24–12 multiplexer. As **Figure**

4-6 shows, addresses A[22–29] are used during column accesses and A[9–19] are used for row accesses. Although address lines A[20–21] are not used as part of the column strobe they should still be connected on the multiplexer as they are required during mode register programming. As A9 is output on the SDA10 pin, the connection to the multiplexer is A[10–19] and A[20–29], respectively. The bank address lines are output on the BNKSEL lines. The LE pin latches the 1B and 2B inputs on the falling edge of the ALE signal; the MSC8101 PSDAMUX pin determines the output of the multiplexer.

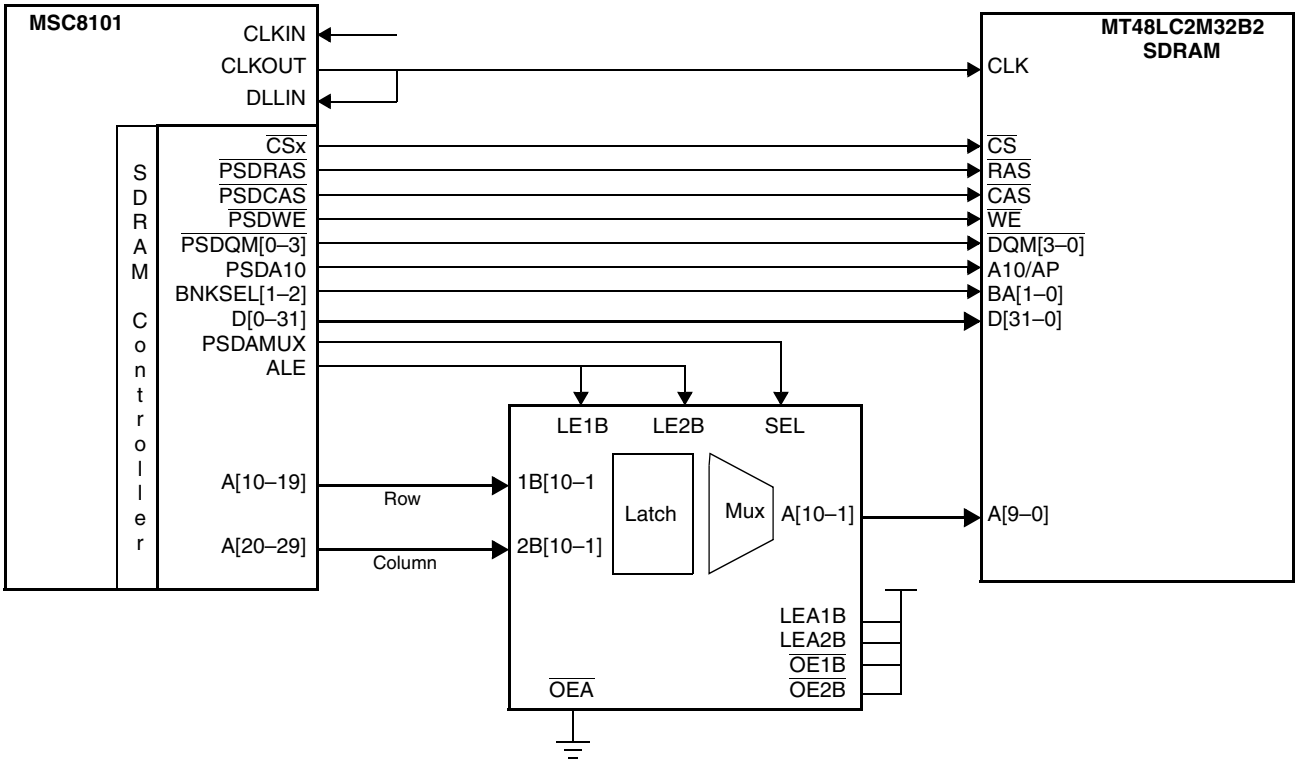


Figure 4-10. MSC8101-To-SDRAM Interconnect in 60x-Compatible Multi-Master Bus Mode

4.5 Connecting to the MSC8101 HDI16 Memory Interface

The MSC8101 HDI16 host port is a programmable 16- or 8-bit wide parallel port that gives an external host device an access window for data transactions. A parallel interface between the external system bus of an MSC8101 host and a target MSC8101 HDI16 slave port is often used as a control and data path. One use of the HDI16 interface is to download bootstrap code at start-up. The MSC8101 is a RAM-based device, so at power-up the DSP initialization instructions must be provided directly by an external ROM or the code must be downloaded into the internal RAM ready for execution. Typically the HDI16 places code/data into the MSC8101 internal DSP RAM under complete control of an MSC8101 external host.

Once the HDI16 initialization code is downloaded, a branch is made to the start of the downloaded initialization code and execution begins. When the DSP is initialized, the MSC8101 external host can use the same HDI16 host port to control and schedule the ongoing DSP tasks.

The HDI16 offers various programmable options for chip selects and data strobe modes. When the HDI16 interface downloads initialization bootstrap code, it is important to select a strobe implementation that the MSC8101 HDI16 interface supports straight from reset. One option is to select the dual strobe, 16-bit bus mode by setting the H8BIT pin = 0 and $\overline{\text{HDDS}}$ pin = 1 at $\overline{\text{PORESET}}$ release.

One of the most important HDI16 host port hardware features is that it is an asynchronous interface, which reduces concerns over clock skew between an external bus master such as the system bus and the HDI16 host port. Since all the host-side registers are accessed by asserting a single chip select and four address lines, the HDI16 MSC8101 host port can be regarded an asynchronous memory-mapped region. In dual strobe mode, the MSC8101 bus master simply asserts a chip select and a data read (or write) strobe to validate an HDI16 access. The hosts read or write strobe is the data latch control to complete the bus transactions, eliminating the need for any handshake signal back from the HDI16 target. Selecting the appropriate modes, an MSC8101 host can fully support the HDI16 feature set through the UPM-controlled signals. The UPM-defined interface can be used with any (or all) of the MSC8101 chip selects to give a 16/8-bit port size and strobe generation matching that of the HDI16 host port.

The standard MSC8101 GPCM can meet the timings required by the HDI16 gluelessly where ACS = 00 and CSNT = 1. However, this ACS setting requires that the address and chip select be driven active on the first rising clock edge of the bus cycle. This requirement may cause an issue in some systems that use data buffering, in that slower devices may not have stopped driving data from the one access before the next. Therefore, while a GPCM chip select is a potential solution, the UPM is illustrated here as a more general solution.

4.5.1 HDI16 Hardware Interconnect

The interconnection of the MSC8101 UPM-controlled system bus operating in Single-Master MSC8101 Bus mode with the MSC8101 HDI16 port is completely glueless, see **Figure 4-11**. Each HDI16 slave interface requires at least one chip select from the MSC8101 host in order to access the memory map of each device separately. During start-up, a common second chip select can enable all HDI16 slaves to receive broadcast bootstrap code. Regardless of the chip select used, the MSC8101 Byte Strobe 0 [BS0n] generates the HDI16 Write Data strobe (HWRn), which must be asserted every 16-bit write transaction. The general-purpose line PGPL2 is programmed high in a UPM write cycle and low for a UPM read cycle to generate the MSC8101 HDI16 read strobe line (HRDn). These pins correspond to the GPCM-equivalent memory control pins so that the same pins can use either GPCM- or UPM-derived memory control. Operating in dual strobe mode and with active low signal polarity, the HDI16 can be accessed directly by the MSC8101 host.

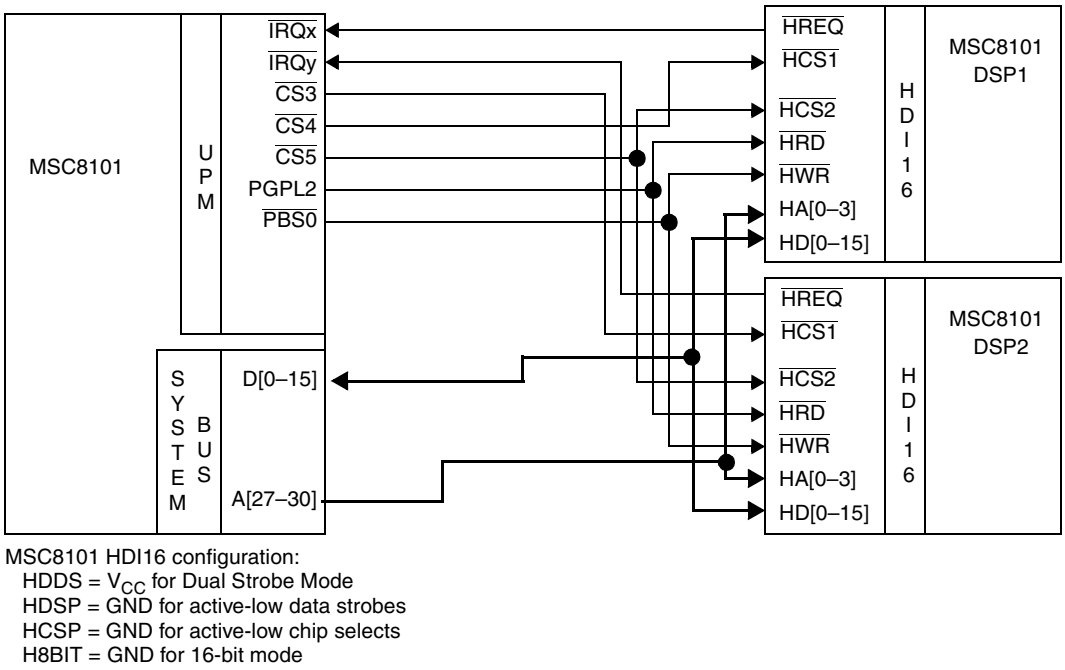


Figure 4-11. Single Master MSC8101 Bus to HDI16 Interface

The timing of the single-master interface allows for 5-cycle read and write accesses between a system bus operating in Single-Master MSC8101 Bus mode and an HDI16 slave. When the system bus operates in Multi-Master Bus mode, arbitration is demanded between the multiple masters to determine which one gains access to the bus. As part of this multi-master bus protocol, the bus must be idle between separate transactions, adding at least one clock cycle of latency. Since the pipelined address must be latched, an external address latch device is required. See **Figure 4-12** for an example of a system bus operating in Multi-Master mode interconnected to a buffered HDI16 interface.

For a few DSPs, the MSC8101 data bus can connect directly to the DSP host port without any glue logic. However, because many applications deal with a multi-DSP concept, appropriate buffering must be added to meet the capacitive loading requirements and make the solution more scalable to a larger bank of DSP devices. The same UPM programming can be used for both buffered and unbuffered systems. The MSC8101 Buffer Control lines ($\overline{\text{BCTL[0–1]}}$) control the direction and output enable of the 74LCX245 bidirectional buffer. To enable $\overline{\text{BCTL1}}$ set SIUMCR[CS5PC] = 1, and for an active low output enable, SIUMCR[BCTLC] = 00. Again, the $\overline{\text{BCTL[0–1]}}$ lines are used so that the same system bus-to-HDI16 interconnect can also be controlled via a GPCM machine.

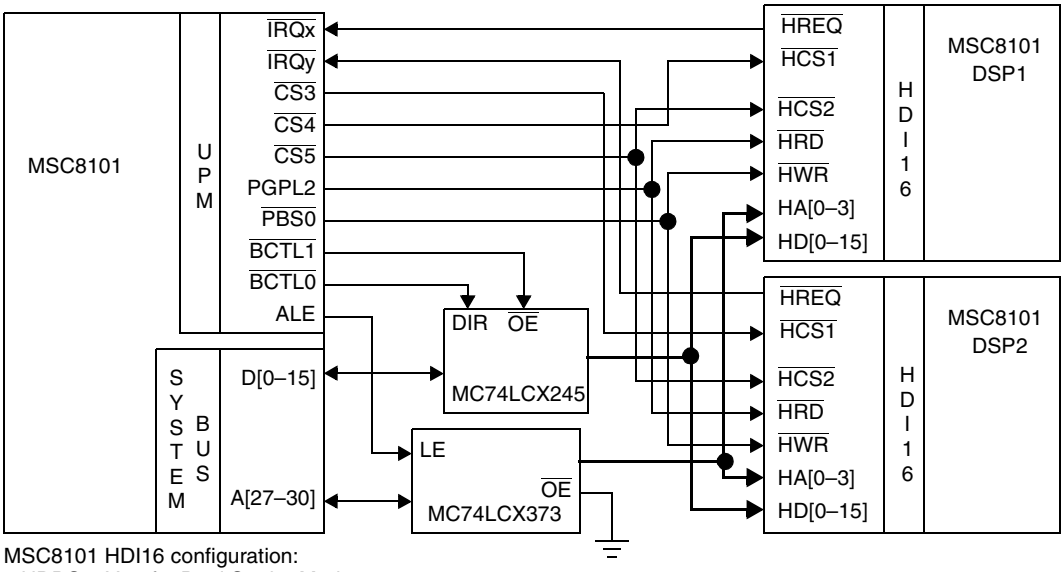


Figure 4-12. Multi-Master System Bus to Buffered HDI16 Interface

4.5.2 Single-Bus Mode HDI16 Timing Settings

The UPM offers some extremely flexible memory control options by which the memory control signals can be controlled to one quarter of one clock resolution. However, depending upon the CPM:Bus clock ratio, the relative phases of this one quarter of one clock granularity may vary. There may be cases where the timing needs change with different clock ratios. To ensure that the timing recommendations developed here hold true at any clock speed or ratio, the analysis is performed using the maximum bus clock of 100 MHz and using only the invariable one half of one clock boundaries (T1 and T3) to change signals. Therefore, the recommendations hold true for anything less than a 100 MHz bus clock.

The UPM-controlled HDI16 read access is illustrated in **Figure 4-13** and the write access in **Figure 4-14**. Both the read and write accesses on the system bus operating in Single-Master MSC8101 mode can be accessed within five clocks.

During a read access, the data into the MSC8101 is latched on the falling edge rather than on the usual rising clock edge. The advantage of this change is a sufficient timing margin to incorporate a further data buffer data delay with the same timing settings still in effect. The DLT3 bit must be set in the corresponding UPM word to indicate the data latch point on the falling clock (falling edge of clock corresponds to rising edge of T3), and MxMR[GPL_x4DIS] must be set to enable this mode.

Furthermore, the read and write strobe negation times are readily met with the illustrated UPM configuration, and this is difficult to achieve with a competitive memory access profile in the alternative GPCM-controlled case.

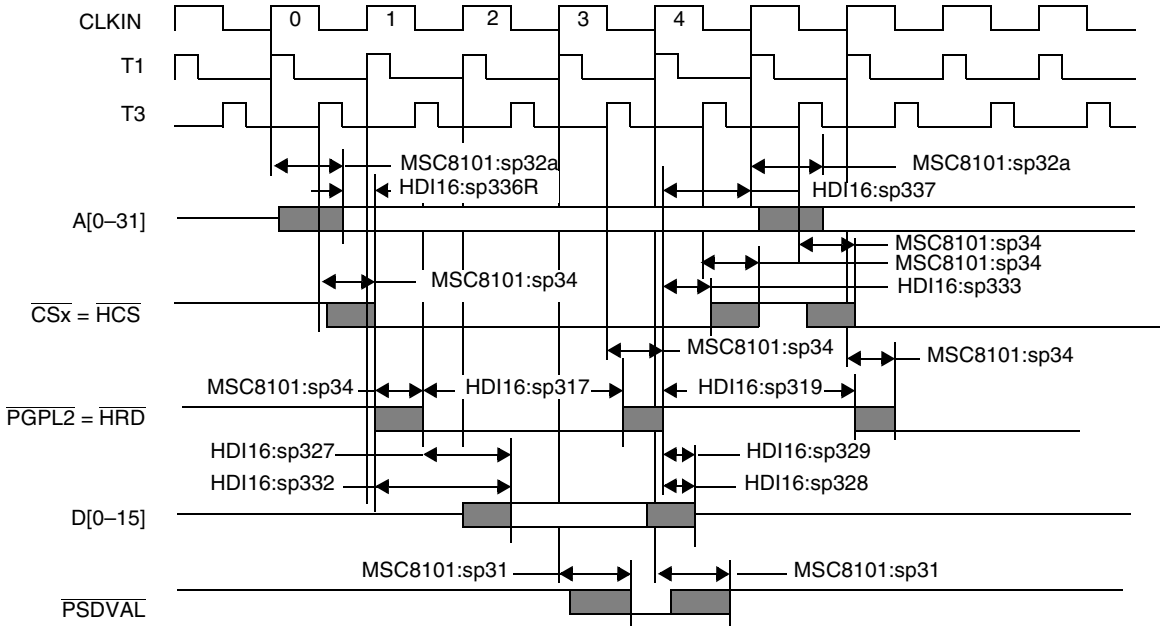


Figure 4-13. HDI16 UPM Read, Single Master

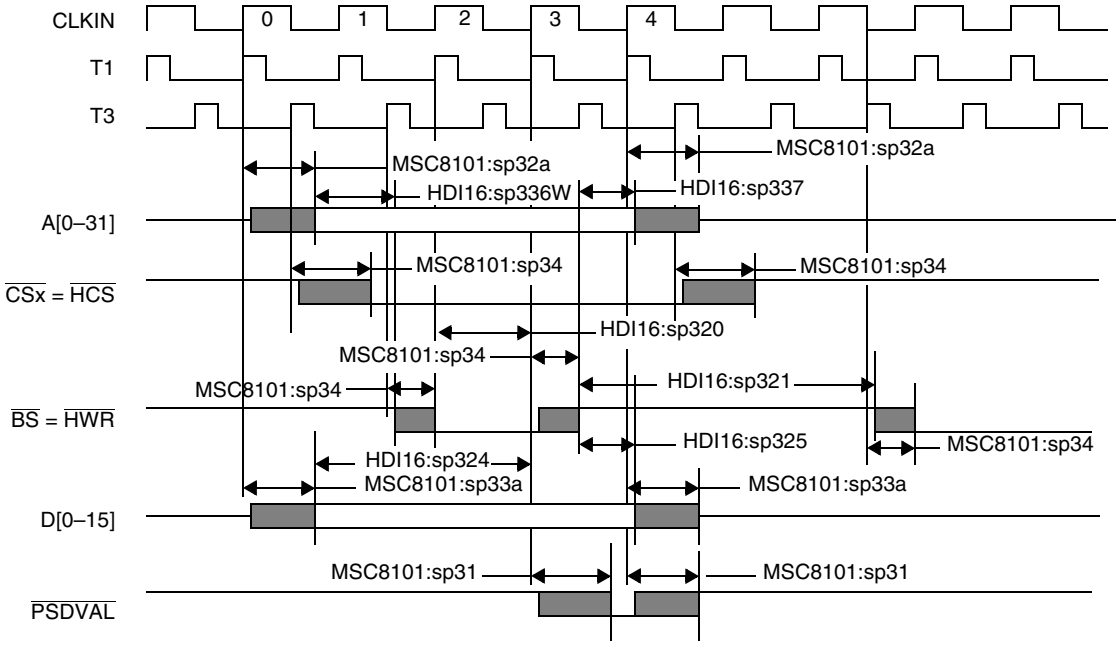


Figure 4-14. HDI16 UPM Write, Single Master

Balancing Between the System and Local Buses 5

The MSC8101 combines the SC140 core with a 60x-compatible system bus and the PowerQUICC II Communications Processor Module (CPM). The system bus makes it possible to place an MSC8101 device directly on an existing 60x-compatible system bus. The MSC8101 has two 64-bit buses: the system bus and the local bus. The system bus accesses external memory and any external bus resources. The local bus provides efficient communication between the MSC8101 SC140 core and the SIU and CPM. This chapter describes the functions of these two buses and their interaction. **Table 5-1** compares the features of the system bus and the local bus. **Figure 5-1** shows the MSC8101 block diagram. Notice the system and local buses in the SIU portion of the diagram.

Table 5-1. Features of the System Bus and Local Bus

System Bus	Local Bus
64/32-bit data and 32-bit address	64-bit data and 32-bit address
Data width selectable in software: 64-bit mode or 32-bit mode	
Support for multiple-master designs	
Support for four-beat burst transfers	Support for four-beat burst transfers
Port size of 64, 32, 16, and 8 bits controlled by the internal memory controller	Port size of 64, 32, 16, and 8 bits
Support for data and address parity	Support for data and address parity
Can access external memory expansion or external peripherals, or can enable an external host device to access internal resources	No external access

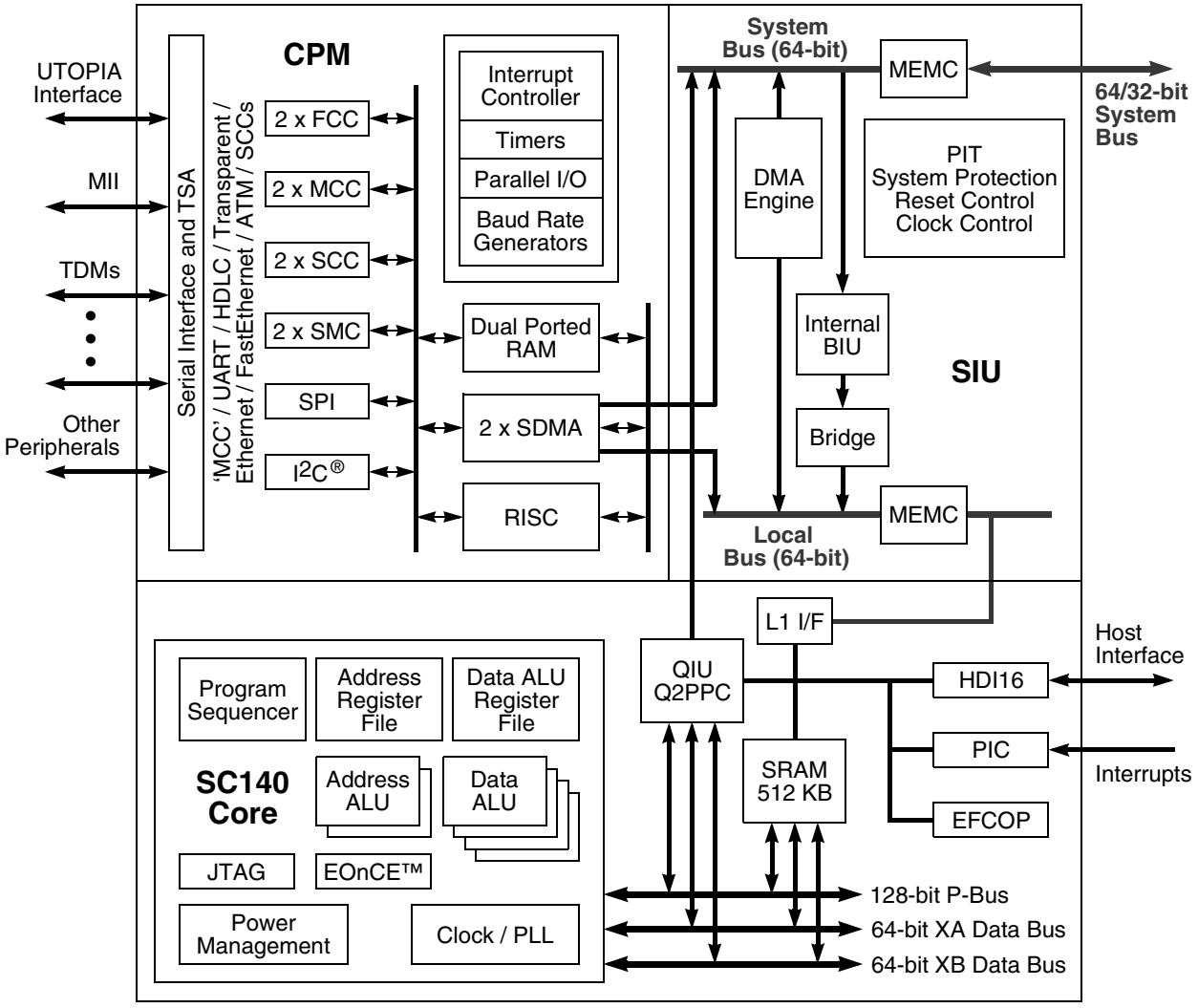


Figure 5-1. MSC8101 Block Diagram

5.1 System Bus

Because the system bus has external access from the device, the MSC8101 can be part of a system communication via the system bus. The MSC8101 can act as a master on a system bus, directing other system devices for a given application. Also, the MSC8101 can act as a slave on a system bus so that another device on the bus acts as a master. The system bus master can gain access to the MSC8101 memory map and direct MSC8101 functions as needed for a given application.

The MSC8101 memory controller uses the system bus to access external memories. **Figure 5-2** shows one example of an interface to external memories that uses the general-purpose chip-select machine (GPCM) and user-programmable machine (UPM) memory controllers. Since the local bus does not have access external to the MSC8101, any external memory accesses must use the system bus.

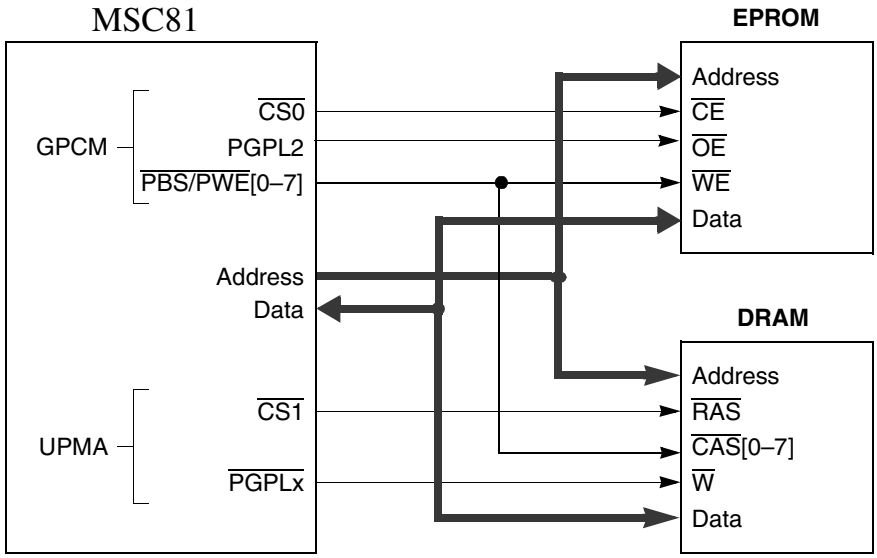


Figure 5-2. System Bus External Memory Access Example

In addition to the system bus and the local bus, the MSC8101 contains the QBus, which is the SC140 core interface. It handles all communication between the SC140 core and the peripherals: boot ROM, PIC, EFCOP, and the HDI16 host interface. It also connects to the system external bus interface via a interface called the QIU. **Figure 5-3** shows the QBus interface. Notice that the QBus does not have access to the local bus.

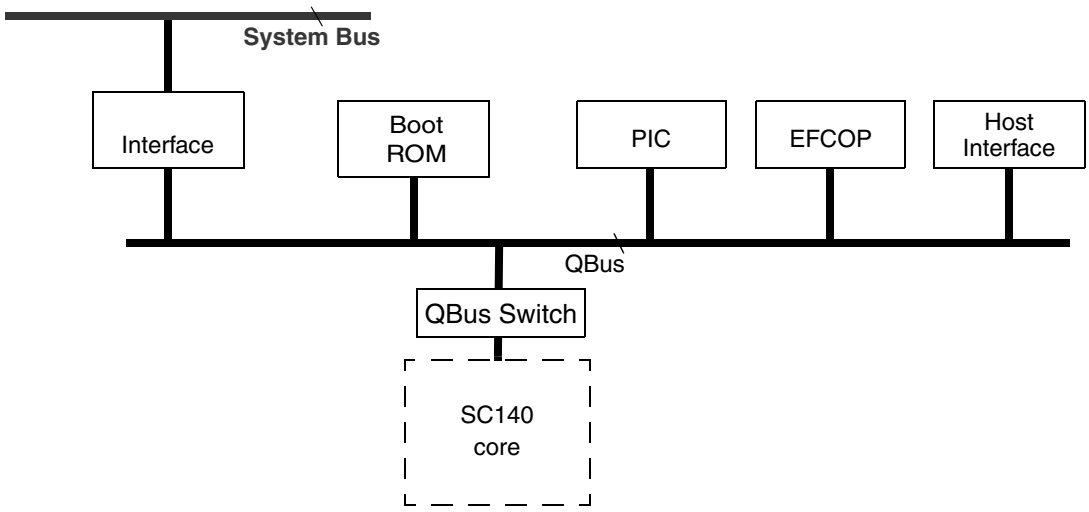


Figure 5-3. QBus-to-System Bus Interface

Access from the SC140 core to the QBus goes through a QBus switch. Since the QBus is part of the core and is clocked at the same speed as the core, accesses to banks on the QBus are fast. However, there is some latency inherent in accessing the QBus banks. For example, it takes three DSP core clocks to access an EFCOP register from the SC140 core.

When the SC140 core initiates an access, any transaction that does not match the SC140 core internal address space or the QBus banks is routed to the system bus. This routing occurs through the interface on the QBus. Accesses to the system bus can take 16–18 core clock cycles. For example, if the SC140 core needs to access a DMA register on the system bus, it requires 2–4 core clocks to get through the QBus switch and interface to the system bus, 4 bus clocks (12 core clocks assuming a 3:1 core/bus clock ratio) for a fast transaction on the system bus, and 2 core clocks to complete the transaction. Although the QBus interface enables the SC140 core to access information external to the SC140 core, such QBus access are costly because each access through the interface requires a significant number of cycles to complete. You should use system bus accesses from the QBus sparingly so that the focus of the SC140 core remains on DSP data processing tasks.

5.2 Local Bus

The MSC8101 local bus is the interface between the SC140 core and the SIU and CPM blocks. DMA data exchanges between the core peripherals (HDI16 and EFCOP), SRAM, and other modules of the MSC8101 occur through the local bus. As the MSC8101 block diagram in **Figure 5-1** shows, the local bus does not have direct access to the SC140 core. **Figure 5-4** shows the local bus interface to the core.

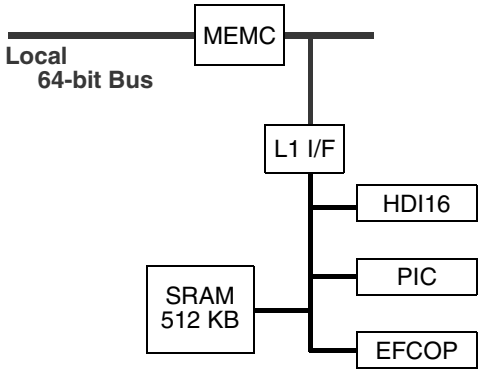


Figure 5-4. Interface From Local bus to the SC140 Core

5.3 Bus Interaction

The buses reside in the SIU portion of the MSC8101. The local bus is synchronous to the system bus and runs at the same frequency as the system bus. Three SIU components interact with the two buses: the bridge, the memory controllers for each bus, and the DMA engine. **Figure 5-5** shows a block diagram of the SIU.

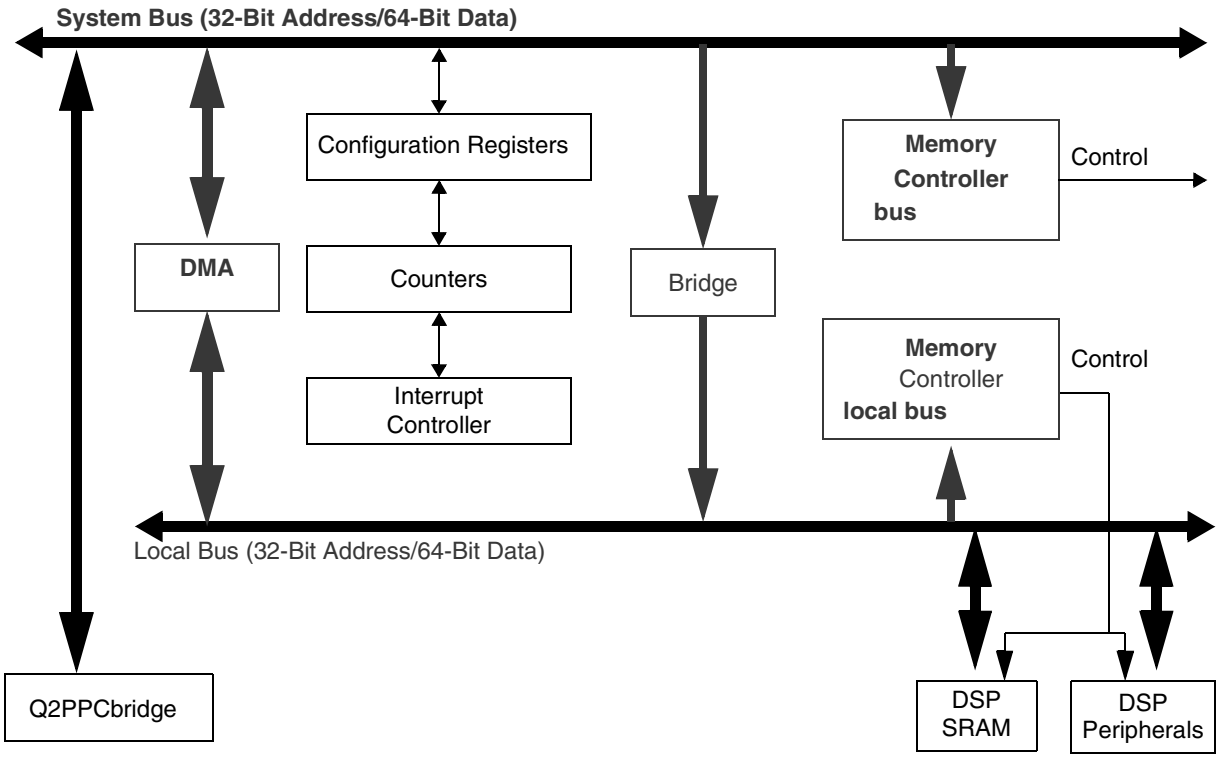


Figure 5-5. SIU Block Diagram

The bridge between the system bus and local bus allows a master on the system bus access to memory-mapped devices residing on the local bus. An external master can send information to the HDI16 port or EFCOP through the system external bus interface. Assuming bus mastership, it takes five bus clocks for a host to transmit data through the bus bridge to the local bus. Since this bridge is unidirectional, the peripheral modules cannot send data to the system bus through the bus bridge. The SC140 core must use the DMA channels to transmit data back to the system bus.

Figure 5-6 shows the interaction between the system bus and local bus through the memory controller. No data flows between the buses through the memory controller. The external system bus has eight memory banks (Bank 0–7); the internal local bus has two memory banks (Bank 10 and Bank 11). Banks 0–7 are allocated to the system bus. User-Programmable Machine C (UPMC) controls Bank 10, which is assigned for the internal DSP RAM. The GPCM controls Bank 11 which is assigned to the DSP peripherals EFCOP and HDI16. The GPCM can also control external memories on the system bus. Bank 8 and Bank 9 are reserved for future use.

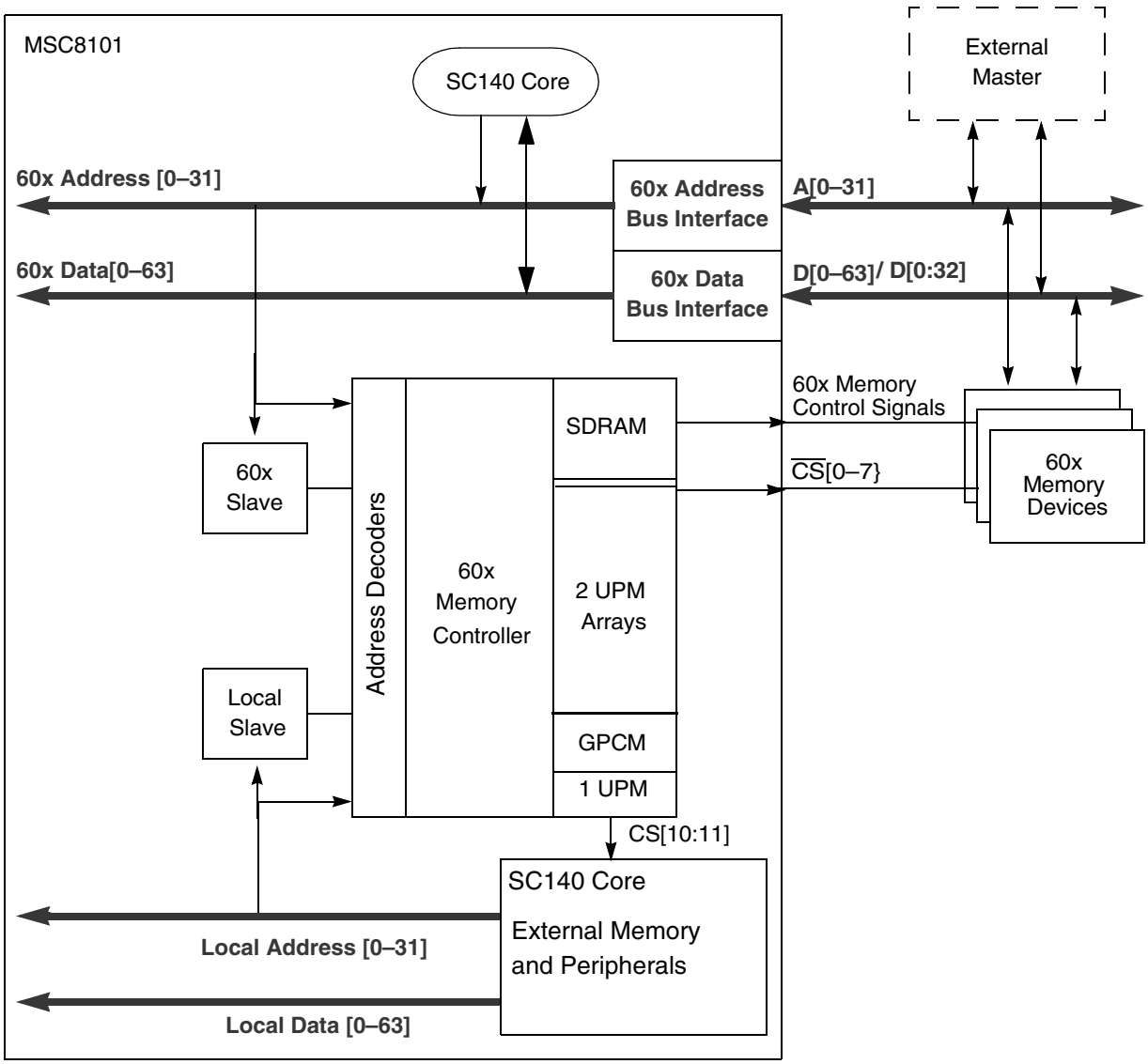


Figure 5-6. Bus Architecture

Each memory bank can be assigned to any one of the SDRAM, GPCM, and UPM machines via BR*x*[MSEL]. For UPM machines, the M*x*MR[BSEL] bit is configured to assign banks to the system bus or the local bus. The memory controller machine selection is shown in **Figure 5-7**.

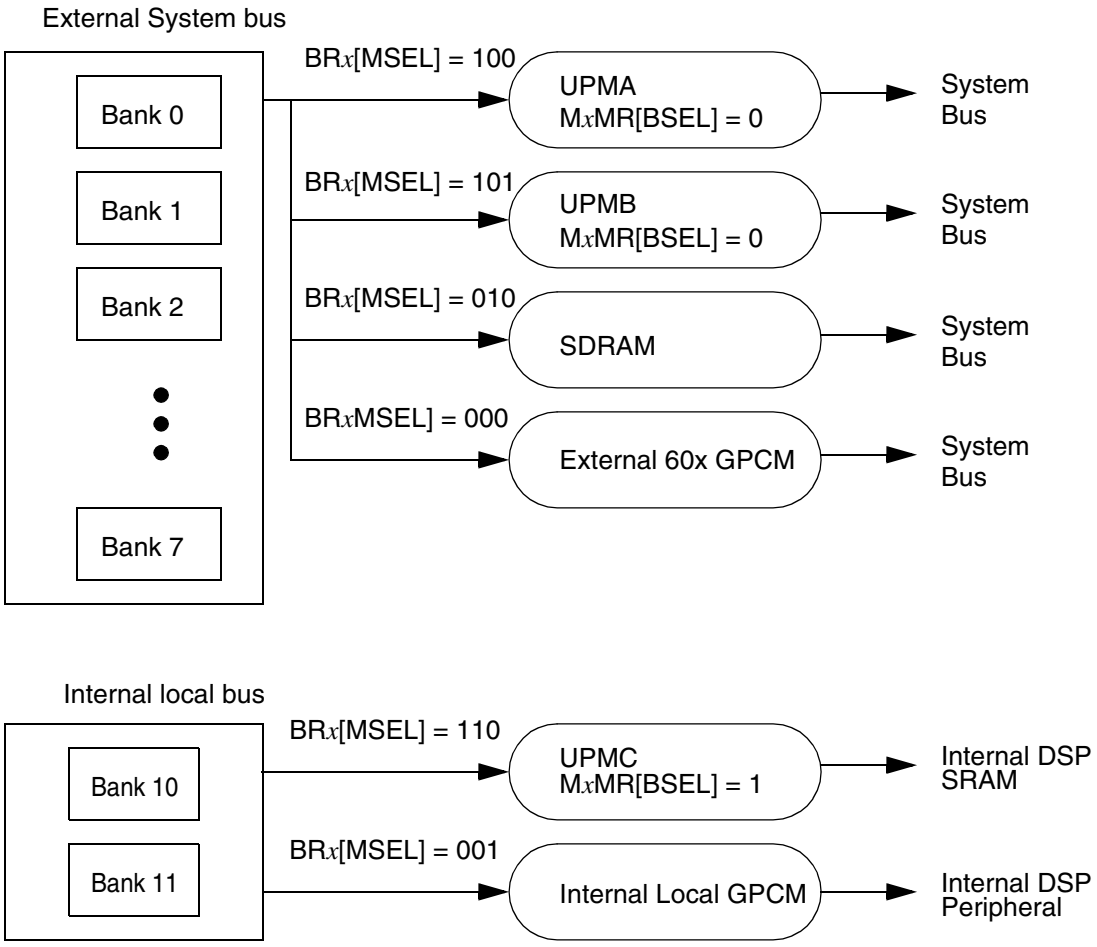


Figure 5-7. Memory Controller Machine Selection

Addresses are decoded by comparing the values on the address pins A[0–16] with the following values:

- *Base address*. Specified by BR*x*[BA]. The base address indicates the start address of the bank in memory; it specifies the upper 17 bits of the base address register.
- *Address mask*. Specified by OR*x*[AM] or OR*x*[SDAM]. The address mask determines the bank size. In the UPM or GPCM mode, OR*x*[AM] specifies the 17-bit address mask. Any set bit causes the corresponding bit to be used in comparison with the address pins. Any cleared bit masks the corresponding address bit. In SDRAM mode, OR*x*[SDAM] specifies the SDRAM address mask and OR*x*[LSDAM] specifies the lower SDRAM address mask.

If an address match occurs in multiple banks, the lowest-numbered bank has priority. If the system bus attempts to access a bank allocated to the local bus, the access is transferred to the local bus. local bus access attempts to system assigned banks are ignored.

5.3.1 DMA Controller

The multi-channel DMA controller connects to both the system bus and the local bus. Data from the core transfers from the local bus to the system bus and *vice versa*.

5.3.1.1 Selecting a Bus

The DMA Channel Configuration Register DCHCR*x*[PPC] bit selects the bus associated with the channel. Clearing this bit assigns the channel to the local bus, and setting this bit assigns the channel to the system bus. For example, when data is transferred from an external peripheral on the system bus to the DMA FIFO, the DCHCR*x*[PPC] bit is set to select the system bus. When data is transferred from an internal peripheral such as the EFCOP, which is located on the local bus to internal SRAM, the DCHCR*x*[PPC] is cleared to select the local bus.

5.3.1.2 DMA FIFO

The DMA uses a FIFO for its data transfers. Therefore, one bus can transfer its data to the DMA FIFO and be free of the data transaction instead of waiting with the data until the other bus is free. For example, in a transfer from the EFCOP data output register to memory on the system bus, the data is transferred on the local bus to the DMA FIFO, and the local bus is released. Subsequently, the DMA arbitrates for access to the system bus. When access is granted, the DMA transfers the data from the DMA FIFO to external memory, completing the transfer. The local bus does not have to wait for access to the system bus before it can execute a second transfer, which could be from the CPM to internal SRAM, for example.

5.3.1.3 Chained Buffers

A chained buffer is a type of buffer that jumps to the address of the next buffer when its size reaches zero. If the buffers use different buses—for example, one buffer maps to the system bus while the other maps to the local bus—the flush option should be used to prevent out-of-sequence transactions from crossing the buses. When data in the FIFO is flushed, data is transferred to the destination. The Buffer Attributes BD_ATTR[FLS] bit configures the behavior of the DMA FIFO when BD_SIZE reaches zero. Clearing this bit does not flush the FIFO, and setting this bit flushes the FIFO.

5.3.1.4 Bus Errors

A non-maskable interrupt is generated and the DMA TEA Status Register (DTEAR) is updated whenever a system bus or a local bus error occurs on a DMA access.

The DTEAR[DBER_P] bit is set when a system bus error occurs. The DMA transfer error address is read from the DMA Transfer Error Address (PDMTEA) Register. The channel that caused the error is read from the DMA Transfer Error Requestor Number Register, PDMTER[RQNUM]. The DTEAR[DBER_L] bit is set when a local bus error occurs. The DMA transfer error address is read from the Local DMA Transfer Error Address (LDMTEA) Register.

The channel that caused the error is read from the Local DMA Transfer Error Requestor Number Register, LDMTER[RQNUM].

5.3.2 SDMA Channels

The CPM uses its own SDMA channels to transfer data to and from the CPM and the rest of the MSC8101. The CPM SDMA channels are separate from the DMA channels in the SIU. Each SDMA channel has access to a bus. SDMA1 accesses the system bus so that a master on the system bus can send information to the CPM regarding incoming data on a CPM port. Data from the CPM can be sent directly to the core for processing using the SDMA2 channel and the local bus. **Figure 5-8** shows the interaction of the SDMA module with the system and local buses.

Although the CPM has only two physical SDMA channels, the communications processor (CP) within the CPM can implement many dedicated virtual SDMA channels for each FCC, MCC, SCC, SMC, SPI, and I^2C. Each channel is permanently assigned to service either the receive or transmit operation of an FCC, MCC, SCC, SMC, SPI, or I^2C. As **Figure 5-8** shows, data from the peripheral controllers can be routed to external RAM using the system bus (path 1). To route data to internal SRAM, the local bus (path 2) must be used.

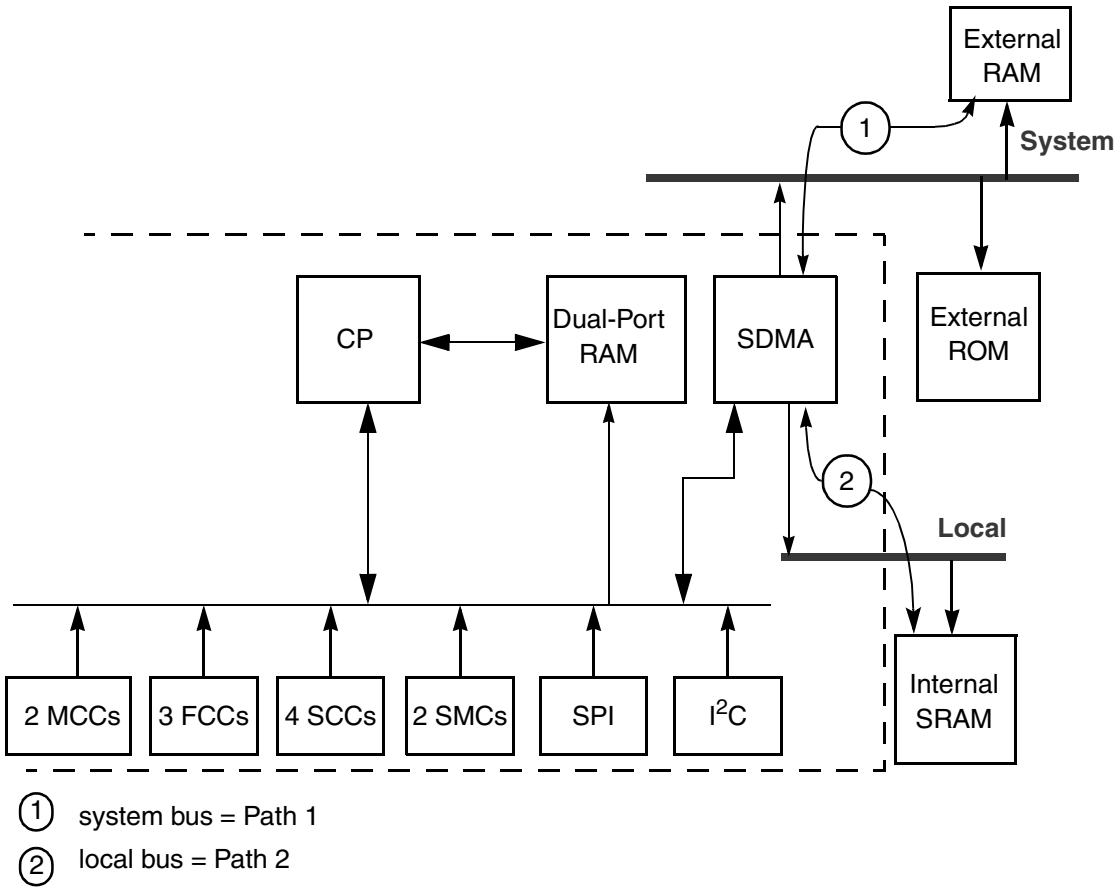


Figure 5-8. SDMA Data Paths

On a path 1 access, the SDMA channel must acquire the system bus. On a path 2 access, the local bus is acquired and the access is not seen on the external system bus. Thus, the local bus transfer occurs at the same time as other operations on the external system bus. SDMA access times to memory on the buses varies depending on the memory used. One example of transfer time is from the CPM to internal SRAM. A single access to internal SRAM from the CPM using the local bus requires 4 bus clocks.

5.3.2.1 Bus Errors from SDMA access

If a system bus or local bus error occurs during a CP-related access by the SDMA, the CP generates a unique interrupt in the SDMA Status Register (SDSR). The interrupt service routine then reads the appropriate DMA transfer error address register (PDTEA for the system bus or LDTEA for the local bus) to determine the address at which the bus error occurred.[1] The channel that caused the bus error is determined by reading the channel number from the SDMA Transfer Error MSNUM Registers, PDTEM and LDTEM. If an SDMA bus error occurs on a CP-related transaction, all CPM activity stops and the entire CPM must be reset in the CP Command Register (CPCR).

5.3.2.2 SDMA Bus Arbitration and Bus Transfers

On the MSC8101, theSC140 and SDMA can become external bus masters. Therefore, any SDMA channel can arbitrate for the bus against the other internal devices and any external devices present. Once an SDMA channel becomes system bus master, it remains bus master for one transaction (which can be a byte, half-word, word, burst, or extended special burst) before it releases the bus.

1. For details on the SDMA Transfer Error Address Registers (PDTEA and LDTEA), consult the “SDMA Programming Model” section of the chapter on SDMA Channels in the *MSC8101 Reference Manual*.

DMA Channels 6

The internal direct memory access (DMA) controller channels transport data between the various modules of the MSC8101 so that the data is available for processing when needed. The DMA controller can transfer data to and from memory or enable communication between peripherals directly without passing through memory. The DMA requestors can be accessed by one of the following:

- Host interface (HDI16), internal peripheral
- Enhanced filter coprocessor (EFCOP), internal peripheral
- External Peripherals (up to 4)
- DMA FIFO (each channel is a requestor)

This chapter describes the purpose and use of DMA on the MSC8101 and provides numerous programming examples.

6.1 DMA Programming Basics

The MSC8101 DMA controller is located in the system interface unit (SIU) between the system and local buses (see **Figure 6-1**). The DMA controller transfers data to and from the system bus, the local bus, or between the two buses, so it is a highly flexible data transfer mechanism. The MSC8101 DMA controller handles hot swap operation, so it can service one channel in the current clock cycle and service a different channel in the following clock cycle with no addition of wait states or delay between the two. Hot swap thus increases the efficiency of the DMA transfers.

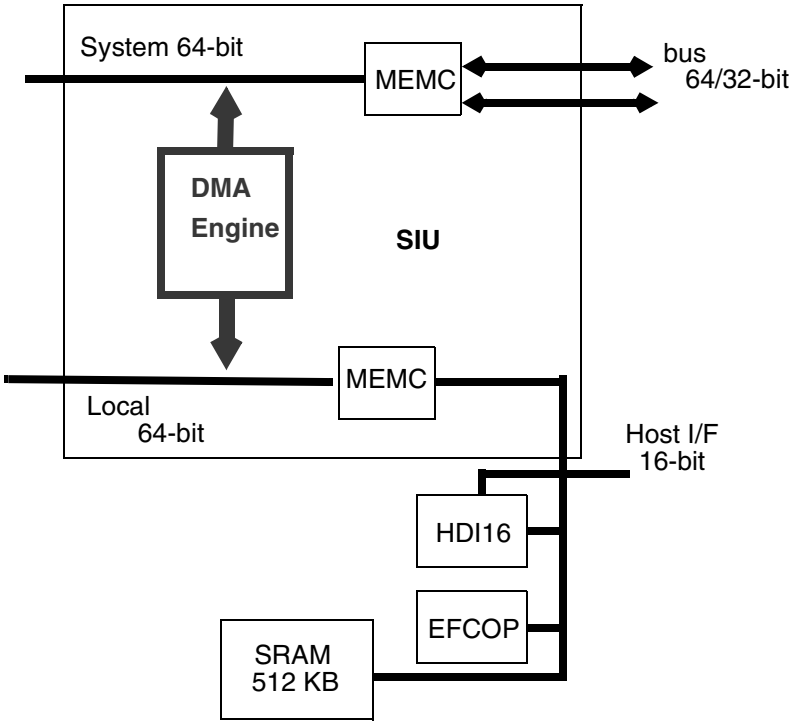


Figure 6-1. DMA Engine Interfaces

6.1.1 Operating Modes

The DMA controller operates in two modes: Normal (dual-access) mode and Flyby (single-access) mode.

6.1.1.1 Normal Mode (Dual Access)

In Normal mode, data is read from the source and written to the destination through the DMA FIFO. Two DMA channels are required in this mode (as illustrated in **Figure 6-2**), so it is called a “dual-access.” The even-numbered channel is always the read path, and the odd-numbered channel defines the transaction write path. For example, if channel 4 were selected to read data from external memory to the DMA FIFO, then channel 9 could not be used to write the data from the DMA FIFO to the EFCOP. Instead, DMA channel 5 must be used.

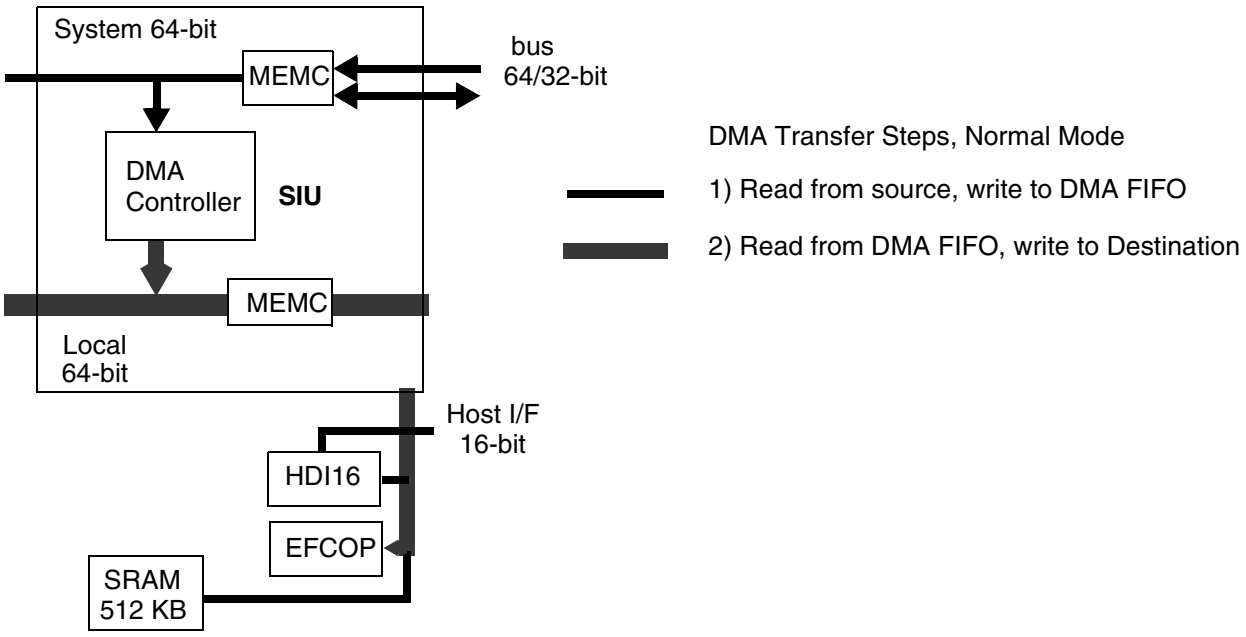


Figure 6-2. Normal Mode Example

6.1.1.2 Flyby Mode (Single Access)

Flyby mode does not require two DMA channels to complete a data transfer between a peripheral and a memory module. In this mode, the transaction occurs between two resources with the same port size[1] on the same bus so that it can be executed by a single channel without going through the DMA FIFO. The read cycle data is transferred directly "on the fly" to its destination.

There are constraints on which DMA channels can be used during a flyby transaction. If the transaction is a read transaction from memory, then an even-numbered DMA channel must be programmed for the transfer. If a write transaction to memory is required, an odd-numbered DMA channel must be programmed. The channel must be programmed to external request mode by clearing the (DCHCRx[INT]) bit, and the corresponding BD_ADDR field is programmed to the memory address. The DCHCR requestor number field points to the peripheral.

6.1.2 Transfer Types

Each DMA channel is configured in one of six possible ways. This section describes each configuration.

1. Port size is programmed in the BRx registers. It can be 8, 16, 32, or 64 bits.

6.1.2.1 Memory to DMA FIFO

A memory transfer to the DMA FIFO can occur from either external or internal memory. **Figure 6-3** shows both possibilities. Note that in memory/DMA FIFO transactions, both directions, the channel is programmed to internal request mode (DCHCRx[INT] = 1).

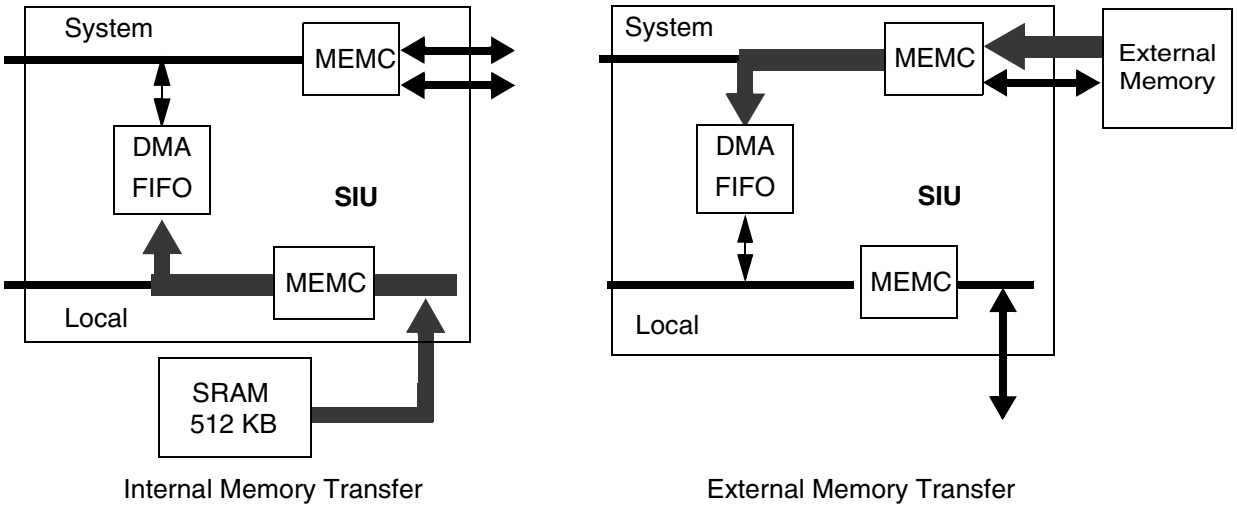


Figure 6-3. Memory to DMA FIFO Transfers

6.1.2.2 DMA FIFO to Memory

A DMA FIFO to memory can access either external or internal memory. **Figure 6-4** shows both possibilities.

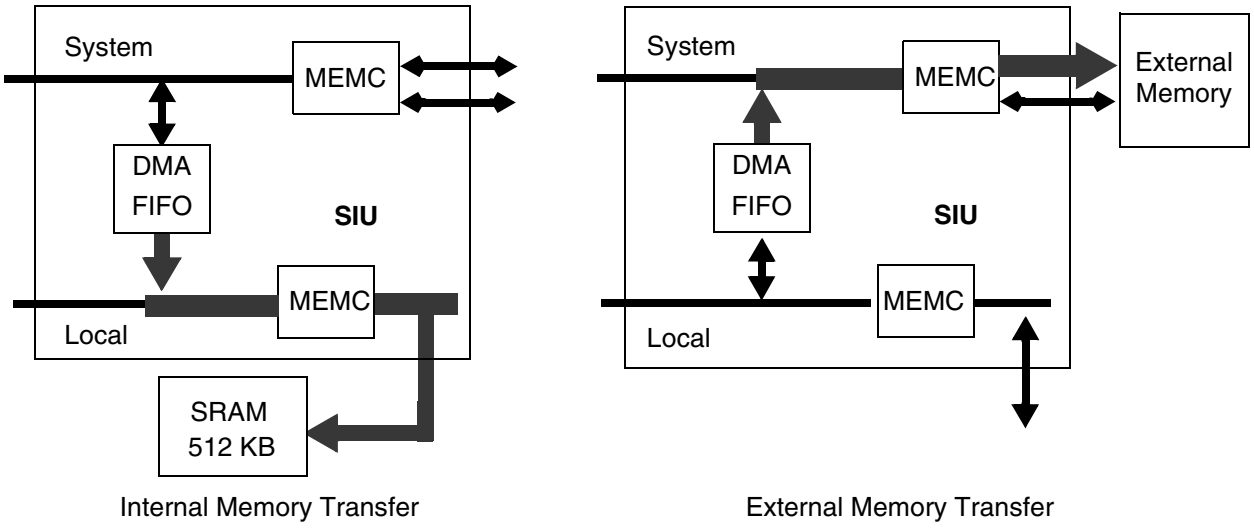


Figure 6-4. DMA FIFO to Memory Transfers

6.1.2.3 Peripheral to DMA FIFO

The DMA controller transfers data from the internal peripherals HDI16 and EFCOP, which reside on the local bus, to the DMA FIFO. It also transfers data from peripherals on the external system bus to the DMA FIFO. **Figure 6-5** shows both of these options. The relevant DMA channel can be programmed either to external or internal request mode, depending on the peripheral type.

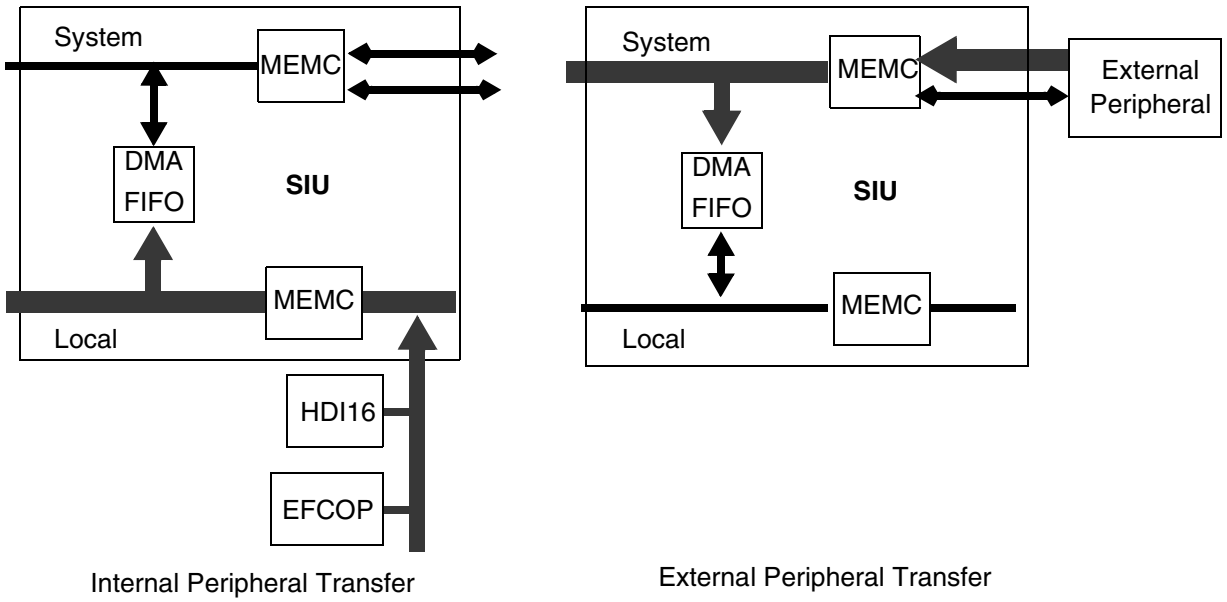
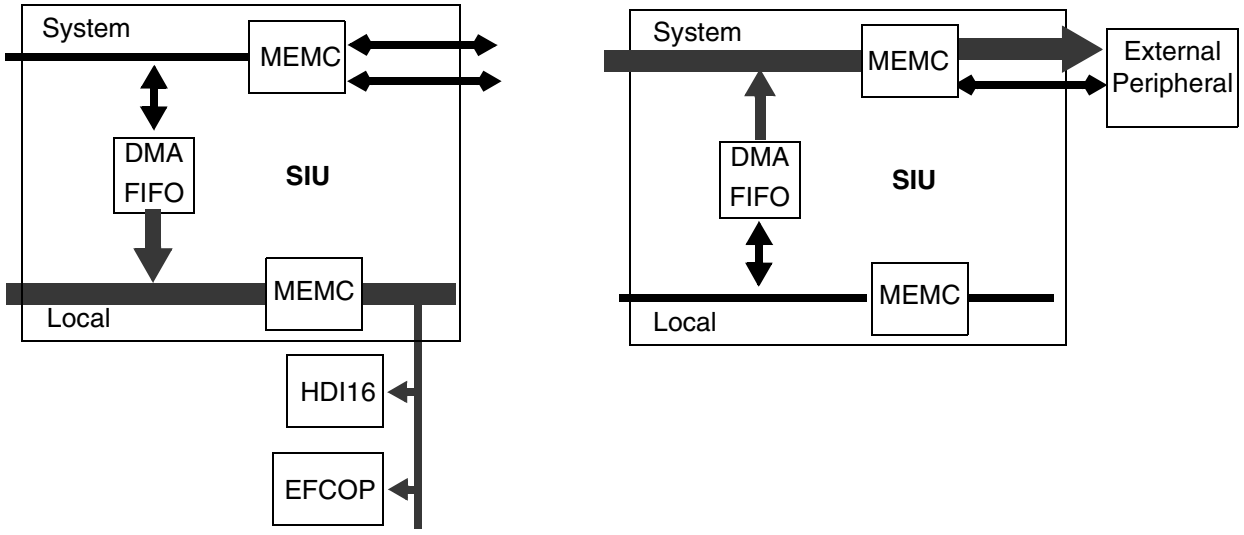


Figure 6-5. Peripheral to DMA FIFO Transfers

6.1.2.4 DMA FIFO to Peripheral

The DMA controller transfers data from the DMA FIFO to such internal peripherals as the HDI16 and EFCOP. It also transfers data from the DMA FIFO to external peripherals on the system bus. **Figure 6-6** shows both of these options.

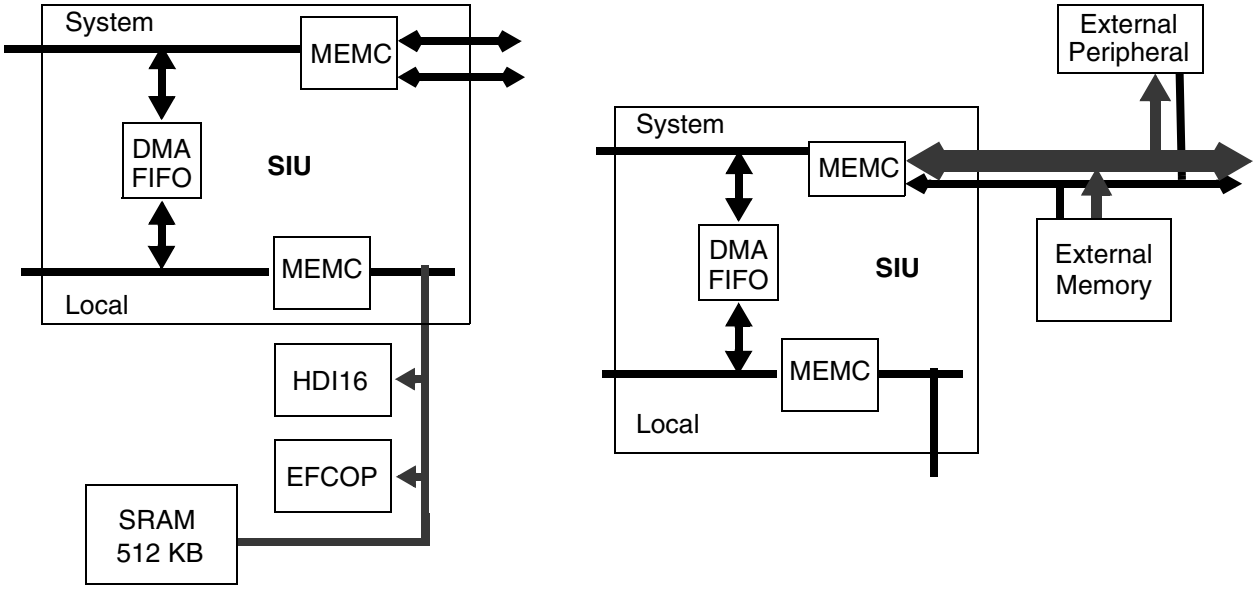


Internal Peripheral Transfer External Peripheral Transfer

Figure 6-6. DMA FIFO to Peripheral Transfers

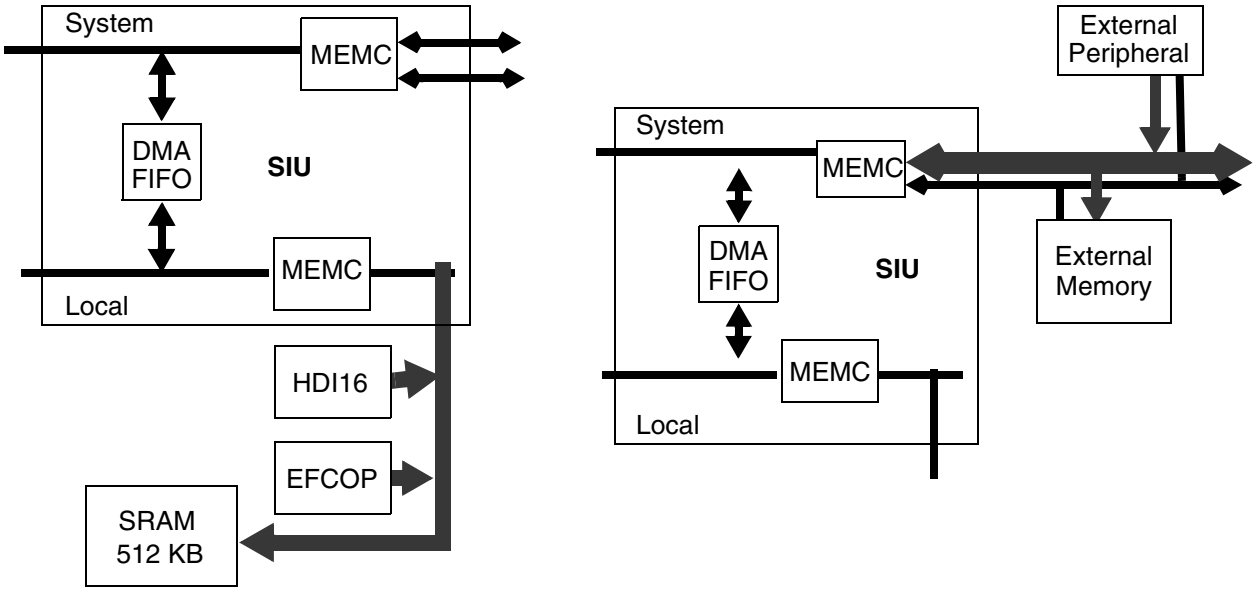
6.1.2.5 Memory to Peripheral, Flyby Mode

Flyby transactions can be executed from memory connected to the system bus to a peripheral connected to the same bus and to the DMA request/acknowledge lines. They can also be executed from the SRAM connected to the local bus, to the HDI16, or to the EFCOP. **Figure 6-7** shows DMA transfers from memory to a peripheral on the same bus, and **Figure 6-8** shows a DMA transfer from a peripheral to memory.



Internal Memory to Internal Peripheral External Memory to External Peripheral

Figure 6-7. Memory to Peripheral, Flyby Mode Data Transfer



Internal Memory to Internal Peripheral

External Memory to External Peripheral

Figure 6-8. Peripheral to Memory, Flyby Mode Data Transfer

6.2 Initializing the DMA

The DMA controller uses registers and DMA Channel Parameters RAM (DCPRAM) to configure each DMA channel. **Table 6-1** summarizes the DMA registers involved in initializing the DMA. For details on programming the DMA registers, consult the DMA chapter of the *MSC8101 Reference Manual*.

Table 6-1. DMA Registers

Mnemonic	Name	Description
DCHCRx	DMA Channel Configuration Registers	Configures the connection between a DMA requestor and the corresponding DMA channel. There is one register per channel.
DCPRAM	DMA Channel Parameters RAM	Holds the buffer parameters for all the channels
DPCR	DMA Pin Configuration Register	Selects the functionality of the $\overline{\text{DONE}}$/$\overline{\text{DRACK}}$ pins.
DSTR	DMA Status Register	Reflects the interrupt requests of the various channels
DIMR	DMA Internal Mask Register	Enables interrupt requests of the corresponding channel on the PIC.
DEMR	DMA External Mask Register	Enables interrupt requests of the corresponding channel to the SIC_EXT.

6.2.1 DMA Channel Configuration Registers (DCHRx)

Each DMA channel has a DCHCR that defines whether the channel is active (ACTV), the active bus (PPC), settings of the DMA request/acknowledge signals, Flyby mode (FLY), active

requestor (INT, RQNUM), and channel priority (PRIO). Since bit 0 of this register activates the DMA channel, it should be set to 1 only after all registers are programmed.

Table 6-2. DCHCRx Bits

	Bit 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	ACTV	PPC	—			EXP			DRS	DPL	BDPTR					
TYPE	R/W															
RESET	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	DRACK	FLY	—	RQNUM					FRZ	INT	—		PRIO			
TYPE	R/W															
RESET	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Four DCHCRx bit groupings control DMA signal usage:

- EXP defines the number of cycles to ignore asserted level DREQ after $\overline{\text{DRACK}}$ or $\overline{\text{DACK}}$ signal is asserted.
- DRS defines the type of trigger used with the DREQ signal. It can be either edge-triggered or level-triggered.
- DPL defines the polarity of the DREQ signal.
- $\overline{\text{DRACK}}$ is used to acknowledge a request before its execution. The external peripheral must support the DRACK protocol.

These bits do not affect DMA transactions involving internal peripherals. They are for use in programming a DMA interface to external devices. See **Section 6.3** for further discussion of the DMA signals. When a flyby transaction (FLY = 1) is implemented, the DMA cannot initiate the data transfer. Instead, the INT bit must be cleared so that a peripheral initiates the request. The specific peripheral requesting the transfer is defined by RQNUM. If INT is set to 1, the transaction does not start.

6.2.2 DMA Pin Configuration Register (DPCR)

The DPCR is an 8-bit register with two programmable bits. This register selects between the $\overline{\text{DRACK}}$/$\overline{\text{DONE}}$ signals, which are available only for external requestors 1 or 2. When an external DMA requestor is in use and an external peripheral requires the DMA to notify it when a transfer completes, the DMA activates a $\overline{\text{DONE}}$ signal to alert the peripheral that it will not be serviced further. On the other hand, if the peripheral requires the DMA channel to terminate before the DMA transfer completes, the peripheral generates the $\overline{\text{DONE}}$ signal to alert the DMA that the channel must be terminated. Instead of using the $\overline{\text{DONE}}$ signal, an external peripheral can use the $\overline{\text{DRACK}}$ signal. In this case, it receives acknowledgment when the DMA samples its DMA request, and the peripheral either asserts a new request or resumes other processes.

6.2.3 DMA Status Register (DSTR)

The bits in the DSTR indicate whether an interrupt request is pending for the associated DMA channel. If the bit is set, the channel has requested service from the core processor.

6.2.4 DMA Internal/External Mask Registers (DIMR/DEMR)

The DIMR/DEMR enable generation of interrupt requests to their associated interrupt controllers. The DIMR enables an interrupt to the Peripheral Interrupt Controller (PIC). The DEMR enables an interrupt to the SIU-CPM Interrupt Controller (SIC_EXT). Interrupts from the SIC_EXT generate interrupt requests to an external host on the system bus.

Note: For each DMA channel, the respective bits in the DIMR and DEMR should have different values. For example, if DIMR[Mx] = 0, ensure that the corresponding DEMR[Mx] = 1 to avoid undefined system behavior. Enable each channel for an internal or an external interrupt only.

6.2.5 DMA Channel Parameters RAM (DCPRAM)

The DCPRAM\) holds 64 buffer descriptors. Each buffer descriptor includes buffer address, transfer size down counter, buffer base size, and buffer attributes. Each buffer descriptor is allocated four 32-bit fields for these transaction parameters. **Table 6-3** shows addresses for the BD_ADDR, BD_SIZE, and BD_ATTR for 17 of the 64 buffer descriptors. Note that there are 64 buffer descriptors that can be programmed but only 16 DMA channels.

Table 6-3. DCPRAM Addressing

DMA Channel	Memory Map Address*	Channel Buffer Address	Memory Map Address*	Channel Transfer Size	Memory Map Address*	Channel Attributes	Memory Map Address*	Channel Transfer Base Size
0	0xF0010800	BD_ADDR0	0xF0010804	BD_SIZE0	0xF0010808	BD_ATTR0	0xF001080C	BD_BSIZE0
1	0xF0010810	BD_ADDR1	0xF0010814	BD_SIZE1	0xF0010818	BD_ATTR1	0xF001081C	BD_BSIZE1
2	0xF0010820	BD_ADDR2	0xF0010824	BD_SIZE2	0xF0010828	BD_ATTR2	0xF001082C	BD_BSIZE2
3	0xF0010830	BD_ADDR3	0xF0010834	BD_SIZE3	0xF0010838	BD_ATTR3	0xF001083C	BD_BSIZE3
4	0xF0010840	BD_ADDR4	0xF0010844	BD_SIZE4	0xF0010848	BD_ATTR4	0xF001084C	BD_BSIZE4
5	0xF0010850	BD_ADDR5	0xF0010854	BD_SIZE5	0xF0010858	BD_ATTR5	0xF001085C	BD_BSIZE5
6	0xF0010860	BD_ADDR6	0xF0010864	BD_SIZE6	0xF0010868	BD_ATTR6	0xF001086C	BD_BSIZE6

* These addresses assume that the Bus memory map is based at 0xF0000000. This is the default value for the Internal Memory Map Register (IMMR) at reset.

Table 6-3. DCPRAM Addressing (Continued)

DMA Channel	Memory Map Address*	Channel Buffer Address	Memory Map Address*	Channel Transfer Size	Memory Map Address*	Channel Attributes	Memory Map Address*	Channel Transfer Base Size
7	0xF0010870	BD_ADDR7	0xF0010874	BD_SIZE7	0xF0010878	BD_ATTR7	0xF001087C	BD_BSIZE7
8	0xF0010880	BD_ADDR8	0xF0010884	BD_SIZE8	0xF0010888	BD_ATTR8	0xF001088C	BD_BSIZE8
9	0xF0010890	BD_ADDR9	0xF0010894	BD_SIZE9	0xF0010898	BD_ATTR9	0xF001089C	BD_BSIZE9
10	0xF00108A0	BD_ADDR10	0xF00108A4	BD_SIZE10	0xF00108A8	BD_ATTR10	0xF00108AC	BD_BSIZE10
11	0xF00108B0	BD_ADDR11	0xF00108B4	BD_SIZE11	0xF00108B8	BD_ATTR11	0xF00108BC	BD_BSIZE11
12	0xF00108C0	BD_ADDR12	0xF00108C4	BD_SIZE12	0xF00108C8	BD_ATTR12	0xF00108CC	BD_BSIZE12
13	0xF00108D0	BD_ADDR13	0xF00108D4	BD_SIZE13	0xF00108D8	BD_ATTR13	0xF00108DC	BD_BSIZE13
14	0xF00108E0	BD_ADDR14	0xF00108E4	BD_SIZE14	0xF00108E8	BD_ATTR14	0xF00108EC	BD_BSIZE14
15	0xF00108F0	BD_ADDR15	0xF00108F4	BD_SIZE15	0xF00108F8	BD_ATTR15	0xF00108FC	BD_BSIZE15
...	...	...	...	...	...	...	...	...
64	0xF0010BF0	BD_ADDR15	0xF0010BF4	BD_SIZE15	0xF0010BF8	BD_ATTR15	0xF0010BFC	BD_BSIZE15

* These addresses assume that the Bus memory map is based at 0xF0000000. This is the default value for the Internal Memory Map Register (IMMR) at reset.

Each DMA channel uses buffer descriptors in the DCPRAM to point to a buffer and characterize it. The buffer descriptor contains four distinct parameters: address (BD_ADDRn), size (BD_SIZEn), base size (BD_BSIZEn), and attributes (BD_ATTRn). Each DMA channel selects the buffer descriptor by configuring the DCHCRx[BDPTR] bits. Therefore, DMA channel 15 and DMA channel 3 can both use the same buffer descriptor if the parameters for both transfers are the same, but these channels can be requested by different sources. **Table 6-4** describes the four types of buffer descriptor parameters for DMA data transactions.

Table 6-4. Buffer Descriptor Parameters

Parameter	Description
BD_ADDR	**Address** Describes either the source or destination of the DMA data transfer. For a read cycle, BD_ADDR describes the source address of the DMA transfer. For a write cycle, BD_ADDR describes the destination address of the transfer. The BD_ADDR for a flyby request must be programmed to the memory address.
BD_SIZE	**Size** A transfer byte size down counter. BD_SIZE is always the number of bytes left to transfer even though the transfer size parameter may vary between eight bits to one burst.
BD_BSIZE	**Base Size** Required only for programming continuous buffers. When the DMA transfer size reaches zero (the complete buffer is transferred), BD_SIZE is updated with the value of the BD_BSIZE parameter, and the transfer can resume.
BD_ATTR	**Attributes** Defines the buffer characteristics.

The BD_ATTR bits are as follows.

BD_ATTR Buffer Attributes Parameter

	Bit 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	INTRPT	CYC	CONT	—	NO_INC	BP		—	NBUS	NBD						
TYPE	R/W															
RESET	Undefined															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	—						TSZ		—	FLS	RD	—	TC	—	GBL	
TYPE	R/W															
RESET	Undefined															

Some of the bit functions are straightforward:

- INTRPT defines whether the DMA issues an interrupt when BD_SIZE reaches zero.
- NO_INC defines the constant address for port access applications.
- TSZ determines the transfer size.
- FLS indicates whether the FIFO is flushed when BD_SIZE reaches zero.
- RD must be set if data should be read from the buffer.

Other BD_ATTR bits work together in a particular mode. CYC, NBUS, and NBD must be considered if CONT is set. CONT defines whether the buffer is closed when the transfer is complete or whether the DMA transaction continues when BD_SIZE reaches zero. If a continuous buffer is chosen (CONT = 1), then BD_SIZE is reloaded with a BD_BSIZE value.

CYC defines whether BD_ADDR is a cyclic address. NBUS defines which bus is used for the next transaction, and NBD defines the next buffer to be used.

Another grouping of bits that relate to each other are the BP, TC, and GBL bits. These bits all regulate bus action for the DMA transfer. BP defines the priority for a given transfer on the bus. This priority value goes into effect when the DMA arbitrates for mastership of the bus. DMA bus priority 0 corresponds to BP = 00. This is the highest bus priority level. BP = 10 is DMA bus priority 2, the lowest DMA bus priority level. The transfer code bits (TC) give the bus extra information about the source of a DMA transaction. Both settings tell the bus that the transfer is a DMA transfer. Additional bus signals, TT[0–4], TSIZ[0–3], and $\overline{\text{TBST}}$, are used to define the bus transaction. These signals are described in detail in the chapter on the system bus in the *MSC8101 Reference Manual*. When GBL is set, the bus transaction is global. This bit is set only when the DMA transfer accesses a memory shared by multiple devices. Otherwise, the GBL bit is cleared. For details, consult the chapter on the system bus in the *MSC8101 Reference Manual*.

6.2.6 FIFO Requests

The DMA FIFO issues two types of requests that are generated by hardware within the MSC8101: watermark requests and hungry requests. The DMA FIFO generates a watermark request to notify the channel that the FIFO contains data to be transferred to a destination by the DMA channel. Therefore, the DMA channel should write the data in the FIFO to the destination. If the DMA FIFO has room for more data, it generates a hungry request to notify the channel that it can accept more data.

The watermark request is asserted when the FIFO holds 32 bytes or more to notify the destination DMA channel that the FIFO contains data to be transferred to the destination. A hungry request is asserted when the FIFO contains less than 56 bytes of data to notify the source DMA channel that the FIFO can receive more data. Both watermark and hungry requests can be generated simultaneously to alert both DMA channels associated with a given FIFO that there is room for more data and that data is ready to be transferred.

6.2.7 Multiple Pending DMA Requests

If multiple channels are needed, you can program the order in which each channel executes by assigning each channel a priority by the DCHCRx[PRIO] bits. Correct multiple-channel prioritization enables smooth operation. Lower-bandwidth channels should be assigned a higher priority, and the same rules goes for lower-latency channels, such as voice channels.

If all channels are given the same priority, the channels are prioritized based on their number. A lower channel number has priority over a higher channel number. For example, if channels 0, 1, 14, and 15 are active, channel 0 has the highest priority and channel 15 has the lowest. The arbitration between channels is done on a per cycle basis. When a channel wins the arbitration, it can issue one transaction only. Another channel may win the arbitration on the next cycle. If a channel has the highest priority (or the same priority and a lower channel number), then it wins

all the arbitration phases until it is stopped by one of the following events: the FIFO becomes empty/full, or BD_SIZE for the channel reaches zero. When a channel stops, any other channel can issue a transaction on the bus.

6.2.8 Buffering and Bursting

The MSC8101 DMA module supports five types of buffering: simple, cyclic, chained, incremental, and dual cyclic. For details regarding these buffer types, consult the DMA chapter in the *MSC8101 Reference Manual*. For programming examples of cyclic and chained buffers, refer to **Example 6-2**, *External Memory to External Memory, Burst Mode, Cyclic Buffer*, on page 6-19 and **Example 6-4**, *External Flash Memory to External SDRAM Memory, Dual Access Mode*, on page 6-23. **Figure 6-9** shows a diagram of all five buffer types for quick reference.

The MSC8101 DMA module can execute burst data transfers of up to 32 bytes per burst. Timing of the transfer depends on the setting of the memory controller. Burst transfers can be programmed to external devices as long as they are burst-capable controlled by either the SDRAM controller or the UPM. The GPCM is not burst-capable. If a peripheral or memory device is controlled by the GPCM and programmed for a burst transfer, the burst is split into a single-beat transfer, and the address is incremented.

6.2.9 Interrupts

The DMA module controls the $\overline{\text{IRQ18}}$ input to the Programmable Interrupt Controller (PIC). Perform the following steps to initialize interrupts:

1. Set up the interrupt routine by placing code to handle the interrupt at the appropriate interrupt vector address.

   The location of the vector address for $\overline{\text{IRQ18}}$ is 0xC80 offset from VBA.

2. Enable the interrupts by setting bits in various control registers:

   a. To enable interrupts by a buffer, when BD_SIZE reaches zero, set the INTRPT (bit 0) bit in the BD_ATTR field.

b. To specify which interrupt priority levels are allowed, set the interrupt mask bits (I0–2) of the Status Register (SR) in the SC140 core. For details, refer to the *SC140 Core Reference Manual*.

c. To define the priority level for each enabled interrupt, set the PIC Edge/Level-Triggered Interrupt Priority Register E (ELIRE) Interrupt Priority Level (PILxx) bits.

d. To set the interrupt trigger mode, set the PED18 bits of the ELIRE registers. $\overline{\text{IRQ18}}$ interrupts are edge-triggered.

e. To enable the transaction complete interrupt, set the mask bit in the DIMR register corresponding to the DMA channel.

f. To enable interrupts, issue an **ei** (enable interrupts) instruction.

For an example of DMA programming with interrupts, see **Example 6-4**.

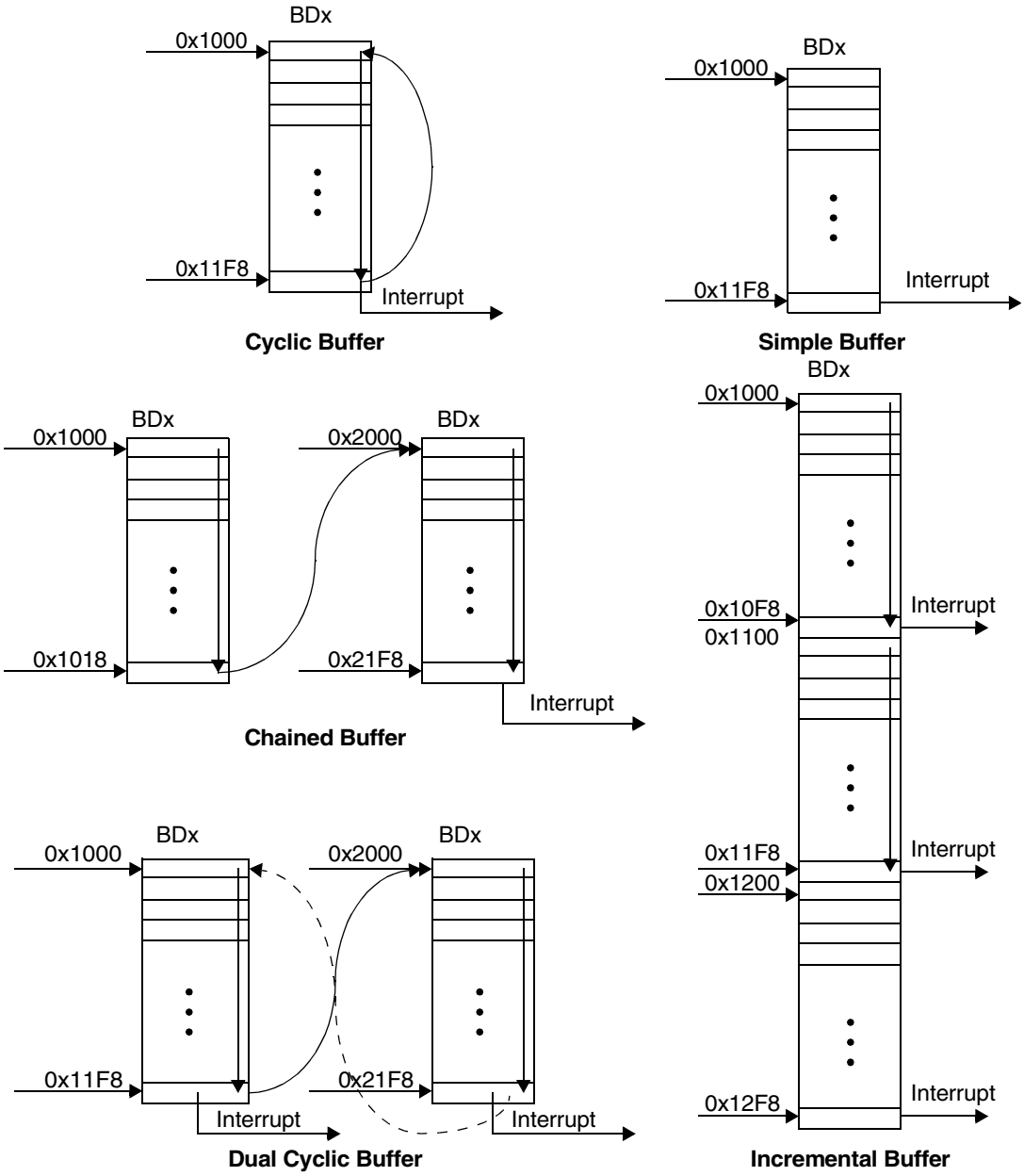


Figure 6-9. DMA Buffer Types

6.3 Using DMA Signals to Initiate and Control DMA Transfers

DMA has four signal types to initiate and control DMA transfers by external peripherals. These signals are:

- DREQ[1–4] (DMA request)
- $\overline{\text{DACK[1–4]}}$ (DMA acknowledge)
- $\overline{\text{DONE[1–2]}}$ (DMA done)/$\overline{\text{DRACK[1–2]}}$ (DMA request acknowledge)

DREQ[3–4] and DACK[3–4] are multiplexed with IRQ[4–7], EXT_BG3, and EXT_DBG3 lines on the MSC8101 pins, as shown in the external signals chapter of the *MSC8101 Reference Manual*. The remaining DMA signals are multiplexed on the CPM ports C and D. Since there are four groupings of DMA signals, up to four external devices can request DMA service via DMA request lines. However, only two of the devices can use the DONE/DRACK protocol. Also, if a system requires use of the DONE/DRACK signals, then the SCC1:RXD and SCC1:TXD signals are not available due to multiplexing on the CPM ports. Also, the DREQ[1–2] and DACK[1–2 signals are multiplexed with BRG7, BRG8, and CLK[7–10]. Therefore, when using these DMA signals, a system designer must use other baud-rate generators or clocks for the application design.

6.4 DMA Programming Examples

The code examples in this section illustrate how to program the DMA controller in various modes:

- A simple buffer to transfer data from internal memory to external memory.
- Burst mode transactions between two external memory locations. It also uses a cyclic buffer.
- A simple buffer to transfer data from an internal peripheral (HDI16) to external memory.
- A data transfer between external Flash memory, external memory, and internal memory. The two transfers are chained. This example also shows how interrupts are used. For an example of a flyby data transfer with an internal peripheral, see .

The examples in this section use equate labels for the location of the DMA registers and buffer descriptors. It is assumed that these equates are declared before the example code. Equate labels include the register or field name preceded with “M_”.

6.4.1 Internal to External Dual Access, Simple Buffer

Example 6-1 uses a DMA channel to transfer data from internal to external memory with a simple buffer. Since data is transferred from internal memory to external SDRAM, the DMA transfer is a dual transaction. A transaction transfers SIZE bytes of data 32-bits at a time. Bank 2 and bank 10 of the memory controller are configured to allow the DMA channel to access external memory through the SDRAM and the internal DSP SRAM through the UPMC, respectively. The memory buffer to which the OUT_ADDR equate points must be within the memory range of bank 2; the memory buffer to which the IN_ADDR equate points must be within the memory range of bank 10.

1. Channel 0 reads data from internal memory to the DMA FIFO. The DMA control registers for channel 0 are programmed as follows:
   a. The address location of the data, IN_ADDR, is written to the DMA buffer address pointer field (BD_ADDR0). Note that the buffer descriptor is associated to the channel by the DCHCR[BDPTR] bits.

b. The total number of bytes to transfer, SIZE, is written to the DMA buffer size field (BD_SIZE0).

The DMA buffer base size (BD_BSIZE0) does not need to be programmed since this is not a continuous buffer.

c. To configure DMA channel 0 for 32-bit read transactions, flush mode, the value 0x000001b0 is written to the DMA attribute field (BD_ATTR0).

d. To enable channel 0 as a dual transaction initiated by the DMA controller, the value 0x80000040 is written to the DCHCR0.

2. Channel 1 writes data from the DMA FIFO to external SDRAM. The DMA control registers for channel 1 are programmed as follows:

a. The address location of output data, OUT_ADDR, is written to the DMA buffer address pointer field (BD_ADDR1).

b. The total number of bytes to transfer, SIZE, is written to the DMA buffer size field (BD_SIZE1).

The DMA buffer base size (BD_BSIZE1) does not need to be programmed since this is not a continuous buffer.

c. To configure channel 1 for 32-bit write transactions, the value 0x00000180 is written to the DMA attribute field (BD_ATTR1). Note that the flush bit should be set, if desired, only for the read channel.

d. To enable channel 1 as a dual transaction initiated by the DMA controller, the value 0xc0010040 is written to DCHRC1.

3. The DMA Status register is polled to see if channels 0 and 1 have completed their transactions.

Bits 0 and 1 are checked in the DSTR. If bits 0 and 1 are clear, then the code continues to run. If bits 0 and 1 are set, then DMA channels 0 and 1 have completed their transactions.

Once the DMA channels are programmed, the data is transferred without intervention of the SC140 core.

Example 6-1. Internal Memory to External Memory, Simple Buffer

```
;DMA0 init to input DATA to DMA Buffer
move.l #IN_ADDR,d0          ;Init source address
move.l d0,M_BDADDR0
move.l #SIZE,d0             ;Init transfer size
move.l d0,M_BDSIZE0
move.l #ATTR0,d0            ;Init channel 0 attrib
move.l d0,M_BDATTR0

;DMA1 init to output DATA from DMA Buffer
```

```
move.l #OUT_ADDR,d0          ;Init destination
address
move.l d0,M_BDADDR1
move.l #SIZE,d0              ;Init transfer size
move.l d0,M_BDSIZE1
move.l #ATTR1,d0             ;Init channel 1 attrib

DMA_START
move.l d0,M_BDATTR1
moveu.l #dchcr0,d0           ;Init channel 0 config
move.l d0,M_DCHCR0
moveu.l #dchcr1,d0           ;Init channel 1 config
move.l d0,M_DCHCR1

CONT
move.l M_DSTR,d5
bmtstc #0xc000,d5.h
jt CONT                      ;Wait until output done
```

6.4.2 External to External Dual-Access Burst Transfer, Cyclic Buffer

Example 6-2 transfers data from external memory to external memory. Data is transferred from SDRAM to SDRAM, so this is a dual access transaction with a cyclic buffer in burst mode. Channel 0 reads data from external memory to the DMA FIFO; channel 1 writes the 64 bytes of data from the DMA FIFO to external memory. Thus, the first 32 bytes from the source address are transferred to the destination twice. Bank 2 of the memory controller is configured to allow the DMA channel to access external memory through the SDRAM. The memory buffer to which the IN_ADDR and OUT_ADDR equates point must be within the memory range of bank 2.

1. Channel 0 reads data from cyclic buffer external SDRAM to the DMA FIFO. The DMA control registers for channel 0 are programmed as follows:
   a. The address location of the data, IN_ADDR, is written to the DMA buffer address pointer field (BD_ADDR0).
   b. The total number of bytes to transfer, SIZE0, is written to the DMA buffer size field (BD_SIZE0).
   c. The buffer is a continuous cyclic buffer. The buffer size to be reloaded, BSIZE0, is written to the DMA buffer base size field (BD_BSIZE0)
   d. To configure channel 0 for burst cyclic read transactions, the value 0xe0400210 is written to the DMA attribute field (BD_ATTR0).
   e. To enable channel 0 as a dual transaction initiated by the DMA, the value 0xc0000040 is written to the DCHCR0.
2. Channel 1 writes data from the DMA FIFO to external SDRAM. The DMA control registers for channel 1 are programmed as follows:

a. The address location of output data, OUT_ADDR, is written to the DMA buffer address pointer field (BD_ADDR1).

b. The total number of bytes to transfer, SIZE1, is written to the DMA buffer size field (BD_SIZE1).

The DMA buffer base size (BD_BSIZE1) does not need to be programmed since this is not a continuous buffer.

c. To configure channel 1 for burst write transactions, the value 0x00000200 is written to the DMA attribute field (BD_ATTR1).

d. To enable channel 1 as a dual transaction initiated by the DMA controller, the value 0xc0010040 is written to DCHRC1.

3. The DMA Status register is polled to see if channels 0 and 1 have completed their transactions. Bits 0 and 1 are checked in the DSTR. If bits 0 and 1 are clear, then the code continues to run. If bits 0 and 1 are set, then DMA channels 0 and 1 have completed their transactions.

Once the DMA channels are programmed, the data is transferred without intervention of the SC140 core.

Example 6-2. External Memory to External Memory, Burst Mode, Cyclic Buffer

```
;DMA0 init to input DATA to DMA Buffer

move.l #IN_ADDR,d0          ;Init source address
move.l d0,M_BDADDR0
move.l #BSIZE0,d0           ;Init base size
move.l d0,M_BDBSIZE0
move.l #SIZE0,d0            ;Init transfer size
move.l d0,M_BDSIZE0
move.l #ATTR0,d0            ;Init channel 0 attrib
move.l d0,M_BDATTR0

;DMA1 init to output DATA from DMA Buffer

move.l #OUT_ADDR,d0         ;Init destination
address
move.l d0,M_BDADDR1
move.l #SIZE1,d0            ;Init transfer size
move.l d0,M_BDSIZE1
move.l #ATTR1,d0            ;Init channel 1 attrib
move.l d0,M_BDATTR1


DMA_START
moveu.l #dchcr0,d0          ;Init channel 0 config
move.l d0,M_DCHCR0
moveu.l #dchcr1,d0          ;Init channel 1 config
```

```
move.l d0,M_DCHCR1
CONT
move.l M_DSTR,d5
bmtsts #0xc000,d5.h
jf CONT
```

6.4.3 Internal Peripheral to External Dual Access, Simple Buffer

Example 6-3 transfers 16-bit data from an internal peripheral to external memory as a dual transaction with a simple buffer. The data is read from the host via the HDI16 interface to the DMA FIFO. The DMA FIFO writes the data to external SDRAM. Bank 2 and bank 11 of the memory controller are configured to allow the DMA channel to access external memory through the SDRAM and to the HDI16 through the GPCM, respectively. The memory buffer to which the BUFF_START equate points must be within the memory range of bank 2.

1. The HDI16 control registers are initialized in the following manner:
   a. The Host Control Register (HCR) is programmed for Host DMA mode by the value (INIT_HCR).
   b. The Host Port Control Register (HPCR) is programmed for 16-bit mode by the value (INIT_HPCR).
   c. After the control registers are programmed, the HDI16 is enabled as a host interface by setting bit 8 of the HPCR.
2. Channel 0 reads data from the Host Receive Data Register (HORX) as a simple buffer to the DMA FIFO. The DMA control registers for channel 0 are programmed in the following manner:
   a. The address location of the data, M_HORX, is written to the DMA buffer address pointer field (BD_ADDR0).
   b. The total number of bytes to transfer, PATT_SIZE, is written to the DMA buffer size field (BD_SIZE0).

      The DMA buffer base size (BD_BSIZE0) does not need to be programmed since this is a simple buffer.
   c. To configure channel 0 to perform 16-bit read transactions with no increment of the address and a flush of the FIFO, the value 0x08000130 is written to the DMA attribute field (BD_ATTR0).
   d. To enable channel 0 as a dual transaction initiated by an HDI16 read request, the value 0x81800005 is written to DCHCR0.
3. Channel 1 writes data from the DMA FIFO to external SDRAM. The DMA control registers for channel 1 are programmed as follows:
   a. The address location of output data, BUFF_START, is written to the DMA buffer address pointer field (BD_ADDR1).

b. The total number of bytes to transfer, PATT_SIZE, is written to the DMA buffer size field (BD_SIZE1).

The DMA buffer base size (BD_BSIZE1) does not need to be programmed since this is a simple buffer.

c. To configure channel 1 for 16-bit write transactions, the value 0x80000120 is written to the DMA attribute field (BD_ATTR1).

d. To enable channel 1 as a dual transaction initiated by the DMA, the value 0xc0010045 is written to DCHRC1.

4. The DMA Status register is polled to see if the channels 0 and 1 have completed their transactions.

Bits 0 and 1 are checked in the DSTR. If bits 0 and 1 are clear, then the code continues to run. If bits 0 and 1 are set, then DMA channels 0 and 1 have completed their transactions.

Example 6-3. Internal Peripheral to External Memory, Simple Buffer

```
;setup HDI16 registers

move.w #INIT_HCR,r0          ; Init HDI16 host
control
                             ; register
move.w r0,M_HCR
move.w #INIT_HPCR,r0         ; Init HDI16 host port
control
                             ; register
move.w r0,M_HPCR
bmset.w #0x80,M_HPCR         ;Enable HDI16

;DMA0 init to input DATA from HDI16 to DMA Buffer

move.l #M_HORX,d0            ;Init source address
move.l d0,M_BDADDR0
move.l #PATT_SIZE,d0         ;Init transfer size
move.l d0,M_BDSIZE0
move.l #ATTR0,d0             ;Init channel 0 attrib
move.l d0,M_BDATTR0

;DMA1 init to output DATA from DMA Buffer

move.l #BUFF_START,d0        ;Init destination
address
move.l d0,M_BDADDR1
move.l #PATT_SIZE,d0         ;Init transfer size
move.l d0,M_BDSIZE1
move.l #ATTR1,d0             ;Init channel 1 attrib
move.l d0,M_BDATTR1
```

```
DMA_START
moveu.l #dchcr0,d0          ;Init channel 0 config
move.l d0,M_DCHCR0
moveu.l #dchcr1,d0          ;Init channel 1 config
move.l d0,M_DCHCR1
CONT
move.l M_DSTR,d5
bmtsts #0xc000,d5.h
jf CONT
```

6.4.4 External to External Dual Access, Chained with Interrupts

Example 6-4 uses a dual access to transfer 100 bytes of data from external SDRAM memory to a second external SDRAM memory location. DMA channel 2 reads data from the first external SDRAM memory address controlled by the SDRAM controller. Then DMA channel 3 reads data from the DMA FIFO and writes it to the second external SDRAM memory location under control of the SDRAM controller. This first transfer is chained to a second transfer that moves the data from the second external SDRAM address to the DMA FIFO using DMA channel 2. DMA channel 3 then transfers the data from the DMA FIFO to internal SRAM and generates an interrupt when the transfer is complete. The code implements the DMA transfer as follows:

Note: This code does not show initialization of the external and internal memory banks. It assumes that this is already complete.

1. The code begins with initialization of interrupts, as follows:
   a. The interrupt mask bits (I0-2) of the Status Register (SR) are cleared. This permits all interrupt priority levels.
   b. ELIRE is programmed with IRQ18 at priority level 5 level-triggered mode.
   c. To allow a DMA Channel 3 interrupt, the associated mask bit in the DIMR is set.
   d. To enable interrupts, an **ei** instruction is issued.
2. The first DMA transaction is initialized. Since it is a dual access transaction, both DMA channel 2 and DMA channel 3 must be programmed. BD_BSIZE is not initialized for any of the transfers because the code is not implementing cyclic buffers.
   a. The DMA buffer descriptor 2 address is initialized to an SDRAM memory location for a 32-bit transfer size, transferring a total of 100 bytes. Once the buffer descriptor 2 read transfer is complete, it invokes the buffer descriptor 8 transfer.
   b. The DMA buffer descriptor 3 address is initialized to another SDRAM memory location for a 32-bit transfer size, transferring a total of 100 bytes. Also,

BD_ATTR3 must include the next buffer pointer since this is a chained buffer. DMA channel 3 invokes DMA buffer descriptor 9 once its transfer is complete.

3. The second DMA transaction is initialized. Since it is a dual-access transaction, both buffer descriptor 8 and buffer descriptor 9 must be programmed.
   a. The buffer descriptor 8 address is initialized to the second SDRAM memory location for a 32-bit transfer size, reading a total of 100 bytes.
   b. The buffer descriptor 9 address is initialized to an internal SRAM memory location for a 32-bit transfer size, writing a total of 100 bytes. BD_ATTR9 must also enable interrupts so that the SC140 core is notified when the entire transaction is complete.
4. The DCHCRx for each channel must be initialized. Each DCHCRx value defines which bus the transaction is occurring on (System or local bus), which buffer descriptor is associated with the channel, and the priority level of the DMA transfer. All of these transactions are internal requests, so none of them are requested by a peripheral. The DCHCRx registers are programmed for the first transfer buffer descriptors. Once the first transfer is complete, the channel remains open, and the second two buffer descriptors define the remainder of the transfer.
5. Processing begins as soon as the activate channel bit is set in DCHCR2.

Example 6-4. External Flash Memory to External SDRAM Memory, Dual Access Mode

```
; Memory Map Base Value
SDRAM_LOC    equ  $20000000                  ; SDRAM
Base Address
SRAM_LOC     equ  $02000000                  ; SRAM
base address on local bus
; Data Addresses
SDRAM_DATA1 equ  SDRAM_LOC+$300     ; Location of
first SDRAM data
SDRAM_DATA2 equ  SDRAM_LOC+$400     ; Location of
second SDRAM data
SRAM_DATA    equ  SRAM_LOC+$4000             ;
Location of final data in SRAM
;-------------------------------------------------
----
MAIN
; Initialize 16 MB SDRAM at address $20000000
INIT_INTER
    ; initialize interrupts
    bmclr #$00e0,sr.h       ; allow all interrupt
levels
    move.l #M_ELIRE,r7
    nop
    move.w #$0500,(r7)      ; set IRQ18 interrupt to
                            ; level 5
    move.l #$10000000,d7    ; enable DMA channel 3
interrupt to PIC
```

```
    move.l d7,M_DIMR
    ei                          ; enable interrupts
INIT_DMA1
    move.l #SDRAM_DATA1,d0
    move.l d0,BD_ADDR2
    move.l #$204801B0,d0        ; continuous next bus
60x,
                                ;       xfer 32 bits,
chained to BD8
    move.l d0,BD_ATTR2
    move.l #100,d0              ; transfer 100 bytes
    move.l d0,BD_SIZE2

    move.l #SDRAM_DATA2,d0
    move.l d0,BD_ADDR3
    move.l #$200901A0,d0        ; continuous,next
local,next buff
                                ; 9,xfer 32, write bits
    move.l d0,BD_ATTR3
    move.l #100,d0              ; transfer 100 bytes
    move.l d0,BD_SIZE3
INIT_DMA2
    move.l #SDRAM_DATA2,d0
    move.l d0,BD_ADDR8
    move.l #$00000190,d0        ; 32-bit xfer
    move.l d0,BD_ATTR8
    move.l #100,d0              ; transfer 100 bytes
    move.l d0,BD_SIZE8

    move.l #SRAM_DATA,d0
    move.l d0,BD_ADDR9
    move.l #$80000180,d0        ; enable interrupt,
32-bit xfer
    move.l d0,BD_ATTR9
    move.l #100,d0              ; transfer 100 bytes
    move.l d0,BD_SIZE9
DMA_XFER
    move.l #$C0020045,d0        ; activate
DMA2,60x,BD2,int req, prio 5
    move.l d0,M_DCHCR2
    move.l #$C0030045,d0        ; activate
DMA3,60x,BD3,int req, prio 5
    move.l d0,M_DCHCR3
LOOP
    bra *                       ; Stay here until the
interrupt
END
    debug
```

The SC140 core can run other application code while the DMA channel completes its data transfer. When the transfer is complete, DMA channel 3 triggers an interrupt to the SC140 core using $\overline{\text{IRQ18}}$. Once this interrupt is triggered, the processor jumps to the appropriate interrupt vector address and begins processing. The interrupt vector code, shown in **Example 6-5**, uses an equate level, I_IRQ18, to define the offset from the vector base address for the interrupt request. All DMA interrupts generate an $\overline{\text{IRQ18}}$ interrupt trigger. The interrupt service routine must determine which DMA channel triggered the interrupt and respond accordingly. Before a jump to the DMA interrupt service routine, all interrupts are disabled so that the DMA service routine can operate without interruption by any other resource on the device. The interrupts are re-enabled when the service routine completes.

Example 6-5. Interrupt Vector Code

```
    org p:I_IRQ18           ; DMA interrupt
    di                      ; disable interrupts
    jsr DMAHANDLER          ; jump to subroutine to
handle
                            ; the interrupt
    ei                      ; enable interrupts
    jmp END                 ; return from interrupt
```

The interrupt service routine shown in **Example 6-6** completes the program as follows:

1. The code gets the value of the DSTR and tests it to see if DMA channel 3 triggered the interrupt.
2. If the interrupt is triggered by DMA channel 3, then the code operates as follows:
   a. To notify the SC140 core that the interrupt has been serviced, the pending IRQ18 interrupt is cleared in Interrupt Pending Register B (IPRB).
   b. The interrupt service bit in the DSTR is cleared. Service of the interrupt is complete, so the code returns from the interrupt.
3. If the interrupt is not triggered by the DMA channel 3, then the code returns from the interrupt and continues to process.

Example 6-6. Interrupt Service Routine

```
DMAHANDLER
    move.l (M_DSTR),d7      ; move dma status reg
                            ; contents to d7
    move.l #I_IPRB,r7
    bmtset #$0040,d7.h      ; test if dma3 int
    bmset.w #$4,(r7)        ; clear pending EMA
interrupt
    jf DMA_END              ; if not dma3 int then
return

    ; if dma3 was pending clear interrupt and finish
    move.l d7,(M_DSTR)      ; clear dma3 interrupt
```

```
DMA_END
    rts
```

6.5 Avoiding DMA and SC140 Core Contentions

The DMA and the SC140 core can access internal memory and peripheral registers independently of each other. Thus, potential contention between the DMA controller and the SC140 core can occur if both are trying to access the same memory location or peripheral register at the same time. Here are some helpful hints to avoid contention:

- *Internal Memory*. Both the DMA channel and the SC140 core interface with the internal SRAM through the UPMC. The L1 bus has the highest priority over the P and X buses. The DMA channel connects to internal SRAM on the L1 bus; the SC140 core connects to internal SRAM on the P and X buses. Therefore, if the DMA channel and SC140 core try to access the same memory location, the DMA channel has priority over the SC140 core. To ensure that both the DMA controller and SC140 core can access internal SRAM without interfering with each other, the DMA controller and core should access different memory locations.
- *Peripheral Registers*. The DMA controller and the SC140 core have access to the HDI16 and EFCOP registers via the GPCM. If the DMA controller and SC140 core both try to access the same peripheral register, the DMA controller has priority over the SC140 core. To ensure that the correct value is written to or read from a peripheral register, the DMA controller and SC140 core should not access the same peripheral register at the same time
- *External Access*. In the BP field of the BD_ATTR register, each DMA channel indicates which bus mastership request will be initiated. The PRKM field of the System Bus Arbiter Configuration Register (PPC_ACR) and the Local Bus Arbiter Configuration Register (LCL_ACR) defines the parked master on the System bus and local bus, respectively.

  For the System bus, the system Bus Arbitration-Level Register (PPC_ALRH and PPC_ALRL) define the arbitration priority for the system bus master. The bus master programmed in priority field 0 has the highest priority. For example, 01 (arbitrate for bus mastership with request 1011) is programmed in the BP field of the BD_ATTR register. 1011 (DMA priority 1) is programmed in priority field 0, and 0101 (SC140 core interface) is programmed in priority field 1. The DMA has priority over the SC140 core on the system bus.

  For the local bus, the Local Bus Arbitration Level Register (LCL_ALRH and LCL_ALRL) define the arbitration priority for the local bus master. These registers are programmed the same way as the PPC_ALRH and PPC_ALRL.
- *DMA Access*. The SC140 core can access the DMA in mid-operation. The DCHCR[INT, PRIO, FRZ, PPC, ACTV] bits can be modified while the DMA channel is active. The DMA can also change the BDPTR and ACTV fields. To prevent the SC140 core from conflicting with the DMA logic and overwriting the DMA modifications, use byte access to the fields when the channel is active. Modifying fields other than the DCHCR[INT,

PRIO, FRZ, PPC, ACTV] bits may result in erroneous results. For a chained buffer, if the DCHCR[BDPTR] bits are written while the PRIO field is modified, the incorrect buffer may be selected.

Interrupts and Interrupt Priorities 7

This chapter describes a step-by-step procedure for handling MSC8101 interrupts. The main steps in this procedure are the software configuration phases for setting up the information in the interrupt controller registers and the interrupt subroutines to be executed. An example driver implementation illustrates both the hardware and software configurations for connecting to a peripheral (the EFCOP) and interrupting it. Refer to the *MSC8101 Reference Manual* for information on features not implemented by the driver.

7.1 Interrupt Basics

The MSC8101 interrupt scheme consists of three different interrupt controllers:

- *Programmable interrupt controller (PIC)*, which operates in the SC140. The PIC receives interrupts from DSP peripherals, DMA, external $\overline{\text{IRQ[2–3]}}$, and the SIC. When the PIC detects an interrupt request ($\overline{\text{IRQ}}$) on one or more of its inputs, it arbitrates each IR according to its priority level.
- *SIU-CPM interrupt controller (SIC)*, which generates interrupt requests to the PIC. The SIC receives interrupts from internal sources, such as the Periodic Interrupt Timer (PIT) or Time Counter Register (TMCNT), from the CPM, and from external sources such as port C parallel I/O pins or IRQs. The SIC generates interrupt requests to the PIC to be handled by the SC140.
- *External SIU-CPM interrupt controller (SIC_EXT)*, which generates interrupt requests to an external host CPU. The SIC_EXT receives interrupts from the same sources as the SIC with additional $\overline{\text{IRQ}}$s from external $\overline{\text{IRQ[2–3]}}$ and DMA, but it generates interrupts externally for handling by an external processor. The use of two SICs increases flexibility since each SIC can handle different interrupt sources. For example, the SC140 core can handle DSP-related interrupts while another processor, such as the MSC8101 device or the PowerQUICC II, handles communication-related interrupts.

This configuration provides maximum flexibility so that interrupts can be handled internally by the SC140, by an external host, or by a combination of the two. **Figure 7-1** shows the MSC8101 interrupt structure.

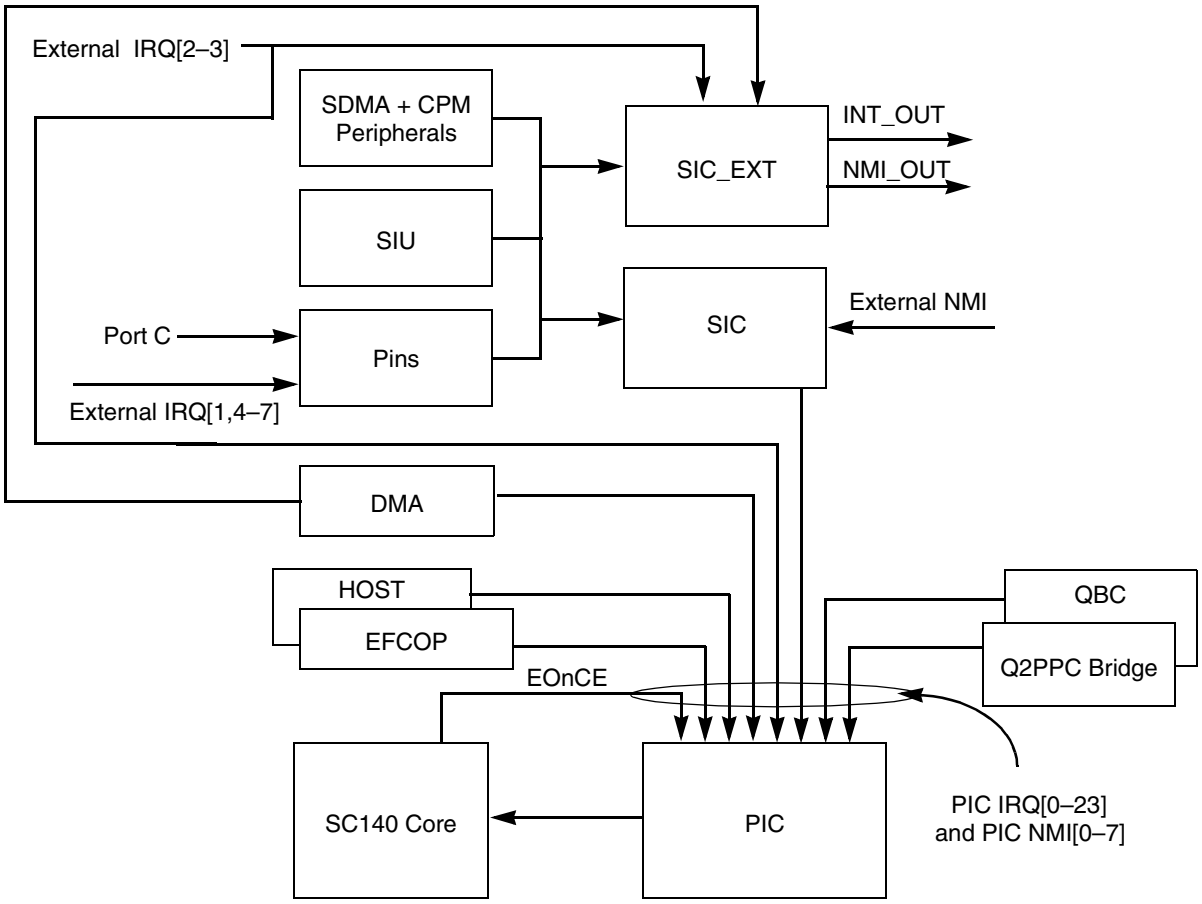


Figure 7-1. MSC8101 Interrupt Structure

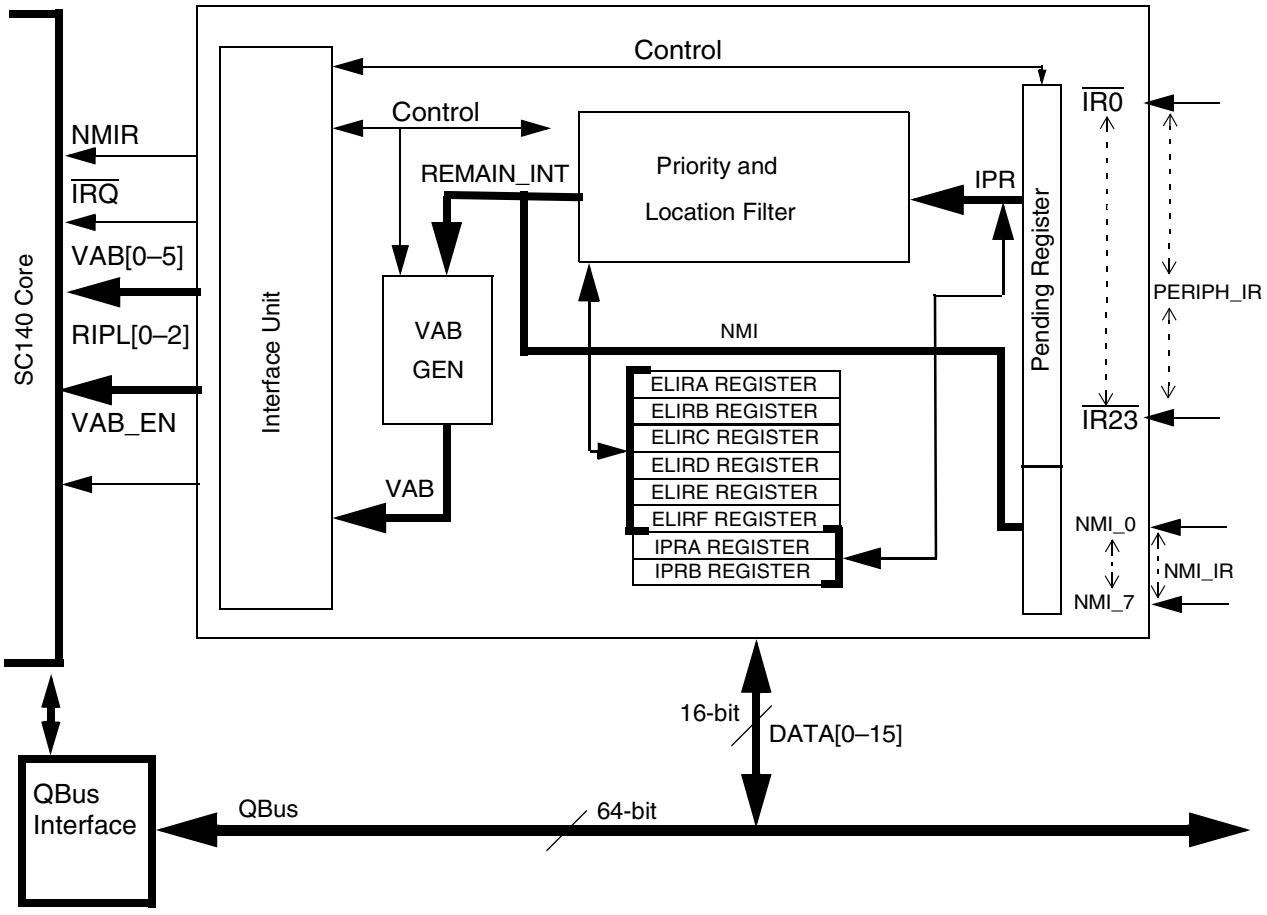
7.2 Programmable Interrupt Controller (PIC)

The MSC8101 PIC is a peripheral module that serves all the $\overline{\text{IRQ}}$s and non-maskable interrupts ($\overline{\text{NMI}}$s) received from MSC8101 peripherals and I/O pins. $\overline{\text{NMI}}$s handled by the SIC can be routed to the SC140 through the PIC or to an external host. The $\overline{\text{NMI}}$ handler is determined according to the NMI OUT bit in the hard reset configuration word. Routing NMIs to an external host makes it possible to build a system in which a single host handles all $\overline{\text{NMI}}$s.

The PIC is memory-mapped to the SC140 and can be accessed via the SC140 QBus. Key features of the PIC include the following:

- 32 inputs for $\overline{\text{IRQ}}$s and $\overline{\text{NMI}}$s, consisting of:
  - 8 asynchronous edge-triggered $\overline{\text{NMI}}$ inputs.
  - 24 asynchronous edge-triggered/level-triggered $\overline{\text{IRQ}}$ inputs.
- Auto-vector interrupt generation enable and disable.
- Support for software acknowledgment of all edge-triggered $\overline{\text{IRQ}}$ and $\overline{\text{NMI}}$.
- Visibility to all pending $\overline{\text{IRQ}}$.

- Support for nine priority levels:
  - Interrupt disabled (level 0).
  - Interrupt enabled (levels 1-7, where 7 is the highest priority).
  - NMI level (8 inputs only).
- Support for location-dependent priority for equi-level $\overline{\text{IRQ}}$ and $\overline{\text{NMI}}$.
- Ability to work with slow peripherals in edge-triggered/level-triggered modes.



NOTE: Bolded lines are bus lines; the thinner lines are control and data lines.

Figure 7-2. PIC Block Diagram

7.3 Programming MSC8101 Interrupts

When the PIC detects an $\overline{\text{IRQ}}$ on one or more of its inputs, it arbitrates each $\overline{\text{IRQ}}$ according to its priority level and location and then generates the following:

- An $\overline{\text{IRQ}}$ signal to the SC140, indicating that an IR input has requested interrupt service from the SC140.
- An RIPL[2–0] signal indicating the priority of the $\overline{\text{IRQ}}$.

- An entry in the predefined Vector Address Bus (VAB), determined by the location of the IR.

Programming MSC8101interrupts consists of the following overall steps:

1. Set the interrupt table base address in the Vector Base Address (VBA) register.
2. Program the PIC ELIRx registers to set the interrupt priority level (IPL) and trigger mode of the interrupt requests.
3. Monitor the status of pending interrupts.
4. Route the interrupts.

7.3.1 Setting the Interrupt Table Base Address

You determine the base address for the interrupt vector table by writing it to the VBA register in the SC140 core. At reset the value of the 20-bit wide VBA register is set to zero. The offset for each exception vector is predefined. There are 64 possible exception vector locations. The spacing between two exception vectors is 32 words (four full execution sets).

7.3.2 Setting the Interrupt Priority Level and Trigger Mode

The SC140 core uses six edge-triggered/level-triggered interrupt priority registers (16-bit read/write registers) to determine the interrupt priority level (IPL) and trigger mode of the interrupt requests received at each of the 24 maskable PIC inputs. These registers are software programmable. Each of the six edge-triggered/level-triggered interrupt priority registers, ELIRA through ELIRF, defines a bank of four maskable IR inputs.

Table 7-1. Edge-Triggered/Level-Triggered Interrupt Priority Registers

Register Name	Description	Bank	IR Inputs
ELIRA	PIC Edge/Level-Triggered Interrupt Priority Register A	A	0–3
ELIRB	PIC Edge/Level-Triggered Interrupt Priority Register B	B	4–7
ELIRC	PIC Edge/Level-Triggered Interrupt Priority Register C	C	8–11
ELIRD	PIC Edge/Level-Triggered Interrupt Priority Register D	D	12–15
ELIRE	PIC Edge/Level-Triggered Interrupt Priority Register E	E	16–19
ELIRF	PIC Edge/Level-Triggered Interrupt Priority Register F	F	20–23

Each register defines the interrupt trigger mode and IPL for four inputs. For each input, three bits define the priority level, and one bit specifies the trigger mode for the interrupt.

Of the 32 PIC inputs, eight are non-maskable interrupts ($\overline{\text{NMI}}$) and cannot be programmed. The $\overline{\text{NMI}}$ are always assigned the highest priority, regardless of their source. Each of the remaining 24 inputs can be programmed to one of seven maskable priority levels, IPL 0 through IPL 6, with a corresponding numeric value of 1 through 7. The highest maskable priority is IPL 6. **Table 7-2**

lists the possible settings for the three interrupt priority level bits, with their corresponding value and IPL. A value of zero in these three bits indicates that interrupts are disabled on this input.

Table 7-2. Interrupt Priority Level Bit Settings

PILxx0	PILxx1	PILxx2	Enabled	Value	IPL
0	0	0	No	0	----
0	0	1	Yes	1	0
0	1	0	Yes	2	1
0	1	1	Yes	3	2
1	0	0	Yes	4	3
1	0	1	Yes	5	4
1	1	0	Yes	6	5
1	1	1	Yes	7	6

7.3.3 Monitoring the Status of Pending Interrupts

The PIC interrupt pending registers, IPRA and IPRB, are 16-bit read/write registers used by the SC140 for two purposes:

- Monitoring pending interrupts.
- Resetting edge-triggered interrupts.

Reading the two interrupt pending registers, you can view the status of all current $\overline{\text{IRQ}}$ and $\overline{\text{NMI}}$. Each bit in the registers represents one of the 32 inputs. If an $\overline{\text{IRQ}}$ is configured as level-triggered, its corresponding interrupt pending (IP) bit reflects the status of the $\overline{\text{IRQ}}$ signal. The IP bit is set if at least one $\overline{\text{IRQ}}$ is pending and reset if there are no IRs pending.

When the corresponding IR is configured as edge-triggered, its IP bit is set for every new negative edge detected on the IR. A value of “1” written to the IP bit indicates that the corresponding IR has been acknowledged. This feature is used for both IRs and NMIs to indicate to the PIC that the SC140 core has acknowledged the corresponding edge-triggered interrupt source and that the PIC should ignore any request from the corresponding interrupt source until its next negative edge.

Each bit in the interrupt pending registers corresponds to an interrupt source, as shown in **Table 7-3**. Interrupt pending register A (IPRA) defines the status of the first 16 programmable IRs, while interrupt pending register B (IPRB) defines the remaining eight programmable IRs and the eight NMIs.

Table 7-3. Interrupt Pending Sources

Register Name	Description	Bank	Inputs	IRQ/NMI
IPRA	PIC Interrupt Pending Register A	A	0–15	IRQ
IPRB	PIC Interrupt Pending Register B	B	16–23 24–31	IRQ NMI

7.3.4 Routing Interrupts

The MSC8101 PIC can serve a total of 24 $\overline{\text{IRQ}}$ and 8 $\overline{\text{NMI}}$. Each $\overline{\text{IRQ}}$ is configured as edge-triggered or level-triggered and can be assigned a priority in the range 0 through 7, where priority 0 masks the interrupt. On reset, all $\overline{\text{IRQ}}$ are masked (set to priority 0) and configured as level-triggered. On bootstrap, $\overline{\text{IRQ[19–20]}}$ are configured as edge-triggered. The $\overline{\text{NMI}}$ relative priority is fixed, with $\overline{\text{NMI0}}$ assigned the lowest priority and $\overline{\text{NMI7}}$ the highest. $\overline{\text{NMI}}$ are always edge-triggered.

To ensure that specific sets are executed in sequence, mask all interrupts using the **di** (disable interrupts) instruction and unmask them using the **ei** (enable interrupts) instruction. You can also mask interrupts up to a specified priority level by setting the selected priority level in the SC140 status register I[2–0] (SR[23–21]). The SC140 core handles only $\overline{\text{NMI}}$ or interrupts with an IPL higher than the current interrupt mask value. At reset these bits are set, and all interrupts are disabled. The interrupt mask bits, I2, I1, and I0, reflect the current IPL of the SC140 core.

The memory allocation for each interrupt routine is 64 bytes, which constitutes four program fetches. SC140 instructions are encoded as two to four bytes, with a minimum instruction size of one word. An average of 20 instructions can be held in the allocated memory area. The address calculation is based on the VBA and VAB registers, as follows:

```
ADDR[0:31] = {VBA[0:19],VAB[0:5],6'b0}
```

Table 7-4 summarizes the routing of MSC8101 interrupts. Unless stated otherwise, all $\overline{\text{IRQ}}$ are level-triggered.

Table 7-4. Routing of MSC8101 Interrupts

VAB[0–5]	Signal	Description	Service Routine Address (Offset from VBA)
0x0	TRAP	Internal exception (generated by trap instruction)	0x0
0x1	–	Reserved	0x40
0x2	ILLEGAL	Illegal instruction or set	0x80
0x3	DEBUG	Debug exception (EOnCE)	0xC0
0x4	–	Reserved	0x100
0x5	OVERFLOW	Overflow exception (DALU)	0x140
0x6	DEFAULT $\overline{\text{NMI}}$	In VAB disabled mode only	0x180

Table 7-4. Routing of MSC8101 Interrupts (Continued)

VAB[0–5]	Signal	Description	Service Routine Address (Offset from VBA)
0x7	DEFAULT $\overline{\text{IRQ}}$	In VAB disabled mode only	0x1C0
0x8-0x1F	–	Reserved	0x200-0x7FF
0x20	$\overline{\text{IRQ0}}$	EFCOP (0): Input FIFO not full	0x800
0x21	$\overline{\text{IRQ1}}$	EFCOP (1): Input FIFO empty	0x840
0x22	$\overline{\text{IRQ2}}$	EFCOP (2): Output FIFO full	0x880
0x23	$\overline{\text{IRQ3}}$	EFCOP (3): Output FIFO not empty	0x8C0
0x24	$\overline{\text{IRQ4}}$	EFCOP (4): Update done	0x900
0x25	$\overline{\text{IRQ5}}$	HDI16 (0): Receive FIFO full	0x940
0x26	$\overline{\text{IRQ6}}$	HDI16 (1): Receive FIFO not empty	0x980
0x27	$\overline{\text{IRQ7}}$	HDI16 (2): Transmit FIFO empty	0x9C0
0x28	$\overline{\text{IRQ8}}$	HDI16 (3): Transmit FIFO not full	0xA00
0x29	$\overline{\text{IRQ9}}$	HDI16 (4): External HOST command	0xA40
0x2A	$\overline{\text{IRQ10}}$	Bus controller (x-y contention)	0xA80
0x2B	$\overline{\text{IRQ11}}$	Bus controller (level1 contention)	0xAC0
0x2C	$\overline{\text{IRQ12}}$	Bus controller (p-x contention)	0xB00
0x2D	$\overline{\text{IRQ13}}$	Bus controller (non-aligned data error)	0xB40
0x2E	$\overline{\text{IRQ14}}$	Reserved	0xB80
0x2F	$\overline{\text{IRQ15}}$	External IRQ2 (edge/level configurable)	0xBC0
0x30	$\overline{\text{IRQ16}}$	SIC interrupt	0xC00
0x31	$\overline{\text{IRQ17}}$	External IRQ3 (edge/level configurable)	0xC40
0x32	$\overline{\text{IRQ18}}$	DMA interrupt (channel/buffer terminated)	0xC80
0x33	$\overline{\text{IRQ19}}$	SMI (TEA) (edge-triggered)	0xCC0
0x34	$\overline{\text{IRQ20}}$	EOnCE interrupt (edge-triggered)	0xD00
0x35	$\overline{\text{IRQ21}}$	Reserved	0xD40
0x36	$\overline{\text{IRQ22}}$	Reserved	0xD80
0x37	$\overline{\text{IRQ23}}$	Reserved	0xDC0
0x38	$\overline{\text{NMI0}}$	HDI16: External Host $\overline{\text{NMI}}$	0xE00
0x39	$\overline{\text{NMI1}}$	Reserved	0xE40
0x3A	$\overline{\text{NMI2}}$	Bus controller (memory write error)	0xE80
0x3B	$\overline{\text{NMI3}}$	Bus controller (non-aligned error)	0xEC0
0x3C	$\overline{\text{NMI4}}$	Reserved	0xF00
0x3D	$\overline{\text{NMI5}}$	Reserved	0xF40
0x3E	$\overline{\text{NMI6}}$	Reserved	0xF80
0x3F	$\overline{\text{NMI7}}$	SIC $\overline{\text{NMI}}$, for example, S/W watchdog, external $\overline{\text{NMI}}$, parity error	0xFC0

7.4 Interrupt Programming Examples

This section describes how to use the PIC programming model for IRs and NMIs. The programming examples include the following functionality:

- Setting the interrupt base address in the VBA register.
- Initializing the stack pointer.
- Masking interrupts in the MSC8101 status register.
- Masking, unmasking, and programming IR properties in the ELIRx registers.
- Clearing a pending $\overline{\text{IRQ}}$ in the IPRx register.
- Using interrupt service routines longer than 64 bytes

The VBA holds the 20 MSB of the interrupt table base address. Consequently, the 12 LSB of this register must be cleared. At bootstrap, the VBA is initialized to the ROM base address (0x00F80000), and the stack pointer is initialized to 0x68000. You can change this value before issuing a call to any subroutine, since, depending on the specific software, this address may not be available for the stack. At reset, the SC140 core disables all interrupts. When an IR occurs, the status register is pushed onto the stack and the interrupt priority level (IPL) of the current IR is written to SR[23–21]. All IRs with a priority level less than or equal to the IPL of the current IR are masked.

The following example programs the interrupt base address in the VBA, initializes the stack pointer, and enables interrupts with priority levels of 6 or 7 only. All interrupts with priority level 5 or less are masked.

```
...
;Programming the VBA register to address 0x5000
move.l #$5000,vba
;Initializing the stack pointer to address 0x68000
move.l #$68000,r0
nop
tfra r0,sp
...
...
; Masking interrupts of priority 0,1,2.
bmclr #$00a0,sr.h
...
```

7.4.1 PIC Programming

In the PIC ELIRA–ELIRF registers, you can configure the priority level and select the trigger mode for each interrupt. On reset, all IRs are masked (set to priority 0) and configured as level-triggered. The following example shows how to assign priority 5 to the SIC interrupt, priority 4 to the DMA interrupt, and priority 6 to the SMI interrupt. This example also configures the SMI interrupt as edge-triggered, and the other two interrupts as level-triggered.

```
...
; BASE0 is 0x00f00000
ELIRE equ $00f01c20
IRQ16 equ $50c00
IRQ18 equ $50c80
IRQ19 equ $50cc0
; VBA is set to 0x50000
move.l #$50000,vba
; assign priority 5 to SIC (irq 16) and priority 4 to DMA (irq 18)
; assign priority 6 to SMI (irq 19) and set to edge trigger.
move.w #$e405,ELIRE
...
org p:IRQ16
; interrupt routine for SIC
rte
org p:IRQ18
; interrupt routine for DMA
rte
org p:IRQ19
; interrupt routine for SMI
rte
```

7.4.2 Clearing Pending Requests

The first task that an interrupt routine normally handles is to clear pending requests by writing to IPRx. If the size of the interrupt routine is larger then 64 bytes, you can use service routines to accommodate unlimited code size. The following example illustrates a typical interrupt routine that uses a service routine. This example also demonstrates the use of the **ei** and **di** instructions, which enable and disable IRs, respectively.

```
IPRB equ $00f01c38
...
```

```
org p:IRQ16
; interrupt routine for SIC
di ; disable any IR
jsr SIC_IRQ
nop
ei ; enable IR
rte
...
org p:SIC_IRQ
; clear pending interrupt in IRPB
move.w #$1,IPRB
; interrupt service routine to handle SIC
...
rts
```

7.4.3 EFCOP Programming Examples

```
;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; Interrupt Initialization                                                    ;
                                                                              ;
ELIR_A          equ             $7007   ;       enable IRQ0 - IRQ3          ;
ELIR_B          equ             $0000   ; ;
IRQ0_SUB        equ             INIT                                        ;
IRQ1_SUB        equ             INIT+$100                                   ;
IRQ2_SUB        equ             INIT+$200                                   ;
IRQ3_SUB        equ             INIT+$300                                   ;
IRQ4_SUB        equ             INIT+$400                                   ;
;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~;


;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~;
;;       NM interrupt handle subroutine (SIC NMI vector)       ;
;; ---------------------------------------------------------------     ;
;
        org p:NMI7                                                      ;
NMI
        nmis    7                                                       ;
```

```
        rte                                                                 ;
;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~;
;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~;
;;      interrupt handle subroutine (input buffer not full)                 ;
                                                                            ;
        org p:IRQ0                                                          ;
        irqs    0                                                           ;
        jsr             IRQ0_SUB                                            ;
        rte                                                                 ;;
;; ------------------------------------------------------------------       ;
;;      interrupt handle subroutine (input buffer empty)                    ;
;; ------------------------------------------------------------------       ;
        org p:IRQ1                                                          ;
        irqs    1                                                           ;
        jsr             IRQ1_SUB                                            ;
        rte                                                                 ;;
;; ------------------------------------------------------------------       ;
;;      interrupt handle subroutine (output buffer full)                    ;
;; ------------------------------------------------------------------       ;
        org p:IRQ2                                                          ;
        irqs    2                                                           ;
        jsr             IRQ2_SUB                                            ;
        rte                                                                 ;;
;; ------------------------------------------------------------------       ;
;;      interrupt handle subroutine (output buffer not empty)               ;
;; ------------------------------------------------------------------       ;
        org p:IRQ3                                                          ;
        irqs    3                                                           ;
        jsr             IRQ3_SUB                                            ;
        rte                                                                 ;;
;; ------------------------------------------------------------------       ;
;;      interrupt handle subroutine (update coefficient done)               ;
;; ------------------------------------------------------------------       ;
        org p:IRQ4                                                          ;
        irqs    4                                                           ;
```

```
        jsr             IRQ4_SUB                                        ;
        rte                                                             ;
                                                                        ;
;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~;

;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~;
        ;; PIC init                             ;
        ; enable VAB in PIC-SR                  ;
        write_w #$0000,PICSR                    ;
                                                ;
        write_w #ELIR_A,ELIRA   ; irq3 enable   ;
        write_w #ELIR_B,ELIRB                   ;
        write_w #$0000,ELIRC                    ;
        write_w #$0000,ELIRD                    ;
        write_w #$0000,ELIRE                    ;
        write_w #$0000,ELIRF                    ;
;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~;
;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~;
;;     enable interupts
      bmclr #$00e0,sr.h                       ;
;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~;
;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~;
;;      interrupt handle subroutine                                     ;
;; ---------------------------------------------------------------      ;
;;      interrupt handle subroutine (input buffer not full)             ;
;; ---------------------------------------------------------------      ;
        org p:IRQ0_SUB                                                  ;
                                                                        ;
        write_l (r8)+,(r5)      ;       write long ( 32 bit ) to FDIR     ;
        nop                                                             ;
        deceq d3                                                        ;
        jf IBNF                                                         ;
        move.l  EFCTL,d5                ;       clear FINFIE bit        ;
        nop                                                             ;
        bmclr   #$0800,d5.l                                             ;
        nop                                                             ;
```

```
        move.l  d5,EFCTL                                                ;
IBNF                                                                    ;
        rts                                                             ;
;; ---------------------------------------------------------------      ;
;;      interrupt handle subroutine (input buffer empty)                ;
;; ---------------------------------------------------------------      ;
        org p:IRQ1_SUB                                                  ;
        dosetup0        loop0                                           ;
                doen0   #4                                              ;
                nop                                                     ;
                nop                                                     ;
        loopstart0                                                      ;
loop0                                                                   ;
        write_l (r8)+,(r5)                                              ;
        deceq d3                ;       decrement the SRC_COUNT         ;
        nop                                                             ;
        nop                                                             ;
        nop                                                             ;
        nop                                                             ;
        loopend0                                                        ;
        cmpeq.w #0,d3                                                   ;
        jf IBE                                                          ;
        move.l  EFCTL,d5                ;       clear FIEIE bit         ;
        bmclr   #$0400,d5.l                                             ;
        move.l  d5,EFCTL                                                ;
IBE                                                                     ;
        rts                                                             ;
                                                                        ;
;; ---------------------------------------------------------------      ;
;;      interrupt handle subroutine (output buffer full)                ;
;; ---------------------------------------------------------------      ;
        org p:IRQ2_SUB                                                  ;
        dosetup1        loop1                                           ;
                doen1   #4                                              ;
                nop                                                     ;
```

```
                nop                                                     ;
        loopstart1                                                      ;
loop1                                                                   ;
        write_l (r4),(r6)+                                              ;
        deceq d4                ;       decrement the DST_COUNT         ;
        nop                                                             ;
        nop                                                             ;
        nop                                                             ;
        nop                                                             ;
        loopend1                                                        ;
        cmpeq.w #0,d4                                                   ;
        jf OBF                                                          ;
        move.l  EFCTL,d5                ;       clear FOFIE bit         ;
        bmclr   #$1000,d5.l                                             ;
        move.l  d5,EFCTL                                                ;
OBF                                                                     ;
        rts                                                             ;
                                                                        ;
;; ---------------------------------------------------------------      ;
;;      interrupt handle subroutine (output buffer not empty)           ;
;; ---------------------------------------------------------------      ;
        org p:IRQ3_SUB                                                  ;
        write_l EFDOR,(r6)+     ;       read single word from FDOR      ;
        deceq d4                                                        ;
        jf OBNE                                                         ;
        move.l  EFCTL,d5                ;       clear FONEIE bit        ;
        bmclr   #$2000,d5.l                                             ;
        move.l  d5,EFCTL                                                ;
OBNE                                                                    ;
        rts                                                             ;
                                                                        ;
;; ---------------------------------------------------------------      ;
;;      interrupt handle subroutine (update coefficient done)           ;
;; ---------------------------------------------------------------      ;
        org p:IRQ4_SUB                                                  ;
```

```
        irqs    4                                                       ;
        write_l EFDOR,(r6)+     ;read single word from FDOR             ;
        write_l (r8)+,EFDIR     ;write single word to FDIR after UCD    ;
        deceq d4                                                        ;
        jf UPD                                                          ;
        move.l  EFCTL,d5                ;       clear FUDIE bit         ;
        bmclr   #$0200,d5.1                                             ;
        move.l  d5,EFCTL                                                ;
UPD                                                                     ;
        rts                                                             ;
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~;
```

7.4.4 PIC Macros

Following is an irqs macro to clear an edge-triggered interrupt request:

```
irqs    MACRO   irq_num
        ;; irq num from 0-23
        nop
        if      irq_num<16
                move.l #irq_num,d6
                move.l #1,d7
                lsll d6,d7
                nop
                move.w d7,IPRA
        else
                move.l #irq_num-16,d6
                move.l #1,d7
                lsll d6,d7
                nop
                move.w d7,IPRB
        endif
        nop
        ENDM
```

Following is an nmis macro to clear the edge-triggered interrupt request:

```
nmis    MACRO   nmi_num
```

```
        ;; nmi num from 0-7
        nop
        move.w PICSR,d6
        nop
        move.l #nmi_num+8,d6
        move.l #1,d7
        lsll d6,d7
        nop
        move.w d7,IPRB
        nop
        ENDM
```

7.4.4.1 Examples of SIC Interrupts

```
#include"Sic.h"
Sic_Branch SIC_BranchTable[64];


main ()
{
...... M A I N - P R O G R A M ......
}
void SPI_InitInterrupt()
{
//create entry in SIC branch table:
SIC_BranchTable[SIC_SPI].Interrupt = SPI_Interrupt;
//the interrupt routine for the SPI must be called SPI_Interrupt
SIC_BranchTable[SIC_SPI].Serial = NULL; //temp
//Configure SPI:
   IMM->spi_spie = 0xFF;  //clear any previous SPI interrupt events in SPI-reg.
   IMM->spi_spim = 0x01;  //enable only Rx interrupt! Although we only use the
        //Tx line, we need the Rx interrupt to make sure that
        //we get an interrupt after all bits have been sent.
        //The Tx Irq is useless here because it appears too
        //early. All this is neccessary to set CS back to 1.
        //Configure the SIU_CPM interrupt controller (SIC):
        //When a SPI interrupt occurs, bit17 of the SIPNR_L will be
        //set. Furthermore the corresponding interrupt code (SPI = 2)
        //0x08000000 is visible in SIVEC (dependent on priority
        //level).
  IMM->ic_sipnr_l = 0x00004000; //clear any previous SPI interrupt
  IMM->ic_simr_l  |= 0x00004000; //enable SPI interrupt
```

```
return;
} // end SPI_InitInterrupt()
/**************/
/* SPI_Interrupt */
/**************/
void SPI_Interrupt(void *spi)
{
/*
This function is called only when a SIC/SPI interrupt occurs.
*/

  QMM->Iprb  = 0x0001; //clear SIC interrupt flag in Irq Pending Register B:
IMM->ic_sipnr_l = 0x00004000; //clear SPI interrupt flag in SIC-reg.
IMM->spi_spie |= 0xFF;         //clear all SPI irq flags in SPIE-reg.
}

typedef struct sicbrancht Sic_Branch;
struct sicbrancht
{
void (*Interrupt)(void*);
void *Serial;
};

/* ------------------------------------------------------------- */
/* Interrupt codes used for SIVEC and SIC branch table */

/* ------------------------------------------------------------- */

enum SIC_IRQ_CODE {  SIC_ERROR,    SIC_I2C,               SIC_SPI,   SIC_RISC,
        SIC_SMC1,            SIC_SMC2,                   SIC_IDMA1,  SIC_IDMA2,
        SIC_IDMA3,           SIC_IDMA4,                  SIC_SDMA, SIC_RESV01,
        SIC_TIMER1,          SIC_TIMER2,                 SIC_TIMER3, SIC_TIMER4,
        SIC_TMCNT,           SIC_PIT, S                  IC_RESV02,   SIC_IRQ1,
        SIC_IRQ2,            SIC_IRQ3,                   SIC_IRQ4,   SIC_IRQ5,
        SIC_IRQ6,            SIC_IRQ7,                   SIC_RESV03, SIC_RESV04,
      SIC_RESV05,       SIC_RESV06,          SIC_RESV07,    SIC_RESV08,
      SIC_FCC1,         SIC_FCC2,            SIC_FCC3,      SIC_RESV09,
      SIC_MCC1,         SIC_MCC2,            SIC_RESV10,    SIC_RESV11,
      SIC_SCC1,         SIC_SCC2,            SIC_SCC3,      SIC_SCC4,
      SIC_RESV12,       SIC_RESV13,          SIC_RESV14,    SIC_RESV15,
      SIC_PC15,         SIC_PC14,            SIC_PC13,      SIC_PC12,
      SIC_PC11,         SIC_PC10,            SIC_PC09,      SIC_PC08,
      SIC_PC07,         SIC_PC06,            SIC_PC05,      SIC_PC04,
      SIC_PC03,         SIC_PC02,            SIC_PC01,      SIC_PC00

                  };

void SIC_InitInterrupt();
void PIC_Code();
void SIC_IrqHandler();
```

7.4.4.2 Examples of SIC Interrupts

```
#define VBA  0x00000000 //Value for Vector Base Address Register (VBA)
#define SET_VBA          asm("move.l #$00000000,vba") //macro to set vba register
#include "netcomm.h"         //Global defines
#include "msc8101.h"
#include "MM.h"
#include "Sic.h"
#include"segmentor.h"
extern BasePtrIMM *IMM;           //Internal Memory Map base pointer
extern QMMBank0 *QMM;

//This table contains all SIC interrupt function addresses.
extern Sic_Branch SIC_BranchTable[64];
//64 entries, each 4 bytes => 256 bytes of memory

/*********************************************/
/* SIC_InitInterrupt                                                 */
/*********************************************/

void SIC_InitInterrupt()
{
/*
   Note! The SIU Interrupt Vector Register (SIVEC) is not used here,
   so the 'SIC_IrqHandler' must find out the reason for
   every SIC interrupt.

  */

/* Establish relevant pointers and registers for SIC-Irq usage in PIC */
   SET_VBA;  // macro to set vba register
   //Configure the Programmable Irq Controller (PIC) to enable SIC Irqs:
   //Make sure that the Vector Base Address Reg. (VBA) is set correctly!
   //At a SIC-Irq the core jumps into the PIC Irq routing table (base is VBA)
   //to the offset 0x0C00.

   //Create a PIC Irq routing table entry for SIC-Irq; that is, copy assembly code
   //to the provided location of the PIC Irq Routing table. (Note! Here the maximum
   //number of bytes per entry are copied, although maybe less are used. The
   //rationale is that the number of bytes does not need to be changed when
   //the assembly code changes.)
   //SIC_BranchTable = (Sic_Branch*)get_seg(DATA,64*sizeof(Sic_Branch));

  memcpy((void*)(VBA + 0x0c00), &PIC_Code,0x50);


  //clear whole SIC branch table

  memset(SIC_BranchTable, 0 , sizeof(SIC_BranchTable));


  //clear any previous SIC interrupt in Irq Pending Register B:

  QMM->Iprb  = 0x0001;
```

```
   //Configure Edge/Level-Triggered Irq Priority Register E to enable IRQ16:
   //Irq16 (SIC), level-triggered, Irq priority level = 3

  QMM->Elire = (QMM->Elire & 0xFFF0) | 0x0003;


   IMM->ic_sipnr_l  = 0xFFFFFFFF; //clear any previous interrupts in SIC-reg.
   IMM->ic_simr_l   = 0x00000000; //disable all interrupts


  asm("ei"); //enable interrupts

}


#pragma interrupt PIC_Code
   void PIC_Code()

 {

   /* This function must be copied to the location 'VBA+0x0C00' within the PIC
   Irq routing table. Ensure that max. 50 bytes (64-(6+6+2)) of assembly code
   are included into this function. */


   //asm("jsr _QCtxtSave");

   //6 bytes for JSR, created by #pragma to push registers into stack
   asm("di");
   SIC_IrqHandler();
   asm("ei");

   //asm("jsr _QCtxtRestore"); //6 bytes for JSR, created by #pragma to pop
                             //registers from stack
   //asm("rte");               //2 bytes, created by #pragma to return from
                             //interrupt


 }

/*********************************************/
/* SIC_IrqHandler                                                    */
/*********************************************/

void SIC_IrqHandler()
{

 /*Ensure that the SIC interrupt branch
   table is allocated (global) with 'UWORD SIC_BranchTable[64]'.*/

UBYTE sivec;

sivec = (IMM->ic_sivec>>26);//IMM->ic_sivec bits 0-31 wheras only bits 0-5 matters

 SIC_BranchTable[ sivec].Interrupt(SIC_BranchTable[ sivec].Serial);

}
```

```
/*  asm("move.l _IMM,r0");       //IMM base address
    asm("move.l #$00010C04,r1"); //SIVEC offset

  asm("nop");

  asm("adda r1,r0");                        //SIVEC address

  asm("nop");

   asm("moveu.b (r0),r1");                  //get SIVEC to figure out the irq source
   asm("move.l #_SIC_BranchTable,r0");      //get branch table base
   asm("nop");
   asm("adda r0,r1");                       //add sivec to branch table base
   asm("nop");
   asm("move.l (r1),r0");                   //get address of irq function from
                                            //branch
                                            //table
   asm("nop");
   asm("jsr r0");                           //call interrupt function
```

Host Interface (HDI16) 8

The HDI16 host port is an MSC8101 DSP peripheral featuring a 16-bit-wide parallel port for communication with a host processor. This parallel port is a full-duplex and double-buffered slave interface. It transfers data between a host or DMA controller and the DSP, and it transfers commands from the host to the DSP. The 16-bit-wide HDI16 data bus handles 16-bit, 32-bit, 48-bit, and 64-bit data transfers. In addition, an 8-bit data mode handles 8-bit, 16-bit, 24-bit, and 32-bit data transfers on an 8-bit-wide data bus. Programmable options provide a glueless connection between the DSP and other MSC8101 devices or MPC860 and MPC8260 host processors. Minimal glue logic is required to interface the HDI16 port to several industry-standard processors and buses, such as the ISA bus and theFreescale 68K family.

This chapter tells you how and why to operate the HDI16 in different modes, discusses the various handshaking protocols for managing data transfers and the pros and cons of each, and examines how you can use the innovative host command feature of the host interface. For information on the HDI16 and power-on reset, see **Section 2.4**, *Configuring a Multi-MSC8101 System Connected Via the Host Port,* on page 1-11. For a detailed discussion of the HDI16 registers and programming model, consult the *MSC8101 Reference Manual*.

8.1 HDI16 Programming Basics

Note: The HDI16 interface is multiplexed with the upper 32 bits of the system data bus. To use the HDI16, you must enable the port by pulling up the HPE/EE1 line up at reset and set the ISPS bit in the Hard Reset Configuration Word to configure the system data bus as 32 bits to release the upper 32 bits for use by the HDI16. After the HDI16 is enabled at reset, you can set the HPCR[HEN] to make the interface active.

The HDI16 interface is a “slave” peripheral, which means that a “master” performs all the external read and write accesses necessary to communicate with the HDI16 port. In this chapter, the master is called the “host.” The internal MSC8101 resources that manipulate the HDI16 interface are referred to as the “DSP.”

As **Figure 8-1** shows, the HDI16 peripheral has two register banks:

- *Host-side register bank*. Accessible only to the host from the external HDI16 bus.
- *DSP-side register bank*. Accessible only to the DSP internal resources.

For host-to-DSP transfers, the host writes the host-side registers and the DSP reads the DSP-side registers; for DSP-to-host transfers, the DSP writes the DSP-side registers and the host reads the host-side registers. The separate receive and transmit data paths are double buffered for efficient, high-speed asynchronous transfers. On the DSP side, the Host Transmit Data Register (HOTX) and the Host Receive Data Register (HORX) are FIFOs of four 64-bit words. The host-side transmit data path (used for transfers from the host to the DSP) is also the DSP-side *receive* path. The host-side receive data path (used for transfers from the DSP to the host) is also the DSP-side *transmit* path.

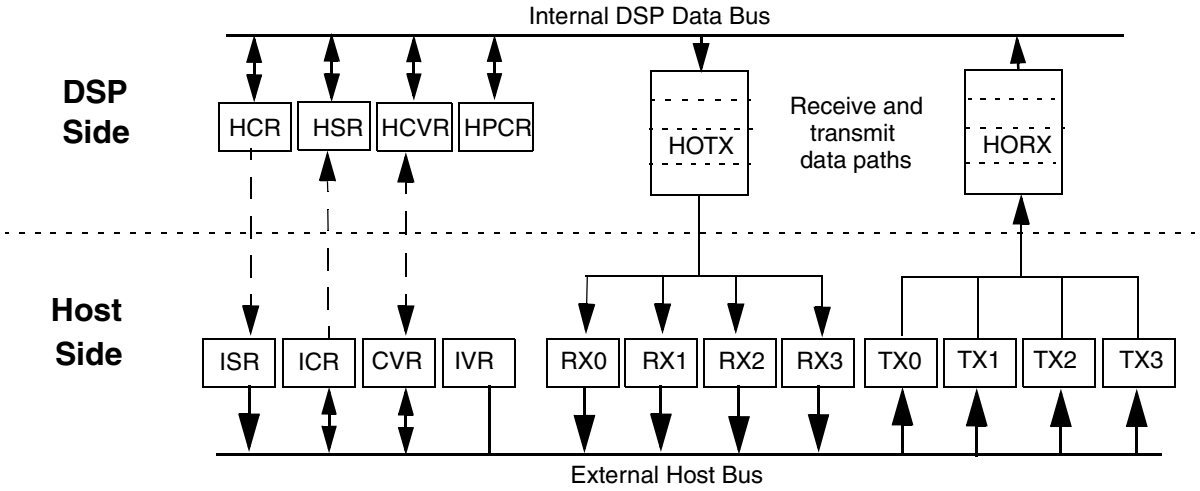


Figure 8-1. HDI16 Programmer's Model

8.1.1 Host-Side Model

To the host, the HDI16 appears as eleven 16-byte-wide locations mapped in its external address space, as indicated in **Table 8-1**. The control registers provide the host with control and status information and allow the use of host commands. The data registers enable data transfers. Finally, the reset configuration registers are used only during reset to configure the MSC8101 hardware.

Table 8-1. Host-Side Programmer's Model

Type	Host Address	Host-Side Register	Mnemonic
Control	0x0	Interface Control Register	ICR
	0x1	Command Vector Register	CVR
	0x2	Interface Status Register	ISR
	0x3	Reserved	—

Table 8-1. Host-Side Programmer's Model (Continued)

Type	Host Address	Host-Side Register	Mnemonic
Data	0x4	Transmit/Receive Register 3	TX3/RX3
	0x5	Transmit/Receive Register 2	TX2/RX2
	0x6	Transmit/Receive Register 1	TX1/RX1
	0x7	Transmit/Receive Register 0	TX0/RX0
Reset Configuration	0x8	Reset Configuration Register 0	RSCFG0
	0x9	Reset Configuration Register 1	RSCFG1
	0xA	Reset Configuration Register 2	RSCFG2
	0xB	Reset Configuration Register 3	RSCFG3
	0xC–0xF	Reserved	—

8.1.2 DSP-Side Model

To the SC140 core, the DSP-side registers appear as six registers mapped in the QBus memory space, as indicated in **Table 8-2**, allowing MSC8101 instructions and addressing modes to access these registers. Four 16-bit control registers provide the SC140 core control of the HDI16 functionality, and two 64-bit data registers transfer the data.

Table 8-2. DSP-Side Programmer's Model

Type	Host Address (HA[0–3])	Host-Side Register	Mnemonic
Control	0x0000	Host Control Register	HCR
	0x0040	Host Status Register	HSR
	0x0060	Host Command Vector Register	HCVR
—	0x0020	Host Port Control Register	HPCR
Data	0x0080	Host Transmit Data FIFO	HOTX
	0x00A0	Host Receive Data FIFO	HORX

8.2 Operating in Different Data Transfer Modes

The HDI16 offers two modes for transferring data between the host and the DSP. Normal mode refers to non-DMA transfers, and DMA mode refers to transfers using an external DMA controller (not be confused with the operation of the internal DMA controller). The Host DMA Mode Enable bit in the Host Port Control Register, HPCR[DMA], defines the mode of operation as shown in **Table 8-3**. The hardware can be set up so that both transfer modes, Normal and DMA, can be used on the same bus, though not simultaneously. In Normal mode, the host transfers data one access at a time, with each access requiring an address and data transaction on the host bus. In host DMA mode, the data is transferred on the host bus without requiring address transactions, thus giving faster data access.

Table 8-3. Normal and DMA Mode Selection

HPCR[DMA]	Mode
0	Normal
1	DMA

To accommodate these modes, each HDI16 pin serves multiple purposes, as **Table 8-4**, **Table 8-5**, and **Table 8-6** show.

Table 8-4. Normal and DMA Mode Pin Functions

Pin	Normal Mode	DMA Mode
HD[15–0]	HD[15–0]	HD[15–0]
HA[3–0]	HA[3–0]	—
HCS1	HCS1	—
HCS2	HCS2	—

The Host Port Control Register (HPCR) and the HDDS, HDSP, and H8BIT pins are used to set the following options:

- *Width of the data bus*. The HDI16 is programmed to use the 8-bit-wide or 16-bit-wide host data bus as indicated by setting the HPCR[H8BIT] bit or by driving the H8BIT pin low to select 16-bit mode. The functionality of the HD[15–0] data pins (indicated in **Table 8-4**) applies to a 16-bit wide host data bus. For an 8-bit host data bus width, only the HD[7–0] data pins are used.
- *Use of single or dual read/write strobes (signals)*. An OR of the HPCR[HDDS] bit and the HDDS pin indicate a single or dual strobe. In a single-strobe bus, the Host Data Strobe pin (HDS) indicates that valid data is present on the bus. The Host Read/Write pin (HRW) indicates the type of transaction in process (read or write). In dual-strobe mode, the Host Read (HRD) and the Host Write (HWR) lines each indicate data validity and transaction type.

Table 8-5. Single- and Dual-Strobe Bus Pin Functionality

HDI16 Pin	Single-Strobe Bus	Dual-Strobe Bus
HRW/HRD	HRW	$\overline{\text{HRD}}$/HRD
HDS/HWR	$\overline{\text{HDS}}$/HDS	$\overline{\text{HWR}}$/HWR

- *Polarity of the read/write strobes*. Signals can be programmed as active high or active low, as indicated by an ORing of the HPCR[HDSP] bit and the HDSP pin.

The HDI16 can be programmed to use a single host request line or dual host request lines.

Table 8-6. Single and Dual Host Request Lines

Pin	Single Request	Dual Request
HREQ/HTRQ	$\overline{\text{HREQ}}$/HREQ	$\overline{\text{HTRQ}}$/HTRQ
HACK/HRRQ	$\overline{\text{HACK}}$/HACK	$\overline{\text{HRRQ}}$/HRRQ

Refer to **Section 8.3.3**, *Host Requests* for information on the host request lines.

8.2.1 Normal Mode

In Normal mode, the HDI16 port appears to the host as eleven 16-bit wide registers mapped to its external memory, much as a 16-bit SRAM would appear. To enable Normal mode, the DSP core must clear the Host Port Control Register's DMA bit (HPCR[DMA]) and set the Host Enable bit (HPCR[HEN]). Once Normal mode is enabled, the HCR[HICR] bit determines whether the DSP-side Host Control Register (HCR) or the host-side Interface Control Register (ICR) defines the size of the data transferred (see **Table 8-7**).

Table 8-7. Transfer Control in Normal Mode

HCR[HICR]	Defines Transfer	Register
0	DSP	HCR[HDM[0–2]]
1	Host	ICR[HDM[0–1]]

If the host defines the Normal mode transfer (HCR[HICR] = 1), the host-side Interface Control Register HDM[0–1] bits define the size as indicated in **Table 8-8**. The ICR HDM[0–1] bits are reflected on the DSP-side HCR[HM] bits, allowing the DSP-side to determine the data transfer size.

Table 8-8. Host-Defined Normal Mode Data Size

ICR[HDM0]	ICR[DM1]	Data Size	Last Address
0	0	64-bit	0x7
0	1	48-bit	0x6
1	0	32-bit	0x5
1	1	16-bit	0x4

Conversely, if the DSP defines the data size of the Normal mode transfer (HCR[HICR] = 0), the DSP-side HCR[HDM] bits select the data size, as indicated in **Table 8-9**. The HCR[HDM] bits are reflected on the host-side ICR[HDM] bits, allowing the host side to determine the DMA data transfer size.

Table 8-9. DSP-Defined Normal Mode Data Size

HCR[HDM1]	HCR[HDM2]	DMA Data Size	Last Address
0	0	16-bit	0x7
0	1	32-bit	0x6
1	0	48-bit	0x5
1	1	64-bit	0x4

Recall that the host can read only the HDI16 host-side Receive Data Registers (RX[0–3]) or write the HDI16 host-side Transmit Data Registers (TX[0–3]). In DSP-to-host transfers, the DSP first writes data to the DSP-side Transmit Data Register (HOTX) FIFO. This data is automatically transferred to the host-side RX[0–3] when these registers are empty. To access this data, the host first reads RX0/TX0 at address offset 0x7 and continues, depending on the data size being used, with reads to RX1/TX1 at address 0x6, RX2/TX2 at address 0x7, and finally RX3/TX3 at address 0x4. The final 16-bit portion of the data is read from the corresponding “last address” register (see **Table 8-8** and **Table 8-9**), automatically causing the transfer of any data available in the DSP-side HOTX FIFO to the host-side RX[0–3] registers.

Similarly, in host-to-DSP transfers, the host begins by writing the host-side RX0/TX0 at address offset 0x7 and continues, depending on the data size being used, with writes to RX1/TX1 at address 0x6, RX2/TX2 at address 0x5, and finally RX3/TX3 at address 0x6. The final 16-bit portion of the data is written to the corresponding “last address” register (see **Table 8-8** and **Table 8-9**), automatically causing the transfer of the entire datum from the host-side TX[0–3] registers to the DSP-side HORX FIFO. Thus, the HDI16 supports 16-bit, 32-bit, 48-bit, and 64-bit transfers.

The minimum hardware set-up necessary for using the HDI16 port in Normal mode is a chip select, four address lines to access the eleven HDI16 host-side registers, sixteen data lines (in 16-bit mode), and two data strobe lines. **Figure 8-2** shows a simple hardware set-up that supports Normal mode. The host bus performs the following actions:

1. Selects the HDI16 device ($\overline{\text{HCS1}}$).
2. Indicates the direction of the transfer (HRD or HWR)
3. Asserts the address of the HDI16 register to be accessed (HA[0–3]).
4. Accesses the data (HD[0-15]).

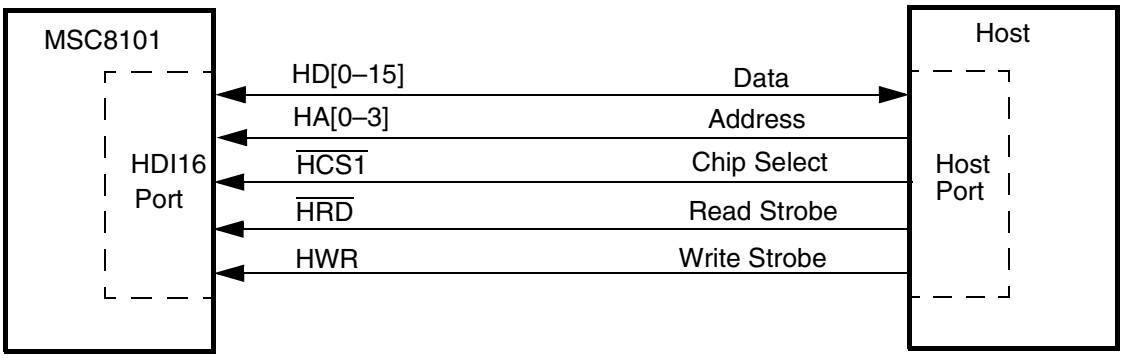


Figure 8-2. Example Hardware Set-up for Normal Mode Transfers

The assembly language equate listed in **Example 8-1** defines the initial values of the Host Port Control Register (HPCR) to set up the HDI16 for the hardware set-up represented in **Figure 8-2**.

Example 8-1. HPCR Values for Example in **Figure 8-2**

```
INIT_HPCR EQU  0x1000
                ; [0] = HAP = 0 => Not used. Write to 0
                ; [1] = HRP = 0 -> Not used. Write to 0
                ; [2] = HCSP = 0 -> HCS1 active low
                ; [3] = HDDS = 1 -> Dual strobe
                ; [4:5] = reserved = 00
                ; [6] = HDSP = 0-> HRD & HWR active low
                ; [7] = reserved = 0
                ; [8] = HEN = 0 -> Host interface disabled
                ; [9] = H8BIT = 0 -> 16-bit mode used
                ; [10:13] = reserved = 0000
                ; [14] = DMA = 0 -> normal mode
                ; [15] = OAD = 0 -> Not used. Write to 0
```

This initialization configures the HDI16 for 16-bit Normal mode transfers by clearing the HPCR[H8BIT] and HPCR[DMA] bits. Bus transactions are programmed to use dual, active low, read ($\overline{\text{HRD}}$) and write (HWR) strobes by setting HPCR[HDDS] and clearing HPCR[HDSP]. Active low Chip Select ($\overline{\text{HCS1}}$) is programmed by clearing HPCR[HCSP].

The HPCR[HEN] bit is initially cleared, disabling the HDI16 port. To assure proper operation, the HPCR[HAP, HRP, HCSP, HDDS, HDSP, and H8BIT] bits should be set only when HPCR[HEN] is cleared. After these bits are set as required, the HDI16 port is enabled by setting the HPCR[HEN] bit. The MSC8101 code listed in **Example 8-2** meets these requirements and initializes and enables the HDI16 port.

Example 8-2. Initializing the HDI16 Port

```
move.l #HPCR_ADDR,r1                    ; r1 = HPCR address
move.w #INIT_HPCR,(r1)                  ; initialize HPCR
bmset.w #$80,(r1)                      ; enable HDI16
```

The following list summarizes the steps the DSP performs to initialize the HDI16 port in Normal mode:

1. Initialize the Host Port Control Register (HPCR), ensuring that HPCR[HEN] is clear and setting the HPCR[HDDS] bit to select the dual-strobe mode.
2. Enable the HDI16 by setting HPCR[HEN].
3. Initialize the HCR[HICR] bit to indicate which register (HCR or ICR) defines the data size.
4. Initialize the DSP-side HCR or the host-side ICR for the desired data size (recall that only the host can access host-side registers)

8.2.2 Host DMA Mode

Host DMA mode supports external DMA controllers connected to the HDI16 on the host bus and should not be confused with the operation of the internal DMA to the MSC8101 DSP. In Host DMA mode, data is transferred in bursts without the need to drive a new address on the host bus address lines for every transfer. To enable Host DMA mode, the DSP core must set the Host Port Control Register DMA bit (HPCR[DMA]). Once DMA mode is enabled, the HCR[HICR] bit determines whether the DSP-side Host Control Register (HCR) or the host-side Interface Control Register (ICR) defines the characteristics (direction and data size) of the DMA transfer (see **Table 8-10**).

Table 8-10. Transfer Control in Host DMA Mode

HCR[HICR]	Defines DMA	Register
0	DSP	HCR[HDM[0–2]]
1	Host	ICR[HDM[0–1]]

If the host (external DMA controller) defines the DMA (HCR[HICR] = 1), the host-side ICR[RREQ] bit defines the direction of the DMA transfers (see **Table 8-11**) and the ICR[HM[0–1]] bits select the DMA data size (see **Table 8-12**). The HM[0–1] bits are reflected on the DSP-side HCR[HM] bits, allowing the DSP side to determine the DMA data transfer size. The RREQ bit is reflected in the HCR[RREQ] bit, so the DSP side can determine the DMA direction.

Table 8-11. Host-Defined DMA Transfer Direction

ICR[RREQ]	DMA Direction
0	Host-to-DSP
1	DSP-to-host

Table 8-12. Host-Defined DMA Data Size

ICR[HM0]	ICR[HM1]	DMA Data Size	Last Address
0	0	16-bit	0x7
0	1	32-bit	0x6
1	0	48-bit	0x5
1	1	64-bit	0x4

If the DSP defines the characteristics of the DMA (HCR[HICR] = 0), the DSP-side HCR HDM[0–2] bits select the DMA data size and direction as indicated in **Table 8-13**. $\overline{\text{HDM0}}$ is reflected in ICR[HDM0] when read, and the HDM[1–2] bits are reflected on the host-side ICR[HDM] bits, allowing the host-side to determine the DMA data transfer size and direction.

Table 8-13. DSP-Defined DMA Data Size and Direction

HCR[HDM0]	HCR[HDM1]	HCR[HDM2]	DMA Direction	DMA Data Size
0	0	0	Host-to-DSP	64-bit
0	0	1		48-bit
0	1	0		32-bit
0	1	1		16-bit
1	0	0	DSP-to-host	64-bit
1	0	1		48-bit
1	1	0		32-bit
1	1	1		16-bit

There are two ways to set up the hardware connection for DMA transfers, depending on the method of acknowledging that valid data is on the host bus. Data always transfers over the HD[0–15] data lines (HD[0–7] in 8-bit mode). The HREQ output pin is always used to request DMA transfers from the host or DMA controller. However, valid data on the host bus can be acknowledged in two different ways, using the HACK input pin or the host address 0x4, as defined in **Table 8-14**.

Table 8-14. DMA Valid Data Acknowledgment

HPCR[OAD]	Data Valid
0	HACK pin is asserted
1	Host address 0x4 is asserted

Consider the hardware set-up shown in **Figure 8-3**. In this example, the hardware set-up uses sixteen data lines, the HREQ pin for DMA data transfer requests to the host, and the HACK pin for valid data acknowledgment. No address lines are required. An internal 2-bit address counter

preloaded with the values in ICR[9–10] determines which host-side data register is selected during the DMA transfer. The address counter is initialized using (ICR[INIT]). After each DMA transfer on the host data bus, the address counter is incremented to the next data register. When the address counter reaches the highest register, the counter is reloaded with the value in ICR[9–10]. Reloading the counter allows 16-bit, 32-bit, 48-bit, and 64-bit circular data transfers and eliminates the need for the host or DMA controller to supply an address transaction for each data transfer.

The HREQ pin is asserted to indicate a request for a data transfer (receive or transmit). The HACK input pin is used as a DMA acknowledge. For DSP-to-host transfers, the HDI16 writes data on the bus when HACK is asserted; for host-to-DSP transfers, the data is valid on the bus when HACK is asserted. The same principle would hold true if host address 0x4 were used for DMA data valid acknowledgment instead of HACK.

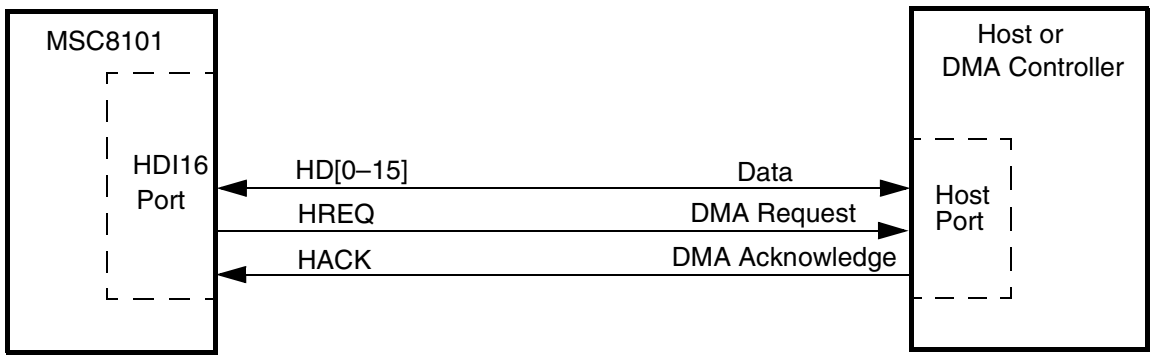


Figure 8-3. HDI16 Hardware set-up for DMA Mode Transfer

The assembly language equate listed in **Example 8-3** defines the initial register values to set up the HDI16 for the hardware set-up represented in **Figure 8-3**.

Example 8-3. Register Values for Example Shown in **Figure 8-3**

```
INIT_HPCR EQU  0xC002
               ; [0] = HAP = 1 -> HACK pin active high
               ; [1] = HRP = 1-> HREQ pin active high
               ; [2] = HCSP = 0 -> Not used. Write to 0
               ; [3] = HDDS = 0-> Not used. Write to 0
               ; [4:5] = reserved = 00
               ; [6] = HDSP = 0 -> Not used. Write to 0
               ; [7] = reserved = 0
               ; [8] = HEN = 0 -> Host interface disabled
               ; [9] = H8BIT = 0 -> 16-bit mode used
               ; [10:13] = reserved = 0000
               ; [14] = DMA = 1-> DMA mode
               ; [15] = OAD = 0 -> HACK pin for DMA acknowledge
INIT_HCR EQU   0x0600
               ; [0] = HF4 = 0 -> Not used. Write to 0
```

```
; [1] = HF5 = 0 -> Not used. Write to 0
; [2] = HF6 = 0 -> Not used. Write to 0
; [3] = HF7 = 0 -> Not used. Write to 0
; [4] = HICR = 0 -> HCR defines DMA direction and size
; [5-7] = HDM[0-2] = 110 -> DSP-to-host DMA,32-bit data
; [8] = reserved = 0
; [9] = DBTE = 0 -> DMA transmit burst disabled
; [10] = DBRE = 0 -> DMA receive burst disabled
; [11] = HCIE = 0 -> Host command interrupt disabled
; [12] = HFTIE = 0 -> Host transmit not full interrupt disabled
; [13] = HTEIE = 0 -> Host transmit empty interrupt disabled
; [14] = HRFIE = 0 -> Host receive full interrupt disabled
; [15] = HREIE= 0 -> Host receive not empty interrupt disabled
```

This initialization configures the HDI16 for a 16-bit bus and DMA mode transfers by clearing HPCR[H8BIT] and setting HPCR[DMA]. Bus transactions are programmed to use active high HACK and HREQ signals by setting HPCR[HAP] and HPCR[HRP]. The HPCR[HEN] bit is initially cleared, disabling the HDI16 port. To assure proper operation, the HPCR[HAP, HRP, HCSP, HDDS, HDSP] bits should be set only when HPCR[HEN] is cleared. After these bits are set as appropriate, then enable the HDI16 by setting the HPCR[HEN] bit. The MSC8101 code listed in **Example 8-4** meets these requirements and initializes and enables the HDI16 port. The HCR[HICR] bit is cleared, indicating that the HCR[HDM[0–2]] bits determine the DMA direction to be DSP-to-host and the data size to be 32 bits.

Example 8-4. Initializing the HDI16 Port

```
move.w  #HPCR_ADDR,r1          ; r1 = HPCR address
move.l  #HCR_ADDR,r2           ; r2 = HCR address
move.w  #INIT_HPCR,(r1)        ; initialize HPCR
bmset.w #8,(r1)                ; enable HDI16
move.w  #INIT_HCR,(r2)         ; initialize HCR
```

The following list summarizes the steps the DSP follows to initialize the HDI16 port in DMA mode:

1. Initialize the Host Port Control Register (HPCR), ensuring that the HPCR[HEN] bit is clear.
2. Enable the HDI16 by setting HPCR[HEN].
3. Initialize the HCR[HICR] bit to indicate which register (HCR or ICR) defines the DMA direction and data size
4. Initialize the DSP-side HCR or the host-side ICR for the desired DMA direction and data size (recall that only the host can access host-side registers)

8.3 Managing Data Transfers Via Handshaking Protocols

The HDI16 interface port is slave-only, so the host is the master of all bus transfers. In host-to-DSP transfers, the host writes data to the Transmit Word Registers (TX[0–3]). In DSP-to-host transfers, the host reads data from the Receive Word Registers (RX[0–3]). The DSP side has access only to the Host Receive Data Register (HORX) and the Host Transmit Data Register (HOTX). Available data automatically moves between the host-side data registers and the DSP-side data registers. This double-buffered mechanism allows for fast data transfers, but it creates a “pipeline” that can stall (if the pipeline is either full or empty) or cause erroneous data transfers (overwriting new data or reading old data). Several HDI16 port handshaking mechanisms are available to reduce the possible occurrence of such stalls and erroneous transfers.

For example, a host writing several pieces of data to the HDI16 port should first determine whether any data previously written to the Transmit Word Registers (TX[0–3]) has successfully transferred to the DSP side. A handshaking protocol makes this possible. If the host-side Transmit Word Registers (TX[0–3]) are empty, the host writes the data to these registers. The transfer to the DSP-side Host Receive Data Register (HORX) occurs only if HORX is not full (recall that the HORX is a FIFO of four 64-bit words). Similarly, the DSP core uses an appropriate handshaking protocol to move data from HORX to the receiving memory buffer or register. If the handshaking protocol were not used, the host might overwrite data not yet transferred to the DSP side, or the DSP might receive bogus data.

A similar situation occurs when the host performs multiple reads from the HDI16 port Receive Word Registers (RX[0–3]). The DSP side uses an appropriate handshaking protocol to determine whether the 64-bit Host Transmit Register (HOTX) FIFO is not full. If HOTX is not full, the DSP writes the data to this register. Data is transferred to the host-side Receive Word Registers (RX[0–3]) only if they are empty (that is, the host has previously read them). The host can then use any of the available handshaking protocols to determine when data is ready to be read.

The MSC8101 HDI16 port offers the following handshaking protocols for data transfers with the host:

- Software polling (DSP and host)
- DSP interrupts
- Host requests
- Direct Memory Access (MSC8101 internal DMA and host DMA)

The following sections discuss several factors that determine which protocol to use, including:

- The amount of data to be transferred
- The timing requirements for the transfer
- The availability of resources such as processing bandwidth and DMA channels

Recall that the transfers described here occur between the host and the DSP asynchronously. Each side transfers data at its own pace. However, using an appropriate handshaking protocol allows data to be transferred at optimal rates. Furthermore, the DSP and the host can use different handshaking techniques.

8.3.1 Software Polling

Software polling is the simplest handshaking protocol, but it can consume the most processing power. In software polling, the host or the DSP core reads status bits to determine the state of the HDI16 registers. While polling these status bits, the DSP or host consumes processing clocks.

8.3.1.1 DSP Polling

On the DSP-side, four bits are available for polling. In DSP-to-host transfers (host reads), the DSP core can determine whether the HOTX is empty, partially empty/full, or full. To determine whether HOTX is empty, the DSP core polls the Host Transmit Empty bit in the Host Status Register (HSR[HTFE]):

- If HTFE is clear, the HOTX FIFO is not empty (it is either partially empty or full).
- If HTFE is set, the HOTX FIFO is empty.

To determine whether HOTX is full, the DSP core polls the Host Transmit Not Full bit in the Host Status Register (HSR[HTFNF]):

- If HTFNF is clear, the HOTX FIFO is full, and the core should not write to it.
- If HTFNF is set, the HOTX FIFO is not full (it is either partially full or empty).

The MSC8101 assembly code in **Example 8-5** implements the polling and transfer of 16-bit data from a buffer in memory to the HOTX register:

Example 8-5. Implementing Polling and Data Transfer to HOTX

```
        move.l  #HSR_ADDR,r2            ; r2 = HSR address
        move.l  #BUFF_ADDR,r3           ; r3 = buffer start address
label1  bmtstc.w#4,(r2)                 ; test HSR[HTFE]
        bt      label1                  ; loop if HSR[HTFE] = 0
        move.w  p:(r3),d1               ; d1 = data from buffer in memory
        move.w  d1,p:HOTX_ADDR          ; move d1 to HOTX
```

For host-to-DSP transfers (host writes), the DSP side should determine whether the HORX is full, partially full/empty, or empty. To determine whether HORX is full, the DSP core polls the Host Receive Full bit in the Host Status Register (HSR[HRFF]):

- If HRFF is clear, the HORX FIFO is not full (it is either partially full or empty)
- If HRFF is set, the HORX FIFO is full and the DSP core should read it.

To determine whether HORX is empty, the DSP core polls the Host Receive Not Empty bit in the Host Status Register (HSR[HRFNE]):

- If HRFNE is clear, the HORX FIFO is empty.
- If HRFNE is set, the HOTX FIFO is not empty (it is either partially empty or full).

The MSC8101 assembly code in **Example 8-6** implements the polling and transfer of 16-bit data from the HORX register to a buffer in memory:

Example 8-6. Implementing Polling and Data Transfer From HORX

```
        move.l  #HSR_ADDR,r2          ; r2 = HSR address
        move.l  #BUFF_ADDR,r3         ; r3 = buffer start address
label1  bmtstc.w#1,(r2)               ; test HSR[HRFNE]
        bf      label1                ; loop if HSR[HRFNE] = 0
        move.w  p:HORX_ADDR,d1        ; d1 = HORX data
        move.w  d1,p:(r3)             ; move d1 to buffer in memory
```

8.3.1.2 Host Polling

A polling mechanism similar to that for the DSP is available for host use. When data is transferred to the DSP (host writes), the host polls the Transmit Data Empty bit in the Interface Status Register (ISR[TXDE]). If TXDE is set, the Transmit Data Registers (TX[0–3]) are empty, and the host can write to them. Otherwise, it must wait until the data in these registers is transferred to the DSP-side HORX. In DSP-to-host transfers, (host reads), the host can poll the Receive Data Full bit in the Interface Status register (ISR[RXDF]). When RXDF is set, there is valid data in the Receive Data registers (RX[0–3]).

8.3.1.3 Host Flags

The HDI16 control registers, HCR on the DSP-side and ICR on the host-side, each have four general-purpose flags for communication between the host and the DSP:

- *DSP side*. The HCR Host Flag bits (HCR[HF[4–7]]) can pass application-specific information to the host. The host-side ISR Host Flag bits (ISR[HF[4–7]]) reflect the status of HCR Host Flag bits.
- *Host side*. The ICR Host Flag bits (ICR[HF[0-1]] and ICR[HF[2-3]]) can pass application-specific information to the DSP. The DSP-side HSR Host Flag bits (HSR[HF[0–3]]) reflect the status of the ICR Host Flag bits.

8.3.1.4 Transmit Ready bit

The ISR[TRDY] bit allows the host to determine whether the Transmit Data Registers (TX[0–3]) and the HORX FIFO are empty. When TRDY is set, the TX[0–3] and HORX are empty, so any information written from the host to the TX[0–3] immediately transfers to the DSP side, thus ensuring that the DSP receives this data.

8.3.2 DSP Interrupts

HDI16 interrupts allow the DSP core to perform other processing tasks while waiting for HDI16 resources to become ready. An enabled interrupt automatically occurs when the HDI16 data resources are available for transfer. The interrupt routine can then transfer the data to and from the HDI16 host port. **Table 8-15** lists all the HDI16 sources that can be used to interrupt the SC140 core. Notice that the interrupt sources associated with HORX and HOTX are triggered by the same status bits the SC140 core reads when polling techniques are used. The DSP uses these interrupts to move data to or from the HOTX and HORX data registers. The host command interrupt and the non-maskable interrupt allow the host to force execution of a DSP interrupt routine. (NMI and host commands are addressed in **Section 8.4**, *Issuing Host Commands and Non-Maskable Interrupts,* on page 8-23).

Table 8-15. HDI16 Interrupt Sources

Interrupt Source	HCR Masking Bit
Host Command (HCVR[HCP])	Host Command Interrupt Enable (HCR[HCIE])
HOTX transmit FIFO not full (HSR[HTFNF])	Host Transmit Not Full Interrupt Enable (HCR[HTFIE])
HOTX transmit FIFO empty (HSR[HTFE])	Host Transmit Empty Interrupt Enable (HCR[HTEIE])
HORX receive FIFO full (HSR[HRFF])	Host Receive Full Interrupt Enable (HCR[HRFIE])
HORX receive FIFO not empty (HSR[HRFNE])	Host Receive Not Empty Interrupt Enable (HCR[HREIE])
Host non-maskable interrupt ($\overline{\text{NMI}}$)	None

Figure 8-4 depicts how the interrupt source status bits and the masking bits operate to generate an interrupt. When the appropriate interrupt mask bit in the HCR is set, the interrupt is enabled. An event that causes the corresponding status bit in the HSR to be set therefore generates an interrupt request to the Program Interrupt Controller (PIC).

For each interrupt service routine (ISR), the MSC8101 PIC must be programmed. Each HDI16 interrupt source can cause an ISR to execute at a distinct offset from the vector base address (VBA). The PIC Edge/Level triggered Interrupt Priority Registers (ELIRx) enable you to mask and define the relative priority level of the interrupt, as **Table 8-16** shows. Refer to the chapter on interrupts for a further discussion of the PIC and MSC8101 interrupt service routines.

Table 8-16. PIC Interrupts

ISR Address (VBA offset)	PIC Interrupt Source	PIC Priority Level Bits	PIC Trigger Mode Bits	PIC Interrupt Pending Bits
0x940	$\overline{\text{IRQ5}}$–HDI16: Receive FIFO full	ELIRB[9–11]	ELIRB[8]	IPRA[10]
0x980	$\overline{\text{IRQ6}}$–HDI16: Receive FIFO not empty	ELIRB[5–7]	ELIRB[4]	IPRA[9]
0x9C0	$\overline{\text{IRQ7}}$–HDI16: Transmit FIFO empty	ELIRB[1–3]	ELIRB[0]	IPRA[8]
0xA00	$\overline{\text{IRQ8}}$–HDI16: Transmit FIFO not full	ELIRC[13–15]	ELIRC[12]	IPRA[7]

Table 8-16. PIC Interrupts

ISR Address (VBA offset)	PIC Interrupt Source	PIC Priority Level Bits	PIC Trigger Mode Bits	PIC Interrupt Pending Bits
0xA40	$\overline{\text{IRQ9Q}}$–HDI16: External host command	ELIRC[9–11]	ELIRC[8]	IPRA[6]
0xE00	NMI0–HDI16: External host NMI	-	-	IPRB[7]

To clear the interrupt, the SC140 core interrupt service routine must read or write the appropriate HORX or HOTX register or, for host commands and NMI interrupts, the pending interrupt condition is cleared when the HCVR is read. Furthermore, the interrupt service routine must also clear the interrupt request in the PIC Interrupt Pending Registers (IPRx) if the PIC Trigger Mode is set to Edge Triggered. The exception routine may also need to determine whether the current data is the last data to be transferred, since this is a good place to decide whether to disable the interrupt.

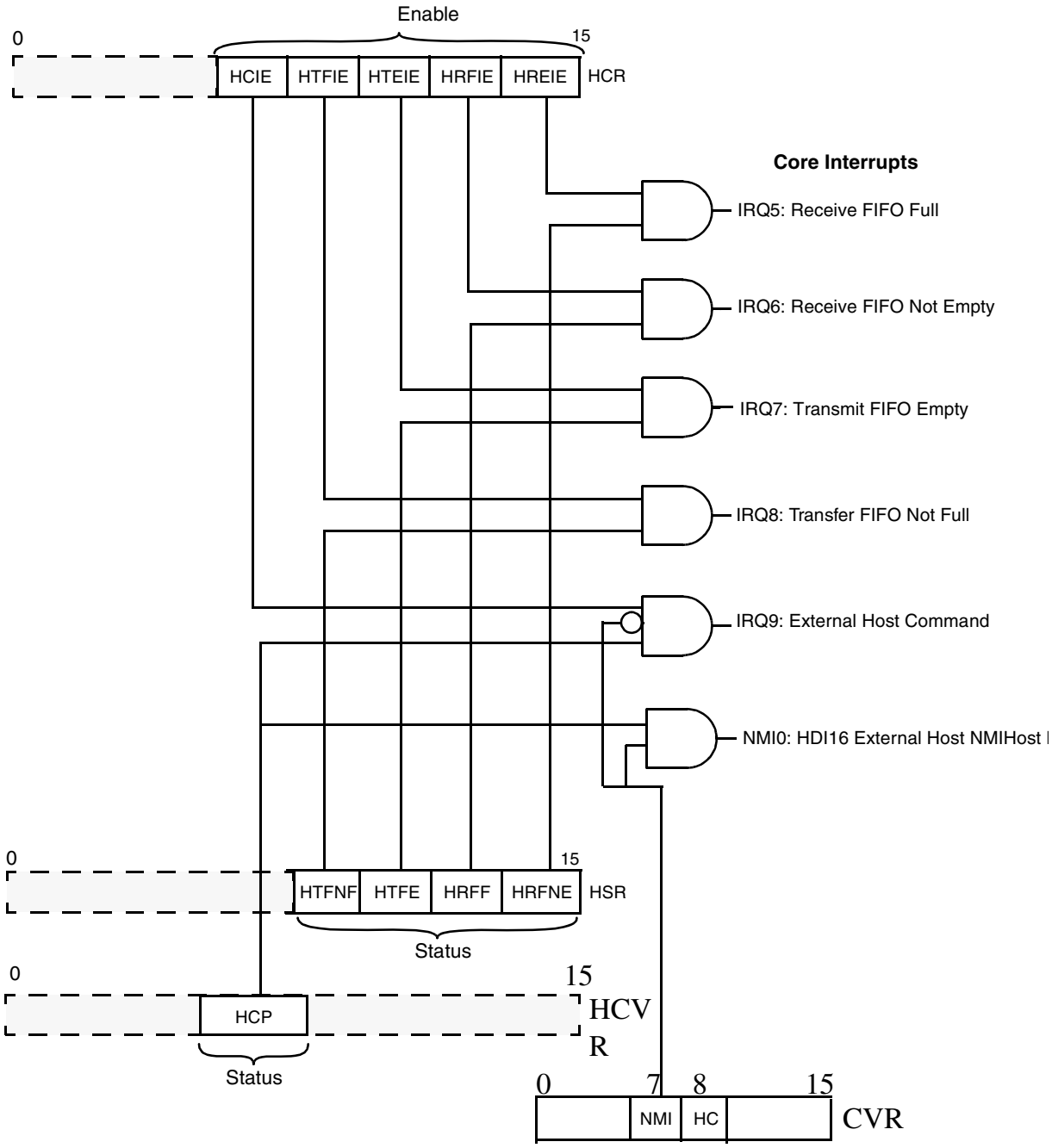


Figure 8-4. HDI16 DSP-Side Interrupt Operation

Example 8-7 shows the set-up code for a simple interrupt that receives 16-bit data from the HDI16 HORX and places it into a buffer in memory. For simplicity, the r3 register used as a pointer into memory is assumed to retain its state throughout.

Example 8-7. Receive Interrupt Set-up Code

```
; setup HI16 registers
move.l    #M_HPCR,r1          ; r1 = HPCR address
move.l    #M_HCR,r0           ; r0 = HCR address
```

```
        move.w   #INIT_HPCR,(r1)       ; init HI16 HPCR
        bmset.w  #M_HEN,(r1)           ; enable HI16
        move.w   #INIT_HCR,(r0)        ; init HI16 HCR

        move.w   #BUFF,r3              ; r3 = pointer to buffer in memory

        ; set-up and enable interrupt
        di
        move.l   #I_ELIRB,r1
        bmclr    #$00e0,sr.h           ; enable all IPLs
        move.w   #INIT_ELIRB,(r1)      ; set hi16 rxne IPL
        bmset.w  #M_HREIE,(r0)         ; enable HI16 RX interrupt
        ei                             ; enable interrupts
```

Example 8-8 shows code for a simple interrupt service routine that receives 16-bit data from the HDI16 HORX into a buffer in memory. For simplicity, the initialization of buffer pointer r3 and the saving of the SC140 core's context are not shown.

Example 8-8. Receive Interrupt Service Routine

```
        org     p:#VBA_INIT+0x980      ; Receive FIFO not empty vector address
        jsr     hi16_rxne              ; jump to ISR
        rte                            ; return from interrupt

        org     p:HDI16_ISR            ; ISR program code
hi16_rxne
        move.w  p:HORX_ADDR,d1         ; d1 = HORX data
        move.w  d1,p:(r3)+             ; move d1 to buffer in memory
        rts                            ; return form ISR
```

8.3.3 Host Requests

The host request mechanism provides a set of signal lines by which the DSP can request data transfers from the host. The request signal lines from the DSP normally connect to the host's interrupt request pins ($\overline{\text{IRQx}}$) and indicate to the host when a HDI16 port requires service. The DSP side can be configured to use either a single request line (HREQ) for both receive and transmit requests or two signal lines, a Host Transmit Request (HTRQ) and a Host Receive Request (HRRQ), one for each direction of transfer.

The host enables host requests using the Interface Control Register (ICR) as follows:

1. Configure the HDI16 for single (HREQ) or double (HRRQ and HTRQ) requests using the ICR Host Double Request bit as indicated in **Table 8-17**. This bit is available only in Normal transfer modes.

Table 8-17. Single or Double Request Configuration

ICR[HDRQ]	Host Request
0	Single
1	Double

2. Enable receive requests by setting the ICR Receive Request Enable bit (ICR[RREQ]) and/or enable transmit requests by setting the ICR Transmit Request Enable bit (ICR[TREQ]).

   With host requests enabled, the host request pins operate as **Figure 8-5** shows.

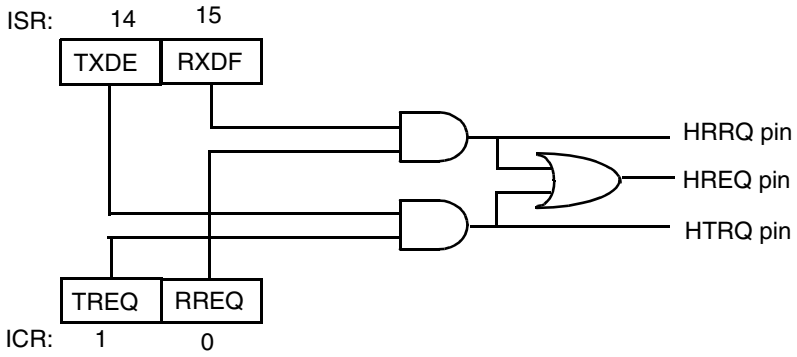


Figure 8-5. HDI16 Host Request Operation

Table 8-18 shows how the HREQ pin operates in single request mode.

Table 8-18. HREQ Pin In Single Request Mode (ICR[HDRQ] = 0)

ICR[TREQ]	ICR[RREQ]	HREQ Pin
0	0	No host requests enabled
0	1	ISR[RXDF] request enabled
1	0	ISR[TXDE] request enabled
1	1	ISR[RXDF] and ISR[TXDE] request enabled

Table 8-19 shows how the transmit request (HTRQ) and receive request (HRRQ) lines operate with double host requests enabled.

Table 8-19. HTRQ and HRRQ Pins In Double Request Mode (ICR[HDRQ] = 1)

ICR[TREQ]	ICR[RREQ]	HTRQ Pin	HRRQ Pin
0	0	No interrupts	No interrupts
0	1	No interrupts	ISR[RXDF] request enabled
1	0	ISR[TXDE] Request enabled	No interrupts
1	1	ISR[TXDE] Request enabled	ISR[RXDF] request enabled

The request signal lines from the DSP normally connect to the host's interrupt request pins ($\overline{\text{IRQx}}$), which generate an interrupt in the host. Generally, the host interrupt service routine must test the status bits in the HDI16 host-side ISR to determine the interrupt source. To clear the interrupt request, the host must read or write the appropriate HDI16 host-side data registers, TX[0–3] and RX[0–3].

8.3.4 Direct Memory Access (DMA)

Two distinct DMA mechanisms are associated with the HDI16: external DMA and internal DMA. Externally, the host or an external DMA controller connected to the HDI16 host bus can transfer data between itself and the HDI16 port. External DMA operation is described in **Section 8.2.2**, *Host DMA Mode,* on page 8-8. The DMA controller that is internal to the DSP is the subject of this section.

The MSC8101 DMA controller performs data transfers between memory (either external on the system bus or internal) and the HDI16 HORX and HOTX data registers with no SC140 core intervention. The DMA controller frees the core to use its processing power on functions other than polling or interrupt routines associated with the HDI16. DMA may well be the most efficient and least costly method to use for data transfers, but it requires available DMA channels. If the HDI16 DMA controller transfers data to and from the internal SRAM, a single DMA channel can be used in flyby mode.[1] If the source or destination is on the system bus, two DMA channels are required, one to transfer data between the memory and the DMA FIFO and the other to transfer the data between the HDI16 data register and the DMA FIFO.

The details of the MSC8101 internal DMA are beyond the scope of this chapter. The overall steps involved in programming a DMA channel for access of the HDI16 in flyby mode are as follows:

1. Initialize a DMA Channel Configuration Register (DCHCRx) for the selected DMA. In flyby mode, the DCHCRx describes the HDI16 peripheral; thus, the Request Number bits (DCHCRx[RQNUM]) identify the HDI16 as indicated in **Table 8-20** (this field is valid only when an internal requestor is defined by setting DCHCRx[INT]). The flyby

1. A flyby transfer is also known as a "single access data transaction." The data path is between a peripheral and memory with the same port size, located on the same bus. On the MSC8101, flyby transactions can occur only between external peripherals and external memories located on the bus, or between internal peripherals and internal SRAM located on the local bus. Flyby operations do not require access to the DMA FIFO.

transaction bit is asserted to enable flyby mode and the DMA Active Channel bit is negated until later.

Table 8-20. DMA Request Sources

Requesting Device	DCHCRx[RQNUM]
HDI16 read request	00000
HDI16 write request	00001

2. Configure the DMA Channel Parameter RAM (DCPRAM) for the buffer descriptor to which the Buffer Pointer bits in the DCHCRx point.

   In flyby mode the DCPRAM parameters describe the memory, so the BD_ADDR parameter is initialized with the address in SRAM to be used as a source/destination of the DMA. The BD_SIZE and BD_BSIZE parameters are initialized with the size of the DMA transfer. The BD_ATTR parameter is initialized with the characteristics of the access of the buffer in SRAM (simple, cyclic, incremental, chained, or a combination of these).

3. Activate the DMA by setting the Active DMA Channel bit, DCHCRx[ACTV].

   You can also specify whether to transfer the data in bursts or single accesses via the DMA Transmit Burst Enable (DBTE) and the DMA Receive Burst Enable (DBRE) in the Host Control Register (HCR). These bits define what condition associated with the HOTX and HORX registers requests a DMA access, thus indicating to the DMA controller whether to use burst for the data access. **Table 8-21** shows the behavior of the DBTE and DBRE bits.

Table 8-21. DMA Single and Burst Mode Access

ICR[DBTE]	ICR[DBRE]	DMA Request Condition	DMA Transfer Type
0	n/a	HSR[HTFNF]	Single access to HOTX
1	n/a	HSR[HTFE]	Burst access to HOTX
n/a	0	HSR[HRFNE]	Single access from HORX
n/a	1	HSR[HRFF]	Burst access from HORX

Example 8-9 shows the code necessary to set up a dual DMA to receive BUFF_SIZE 16-bit data elements from the HDI16 and place them into a buffer in the internal SRAM located at BUFF_START. An interrupt is generated when the DMA is finished.

Example 8-9. Receive Interrupt Service Routine

```
INIT_ATTR0      EQU     $08000010
INIT_ATTR1      EQU      $80000000
INIT_DCHCR0     EQU      $80000005
```

```
INIT_DCHCR1     EQU     $80010045
INIT_DIMR       EQU     $40000000
INIT_ELIRE       EQU     $0c00

; setup source DMA DCPRAM
move.l    #BD_ADDR0,r1          ; set source address base
moveu.l   #I_GPCM+HORX,d0
move.l    d0,(r1)
move.l    #BD_SIZE0,r1          ; set size of source transfer
moveu.l   #BUFF_SIZE,d0
move.l    d0,(r1)
move.l    #BD_BSIZ0,r1          ; set source base size
moveu.l   #$00000000,d0
move.l    d0,(r1)
move.l    #BD_ATTR0,r1          ; set source channel attributes
moveu.l   #INIT_ATTR0,d0
move.l    d0,(r1)
; setup dstination DMA DCPRAM
move.l    #BD_ADDR1,r0          ; set destination address base
moveu.l   #BUFF_START,d0
move.l    d0,(r0)
move.l    #BD_SIZE1,r0          ; set size of destination transfer
moveu.l   #PATT_SIZE,d0
move.l    d0,(r0)
move.l    #BD_BSIZ1,r0          ; set destination base size
moveu.l   #$00000000,d0
move.l    d0,(r0)
move.l    #BD_ATTR1,r0          ; set destination channel attributes
moveu.l   #INIT_ATTR1,d0
move.l    d0,(r0)
; setup DMA internal mask register
move.l    #M_DIMR,r1            ; set DIMR
moveu.l   #INIT_DIMR,d0
move.l    d0,(r1)
; setup DMA channel configuration and activate
move.l    #M_DCHCR0,r1          ; set DCHCR0
moveu.l   #INIT_DCHCR0,d0
move.l    d0,(r1)
move.l    #M_DCHCR1,r0          ; set DCHCR1
moveu.l   #INIT_DCHCR1,d0
move.l    d0,(r0)

; setup PIC registers
move.l    #M_ELIRE,r3
bmclr           #$00a0,sr.h     ; mask priorities < 2
move.w    #INIT_ELIRE,(r3)      ; init PIC ELIRE reg
```

The internal MSC8101 DMA controller does not access the host bus, so the host must determine when data is available in the host-side data registers using an appropriate polling mechanism.

8.4 Issuing Host Commands and Non-Maskable Interrupts

The innovative host command feature of the HDI16 host interface allows the host to issue any of 128 pre-programmed functions for the DSP to execute. For example, the host can issue a host command that sets up and enables a DMA data transfer. This flexibility is independent of the data transfer mechanisms in the HDI16; it enables the host processor to read or write DSP registers or memory locations, perform control status or debugging operations, and start DMA transfers, among other functions.

To enable host command interrupts, set the Host Command Interrupt Enable bit (HCR[HCIE]) on the DSP-side Host Control Register (HCR). The MSC8101 PIC must be programmed accordingly (refer to **Section 8.3.2**, *DSP Interrupts,* on page 8-15). The PIC Edge/Level triggered Interrupt Priority Registers (ELIRx) allow you to mask and define the relative priority level of the host command interrupts, as shown in **Table 8-16**. The host can then issue a host command by writing the CVR[HV] bits with the pointer to the interrupt service routine to execute and setting the Host Command (CVR[HC]) to request the interrupt. The host can write the HC and HV bits simultaneously.

When the MSC8101 Programmable Interrupt Controller (PIC) on the extended core recognizes the host command interrupt request, the External Host Command interrupt service routine at VBA offset 0xA40 is executed. This interrupt service routine must then read the Host Vector bits of the DSP-side Host Command Vector Register (HCVR[HV]), which reflect the CVR[HV] bits, to determine which command to execute. The Host Command Pending bit, HCVR[HCP], reflects the status of the CVR[HC] bit. HCVR[HCP] and CVR[HC] are cleared when the interrupt service routine reads the HCVR. The host must not clear CVR[HC] (which clears HCVR[HCP]) until the interrupt service routine clears them. However, the host can poll this bit to determine when the PIC accepts this command. The ISR must also clear the interrupt request in the PIC Interrupt Pending Registers (IPRx).

The operation is very similar for non-maskable interrupts (NMIs), except that the ISR cannot be masked in the HCR. Typically, the host writes the CVR[HV] bits with the pointer to the pre-programmed function and also sets the HC and NMI bits. This causes the PIC on the MSC8101 extended core to execute the External Host NMI ISR at VBA offset 0xE00. The pending interrupt condition is cleared when the interrupt service routine reads the HCVR. As with host command interrupt service routines, the host NMI ISR must also clear the interrupt request in the PIC Interrupt Pending Registers (IPRx).

8.5 Related Reading

Table 2-22.

MSC8101 User's Guide (This manual)

Chapter 7, Interrupts and Interrupt Priorities

MSC8101 Reference Manual

Chapter 5, *Reset*

Chapter 14, *Host Interface (HDI16)*

Chapter 15, *Direct Memory Access (DMA)*

Chapter 16, *Interrupt Scheme*
Especially the section on the Programmable Interrupt Controller (PIC)

Enhanced Filter Coprocessor (EFCOP) 9

The MSC8101 EFCOP module is a general-purpose, fully programmable filter with 32-bit resolution. It has optimized modes of operation to perform real and complex finite impulse response (FIR) filtering, infinite impulse response (IIR) filtering, adaptive FIR filtering, and multichannel FIR filtering. EFCOP filter operations complete concurrently with SC140 operations, with minimal CPU intervention. For optimal performance, the EFCOP has one dedicated filter multiplier accumulator (FMAC) unit. As a result, the SC140/EFCOP combination offers multiple multiply-accumulate (MAC) filtering capabilities.

Its dedicated modes make the EFCOP a very flexible filtering coprocessor with operations optimized for cellular base station applications. In a transceiver base station, the EFCOP performs complex matched filtering to maximize the signal-to-noise ratio (SNR) in an equalizer. The coefficients of the matched filter can be determined by a cross-correlation filtering process between a received training sequence and a known reference sequence. In a transcoder base station or a mobile switching center, the EFCOP can perform all types of FIR and IIR filtering within a vocoder, as well as LMS-type echo cancellation.

This chapter discusses how to program the EFCOP to operate in different modes and how to use the different methods for transferring data into and out of the EFCOP. Code examples demonstrate how the EFCOP can be programmed to complete a variety of tasks. The focus is on programming the EFCOP internally from the SC140 core. The EFCOP can also be programmed from an external device on the system bus, and all the programming issues discussed here still apply.

9.1 Programming the Control Registers

The EFCOP is programmed by writing the desired settings to the memory-mapped control registers. **Table 9-1** summarizes the EFCOP control registers.:

Table 9-1. EFCOP Control Registers

Register Name	Description
Filter Count Register (FCNT)	A 16-bit read/write register that specifies the number of filter taps. The count stored in the FCNT register is used by the EFCOP address generation logic to generate correct addressing to the filter data memory (FDM) and filter coefficient memory (FCM).
EFCOP Control Register (FCTL)	A 16-bit read/write register used by the SC140 to program the EFCOP.

Table 9-1. EFCOP Control Registers

Register Name	Description
EFCOP ALU Control Register (FACR)	A 16-bit read/write register used by the SC140 to program the EFCOP data ALU operating modes.
EFCOP Data Base Address Register (FDBA)	A 16-bit read/write register used by the SC140 to indicate the EFCOP data buffer base start address pointer in FDM RAM.
EFCOP Coefficient Base Address Register (FCBA)	A 16-bit read/write register by which the SC140 indicates the EFCOP coefficient buffer base start address pointer in FCM RAM.
EFCOP Decimation/Channel Count Register (FDCH)	A 16-bit read/write register that sets the number of channels in multichannel mode and the filter decimation ratio. The EFCOP address generation logic uses this information to supply the correct addressing to the FDM and FCM.
EFCOP Status Register (FSTR)	A 16-bit read-only register used by the SC140 to examine the status of the EFCOP module.
Filter Data Input Register (FDIR)	An 8-word deep 32-bit wide FIFO used for core-to-EFCOP and DMA data transfers. Data from the FDIR is transferred to the FDM for filter processing.
Filter Data Output Register (FDOR)	An 8-word deep 32-bit wide FIFO used for EFCOP-to-core and DMA data transfers. Data is transferred to FDOR after processing of all filter taps is completed for a specific set of input samples.
Filter K-Constant Input Register (FKIR)	A 32-bit read/write register for core-to-EFCOP constant transfers.

The EFCOP operates in many different modes based on the settings of these control registers. However, the EFCOP performs only two basic modes of processing, FIR filter type and IIR filter type processing. Various operating modes are available for each filter type. The following sections discuss the two basic operating modes.

9.2 Specifying the Operating Modes for the FIR Filter Type

This section discusses the various operating modes that are available for the FIR filter type. The FIR filter type is selected by clearing the FCTL[FLT] bit, and it performs the processing shown in **Figure 9-1** using the following equation:

$$w(n) = \sum_{i=0}^{N} B_i x(n-i)$$

For each sample to be filtered, the EFCOP completes the following steps:

1. Take an input, *x(n)*, from the FDIR.
2. Save the input while shifting the previous inputs down in the FDM. The shifting down of the previous inputs is accomplished by incrementing the value in the FDBA register by one.
3. Multiply each input in the FDM by the corresponding coefficient, B_i, stored in the FCM.

4. Accumulate the multiplication results.
5. Place accumulation result, *w(n)*, into the FDOR.

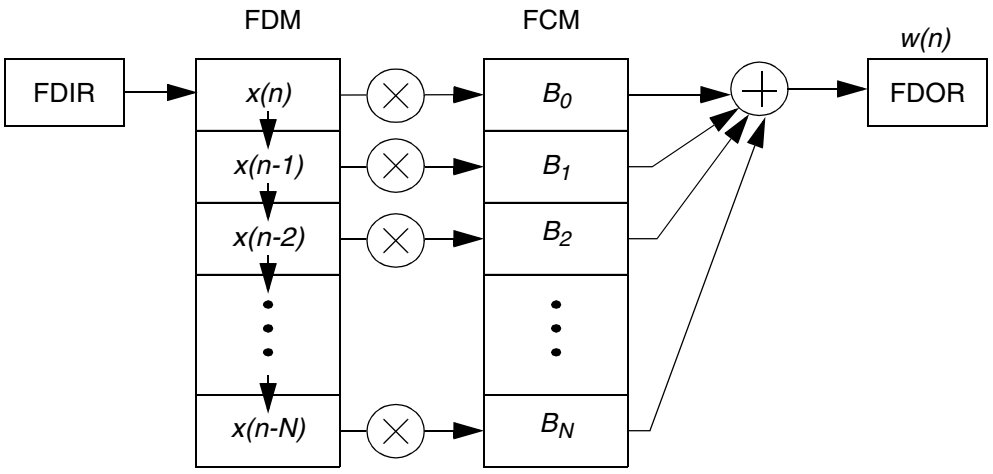


Figure 9-1. FIR Filter Block Diagram

Four operating modes are available for the FIR filter type: real, complex, alternating complex, and magnitude mode.

9.2.1 Real Mode

Real mode performs FIR type filtering with real data and is selected by clearing both FCTL[FOM] bits. For each sample (the real input) written to the FDIR, one sample (the real output) is read from the FDOR. In Real mode, the number written to the FCNT register should be one minus the number of filter coefficients.

Two other options are available with the real FIR filter type: Adaptive and Multichannel modes. These modes can be used singly or together.

9.2.1.1 Adaptive Mode

The Adaptive mode provides a way to update the coefficients based on filter input, *x(n)*, using the following equation:

$$h_{n+1}(i) = h_n(i) + K_e(n)x(n-i)$$

where $h_n(i)$ is the *i*th coefficient at time *n*. The coefficients are updated when the FCTL[FUPD] bit is set. When this bit is set, the EFCOP checks to see if a value has been written to the FKIR. If no value is written, the EFCOP halts processing until a value is written to the FKIR. When a value is written to the FKIR, the EFCOP updates all the coefficients based on the preceding equation using the value in the FKIR for $K_e(n)$. The EFCOP automatically clears the FCTL[FUPD] bit when the coefficient update completes.

If the coefficients are to be updated after every input sample, Adaptive mode is enabled by setting the FCTL[FADP] bit. In Adaptive mode, the EFCOP automatically sets the FCTL[FUPD] bit after each input sample is processed. This allows for continuous processing using interrupts that includes a filter session and a coefficient update session with minimal core intervention.

Additionally, the EFCOP can generate an interrupt request to the SC140 core when the coefficient update is complete. This interrupt is controlled by the FCTL[FUDIE] bit. If this bit is clear, the interrupt is disabled. If this bit is set, the interrupt is enabled. When the interrupt is enabled, it is triggered by the FSTR[FUDN] bit. The EFCOP sets FSTR[FUDN] when the coefficient update session completes. Thus, when both FSTR[FUDN] and FCTL[FUDIE] are set, the EFCOP requests the coefficient update complete interrupt from the SC140 core.

9.2.1.2 Multichannel Mode

In Multichannel mode, several channels of data are processed concurrently. This mode is selected by setting the FCTL[FMLC] bit. The number of channels to process is one plus the number in the FDCH[FCHL] bits. The number of channels can be programmed to be from 1 to 64. For each time period, the EFCOP expects to receive the samples for each channel sequentially. This process repeats for consecutive time periods.

Filtering is performed with the same filter or different filters for each channel using the FACR[FSCO] bit. If this bit is set, the same set of coefficients is used for all channels. If FACR[FSCO] is clear, the coefficients for each channel are stored sequentially in memory with the beginning address of each coefficient buffer at the next 2^k address (where $2^{k-1} \leq$filter length $\leq 2^k$). If the filter length is less than 2^k there is a space between the sequential buffers of 2^k minus the filter length.

9.2.2 Complex Mode

Complex mode performs FIR type filtering with complex data based on the following equations, in which *Re* is the real part and *Im* is the imaginary part:

$$Re(F(n)) = \sum_{i=0}^{N-1} Re(H(i)) \cdot Re(D(n-i)) - Im(H(i)) \cdot Im(D(n-i))$$

$$Im(F(n)) = \sum_{i=0}^{N-1} Re(H(i)) \cdot Im(D(n-i)) + Im(H(i)) \cdot Re(D(n-i))$$

where *H(n)* is the coefficients, *D(n)* is the input data, and *F(n)* is the output data at time *n*. For every two samples written to the FDIR (the real part followed by the imaginary part of the input), two samples (the real part followed by the imaginary part of the output) are read from the FDOR.

Complex mode is selected by writing 01 to the FCTL[FOM] bits. When Complex mode is used, the number written to the FCNT register should be twice the number of filter coefficients minus

one, (2 × filter length) – 1. Also, the coefficients should be stored in the FCM with the real part of the coefficient in the memory location preceding the location holding the imaginary part.

9.2.3 Alternating Complex Mode

Alternating Complex mode performs FIR type filtering with complex data providing alternating real and complex results based on the following equations:

$$Re(F(n|_{even})) = \sum_{i=0}^{N-1} Re(H(i)) \cdot Re(D(n-i)) - Im(H(i)) \cdot Im(D(n-i))$$

$$Im(F(n|_{odd})) = \sum_{i=0}^{N-1} Re(H(i)) \cdot Im(D(n-i)) + Im(H(i)) \cdot Re(D(n-i))$$

where *H(n)* is the coefficients, *D(n)* is the input data, and *F(n)* is the output data at time *n*. For every two samples (the real part followed by the imaginary part of the input) written to the FDIR, one sample (alternating between the real part and the imaginary part of the output) is read from the FDOR.

Alternating Complex mode is selected by writing 10 to FCTL[FOM] bits. When Alternating Complex mode is used, the number written to the FCNT register should be twice the number of filter coefficients minus one, (2*filter length) –1. Also, the coefficients should be stored in the FCM with the real part of the coefficient in the memory location preceding the location holding the imaginary part.

9.2.4 Magnitude Mode

Magnitude mode calculates the magnitude of an input signal using the following equation:

$$F(n) = \sum_{i=0}^{N-1} D(n-i)^2$$

where *D(n)* is the input data and *F(n)* is the output data at time *n*. For each sample (the real input) written to the FDIR, one sample (the real magnitude of the input signal) is read from the FDOR. Magnitude mode is selected by setting both FCTL[FOM] bits. In Magnitude mode, the number written to the FCNT register should be the number of data samples to compute the magnitude of minus one, and the value in the FCBA register is ignored.

9.2.5 Data and Coefficient Initialization

Before the first sample can be processed, the filter must be initialized, meaning that the input samples for times before $n = 0$ (assuming that time starts at 0) must be loaded into the FDM. The number of samples needed to initialize the filter is the number of filter coefficients.

The Data Initialization mode is selected via the FCTL[FPRC] bit:

- If FCTL[FPRC] is set, initialization is disabled and the EFCOP assumes that the SC140 core wrote the initial input values to the FDM before the EFCOP was enabled. Thus, the first value written to FDIR is the first sample to be filtered.
- If FCTL[FPRC] is clear, initialization mode is enabled and the EFCOP initializes the FDM by receiving the number of coefficient data samples through the FDIR. These samples are loaded into the FDM buffer and after the last value is loaded the EFCOP begins processing the first result.

The EFCOP also allows the coefficient buffer to be initialized before processing begins. The Coefficient Initialization mode is selected via FCTL[FCIM]:

- If FCTL[FCIM] is clear, Coefficient Initialization mode is disabled and processing completes as described earlier, depending on how the FCTL[FPRC] bit is set.
- If FCTL[FCIM] is set, the coefficients are initialized by a coefficient update session with the original coefficients equal to zero, as in the following equation:

$$h(i) = Kx(i)$$

  The data buffer, $x(i)$, should be initialized first, either by the SC140 core or the EFCOP with data initialization. Then, K should be written to FKIR, and the EFCOP initializes the coefficient buffer accordingly.

  If data and coefficient initialization are both enabled, data initialization completes first but the EFCOP does not begin processing the first result. After FKIR is written, the EFCOP initializes the coefficients and then another input sample must be written to FDIR to begin processing.

Coefficient initialization is usually used with Adaptive mode or can be used to clear the coefficients by writing zero to FKIR.

9.2.6 Decimation

Decimation is another option that can be used with any of the four FIR filter type modes. However, decimation cannot be used in conjunction with the Adaptive or Multichannel modes. Decimation, also known as “downsampling,” decreases the sampling rate. The decimation ratio defines the number of input samples per output sample. The decimation ratio is one plus the number in the FDCH[FDCM] bits. The decimation ratio can be programmed from 1 to 16.

For Real and Magnitude modes, the decimation ratio number of the sample must be written to the FDIR before an output sample can be read from the FDOR. For Complex mode, two times the decimation ratio number of samples must be written to the FDIR (one for the real part and one for the imaginary part of the input) before two output samples (one for the real part and one for the imaginary part of the output) can be read from the FDOR. For Alternating Complex mode, two times the decimation ratio number of samples must be written to the FDIR (one for the real part and one for the imaginary part of the input) before one output sample (alternating between the real part and the imaginary part of the output) can be read from the FDOR.

9.3 Specifying the Operating Modes for the IIR Filter Type

This section discusses the various operating modes that are available for the IIR filter type. To process a complete IIR filter, a FIR filter type session followed by an IIR filter type session is needed. The IIR filter type is selected by setting the FCTL[FLT] bit, and it performs the processing shown in **Figure 9-2** using the following equation:.

$$y(n) = w(n) + \sum_{j=1}^{M} A_j y(n-j)$$

For each sample input to the FDIR, the EFCOP completes the following steps:

1. Multiply each previous output value in the FDM by the corresponding coefficient, *A*, stored in the FCM.
2. Accumulate the multiplication results.
3. Add the input, *w(n)*, from the FDIR.
4. Place the accumulation result, *y(n)*, in the FDOR.
5. Save the output while shifting the previous outputs down in the FDM. The shifting down of the previous outputs is accomplished by incrementing the value in the FDBA register by one.

Only the Real and the Multichannel Operation modes are available for the IIR filter type. Thus, the FCTL[FOM] bits are ignored when the IIR filter type is used. The Real Operation mode performs IIR type filtering with real data. For each sample (the real input) written to the FDIR, one sample (the real output) is read from the FDOR. In Real mode, the number written to the FCNT register should be the number of filter coefficients minus one.

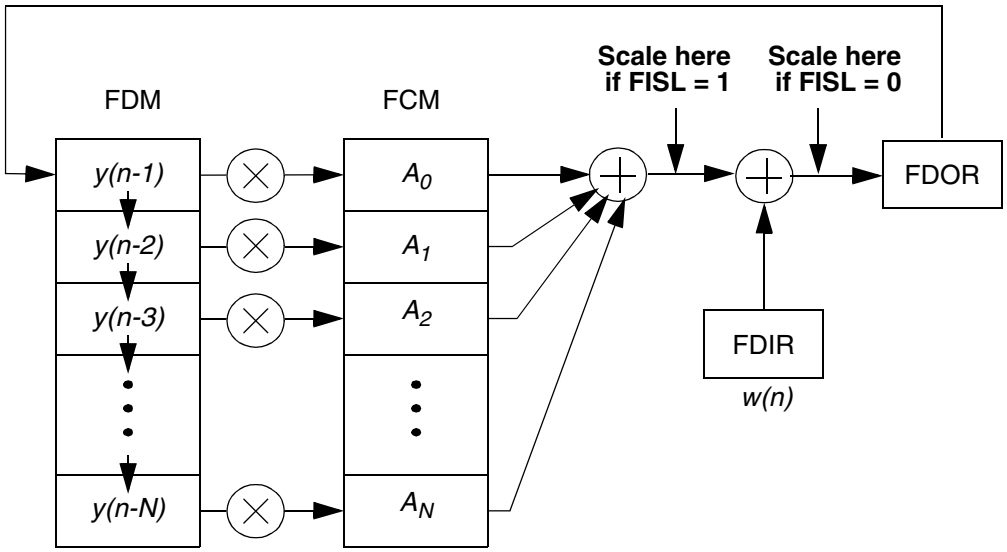


Figure 9-2. IIR Filter Block Diagram

Multichannel mode for the IIR filter type works exactly the same way as for FIR filter type as explained in **Section 9.2.1.2**, *Multichannel Mode,* on page 9-4. Decimation and Adaptive modes are not available with the IIR filter type.

Initialization is always disabled with the IIR filter type, and the FCTL[FPRC] bit is ignored. Thus, the SC140 core must write the initial input values to the FDM before the EFCOP is enabled. The first value written to the FDIR is always the first sample to be filtered.

9.4 Specifying the ALU Modes

Two modes that affect the arithmetic operation of the EFCOP are rounding and input scaling. These ALU modes are independent of the filter type.

9.4.1 Rounding

Rounding mode is selected via the FACR[FRM] bits. These bits select the type of rounding performed by the EFCOP data ALU (DALU) during arithmetic operations. The EFCOP DALU performs the following types of rounding:

- Convergent rounding (FACR[FRM] = 00)
- Twos complement rounding (FACR[FRM] = 01)
- No rounding, that is, truncation (FACR[FRM] = 10)

9.4.2 Input Scaling

The Input Scaling mode affects only IIR filtering and the coefficient update session of adaptive FIR filtering. The FACR[FISL] and FACR[FSCL] bits, determine how the outputs are scaled. The result can be scaled up by the following values:

- One, that is, no scaling (FACR[FSCL] = 00)
- Eight (FACR[FSCL] = 01)
- Sixteen (FACR[FSCL] = 10)

For IIR type filtering, FACR[FISL] determines whether the IIR input is scaled. When FACR[FISL] is set, the IIR feedback terms are scaled, but the IIR input, $w(n)$ is not scaled. This case is represented by the following equation:

$$y(n) = w(n) + S\sum_{j=1}^{M} A_j y(n-j)$$

where S is the scaling factor. When FACR[FISL] is clear, the EFCOP ALU scales both the IIR feedback terms and the IIR input. This case is represented by the following equation:

$$y(n) = S\left(w(n) + \sum_{j=1}^{M} A_j y(n-j)\right)$$

Figure 9-2 also shows where the scaling occurs, depending on the value of FISL.

For coefficient update sessions, FACR[FISL] determines whether the original coefficients are scaled. When FACR[FISL] is set, the EFCOP ALU scales only the input/constant term and not the original coefficients. This is represented by the following equation:

$$h_{n+1}(i) = h_n(i) + SK_e(n)x(n-i)$$

When FACR[FISL] is clear, both the input/constant term and the original coefficients are scaled. This is represented by the following equation:.

$$h_{n+1}(i) = S(h_n(i) + K_e(n)x(n-i))$$

9.5 Transferring Data In and Out of the EFCOP

When the EFCOP is programmed and enabled, it waits until input data is written to the Filter Data Input Register (FDIR). The FDIR is an 8-element deep FIFO, so up to eight 32-bit wide data samples can be written into FDIR at the same address. When the EFCOP finishes processing the input data from the FDIR, it sends the results to the FDOR. The FDOR is an 8-element deep

read-only FIFO, so up to eight 32-bit wide data samples can be read from FDOR at the same address.

There are three methods for transferring data to or from the EFCOP data registers:

- *Polling*. The easiest method, but it demands a large amount of the core processing power. The SC140 core cannot be involved in other processing activities while it is polling the input and output buffer bits.
- *Interrupts.* This method requires more code, but the core can process other routines while the EFCOP is computing.
- *DMA*. This method requires even less core intervention and the set-up code is minimal, but the DMA channels must be available.

The following sections describe each transfer method.

9.5.1 Polling

The EFCOP Status Register (FSTR) contains bits that notify when data is ready for transfer to or from the EFCOP. These bits determine when to interact with the EFCOP. For proper operation, the SC140 core must write to the FDIR only when it is empty or not full and read from the FDOR only when it is full or not empty.

FSTR[FIBNF] and FSTR[FDIBE] determine when to write to the FDIR:

- FSTR[FIBNF] is set when the FDIR is not full (that is, at least one of the locations is empty). Thus, when FSTR[FIBNF] is set, the SC140 core can write only one sample of data to the FDIR.
- FSTR[FDIBE] is set when the FDIR is empty (that is, all eight of the locations are empty). Thus, when FSTR[FDIBE] is set, the core can write up to eight samples of data to FDIR. This bit is set immediately after the EFCOP is enabled by setting FCTL[FEN].

FSTR[FOBNE] and FSTR[FDOBF] determine when to read from the FDOR:

- FSTR[FOBNE] is set when the FDOR is not empty (that is, at least one of the locations is full). Thus, when this bit is set, the SC140 core can read only one sample of data from the FDOR.
- FSTR[FDOBF] is set when the FDOR is full (that is, all eight of the locations are full). Thus, when this bit is set, the SC140 core can read up to eight samples of data from the FDOR.

For an example of EFCOP programming with polling, see **Section 9.6.1**, *Complex FIR Filter with Polling,* on page 9-13.

9.5.2 Interrupts

The EFCOP provides five interrupts. **Table 9-2** describes these interrupts, including how they are enabled and triggered and the location of the vector address.

Table 9-2. EFCOP Interrupts

Signal	Description	Enabled by Setting	Triggered When	Vector Address (Offset from VBA)
IRQ0	Data Input FIFO Not Full	FCTL[FINFIE]	FSTR[FIBNF] and FCTL[FINFIE] are set simultaneously	0x800
IRQ1	Data Input FIFO Empty	FCTL[FIEIE]	FSTR[FDIBE] and FCTL[FIEIE] are set simultaneously	0x840
IRQ2	Data Output FIFO Full	FCTL[FOFIE]	FSTR[FDOBF] and FCTL[FOFIE] are set simultaneously	0x880
IRQ3	Data Output FIFO Not Empty	FCTL[FONEIE]	FSTR[FOBNE] and FCTL[FONEIE] are set simultaneously	0x8C0
IRQ4	Coefficient Update Done	FCTL[FUDIE]	FSTR[FUDN] and FCTL[FUDIE] are set simultaneously	0x900

In general, configuring interrupts requires two steps:

1. Set up the interrupt routine by placing the code to be run during the interrupt at the appropriate interrupt vector address.

   The location of the vector address is shown in **Table 9-2** and depends on the setting of the Vector Base Address (VBA) Register. The memory allocation for each interrupt is 64 bytes. To extend the interrupt code size further, service routines can be used.

2. Enable the interrupts by setting bits in various control registers:

   a. To enable the desired interrupts, set the appropriate bits in the FCTL.

   b. To determine the interrupt priority level of the core, set the interrupt mask bits (I0–2) bits of the Status Register (SR).

   c. To determine the priority level for each enabled interrupt, set the PIC Edge/Level-Triggered Interrupt Priority Registers A and B (ELIRA and ELIRB) Interrupt Priority Level (PILxx) bits.

   d. Clear the Interrupt Trigger Mode (PEDxx) bits of the ELIRx registers because all peripheral interrupts are level triggered.

   e. Enable the interrupts by issuing an **ei** (enable interrupts) instruction.

For an example of EFCOP programming with interrupts, see **Section 9.6.2**, *Adaptive Filter With Interrupts,* on page 9-16.

9.5.3 DMA

The Direct Memory Access (DMA) controller is an internal device that permits data transfers to and from the EFCOP without intervention of the SC140 core. The DMA can move data to the EFCOP input register and from the EFCOP output register. The DMA allows dual access and flyby transactions to the EFCOP. Flyby transactions occur directly between the EFCOP and internal SRAM, and do not require access to the DMA FIFO.

The DMA request source is controlled by the requestor number bits (RQNUM, bits 19-23) in the DMA Channel Configuration Register (DCHCRx). If these bits are equal to 00010, the DMA is triggered by an EFCOP read request (the FDOR needs to be read because it is full or not empty). If these bits are equal to 00011, the DMA is triggered by an EFCOP write request (when the FDIR needs to be written because it is empty or not full).

On the EFCOP side, DMA transfers are controlled by the FCTL[FDIM] and FCTL[FDOM] bits.

- FCTL[FDIM] controls the data input mode:
  - When FCTL[FDIM] is clear, the EFCOP issues a transfer request from the DMA when the input buffer is not full (when FSTR[FIBNF] is set). Use this mode when the DMA is programmed to transfer single 32-bit samples to the FDIR (DMA BD_ATTR field TSZ bits equal to 011). Burst transfers or single transfers larger than 32-bits cause an error when the FCTL[FDIM] is clear.
  - When FCTL[FDIM] is set, the EFCOP issues a transfer request from the DMA when the input buffer is empty (when FSTR[FDIBE] is set). Use this mode when the DMA is programmed to burst mode (TSZ equal to 100).
- FCTL[FDOM] controls the data output mode:
  - When FCTL[FDOM] is clear, the EFCOP issues a transfer request from the DMA when the output buffer is not empty (when FSTR[FOBNE] is set). Use this mode when the DMA is programmed to transfer single 32-bit samples from the FDOR (DMA BD_ATTR field TSZ bits equal to 011). Burst transfers or single transfers larger than 32-bits cause an error when FCTL[FDOM] is clear.
  - When FCTL[FDOM] is set, the EFCOP issues transfer request from the DMA when the output buffer is full (when FSTR[FOBNE] is set). Use this mode when the DMA is programmed to burst transfer mode (TSZ equal to 100).

Note: The EFCOP does not support the DMA 64-bit transfer size.

Figure 9-3 shows EFCOP/Internal memory transfers using DMA configured for 32-bit flyby mode. shows the use of DMA flyby burst mode with the EFCOP.

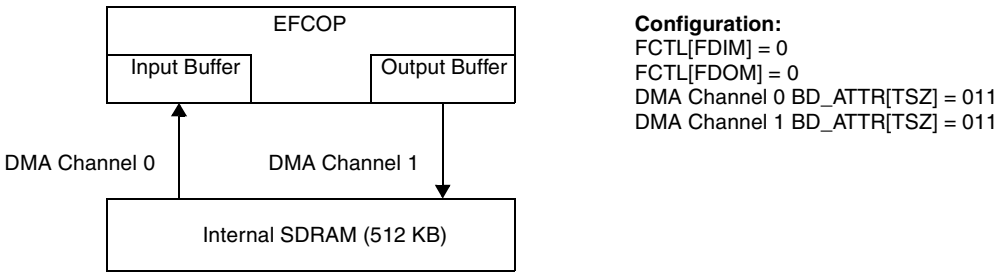


Figure 9-3. Single 32-Bit Transfers—DMA Configured for 32-Bit Flyby Mode

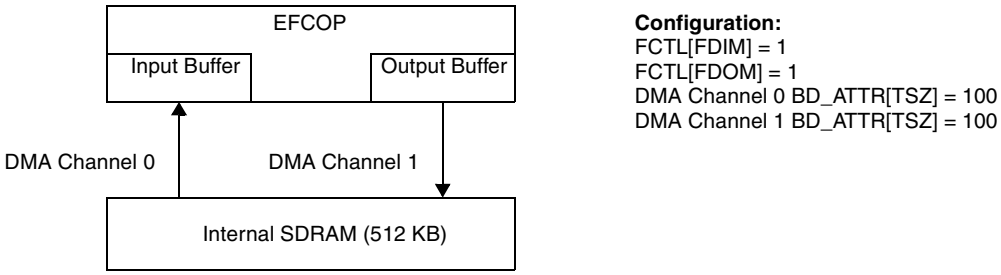


Figure 9-4. Burst Transfers—DMA Configured for Burst Mode

For an example of EFCOP programming with DMA, see **Section 9.6.3**, *Real IIR Filter with DMA,* on page 9-19.

9.6 Programming Examples

The code examples in this section illustrate how to program the EFCOP to complete three basic EFCOP operations: FIR filtering, IIR filtering, and adaptive filtering. These examples also illustrate the three EFCOP data transfer methods. The examples use equate labels for the location of the EFCOP registers and assume that these equates are declared prior to the example code. These labels include the register name preceded with “M_”.

9.6.1 Complex FIR Filter with Polling

The code in **Example 9-1** exhibits a simple way to program the EFCOP using polling to transfer complex data in and out of the EFCOP data registers. In Complex mode, each filter operation begins with a write of two data samples to the FDIR: one for the real part followed by one for the imaginary part of the input data. Two 32-bit data samples are written to the FDIR register using two **move.l** instructions. More efficiently, two 32-bit data samples (the real and the imaginary

part) can be written to the FDIR with one **move.2l** instruction. This code programs the EFCOP to implement a complex FIR filter based on the equation from **Section 9.2.2**, as follows:

1. The address register pointers are initialized for the filter input and output data (`INPUT` and `OUTPUT`) and for the EFCOP input and output registers.
2. EFCOP control parameters are written to the appropriate memory-mapped control registers as follows:
   a. The FDBA and FCBA registers are written with 0x400. To determine where the FDM and FCM are located in memory based on these register settings, recall that these registers contain the offset of the FDM and FCM from their base addresses (0x70000 for the FDM and 0x78000 for the FCM) in four-byte resolution and that memory is addressed in one byte resolution. Therefore, the FDM and FCM are located at 0x400 multiplied by four plus the base address, which is memory location 0x71000 for the FDM and 0x79000 for the FCM.
   b. The `FCNT` constant defines the filter length and is equal to twice the number of complex filter coefficients (that is, if there are five complex filter coefficients, `FCNT` should be 10). `FCNT` –1 is written to the filter count register.
   c. The value 0x0011 is written to FCTL to enable the EFCOP in complex FIR filter mode with data initialization enabled.
3. Before the first filter operation, the FDM buffer must be initialized because the FPRC bit is clear. To initialize the FDM buffer for complex data, `FCNT`/2 complex samples must be written to the FDM through the FDIR. The code uses a short loop to write the first `FCNT/2` complex samples of the input data to the FDIR using **move.2l** instructions.
4. The code waits until an output is available in the FDOR register by polling the FSTR[FOBNE] bit. The code tests the FSTR[FOBNE] bit and jumps back to the test if the bit is not set. When this bit is set, the output of the filter operation is ready and the code continues.
5. The FDOR is read using a **move.2l** instruction, and the complex value is placed into the output data buffer.
6. The next complex input data sample is written to the FDIR, and the process continues until all the input data values are written to the FDIR.
7. The last output value is read, and the filter is complete.

Example 9-1. Complex FIR Filter Code

```
move.w #INPUT,r0move.w #OUTPUT,r1                    ;Init data pointers

move.l #M_FDIR,r2

move.l #M_FDOR,r3
```

```
    move.w #$400,d0                                    ;Init FDBA
    move.w d0,M_FDBA
    move.w #$400,d0                                    ;Init FCBA
    move.w d0,M_FCBA
    move.w #FCNT-1,d0                                  ;Init FCNT
    move.w d0,M_FCNT
    move.w #$0011,d0                                   ;Init FCTL
    move.w d0,M_FCTL

    doensh0 #FCNT/2                                    ;Init data taps
    move.2l (r0)+,d0:d1
    loopstart0
    move.2l d0:d1,(r2)
    move.2l (r0)+,d0:d1
    loopend0

    dosetup0 empty          doen0 #NSAMP
    loopstart0
emptymove.w M_FSTR,d4
    bmtstc #$0040,d4.1
    jt empty                                           ;Wait until out not empty

    move.2l (r3),d2:d3
    move.2l d2:d3,(r1)+                                ;Read output
    move.2l d0:d1,(r2)
    move.2l (r0)+,d0:d1                                ;Write input
    loopend0

endempty
    move.w M_FSTR,d4
    bmtstc #$0040,d4.1
    jt endempty                                        ;Wait until out not empty
    move.2l (r3),d2:d3
    move.2l d2:d3,(r1)                                 ;Read last output
```

Before this code can run, the SC140 core must initialize the coefficient buffer because EFCOP coefficient initialization is disabled for this example. **Example 9-2** shows an easy way to do this. Here the coefficients are positioned, with **org** and **dcl** directives, at the location of the FCM (which is memory location 0x79000 as described earlier). For each complex coefficient, the real and imaginary parts are stored in memory separately with the real part first. *The coefficients are stored in reverse order* so that the coefficient with the largest index is stored first and the coefficient with the smallest index is stored last.

Example 9-2. Coefficient Initialization

```
org P:$00079000                                    ;Init coefficients
dcl [Re H(FCNT-1)]
dcl [Im H(FCNT-1)]
   •
   •
   •
dcl [Im H(0)]
dcl [Im H(0)]
```

Code such as shown in **Example 9-1** and **Example 9-2** may not appear in an actual application, but this code shows how the EFCOP works in a very simple way. The next example shows a more sophisticated way to use the EFCOP.

9.6.2 Adaptive Filter With Interrupts

The code in **Example 9-3** shows how to program the EFCOP to implement a real FIR filter using interrupts as the transfer method. In Real mode, each filter operation begins by writing one 32-bit data sample to the FDIR register using a **move.l** instruction. This code uses Adaptive mode and the data output not empty interrupt ($\overline{\text{IRQ3}}$) to update the coefficients as shown in **Section 9.2.1.1**, as follows:

1. Address register pointers are initialized for the filter input and output data (`INPUT` and `OUTPUT`).
2. The EFCOP control parameters are written to the appropriate memory mapped control registers as follows:
   a. FDM and FCM are located at the beginning of the shared memory, so the FDBA and FCBA registers are written with zero.

b. The FCNT constant defines the filter length and is equal to the number of real filter coefficients. FCNT - 1 is written to the filter count register.

c. The value 0x2105 is written to the control register to enable the EFCOP in real FIR filter mode with Adaptive mode enabled and data and coefficient initialization enabled. This value also enables the Data Output Not Empty ($\overline{\text{IRQ3}}$) interrupt.

3. The code enables interrupts as follows:

a. The appropriate bits in the core control registers are set. The interrupt mask bits (I0–2) of the SR are cleared to permit all interrupt priority levels.

b. The value 0x7000 is written to the ELIRA. This value sets the PIL[30–32] bits to assign $\overline{\text{IRQ3}}$ with a priority level a six. This value also clears the PED3 bit to assign $\overline{\text{IRQ3}}$ to be level triggered.

c. Interrupts are enabled by issuing an **ei** instruction.

4. The FDM buffer must be initialized because the FPRC bit is clear. To initialize the FDM buffer, FCNT samples are written to the FDM through the FDIR. The code uses a short loop to write the first FCNT input samples to the FDIR using a **move.l** instruction.

5. The FCM buffer must be initialized because the FCTL[FCIM] bit is set. After the FKIR value is written to the K-constant input register, the EFCOP initializes the FCM buffer using a coefficient update session with the original coefficients equal to zero (so it does not matter what the values are in the FCM buffer memory locations before the code begins).

6. Processing begins with the write of the first input data sample to the FDIR.

Example 9-3. Adaptive Filter Code

```
move.w #INPUT,r0move.w #OUTPUT,r1        ;Init data pointers


move.w #0,d0                                          ;Init FDBA
move.w d0,M_FDBA
move.w #0,d0                                          ;Init FCBA
move.w d0,M_FCBA
move.w #FCNT-1,d0                                     ;Init FCNT
move.w d0,M_FCNT
move.w #$2105,d0                                      ;Init FCTL
move.w d0,M_FCTL


bmclr #$00E0,sr.h                                     ;Enable all IPL
move.w #$7000,d0                                      ;Out not empty IPL 6
move.w d0,M_ELIRA
```

```
ei                                                    ;Enable interrupts

doensh0 #FCNT
loopstart0                                            ;Init data taps
move.l (r0)+,d0
move.l d0,M_FDIR
loopend0

move.l #FKIR,d0                                       ;Init coeffs
move.l d0,M_FKIR

  move.l (r0)+,d0                                     ;Write first input
move.l d0,M_FDIR
```

The main code is now complete and the SC140 core can run other application code while the EFCOP completes the filter operation. When the EFCOP completes the filter operation, it places the result in the FDOR, which triggers the data output not empty interrupt. The processor jumps to the appropriate interrupt vector address.

The interrupt vector code, shown in **Example 9-4**, uses an equate label, I_IRQ3, for the location of the EFCOP data output not empty interrupt vector address. The example assumes that this equate is declared prior to the example code. The interrupt vector code includes the command to jump to the interrupt service routine.

Example 9-4. Interrupt Vector Code

```
org P:I_IRQ3                                          ;Output not empty vector
jsr OBNE_ISR
rte
```

The interrupt service routine code, shown in **Example 9-5**, completes the processing as follows:

1. The code moves the filter output from FDOR to the `output` data buffer.
2. The step parameter is loaded into FKIR. Once FKIR is loaded, the EFCOP performs the coefficient update session, as discussed in **Section 9.2.1.1**, and replaces the filter coefficients with the updated coefficients.
3. The next input sample is written from the `input` data buffer to the FDIR to begin the next filter operation, and the process begins again.

Example 9-5. Interrupt Service Routine Code

```
OBNE_ISR                                        ;Output not empty ISR
   move.l M_FDOR,d0                             ;Read output
   move.l d0,(r1)+
   move.l #FKIR,d0                              ;Write coef update param
   move.l d0,M_FKIR
   move.l (r0)+,d0                              ;Write input
   move.l d0,M_FDIR
   rts
```

Example 9-5 shows the basics of adaptive filtering. The transfer method of this example is more complex than in the polling example, but the SC140 core can process other routines while the EFCOP is computing. Code similar to this example may be seen in an actual application. This example is not complete, because it does not contain code to determine how many samples to process. This example also uses a simple constant for the coefficient update parameter. Other applications may calculate a value to update the coefficients.

9.6.3 Real IIR Filter with DMA

The code in **Example 9-6** and **Example 9-7** shows how to program the EFCOP to implement a real IIR filter using DMA to transfer data into and out of the EFCOP data registers. Processing a complete IIR filter requires two sessions: a FIR filter session (see **Section 9.2**) followed by an IIR filter session (see **Section 9.3**). This example uses dual-access DMA transactions for the FIR session and flyby DMA transactions for the IIR session. Each DMA transaction transfers `NSAMP` bytes, so the EFCOP processes NSAMP/4 32-bit samples.

Example 9-6 and **Example 9-7** use equate labels for the location of the DMA channel configuration registers and DMA Channel Parameter RAM fields. The code assumes that these equates are declared prior to the example code. These labels include the register or field name preceded with “M_”. This code also assumes that banks 10 and 11 of the memory controller are configured to allow the DMA to access the internal DSP SRAM through the UPMC and to access the EFCOP registers through the GPCM, respectively. The memory buffers to which the `IN_ADDR`, `TMP_ADDR`, and `OUT_ADDR` equates point must be within the memory range of bank 10. The `FDIR_ADDR` and `FDOR_ADDR` equates must point to the EFCOP input and output register locations in bank 11.

The code in **Example 9-6** shows the code for the FIR session using dual-access DMA transactions. Four DMA channels are used for the FIR session: two channels (0 and 1) to transfer the data to the FDIR and two channels (2 and 3) to transfer the data from the FDOR. The DMA transfers occur in single transfer mode, that is, the DMA transfers one 32-bit sample to the FDIR whenever the FDIR is not full, and the DMA transfers one 32-bit sample from the FDOR whenever the FDOR is not empty. The FIR session proceeds as follows:

1. The following control parameters are written to the EFCOP control registers:
   a. The FDM and FCM are located at an offset from the beginning of the shared memory defined by the `FIR_FDBA` and `FIR_FCBA` constants.
   b. The `FIR_FCNT` constant defines the FIR filter length and is equal to the number of real filter coefficients. `FIR_FCNT` – 1 is written to the Filter Count Register.
   c. The value 0x0081 is written to FCTL to enable the EFCOP in real FIR filter mode with data initialization disabled. This value also sets the input and output data modes to single-transfer.
2. DMA Channel 0 transfers the data from memory to the DMA FIFO. The channel 0 DMA control registers are programmed as follows:
   a. The address location of the input data, `IN_ADDR`, is written to the DMA buffer address pointer field (BD_ADDR0).
   b. The total number of bytes to transfer, `NSAMP`, is written to the DMA buffer size field (BD_SIZE0).
   c. To configure channel 0 for 32-bit read transactions, the value 0x00000190 is written to the DMA attribute field (BD_ATTR0).
   d. To enable channel 0 as dual access transaction initiated by the DMA, the value 0x80000045 is written to the DMA Channel Configuration Register (DCHCR0).
3. DMA Channel 1 transfers the data from the DMA FIFO to the FDIR. The channel 1 DMA control registers are programmed as follows:
   a. The address location of the FDIR in bank 11, `FDIR_ADDR`, is written to the DMA buffer address pointer field (BD_ADDR1).
   b. The total number of bytes to transfer, `NSAMP`, is written to the DMA buffer size field (BD_SIZE1).
   c. To configure channel 1 for 32-bit write transactions without incrementing the buffer address (always transfers to the FDIR), the value 0x08000180 is written to the DMA attribute field (BD_ATTR1).
   d. To enable channel 1 in dual access mode triggered by an EFCOP write request, the value 0x80010305 is written to the DMA Channel Configuration Register (DCHCR1).
4. DMA Channel 2 transfers the data from the FDOR to the DMA FIFO. The channel 2 DMA control registers are programmed as follows:
   a. The address location of the FDOR in bank 11, `FDOR_ADDR`, is written to the DMA buffer address pointer field (BD_ADDR2).

b. The total number of bytes to transfer, NSAMP, is written to the DMA buffer size field (BD_SIZE2).

c. The value 0x08000190 is written to the DMA attribute field (BD_ATTR2). This value configures channel 2 for 32-bit read transactions without incrementing the buffer address (always transfers from the FDOR).

d. The value 0x80020205 is written to the DMA Channel Configuration Register (DCHCR2). This value enables channel 2 in dual access mode triggered by an EFCOP read request.

5. DMA Channel 3 transfers the data from the DMA FIFO to memory. The channel 3 DMA control registers are programmed as follows:

a. The address location of the FIR output data, TMP_ADDR, is written to the DMA buffer address pointer field (BD_ADDR3).

b. The total number of bytes to transfer, NSAMP, is written to the DMA buffer size field (BD_SIZE3).

c. To configure channel 3 for a 32-bit write transaction, the value 0x00000180 is written to the DMA attribute field (BD_ATTR3).

d. To enable channel 3 as dual access transaction initiated by the DMA, the value 0x80030045 is written to the DMA Channel Configuration Register (DCHCR3).

Once the DMAs and EFCOP are programmed, they work together requesting and sending data without intervention of the SC140 core. After all NSAMP/4 data samples are processed, the IIR session begins.

Example 9-6. FIR Filter Session

```
   move.w #FIR_FDBA,d0                          ;Init FDBA
   move.w d0,M_FDBA
   move.w #FIR_FCBA,d0                          ;Init FCBA
   move.w d0,M_FCBA
   move.w #FIR_FCNT-1,d0                        ;Init FCNT
   move.w d0,M_FCNT
   move.w #$0081,d0                             ;Init FCTL
   move.w d0,M_FCTL


;DMA0 init to transfer Memory to DMA FIFO
   move.l #IN_ADDR,d0                           ;Init source address
   move.l d0,M_BDADDR0
   move.l #NSAMP,d0                             ;Init transfer size
   move.l d0,M_BDSIZE0
```

```
    move.l #$00000190,d0                    ;Init channel 0 attrib
    move.l d0,M_BDATTR0
    move.l #$80000045,d0                    ;Init channel 0 config
    move.l d0,M_DCHCR0

;DMA1 init to transfer DMA FIFO to FDIR
    move.l #FDIR_ADDR,d0                    ;Init destination address
    move.l d0,M_BDADDR1
    move.l #NSAMP,d0                        ;Init transfer size
    move.l d0,M_BDSIZE1
    move.l #$08000180,d0                    ;Init channel 1 attrib
    move.l d0,M_BDATTR1
    move.l #$80010305,d0                    ;Init channel 1 config
    move.l d0,M_DCHCR1
;DMA2 init to transfer FDOR to DMA FIFO
    move.l #FDOR_ADDR,d0                    ;Init source address
    move.l d0,M_BDADDR2
    move.l #NSAMP,d0                        ;Init transfer size
    move.l d0,M_BDSIZE2
    move.l #$08000190,d0                    ;Init channel 2 attrib
    move.l d0,M_BDATTR2
    move.l #$80020205,d0                    ;Init channel 2 config
    move.l d0,M_DCHCR2
;DMA3 init to transfer DMA FIFO to Memory
    move.l #TMP_ADDR,d0                     ;Init destination address
    move.l d0,M_BDADDR3
    move.l #NSAMP,d0                        ;Init transfer size
    move.l d0,M_BDSIZE3
    move.l #$00000180,d0                    ;Init channel 3 attrib
    move.l d0,M_BDATTR3
    move.l #$80030045,d0                    ;Init channel 3 config
    move.l d0,M_DCHCR3
```

Example 9-7 shows the code for the IIR session using flyby DMA transactions. Two DMA channels are used for the IIR session: channel 0 to transfer the data to the FDIR and channel 1 to transfer the data from the FDOR. The DMA transfers occur in burst transfer mode, that is,

channel 0 transfers eight 32-bit samples to the FDIR whenever the FDIR is empty and channel 1 transfers eight 32-bit samples from the FDOR whenever the FDOR is full. The IIR session proceeds as follows:

1. The EFCOP is disabled by clearing the FCTL before the IIR session parameters are programmed into the FCTL.
2. The following control parameters are written to the EFCOP control registers:
   a. The FDM and FCM are located at an offset from the beginning of the shared memory defined by the `IIR_FDBA` and `IIR_FCBA` constants.
   b. The `IIR_FCNT` constant defines the IIR filter length and is equal to the number of real filter coefficients. `IIR_FCNT` – 1 is written to the Filter Count Register.
   c. The `IIR_FACR` constant is written to the Filter ALU Control Register. This constant can be used to enable scaling for the IIR output if necessary.
   d. The value 0xC083 is written to FCTL to enable the EFCOP in IIR filter mode. This value also sets the input and output data modes to burst transfer.
3. DMA Channel 0 of the DMA is used in flyby mode to transfer the input data from memory to the FDIR in burst mode; that is, the DMA transfers eight 32-bit samples to the FDIR whenever the FDIR is empty. The DMA control registers are programmed as follows:
   a. The address location of the input data (the FIR session output), `TMP_ADDR`, is written to the DMA buffer address pointer field (BD_ADDR0).
   b. The total number of bytes to transfer, `NSAMP`, is written to the DMA buffer size field (BD_SIZE0).
   c. To configure the DMA for a burst read transaction, the value 0x00000210 is written to the DMA attribute field (BD_ATTR0).
   d. To enable DMA channel 0 in flyby mode triggered by an EFCOP write request, the value 0x80004305 is written to the DMA Channel Configuration Register (DCHCR0).
4. DMA Channel 1 of the DMA is used in flyby mode to transfer the output data from the FDOR to memory in burst mode; that is, the DMA transfers eight 32-bit samples from the FDOR whenever the FDOR is full. The DMA control registers are programmed as follows:
   a. The address location of the output data, `OUT_ADDR`, is written to the DMA buffer address pointer field (BD_ADDR1).

b. The total number of bytes to transfer, NSAMP, is written to the DMA buffer size field (BD_SIZE1).

c. The value 0x00000200 is written to the DMA attribute field (BD_ATTR1). This value configures the DMA for a burst write transaction.

d. The value 0x80014204 is written to the DMA Channel Configuration Register (DCHCR1). This value enables DMA channel 1 in flyby mode triggered by an EFCOP read request.

Example 9-7. IIR Filter Session

```
   move.w #0,d0                                    ;Disable EFCOP
   move.w d0,M_FCTL

   move.w #IIR_FDBA,d0                             ;Init FDBA
   move.w d0,M_FDBA
   move.w #IIR_FCBA,d0                             ;Init FCBA
   move.w d0,M_FCBA
   move.w #IIR_FCNT-1,d0                           ;Init FCNT
   move.w d0,M_FCNT
   move.w #IIR_FACR,d0                             ;Init FACR
   move.w d0,M_FACR
   move.w #$C083,d0                                ;Init FCTL
   move.w d0,M_FCTL

;DMA0 init to input DATA to EFCOP
   move.l #TMP_ADDR,d0                             ;Init source address
   move.l d0,M_BDADDR0
   move.l #NSAMP,d0                                ;Init transfer size
   move.l d0,M_BDSIZE0
   move.l #$00000210,d0                            ;Init channel 0 attrib
   move.l d0,M_BDATTR0
   move.l #$80004305,d0                            ;Init channel 0 config
   move.l d0,M_DCHCR0

;DMA1 init to output DATA from EFCOP
   move.l #OUT_ADDR,d0                             ;Init destination address
   move.l d0,M_BDADDR1
```

```
move.l #NSAMP,d0                                    ;Init transfer size
move.l d0,M_BDSIZE1
move.l #$00000200,d0                                ;Init channel 1 attrib
move.l d0,M_BDATTR1
move.l #$80014204,d0                                ;Init channel 1 config
move.l d0,M_DCHCR1
```

Once the DMAs and EFCOP are programmed, they work together requesting and sending data without intervention of the SC140 core. When all NSAMP/4 data samples are processed, the filter is complete.

Example 9-6 and **Example 9-7** show the basics of IIR filtering and DMA transactions with the EFCOP. The transfer method of this example requires the least core intervention once the EFCOP and DMAs are programmed and requires no address registers. Code similar to this may appear in an actual application. However, this example is not complete because it does not contain code to initialize the FDM and FCM. This example also does not contain code to determine when the DMA transfers of each session are complete, which can be done with DMA interrupts or polling.

9.7 Related Reading

MSC8101 User's Guide (This manual)

Section 1.3.7.3, *Buffer Descriptors,* on page 1-11

MSC8101 Reference Manual

Chapter 15, *Direct Memory Access (DMA)*

Chapter 18, *Enhanced Filter Coprocessor (EFCOP)*

Multi- Channel Controllers (MCCs) 10

This chapter describes a step-by-step procedure for setting up a 32-channel T1/E1 link using one of the MSC8101 multi-channel controllers (MCCs). The main steps in this procedure are three software configuration phases for setting up MCC and CPM parameters. An example driver implementation illustrates both hardware and software configuration for connecting to an industry-standard T1/E1 line transceiver. The physical interface between the MSC8101 and the PMC PM6388 T1/E1 transceiver is described.

The procedure for global parameter set-up and channel-specific initialization exemplifies the basic CPM concepts in use throughout the other CPM-supported protocols. This basic procedure can be used for applications with varying numbers of channels and protocols. Refer to the *MSC8101 Reference Manual* for information on features not implemented by the driver discussed here.

10.1 MCC Configuration Basics

The MSC8101 communications processor module (CPM) contains two MCC blocks, each capable of providing up to 128 full-duplex serial data channels routed through the programmable time-slot assigner (TSA) in the serial interface blocks, SI1 and SI2. Target applications of the MCC are mainly time-division multiplexing (TDM) interfaces such as TDM backplanes/interconnects and WAN networks. The SI1 has one TDM interface, and the SI2 has three.

MCC channels are individually configured to handle either transparent or HDLC protocols. For each channel, the serial interface (SI) and its associated RAM (SIRAM) control the routing of time-division multiplexing data through each of the four TDM interfaces to the external network. The channels can be spread across the four TDMs, and a single TDM can handle all 128 channels. The MCC operates in both normal mode, in which a single logical channel is assigned to a single time slot, and a superchannel mode, in which multiple MCC channel slots are assigned to a logical channel. The example implementation illustrates only normal mode channels; for details on super channels, refer to the *MSC8101 Reference Manual*.

The main MCC configuration is through MCC-specific parameters in the internal Dual Port RAM (DPRAM). The MCC utilizes several other CPM resources, as the white boxes in **Figure 10-1** show.

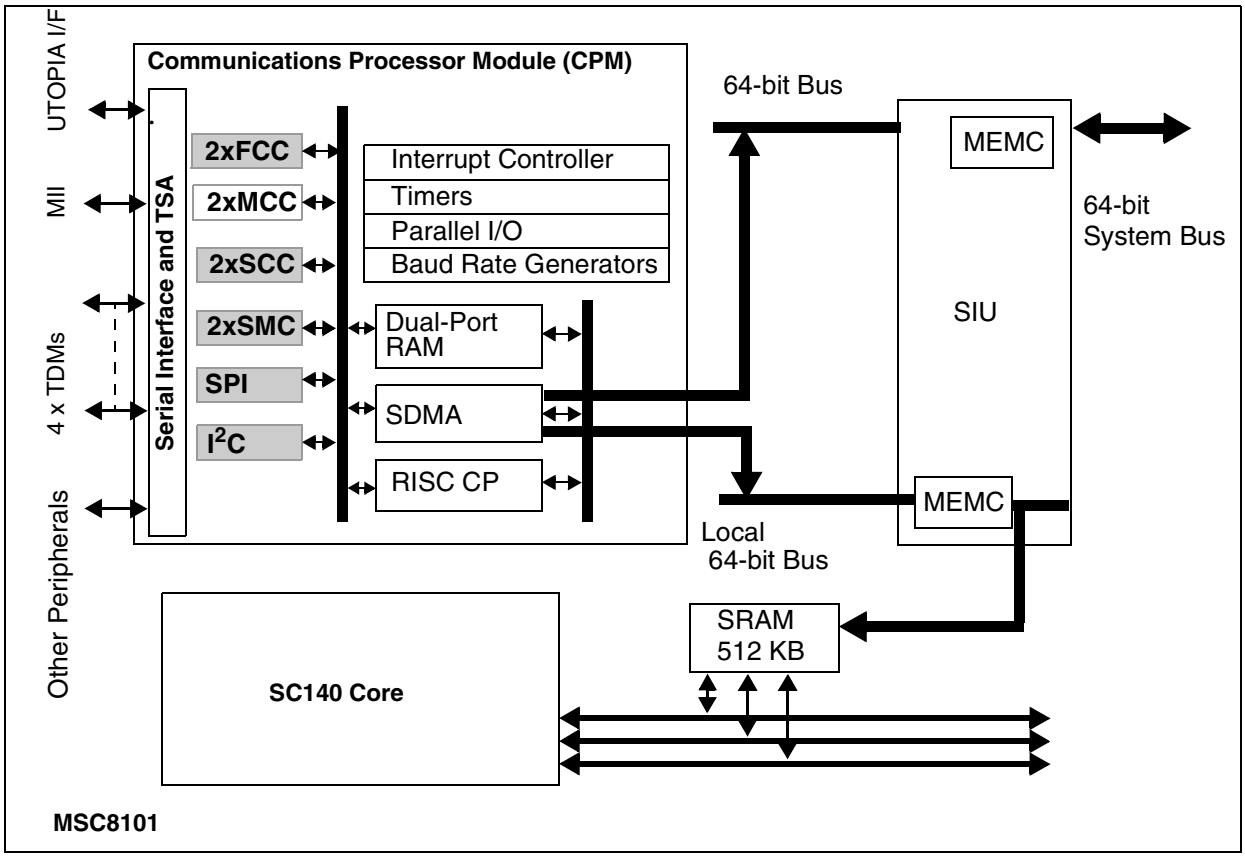


Figure 10-1. MCC Resource Usage

10.1.1 Procedure for Initializing the MCC Resources

The steps for initializing the MCC resources are as follows:

1. Configure the channels:
   a. Initialize the buffer descriptors (BDs).
   b. Set up the global parameters.
   c. Set up the MCC control registers.
   d. Set up the channel-specific parameters.
   e. Initialize the interrupt queues.
2. Select the TSA channel route to a TDM timeslot:
   a. Program the serial interface RAM (SIRAM).
   b. Set up the baud-rate generators (BRGs).
3. Configure the external interface:
   a. Set up the parallel I/O pins.

b. Enable the TDM.

These steps map to the functionality flow of the software driver discussed in this chapter. See **Figure 10-2** for a listing of the driver functions.

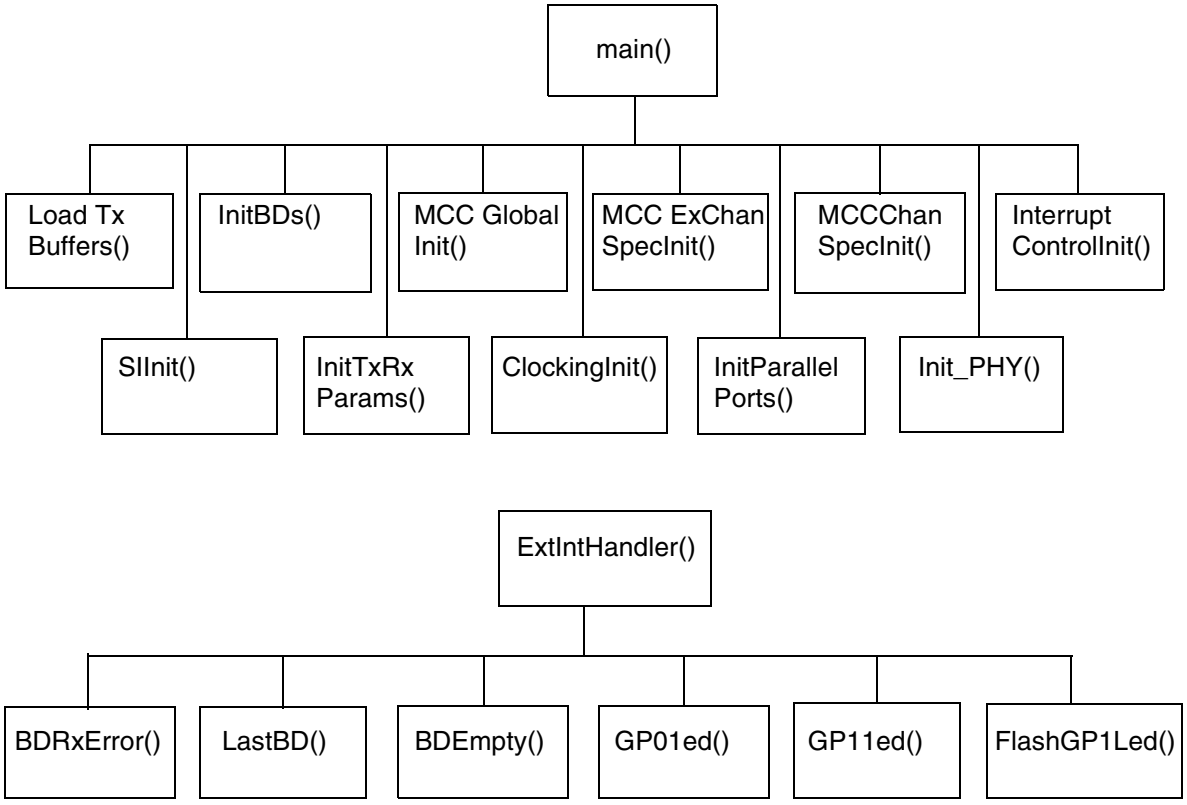


Figure 10-2. Driver Functions

ExtIntHandler() provides an example interrupt handler. The handler checks the event that causes the interrupt and, in the HDLC loopback modes, enables a memory check to ensure that the received data is identical to the transmitted data. **Table 10-1** shows examples of functions that are useful in handling interrupts.

Table 10-1. Interrupt Handler Functions

Function Name	Function Details
BDRxError()	Identifies the reason for the Rx interrupt.
LastBD()	Indicates whether this is the last buffer descriptor in the ring.
BDEmpty()	Determines whether the buffer descriptor is empty.

10.1.2 Driver Memory Map

All values in the driver memory map are set up as offsets to the Internal Memory Map Register (IMMR), indicated in **Figure 10-3** as the DPRAM Base. These values are changed via the `mcc.h` header file, with these exceptions: the MCC2 parameter RAM, in which the global parameters are stored, and the channel-specific parameters, which both have fixed locations. When writing to the parameter RAM, you must ensure that other parameters are not inadvertently overwritten. External memory refers either to the internal SRAM or external memory. The software driver uses only the DPRAM and the 512 KB internal SRAM, shown as external memory in **Figure 10-4**. **Figure 10-3** shows a detailed memory map for the specific driver example.

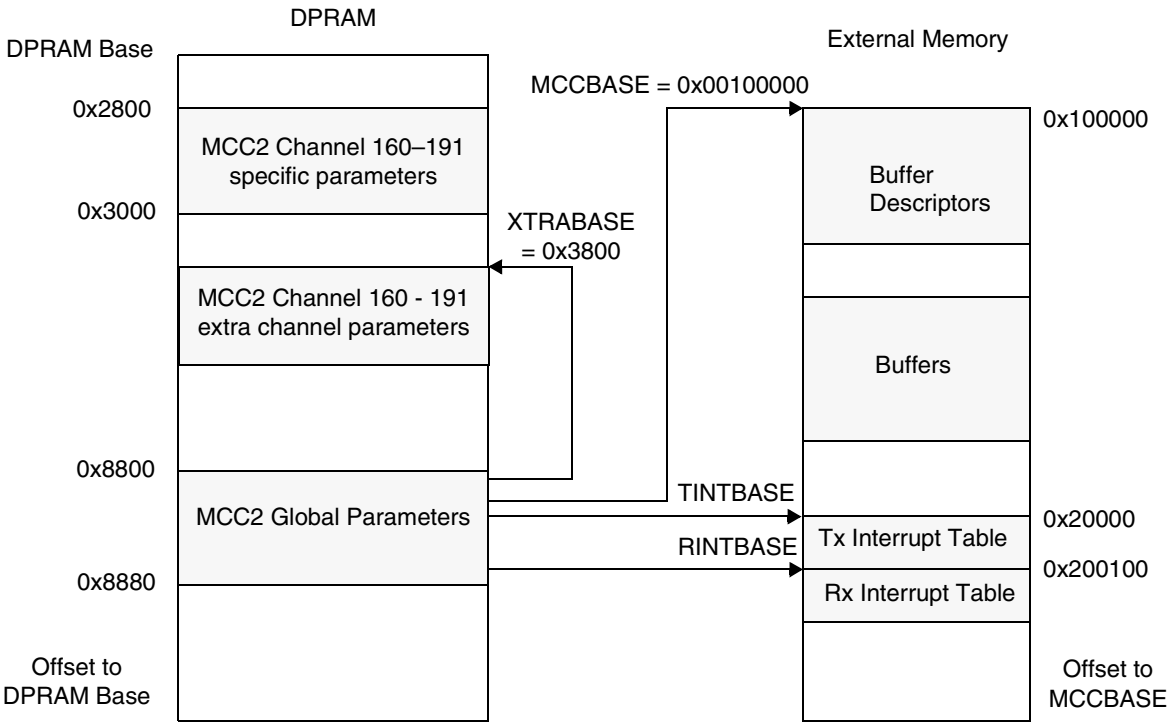


Figure 10-3. Driver Memory Map

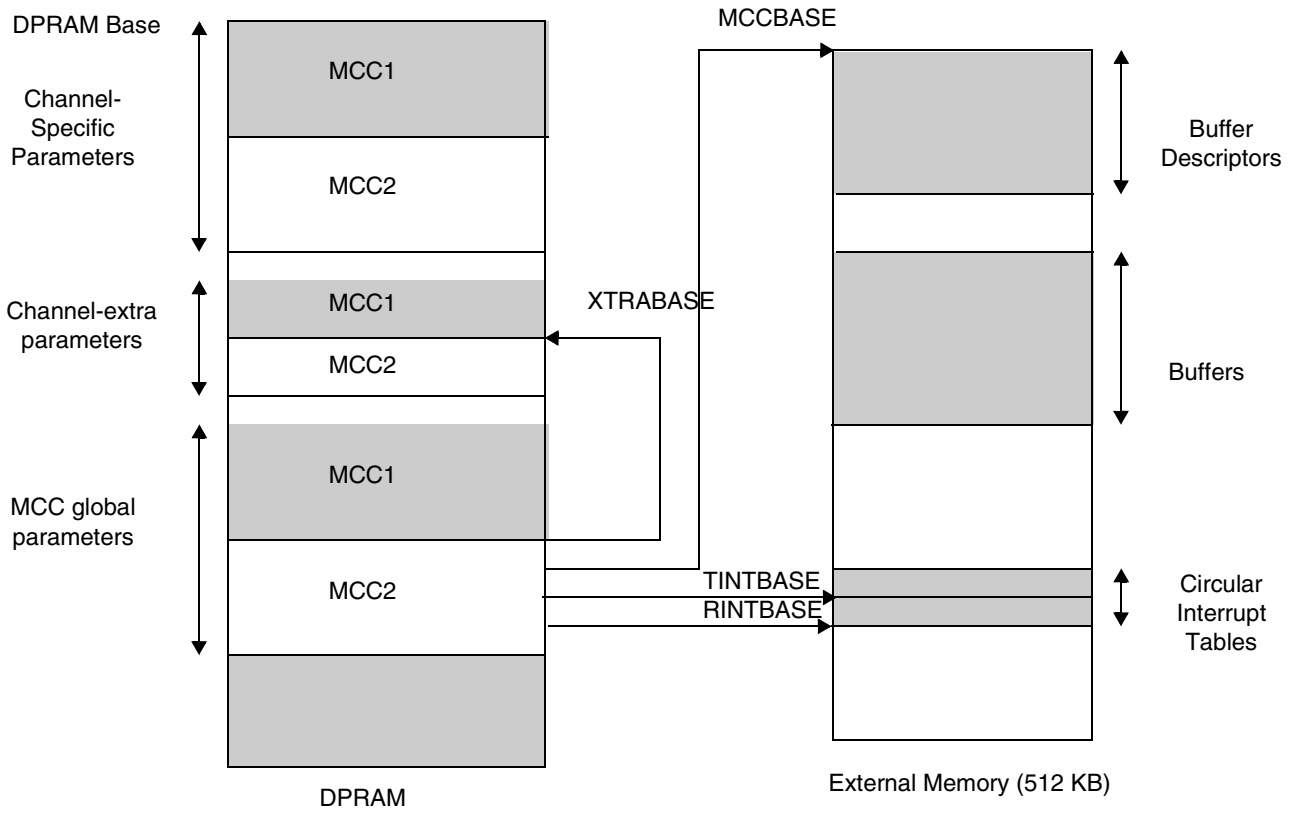


Figure 10-4. Internal and External Memory Usage

10.1.3 Memory Usage

Memory resources can become scarce as the number of MCC channels increases. The size of the BD ring should be varied to suit the protocol being run and the amount of other CPM activity. To avoid a receive overrun or transmit underrun, the circular BD table should be sized to provide enough valid buffers for the available SC140 core servicing rate. See **Section 1.3.7.5**, *BD and Buffer Memory Structure,* on page 1-15.

Table 10-2 details the potential memory usage for the MCC in two cases: (1) maximum number of channels (256) supported; (2) the specific driver example code. It indicates whether the parameters are stored in DPRAM or external memory. The interrupt circular tables are shown in external memory; however they can also be stored in DPRAM memory, space permitting.

Table 10-2. Memory Utilization for MCC Parameters and Resources

Parameters	DPRAM		External Memory	
	Maximum	Driver	Maximum	Driver
MCC global parameters	2 × 128 bytes	128 bytes	N/A	N/A
Channel-extra parameters	256 × 8 bytes	32 × 8 bytes	N/A	N/A

Table 10-2. Memory Utilization for MCC Parameters and Resources (Continued)

Parameters	DPRAM		External Memory	
	Maximum	Driver	Maximum	Driver
Channel-specific parameters	256 × 64 bytes	32 × 64 bytes	N/A	N/A
Buffer descriptors	N/A	N/A	(256 × 8 × 8) × 2 bytes	(32 × 8 × 8) × 2 bytes
Buffers	N/A	N/A	(256 × 8 × 64) × 2 bytes	(32 × 8 × 64) × 2 bytes
Circular interrupt tables			(4 × 64) × 5 bytes	(4 × 64) × 2 bytes
Total memory used	18688 bytes	2432 bytes	296192 bytes	37376 bytes
Notes: 1. External memory utilization assumptions: 8 Tx/Rx buffers per channel, 64 bytes each; 2 INTQs, 64 entries each.				

To simplify the relocation, all BDs for each MCC start from a fixed base in external memory, with the restriction that all BDs for a specific MCC are located within the same 512 KB block.

Rx and Tx BDs have the same structure, consisting of status, length and data buffer address fields. The Empty and Ready status field bits of the Rx and Tx BDs, respectively, are set by the core prior to channel start-up. These fields indicate to the CPM that each buffer is ready to be processed. For the final BD in the ring, the wrap bit is set, indicating to the CPM that the next BD entry wraps back to the first BD in the ring.

10.2 Connect the TDM Interface to T1/E1

This section details the interconnection of the MSC8101 to one port on the PM6388 Octal T1/E1 line transceiver and outlines the loopback configurations available to aid in software driver development.

The standardized TDM lines used for inter-hub backbones or trunks are T1/E1 (CEPT) lines in Europe and T1 lines in the US. An T1/E1 interface implements 32 × 64 Kbps slots, giving a 2.048 Mbps bandwidth. T1 implements 24 × 64 Kbps time slots for a 1.544 Mbps line bandwidth link. Both time-division, multiplexed interfaces can be implemented using the TSA capability of the serial interface in conjunction with an MCC of the MSC8101 CPM. The TDM pins must be carefully selected so that there are no conflicts with other CPM functions multiplexed on the same pins. **Figure 10-5** shows MCC2 on the MSC8101 connected to port 6 on the PM6388 Octal T1/E1 transceiver through TDMB. T1 applications can use the same interconnect, with the PM4388 transceiver as a drop-in alternative.

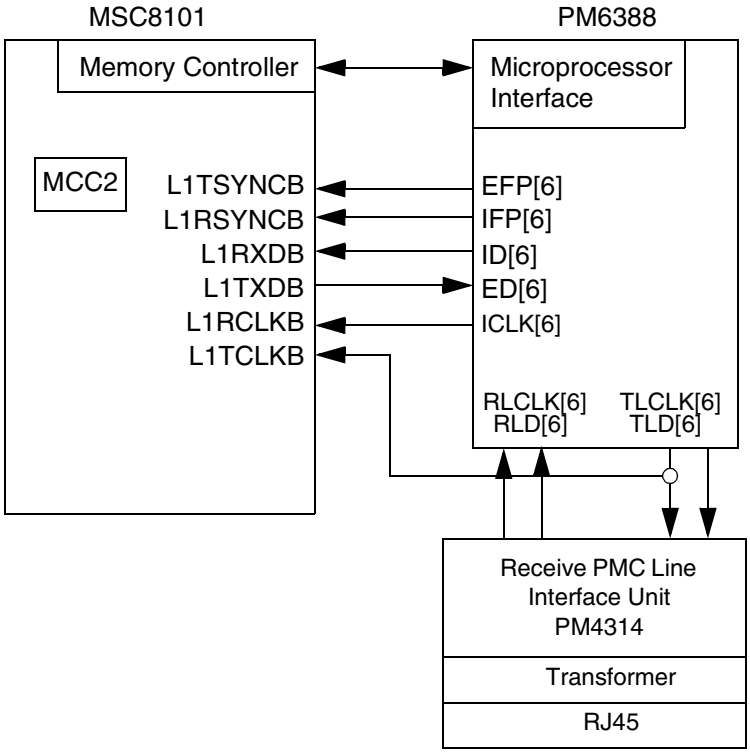


Figure 10-5. T1/E1 Transceiver Interface Example

The TDM interface connection is relatively simple, consisting of a transmit and receive clock, synchronization signals, and data signals. The MSC8101 expects L1RCLKB and L1RSYNCB synchronization to be generated by an external device—in this case, the T1/E1 line transceiver. The T1/E1 frame is delimited by the transceiver synchronization signal (EFP/IFP) that marks the start of the first time slot in the frame and by a clock (ICLK/TLCLK) that controls the line bit rate. The T1/E1 frame is split into multiple time slots, each designated for a different logical channel. **Figure 10-6** illustrates an T1/E1 frame consisting of thirty-two 8-bit logical channels (with common L1RSYNC/L1TSYNC and L1RCLK/L1TCLK used).

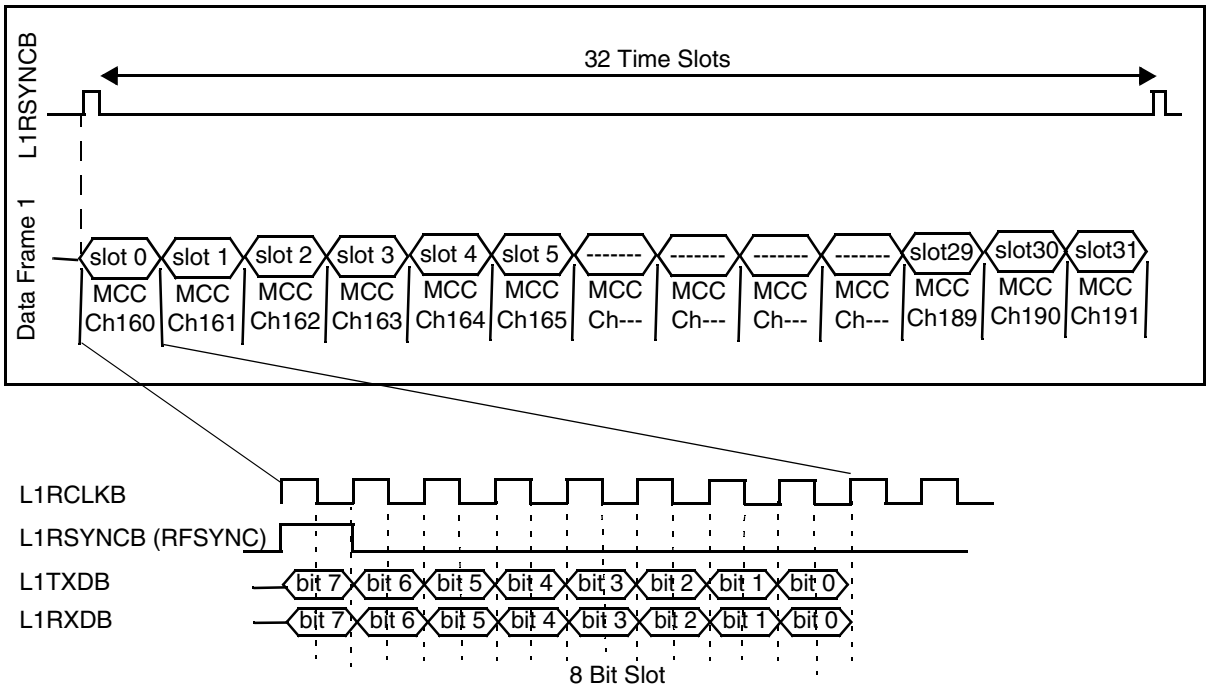


Figure 10-6. T1/E1 Data Frame

10.2.1 Provide Appropriate Signal Polarity and Timing

Table 10-3 defines the SI Mode Register bit settings required on the MSC8101 to provide appropriate signal polarity and timing to the PM6388 line transceiver.

Table 10-3. MSC8101 SI Mode Register Settings

Register Setting	Description
SIxMR[RFSDx] = 00 SIxMR[TFSDx] = 00	L1RSYNCB/L1TSYNC have no L1RCLK/L1TCLK delay from synchronization to data.
SIxMR[FEx] = 0	L1TSYNCB and L1RSYNCB pulses are sampled at the falling edge of TCLK/RCLK.
SIxMR[SLx] = 0	L1TSYNCB and L1RSYNCB are active high signals.
SIxMR[CEx] = 0 SIxMR[DSCx] = 0	Rx Data is latched in on the falling edge, Tx data on the rising edge of L1TCLK/L1RCLK. The double-speed TDM clock is *not* used.

10.2.2 Perform a Phased Test of the Transceiver Interface

During the development process, several loopback options can aid in a phased test of the PHY interface.

- *SIRAM loopback*. Allows verification of the MCC and serial interface programming with loopback at the serial interface.
- *TDM loopback*. Tests the time-slot assigner (TSA) programming with L1TXDB connected internally to L1RXDB.

- *External PHY loopback*. The full external Transmit-to-Receive path tests the physical interface between the MSC8101 and the physical (PHY) device.

Figure 10-7 summarizes the three system-level loopback options: (1) SI-level internal loopback, (2) TDM-level internal loopback, and (3) external PHY loopback.

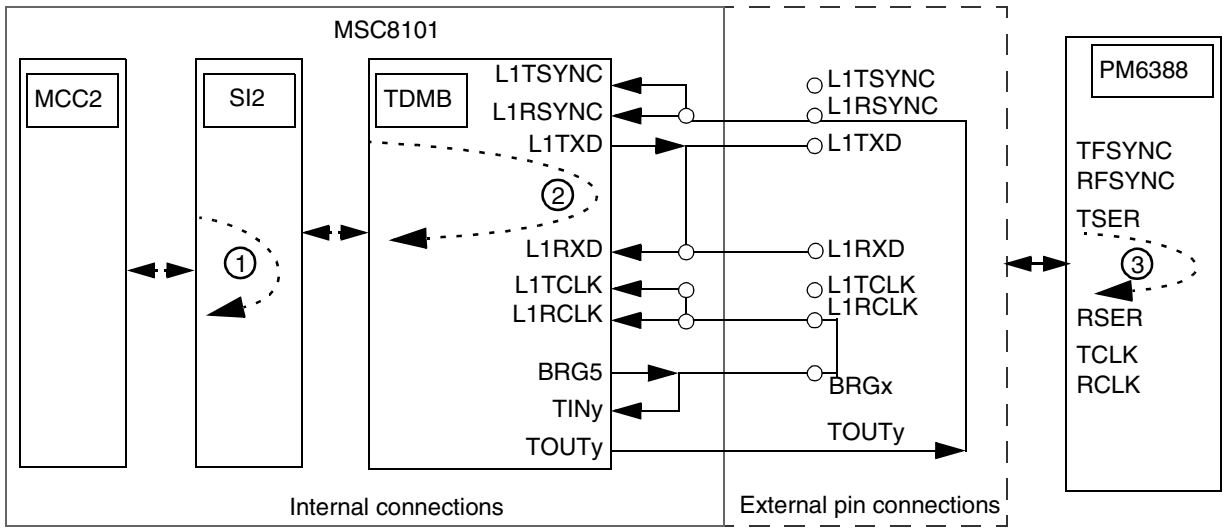


Figure 10-7. Loopback Modes

In the absence of an external PHY, loopback modes (1) and (2) can be used, with clocks and synchronizing signals generated by the MSC8101. The transmit and receive clocks for TDMB are selected through the CMX SI2 Clock Route Register (CMXSI2CR); the selected clock input pin is sourced from one of the flexible internal baud rate generator (BRG) outputs. Frame synchronization is generated from one of the internal timers: the timer registers are programmed to divide the BRGCLK by (32 × 8), thus creating a positive, bit-wide pulse every 32 timeslots.

The MSC8101 has independent receive and transmit clock signals and synchronization signals; however, making these signals common can simplify the interconnection and synchronization. The internal loopback configurations use this concept: setting the SI2MR[CRTx] bit internally connects the transmit clock (L1TCLK) and synchronization (L1TSYNC) to the respective receive clock (L1RCLK) and synchronization (L1RSYNC).

In external PHY loopback mode (3), the PM6388 PHY is used. It generates separate transmit and receive frame syncs, so the physical connection between the timer output (TOUT) and the L1RSYNC is not used. Instead, the Ingress Frame Pulse (IFP) from the PM6388 connects directly to the L1RSYNC in the receive direction, and the Egress Frame Pulse (EFP) sources L1TSYNC for the transmit side. The TDM clocks between the MSC8101 and the T1/E1 transceiver are also generated by the PM6388 with the Transmit Line Clock (TLCLK) sourcing L1TCLK and the Ingress Clock (ICLK) connected to L1RCLK (see **Figure 10-5**).

The PM6388 T1/E1 PHY transceiver is initialized in software through the Init_PHY() function. The driver code sets up the PHY to be in internal loopback mode to enable external PHY loopback.

10.3 Configure the Channels

Channel configuration proceeds in two steps:

- Configure the global MCC resources applicable to all the channels supported by the MCC.
- Configure the individual, channel-specific parameters for the assigned protocol.

The main data structures for programming the MCC are held in the CPM DPRAM. However, several other structures must reside in memory external to the CPM, either in the internal SRAM or in a memory device connected to the system bus. The next sections describe the global and channel-specific data structures for configuring MCC resources.

10.3.1 Set Up the Global MCC Parameters

Each MCC has a set of global parameters that are held in DPRAM and are common to all channels within that MCC. The global parameters define a base for the Transmit (Tx) and Receive (Rx) circular buffer descriptor (BD) tables, the maximum buffer size, the number of receive frame interrupts that cause an interrupt to the SC140 core, and the interrupt queue addresses. The buffer descriptor tables, which are stored in internal SRAM, define the Tx and Rx data buffer locations and maintain status information on received and transmitted data frames. The global parameters provide the common functionality for all active channels on each MCC.

The following parameters must be set up before the channel-specific parameters are assigned:

- *MCCBASE.* Defines the starting address of the 512 KB BD segment. In this case, it is set to 0x1000000 via the variable BDRING_BASE in the header file.
- *MCCSTATE.* Set to all zeros to define the initial MCC state.
- *MRBLR.* Defines the maximum number of bytes written to a receive buffer before a move to the next buffer. For transparent mode, the MRBLR should be assigned the same length as the buffers, 64 bytes in this example.
- *GRFTHR and GRFCNT.* Two parameters relating to the reception of frames. GRFTHR is a threshold value after which an interrupt is generated.
- *XTRABASE*. Defines the offset in the dual-port RAM (DPRAM) that points to the location holding the extra channel-specific parameters. Each channel's extra parameters are stored in order contiguously from this offset. (Offset 0x3800 is used in this example.)

The following parameters relate to interrupt queue set-up and handling:

- *TINTBASE*. Defines the Tx circular interrupt table location. In this driver example, the interrupt tables are held in external memory; however, they can be held in DPRAM.

- *RINTBASE.* Points to the receive circular table location. This example uses RINT Table 0.

Before the interrupts are enabled, the Rx and Tx temporary interrupt queue locations should be initialized to zero, with the wrap bit set in the last entry.

10.3.2 Set Up the MCC Configuration and Control Registers

Part of the global setup is to initialize the three main MCC control registers:

- *MCCFx.* Defines the mapping of MCC channel blocks to a TDM pin interface. **Table 10-4** shows the TDM-to-MCC usage available for the MSC8101. The 128 channels on each MCC are split into four subgroups, each of which can be routed to a particular TDM. All MCC1 channels must be routed through TDMA. The MCC2 subgroups are routed to one of three TDMs (TDMB, TDMC, or TDMD). All channels within a subgroup must be routed to the same TDM, though different subgroups can be routed to the same or different TDMs (see **Table 10-4** for details). The transmit and receive data flow is controlled by the programmable SIRAM and the respective MCCF2[0-7] register, which routes the data to the specified channels. The TDM group channel assignments made in the respective MCCF register must be coherent with the SI register programming. The example driver code configures 32 MCC2 channels in the range 160–191 for TDMB by setting the MCCF2 register to 0x10.

Table 10-4. MCC TDM Usage in MSC8101

	SIRAM1 (MCC1)				SIRAM2 (MCC2)			
MCC Subgroup	A	B	C	D	A	B	C	D
MCC Channel	0–31	32–63	64–95	96–127	128–159	160–191	192–231	231–256
Usable TDM	Yes	No	No	No	No	Yes	Yes	Yes

- *MCCM*. The Interrupt Mask Register filters interrupt event requests to the core. In this example, setting MCCM = 0x4004 enables the RINT0 and TINT interrupts.
- *MCCE.* The Interrupt Event Register reports receive and transmit events. This register is cleared by writing all ones (MCCE = 0xFFFF) at initialization. During interrupt handling, only events serviced at that time should be cleared by writing a one to the relevant register bit; otherwise interrupt events could be lost. Writing all zeros has no effect on the register.

10.3.3 Set Up Channel-Specific Parameters

The main channel-specific characteristics include maximum receive frame size, allowable core interrupts, start-up parameters for a channel, and the transparent or HDLC protocol to be supported by an individual channel. In addition, the RISC CP uses areas of this parameter RAM as temporary variable space. Both HDLC and transparent protocols have parameters that the user

must initialize. Most of these parameters are initialized to predefined values. However, INTMSK, CHAMR, TSTATE and RSTATE are user-configurable. In the example discussed here, you can configure all 32 channels to handle either transparent or HDLC protocols by setting the MODE variable appropriately in the driver.

- *INTMSK*. Determines which non-masked events are passed to the interrupt queue assigned to the channel. The driver code enables interrupts by setting INTMSK to 0x0105 when an RX frame is received (RXF event) or when buffers are transmitted or received (TXB and RXB events).
- *CHAMR*. Selects transparent or HDLC mode for the channel. The bit format of the register varies depending on the protocol and enables receive interrupt queue configuration for a respective channel. For the HDLC protocol, the CHAMR register is programmed to 0xE080 so that no timestamp is added to the data buffers and receive interrupt queue 0 is used. For transparent mode, this register is programmed to 0x7000.
- *RSTATE/TSTATE*. Provides the SDMA transactional details and starts the channel. Bits 0–7 of the RSTATE/TSTATE registers are set to 0x18 to select the byte ordering, the transfer code used during the SDMA channel memory access, and the bus used for data BDs and interrupt queues. In the driver, all the channels are set up in big-endian mode, with all data transactions on the local bus for access to the internal SRAM.

10.3.4 Set Up the Channel Extra Parameters

Each MCC channel has an 8-byte allocation for parameters defining the actual address of the Tx and Rx BDs for a specific channel. These extra parameters are located at an offset from the base address of the DPRAM, defined by XTRABASE, in the global parameters.

The driver sets up all channel-extra parameters for MCC2 from the [XTRABASE+(channel number × 8)] address. The TBASE/RBASE parameters define the base address of the Tx and Rx BDs for a particular channel. For example, for channel 160, the RBASE offset is 0x0500 and the TBASE offset is 0x0600. All channel BDs are contiguous in memory.

10.3.5 Initialize Circular Interrupt Queues

Each unmasked channel interrupt generated during the transmission and reception of data creates an entry in an interrupt queue. The receive and transmit entries are held in separate tables, with four receive interrupt tables and one transmit table that can be allocated to the MCC. The global MCC parameters define the MCC interrupt queue allocation; the queues are stored in internal SRAM.

The interrupt circular queue is initialized to be stored in external memory. Each MCC can be allocated up to five interrupt queues (one transmit and four receive). Each queue's length is user-definable in the global parameters. The final entry in the table is indicated with a set wrap bit to indicate that the next entry to be used is the first in the table.

In the example discussed here, RINT0 and TINT are used for the receive and transmit interrupts respectively, as set up in the global parameters. The Tx and Rx BDs of all channels that use a particular interrupt table must reside on the same bus.

10.4 Select the TSA Channel Route to a TDM Timeslot

Once the MCC is configured to support 32 channels, the channel route to a particular timeslot on the TDM interface must be selected based on the SIRAM entry and the CPM multiplexing settings.

10.4.1 Define the Serial Interface Entries in SIRAM

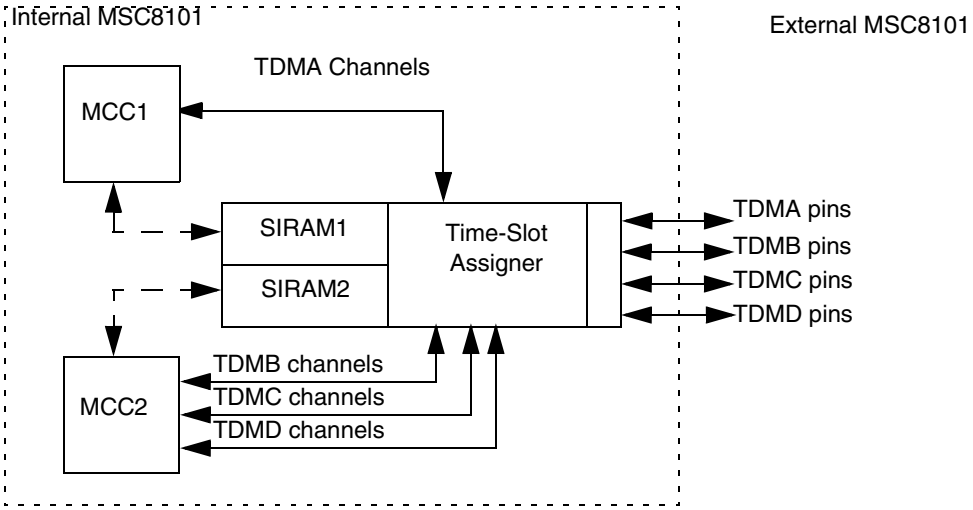


Figure 10-8. Serial Interface

The SIRAM is a block of memory internal to the CPM that routes data from the TDM pins to the MCC. The SIRAM consists of a series of entries, one set for the Tx and one for the Rx flow. **Figure 10-9** shows an example of the bit definitions for the Tx and Rx entries of channel 160 through the last entry for channel 191. To set the Tx and Rx entries, we perform the following steps:

1. For the Rx entry of channel 191, enable SI loopback mode by setting the Loop/Echo bit in the RX entry, thereby looping the transmit back to the receive. Note that the Loop/Echo bit should be set for either the Tx or Rx entry, but not both.
2. Define the MCC channel number in the MCSEL field.
3. Indicate the number of bytes routed using the CNT and BYT fields. A CNT of 000 represents “1,” and BYT indicates whether the CNT is measured in bits or bytes.
4. Set LST to indicate the last entry in the frame.

The TSA set-up is independent of the protocol in use; it simply routes programmed portions of the received data frame from the TDM pins to the MCC. See **Section 10.2**, *Connect the TDM Interface to T1/E1,* on page 10-6 for details on the TSA configuration.

	Loop/ echo	Super	MCSEL	CNT	BYT	LST
Channel 160 Rx entry	1	0	1 0 1 0 0 0 0 0	0 0 0	1	0
Channel 160 Tx entry	0	0	1 0 1 0 0 0 0 0	0 0 0	1	0
Channel 191 Rx entry	1	0	1 0 1 1 1 1 1 1	0 0 0	1	1
Channel 191 Tx entry	0	0	1 0 1 1 1 1 1 1	0 0 0	1	1

Figure 10-9. Serial Interface Entry Definitions for Driver Example

10.4.2 Set up Clocks, Baud Rate Generators (BRG), and Timers

The TDMB clocks (L1RCLKB, L1TCLKB) are always driven from an external source. In our example, they are configured to be driven from the CLK[15–16] pins (the CMX SI2 Clock Route Register [CMXSI2CR] is cleared to 0x0). The reference clock input to these pins is either from the T1/E1 framer, in external PHY loopback, or from the BRG in internal (SI or TDM) loopback. The driver implements internal loopback clock generation via BRG5. The BRG5 output is a fraction of the CPM BRGCLK clock and is determined by the division factors programmed in the BRG Configuration (BRGC) registers. To give a clock rate of ~2.048 MHz from a 150 MHz CPM, BRGC5 is set to 0x00010048. The BRG is set for normal operation, with no prescaler and a clock divider of 37. The clocks produced by the BRGs are sent to the bank-of-clocks logic, where they are either routed to the serial controllers or to the external pins, as in this case. The BRG50 pin must be externally connected to the CLK15 pin.

To provide the synchronization signals, the following CPM timer registers require configuration:

1. Timer Global Configuration Register (TGCR). TGCR must be configured prior to the TMR or erratic behavior can occur. Setting TGCR1 = 0x09 enables Timer1 with normal gate mode.
2. Timer Mode Register (TMR). Next, the TMR1 register is configured with the timer prescaler set to 1, the capture event disabled and the timer counter is reset immediately after the reference value is reached. The input source for the timer is TIN1 (fed by BRG5); therefore TIMR1 is set to 0x000E.

3. Timer Reference Register (TRR). Finally, the TRR1 register is set to contain the timeout reference value, resulting in a configuration of 0x00FF. The timer output TOUT1 must be externally connected to L1RSYNC.

10.5 Set Up the External Interface

The CPM interface is essentially a set of I/O pins that can be configured for either a peripheral or a general-purpose function. The multiplexed peripheral pins for TDMB are configured through the parallel I/O port registers (PPAR, PSOR, PDIR). The driver function InitParallelPorts() details the appropriate port registers assignment.

Note: All the CPM I/O pins default to general-purpose inputs.

Enabling the TDM involves configuring the serial interface registers. The SI Global Mode Register (SIGMR), which is the last register to be set up in the driver before the driver is started, defines the activation of the TDM channels for each SI. Because TDMB is used, this register is set at 0x02.

10.6 Related Reading

MSC8101 User's Guide (This manual)

Section 1.3.7, *Communications Processor Module (CPM),* on page 1-9

Section 1.3.7.3, *Buffer Descriptors,* on page 1-11

MSC8101 Reference Manual

Chapter 19, *Communications Processor Module Overview*

Chapter 20, *Serial Interface With Time-Slot Assigner*

Chapter 22, *Baud-Rate Generators*

Chapter 33, *Multi-Channel Controllers (MCCs)*

Serial Peripheral Interface (SPI) 11

The serial peripheral interface (SPI) is a synchronous serial data protocol that is standard across many Freescale processors and other SPI-compatible devices, including EEPROMs and analog-to-digital A/D converters. It is essentially a shift register that transmits and receives data serially to and from other devices with an SPI. The MSC8101 SPI includes the following features:

- Four-wire interface (SPIMOSI, SPIMISO, SPICLK, $\overline{\text{SPISEL}}$)
- Full-duplex operation
- Double-buffered receiver and transmitter
- Independent programmable baud-rate generator
- Master or slave mode
- Multi-master environment support

This chapter describes how the MSC8101 exchanges data between other devices via the SPI. It tells you how to configure port D for SPI operation and how to configure the SPI baud-rate generator in master mode. It gives examples of the SPI operating as a master and as a slave.

11.1 Configuring the SPI for Use

The first phase of SPI programming is to configure the MSC8101 to use the SPI signals on port D pins, select the SPI peripheral function for port D, and select the pin direction. **Table 11-1** describes the functions of the SPI signals.

Table 11-1. SPI Signals

Signal	Description
SPIMOSI	SPI Master Out Slave In When the SPI is configured as a master, SPIMOSI is the output signal that transmits data to the slave device. When the SPI is configured as a slave, SPIMOSI is the input signal that receives data from the master device.
SPIMISO	SPI Master In Slave Out When the SPI is configured as a master, SPIMISO is the input signal that receives data from the slave device. When the SPI is configured as a slave, SPIMISO is the output signal that transmits data to the master device.
SPICLK	SPI Clock When the SPI is configured as a master, SPICLK is the output signal that shifts received data in from SPIMISO and transmitted data out to SPIMOSI. When the SPI is configured as a slave, SPICLK is the input signal that shifts received data in from SPIMOSI and transmitted data out to SPIMISO.
SPISEL	SPI Select When the SPI is configured as a master, $\overline{\text{SPISEL}}$ should be disabled to prevent a multi-master error. When the SPI is configured as a slave, $\overline{\text{SPISEL}}$ is the input signal that the master device asserts to select the slave device.

The SPI signals are multiplexed with the port D pins, PD[16–19]. These pins can be configured as general-purpose pins or as dedicated peripheral pins. To configure PD[16–19] for SPI, the following registers must be initialized:

- *Port Pin Assignment Register D (PPARD)*. Assigns PD[16–19] as dedicated SPI peripheral signals:

Table 1-2.

Bit	Name	Value	Description
16	DD16	1	PD16 is assigned to SPIMISO
17	DD17	1	PD17 is assigned to SPIMOSI
18	DD18	1	PD18 is assigned to SPICLK
19	DD19	1	PD19 is assigned to $\overline{\text{SPISEL}}$

- *Special Options Register D (PSORD)*. Selects the SPI peripheral function by assigning PD[16–19] to use the option 2 function. The SO[16–19] bits of this register are all set to a value of one.
- *Data Direction Register D (PDIRD)*. Selects the pin direction, as follows:

Table 1-3.

PDIRD Bits	Name	Value	Description
16	DR16	0	SPIMISO is bidirectional.
17	DR17	0	SPIMOSI is bidirectional.
18	DR18	0	SPICLK is bidirectional.
19	DR19	0	$\overline{\text{SPISEL}}$ is input.

11.2 Setting the Clock

In the master mode, the baud rate is determined by the divide by 16 option and the prescale modulus. The SPI baud rate generator (SPI BRG) takes its input from BRGCLK and generates the SPICLK. The SPI BRG provides a divide-by 16 option and a prescale divider option. **Figure 11-1** shows the SPI BRG block diagram.

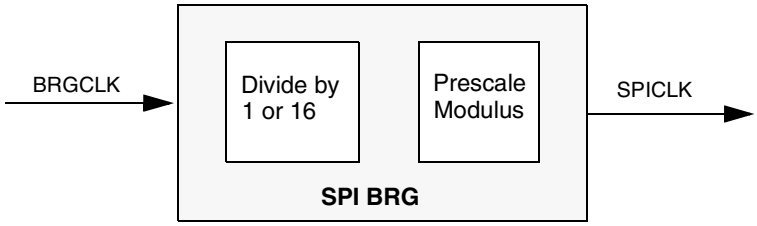


Figure 11-1. SPI BRG Block Diagram

Note: The BRGCLK is an internally derived signal generated from CLKIN. There is no separate BRGCLK input. See the *MSC8101 Technical Data* sheet for details.

To divide the BRGCLK input to the SPI BRG by 16, set the DIV16 bit in the SPI Mode Register (SPMODE). The SPMODE[DIV]16 bit description is as follows.

Table 1-4.

SPMODE Bit	Name	Description	
4	DIV16	Selects the clock source for the SPI BRG.	
		0	BRGCLK is input to the SPI BRG.
		1	BRGCLK/16 is input to the SPI BRG.

To divide the BRGCLK input to the SPI BRG by a value other than 16, configure the PM bits in the SPMODE register. The SPMODE[PM] bits select the prescale modulus for the SPI BRG. BRGCLK is divided by 4 × (PM[0–3])+1. The prescale modulus range is 4 to 64. The synchronous baud rate is calculated by dividing the BRGCLK input to the BRG by the DIV16 and PM options:

```
Sync Baud Rate = BRGCLK / [DIV16 × (4 × PM+1)]
```

For this calculation, use 1 or 16 for the meaning of DIV16, instead of 0 or 1.

11.3 Specifying the Receive and Transmit Buffer Descriptors

The SPI parameter table (see **Table 11-5**) can be placed at any 64-byte aligned address in Banks 1–8 in the dual-port RAM. The SPI parameters specify the receive and transmit buffer descriptors. You must initialize the parameters shown in bold-face. They should be changed only when the SPI is disabled. The remaining parameters are for communications processor (CP) use only, so you do not need to initialize them.

Table 11-5. SPI Parameter Table

IMM + 0x Value in 0x89FC	Name	Width	Description
0x00	**RBASE**	16-bits	RxBD/TxBD table base address Indicates where the BD tables begin in the dual-port RAM. BD tables can be placed in any unused portion of Banks 1–8. RBASE and TBASE must be initialized before the SPI is enabled. These values should be multiples of 8.
0x02	**TBASE**	16-bits	
0x04	**RFCR**	8-bits	Rx/Tx function code Contains the transaction specification associated with SDMA channel accesses to external memory.
0x05	**TFCR**	8-bits	
0x06	**MRBLR**	16-bits	Maximum receive buffer length Defines the maximum number of bytes the MSC8101 writes to a receive buffer before moving to the next buffer. The MSC8101 can write fewer bytes than MRBLR if an error or an end-of-frame occurs. It never writes more bytes than the MRBLR value. MRBLR should be changed only while the receiver is disabled.
0x08	RSTATE	32-bits	Rx internal state For CP use only.
0x0C	—	32-bits	Rx internal data pointer Updated by the SDMA channels to show the next address in the buffer to be accessed.
0x10	RBPTR	16-bits	Current RxBD pointer Points to the RxBD being processed or to the next BD to be serviced when idle. After reset or at the end of the RxBD table, the CP initializes RBPTR to RBASE.
0x12	—	16-bits	**Rx internal byte count**. Down-count value initialized with MRBLR and decremented with each byte written by the supporting SDMA channel.
0x14	—	32-bits	Rx temp For CP use only.
0x18	TSTATE	32-bits	Tx internal state For CP use only.
0x1C	—	32-bits	Tx internal data pointer Updated by the SDMA channels to show the next address in the buffer to be accessed.
0x20	TBPTR	16-bits	Current TxBD pointer Points to the TxBD being processed or to the next BD to be serviced when idle. After reset or at the end of the TxBD table, the CP initializes TBPTR to TBASE.

Table 11-5. SPI Parameter Table (Continued)

IMM + 0x Value in 0x89FC	Name	Width	Description
0x22	—	16-bits	**Tx internal byte count**. Down-count value initialized with `TxBD.bd_length` and decremented with each byte read by the supporting SDMA channel.
0x24	—	32-bits	**Tx temp**. For CP use only.
0x34	—	32-bits	SDMA temp.

Figure 11-2 shows an example of the memory structure of the SPI BDs and buffers. SPI_BASE points to the beginning of the parameter table at location `IMM + 0x89FC` in the dual-port RAM. For example, if the value at location `IMM + 0x89FC` were `0x3800`, then the SPI parameter table would begin at `IMM + 0x3800`. The RxBDs reside in the dual-port RAM, starting at the address in RBASE. The TxBDs also reside in the dual-port RAM, starting at the address in TBASE. For example, if one RxBD is followed by one TxBD, RBASE contains `0x0000`, and TBASE contains `0x0008`, then the RxBD is located at `IMM + 0x0000` and the TxBD is located at `IMM + 0x0008`. The data buffers can reside in the internal dual-port RAM. However, if the data buffers are large, they can reside in external memory. The receive buffer starts at the location to which `RxBD.bd_addr` points, and the transmit buffer starts at location to which `TxBD.bd_addr` points. `RxBD.bd_addr` and `TxBD.bd_addr` are the address fields in the receive and transmit buffer descriptors.

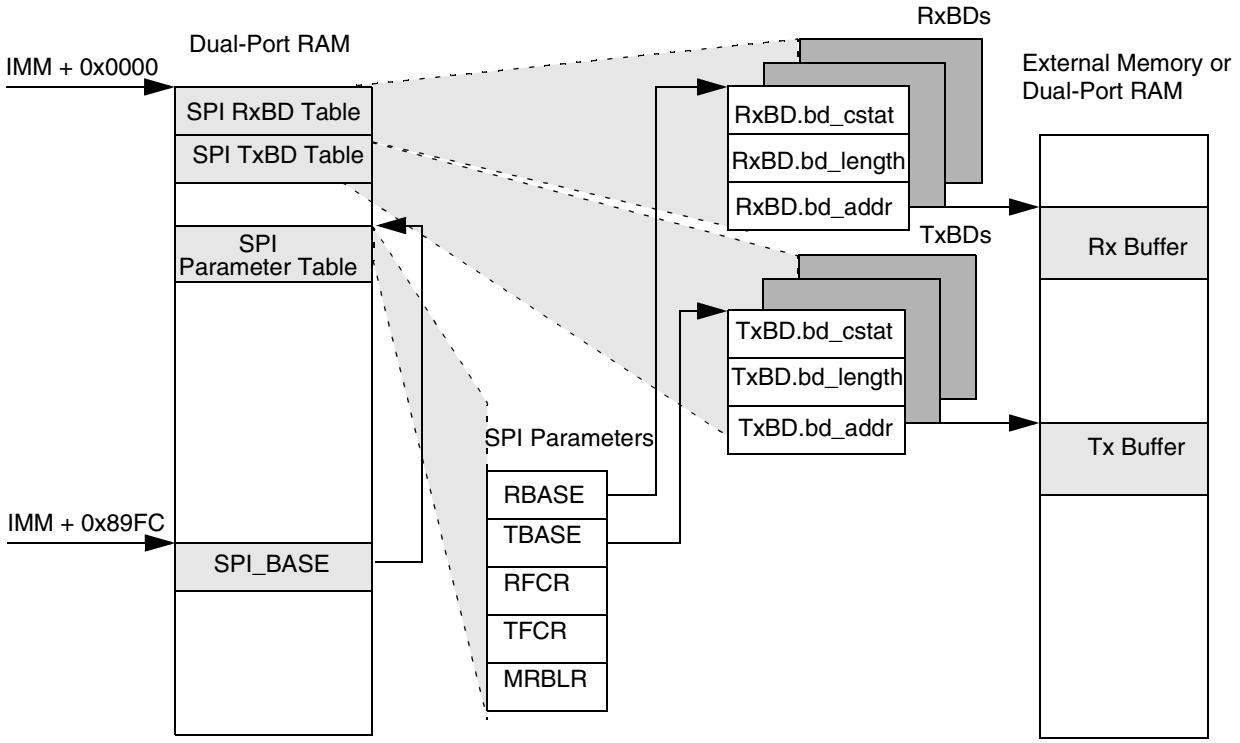


Figure 11-2. SPI BD and Buffer Memory Structure

11.4 Operating the SPI as a Master

The state diagram in **Figure 11-3** shows how the SPI transmits and receives characters as a master. The SPI is enabled by setting SPMODE[EN] = 1.

1. Once the SPI is enabled, the start of data transfer is enabled by setting SPCOM[STR] = 1.
2. TxBD[R] is set to indicate that the buffer is ready for transmission. As a master, the SPI generates clock pulses on SPICLK for each character and simultaneously shifts transmit data out on SPIMOSI and receive data in on SPIMISO. Received data is written into a receive buffer using the next available RxBD.
3. The SPI transmits and receives data until the entire buffer is sent or an error occurs. The CP clears TxBD[R] after the buffer is sent and clears RxBD[E] to indicate that the buffer is full.
4. If the current TxBD[L] is cleared, indicating that the current buffer does not contain the last character of the message, the next TxBD is processed after data from the current buffer is sent. The next TxBD[R] is set to indicate that the next buffer is ready for transmission.
5. If the current TxBD[L] is set, indicating that the current buffer contains the last character of the message, transmission stops after the current buffer is sent. The RxBD is closed after transmission stops even if the receive buffer is not full.
6. To resume transmission, SPCOM[STR] is set.
7. An error occurs when $\overline{\text{SPISEL}}$ is asserted while the SPI is master. To avoid this error, $\overline{\text{SPISEL}}$ must be disabled.
8. The SPI must be initialized after an error occurs by setting the SPCOM[STR] bit.

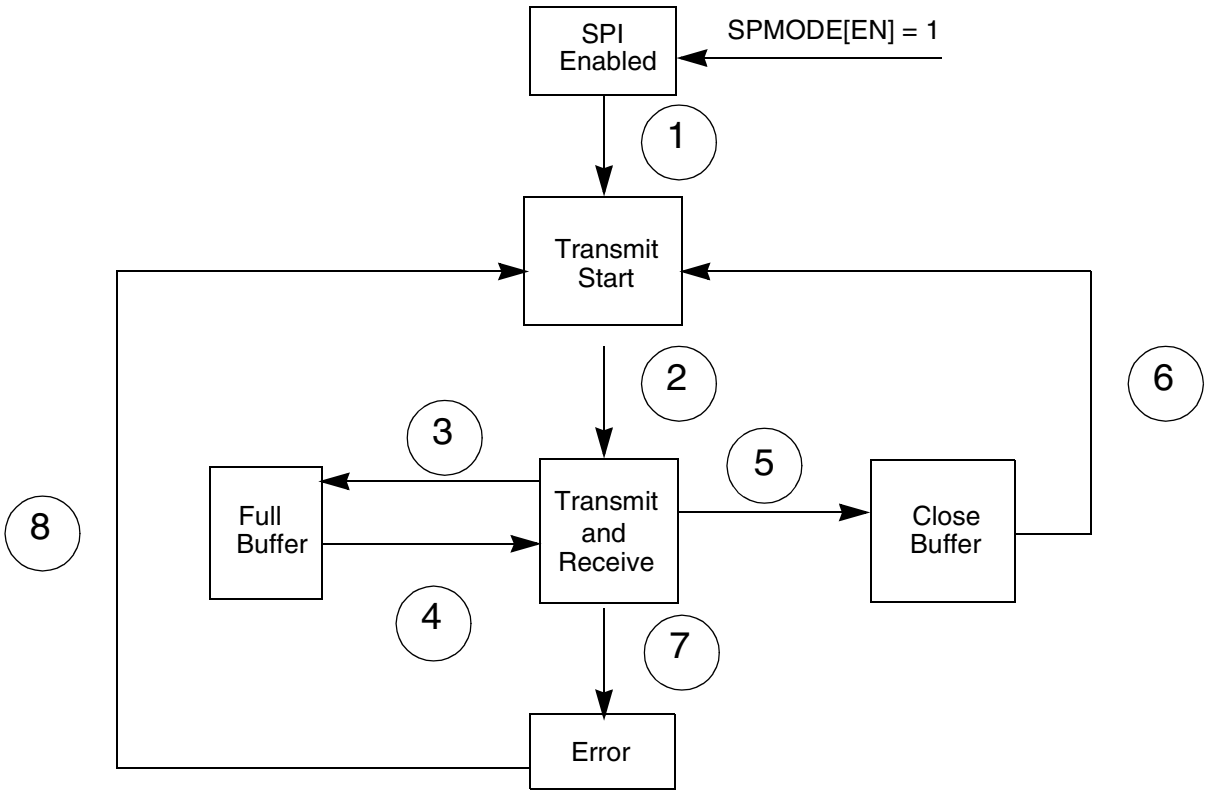


Figure 11-3. SPI as a Master

The following example shows the steps required to initialize the SPI as a master. The master SPI drives the SCLK signal. The assumptions underlying this example are:

- `IMM` is a pointer to the MSC8101 registers in the system bus address spaces and is initialized to `0xF0000000`.
- `SPIPRAM` is a pointer to the SPI parameter RAM.
- `RxTxBD` is a pointer to the buffer descriptors RxBD and TxBD.
- One RxBD and one TxBD are used.

The steps in initializing the SPI as a master are as follows:

1. Configure port D for SPI.

   In the master mode, $\overline{\text{SPISEL}}$ is disabled while SPIMOSI, SPIMISO, and SPICLK are enabled. $\overline{\text{SPISEL}}$ must be configured as a GPIO. The PPARD, PDIRD, and PSORD registers are configured as follows:

   ```
   IMM->io_regs[PORT_D].ppar = 0x0000E000;
   IMM->io_regs[PORT_D].pdir = 0x00001000;
   IMM->io_regs[PORT_D].psor = 0x0000F000;
   ```

2. Assign a pointer to the SPI parameter RAM.

   At location `IMM + 0x89FC`, SPI_BASE points to the SPI parameter RAM area, which can be placed at any 64-byte aligned address in the dual-port RAM's general-purpose

area (Banks 1–8 are located at IMM + 0x0 through IMM + 0x3FFF). In this example, assume that the SPI parameter RAM area resides in Bank 8 at IMM + 0x3800.

```
IMM->pram.standard.spi[0] = 0x38;
IMM->pram.standard.spi[1] = 0x00;
```

3. Configure RBASE and TBASE.

Assume that there is one RxBD followed by one TxBD at the beginning of the dual-port RAM and the RxBD starts at IMM+0. Since each buffer descriptor is 8-bytes, the TxBD starts at IMM+8.

```
SPIPRAM->rbase = 0x0000;
SPIPRAM->tbase = 0x0008;
```

4. Configure RFCR and TFCR.

```
SPIRAM->rfcr = 0x10;
SPIRAM->tfcr = 0x10;
```

5. Configure MRBLR.

Assume the maximum bytes per receive buffer is 16 bytes.

```
SPIRAM->mrblr = 0x0010;
```

a. Configure the RxBD and TxBD.

Since there is only one RxBD, it is the last BD in the table. Assume the buffer is empty and an interrupt is generated after the buffer is filled.

```
RxTxBD->RxBD[0].bd_cstatus = 0xB000;
```

Since there is only one TxBD, it is the last BD in the table. Assuming the buffer is ready, an interrupt is generated after the buffer is filled and the buffer contains the last character of the message.

```
RxTxBD->TxBD[0].bd_cstatus = 0xB800;
```

Assume that five 8-bit characters must be transmitted.

```
RxTxBD->TxBD[0].bd_length = 0x0005;
RxTxBD->RxBD[0].bd_length = 0x0000; (optional)
```

Assume that the receive buffer is located at IMM + 0x1000 and the transmit buffer is located at IMM + 0x2000.

```
RxTxBD->RxBD[0].bd_addr = 0xF0001000;
RxTxBD->TxBD[0].bd_addr = 0xF0002000;
```

b. Execute the INIT RX AND TX PARAMETERS opcode.

This opcode operates on the SPI sub-block code 10 to initialize the transmit and receive parameters.

```
IMM->cpm_cpcr = 0x25410000;
```

c. Clear any previous events.

Write a 1 to clear SPI Event Register (SPIE) bits to clear previous events.

```
IMM->spi_spie = 0xFF;
```

d. Enable all possible interrupts.

Setting SPI Mask Register (SPIM) bits enables the corresponding interrupt.

```
IMM->spi_spim = 0x37;
```

e. Configure the SPI Mode Register (SPMODE).

Assume normal operation, master mode, SPI enabled, 8-bits per character, and the fastest speed possible.

```
IMM->spi_spmode = 0x0370;
```

f. Start the transfer.

Setting the SPCOM[STR] bit causes the SPI to start transferring data to and from the Tx/Rx buffers.

```
IMM->spi_spcom = 0x80;
```

11.5 Operating the SPI as a Slave

The state diagram in **Figure 11-4** shows how the SPI transmits and receives characters as a master. The SPI is enabled by setting SPMODE[EN] = 1.

1. Once the SPI is enabled, the start of data transfer is enabled: SPCOM[STR] = 1.
2. TxBD[R] is set to indicate that the buffer is ready for transmission. Once $\overline{\text{SPISEL}}$ is asserted, the slave shifts data out on SPIMISO and shifts data in on SPIMOSI.
3. The SPI transmits and receives data until the entire buffer is sent.
4. If the current TxBD[L] is cleared, indicating that the current buffer does not contain the last character of the message, the next TxBD is processed after data from the current buffer is sent. The next TxBD[R] is set to indicate that the next buffer is ready for transmission.
5. If the current TxBD[L] is set, the current buffer contains the last character of the message and transmission stops after the current buffer is sent. Then the SPI sends ones as long as $\overline{\text{SPISEL}}$ remains asserted.
6. Transmission resumes when SPCOM[STR] is set.
7. Transmission stops when $\overline{\text{SPISEL}}$ is deasserted.
8. Transmission resumes when $\overline{\text{SPISEL}}$ is reasserted.

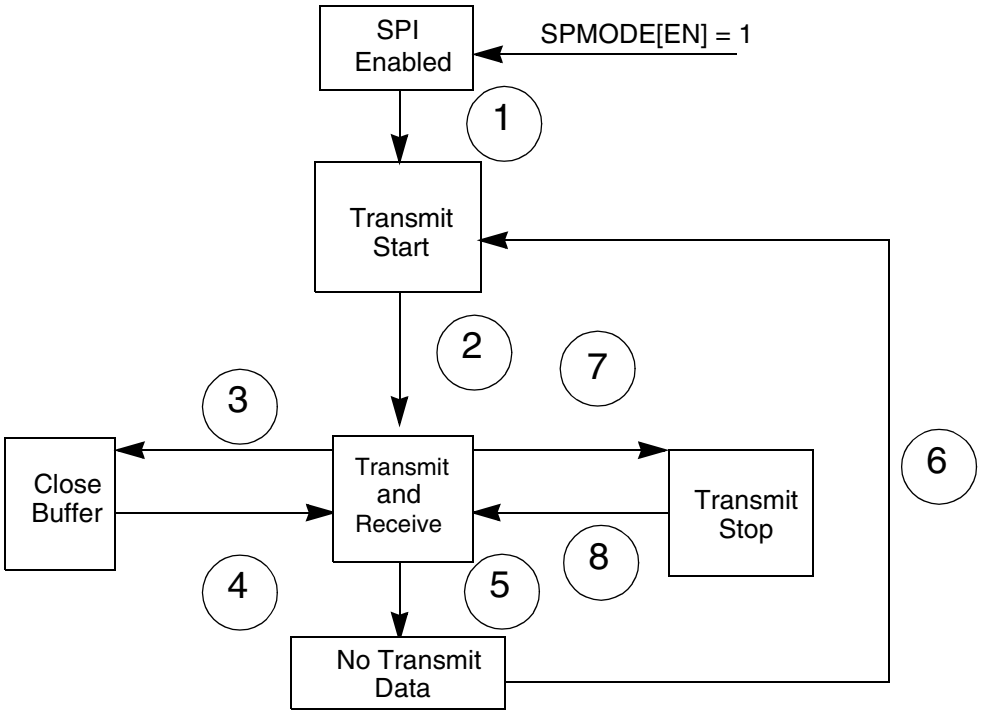


Figure 11-4. SPI as Slave

The following example shows the steps required to initialize the SPI as a slave. The assumptions underlying this example are:

- `IMM` is a pointer to the MSC8101 registers in the system bus and local bus address spaces.
- `SPIPRAM` is a pointer to the SPI parameter RAM.
- `RxTxBD` is a pointer to the buffer descriptors RxBD and TxBD.
- One RxBD and one TxBD are used.

The steps in initializing the SPI as a slave are as follows:

1. Configure Port D for SPI.

   In the slave mode, $\overline{\text{SPISEL}}$, SPIMOSI, SPIMISO, and SPICLK are enabled. The PPARD and PDIRD and PSORD are configured as follows:

   ```
   IMM->io_regs[PORT_D].ppar = 0x0000F000;
   IMM->io_regs[PORT_D].pdir = 0x00000000;
   IMM->io_regs[PORT_D].psor = 0x0000F000;
   ```

2. Assign a pointer to the SPI Parameter RAM.

   At location `IMM+0x89FC`, SPI_BASE points to the SPI Parameter RAM area which can be placed at any 64-byte aligned address in the dual-port RAM's general-purpose area (Banks 1-8 located at `IMM+0x0` through `IMM+0x3FFF`). In this example, assume that the SPI Parameter RAM area is located in Bank 8 at `IMM+0x3800`.

   ```
   IMM->pram.standard.spi[0] = 0x38;
   ```

```
IMM->pram.standard.spi[1] = 0x00;
```

3. Configure RBASE and TBASE.

   Assume that there is one RxBD followed by one TxBD at the beginning of the dual-port RAM and the RxBD starts at `IMM+0`. Since each buffer descriptor is 8-bytes, the TxBD starts at `IMM+8`.

```
SPIPRAM->rbase = 0x0000;
SPIPRAM->tbase = 0x0008;
```

4. Configure RFCR and TFCR.

   Assume big-endian byte ordering is used.

```
SPIRAM->rfcr = 0x10;
SPIRAM->tfcr = 0x10;
```

5. Configure MRBLR.

   Assume the maximum bytes per receive buffer is 16 bytes.

```
SPIRAM->mrblr = 0x0010;
```

6. Configure the RxBD and TxBD.

   Since there is only one RxBD, it is the last BD in the table. Assume the buffer is empty and an interrupt is generated after the buffer is filled.

```
RxTxBD->RxBD[0].bd_cstatus = 0xB000;
```

   Since there is only one TxBD, it is the last BD in the table. Assume the buffer is ready, an interrupt is generated after the buffer is filled and the buffer contains the last character of the message.

```
RxTxBD->TxBD[0].bd_cstatus = 0xB800;
```

   Assume five 8-bit characters need to be transmitted.

```
RxTxBD->TxBD[0].bd_length = 0x0005;
RxTxBD->RxBD[0].bd_length = 0x0000; (optional)
```

   Assume the receive buffer is located at `IMM+0x1000` and the transmit buffer is located at `IMM+0x2000`.

```
RxTxBD->RxBD[0].bd_addr = 0xF0001000;
RxTxBD->TxBD[0].bd_addr = 0xF0002000;
```

7. Execute the `INIT RX AND TX PARAMETERS` opcode.

   This opcode operates on the SPI sub-block code 10 and page 9 to initialize the transmit and receive parameters.

```
IMM->cpm_cpcr = 0x25410000;
```

8. Clear any previous events.

   Write a 1 to clear bits to clear previous events.

```
IMM->spi_spie = 0xFF;
```

9. Enable all possible interrupts.

   Setting SPIM bits enables the corresponding interrupt.

```
IMM->spi_spim = 0x37;
```

10. Configure SPMODE.

Assume normal operation, slave mode, SPI enabled, 8-bits per character, and the fastest speed possible.

```
IMM->spi_spmode = 0x0170;
```

11. Start the data transfer to and from the Tx/Rx buffers by setting the SPCOM[STR] bit.

```
IMM->spi_spcom = 0x80;
```

11.6 Responding to a Multi-master Error

A multi-master error occurs when the $\overline{\text{SPISEL}}$ pin is asserted while the SPI is configured as a master because more than one SPI device is a bus master. To avoid this error, the $\overline{\text{SPISEL}}$ must be disabled as shown in **Section 11.4**, *Operating the SPI as a Master,* on page 11-6. **Figure 11-5** shows how the SPI responds to a multi-master error. When the SPI is operating as a master and the $\overline{\text{SPISEL}}$ is asserted, the SPI sets the SPIE[MME] bit to indicate that a multi-master error has occurred. Then a maskable interrupt is issued to the SC140 core. The SPI operation and output drivers are disabled. The SC140 core must clear SPMODE[EN] before the SPI is used again. The SPIE[MME] bit must be cleared by writing a 1 before the SPI can be re-enabled.

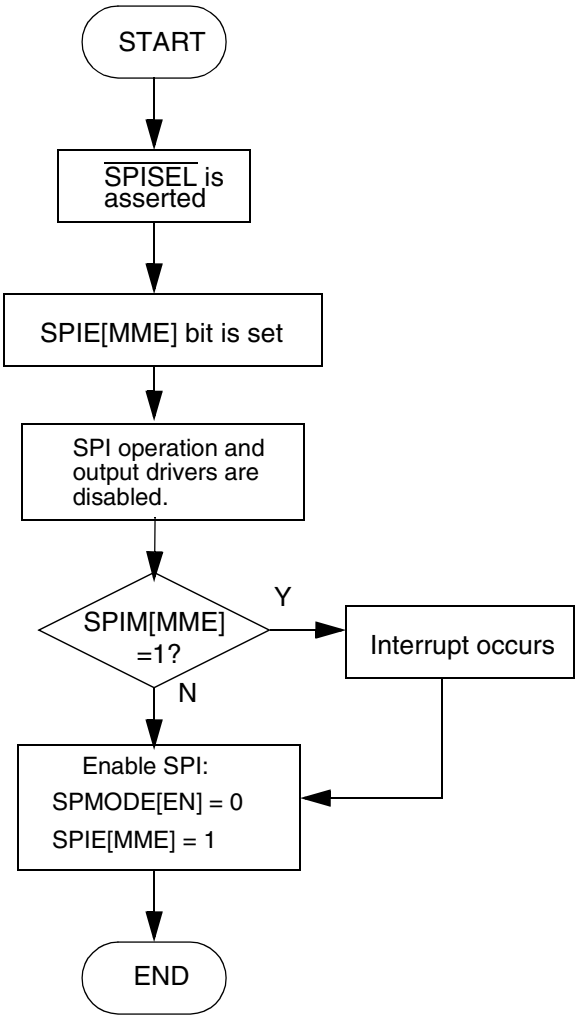


Figure 11-5. SPI Response to Multi-Master Error

11.7 Related Reading

MSC8101 User's Guide (This manual)

Section 1.3.7, *Communications Processor Module (CPM),* on page 1-9

MSC8101 Reference Manual

Chapter 7, *Clocks*

Chapter 19, Communications Processor Module Overview

Chapter 22, *Baud-Rate Generators (BRGs)*

Chapter 39, *Serial Peripheral Interface (SPI)*

System Debugging 12

This chapter presents examples of how the EOnCE port can be used for system-level debugging of real-time systems. The following examples are presented:

- Reading/writing EOnCE registers through JTAG
- Executing a single instruction through JTAG
- Writing to the EOnCE Receive Register (ERCV)
- Reading from the EOnCE Transmit Register (ETRSMT)
- Downloading software
- Reading/writing the trace buffer
- Using the EOnCE to perform profiling functions

12.1 EOnCE/JTAG Basics

In order to access the EOnCE through JTAG, you need to know about the JTAG scan paths, the JTAG instructions, the EOnCE control register value, and the CORE_CMD value. This section gives you these basics, starting with the scan paths.

The host controller transitions from one Test Access Port (TAP) controller state to another by taking one of the following scan paths:

- *Select-IR JTAG scan path*. Used when the host sends the JTAG instructions shown in **Table 12-2** to the MSC8101.
- *Select-DR JTAG scan path*. Used when the host sends data to the MSC8101 or receives status information from the MSC8101.

Figure 12-1 shows the TAP controller state machine, and **Table 12-1** shows the states associated with each scan path. The Test Mode Select (TMS) pin determines whether an instruction register scan or a data register scan is performed. At power-up or during normal operation of the host, the TAP is forced into the Test-Logic-Reset state by driving TMS high for five or more Test Clock (TCK) cycles. When test access is required, TMS is set low to cause the TAP to exit the Test-Logic-Reset and move through the appropriate states. From the Run-Test/Idle state, an instruction register scan or a data register scan can be issued to transition through the appropriate states. The first action that occurs when either block is entered is a Capture operation. The Capture-DR state captures the data into the selected serial data path, and the Capture-IR state

captures status information into the instruction register. The Exit state follows the Shift state when shifting of instructions or data is complete. The Shift and Exit states follow the Capture state so that test data or status information can be shifted out and new data shifted in. Latches in the selected scan path hold their present state during the Capture and Shift operations. The Update state causes the latches to update with the new data that is shifted into the selected scan path.

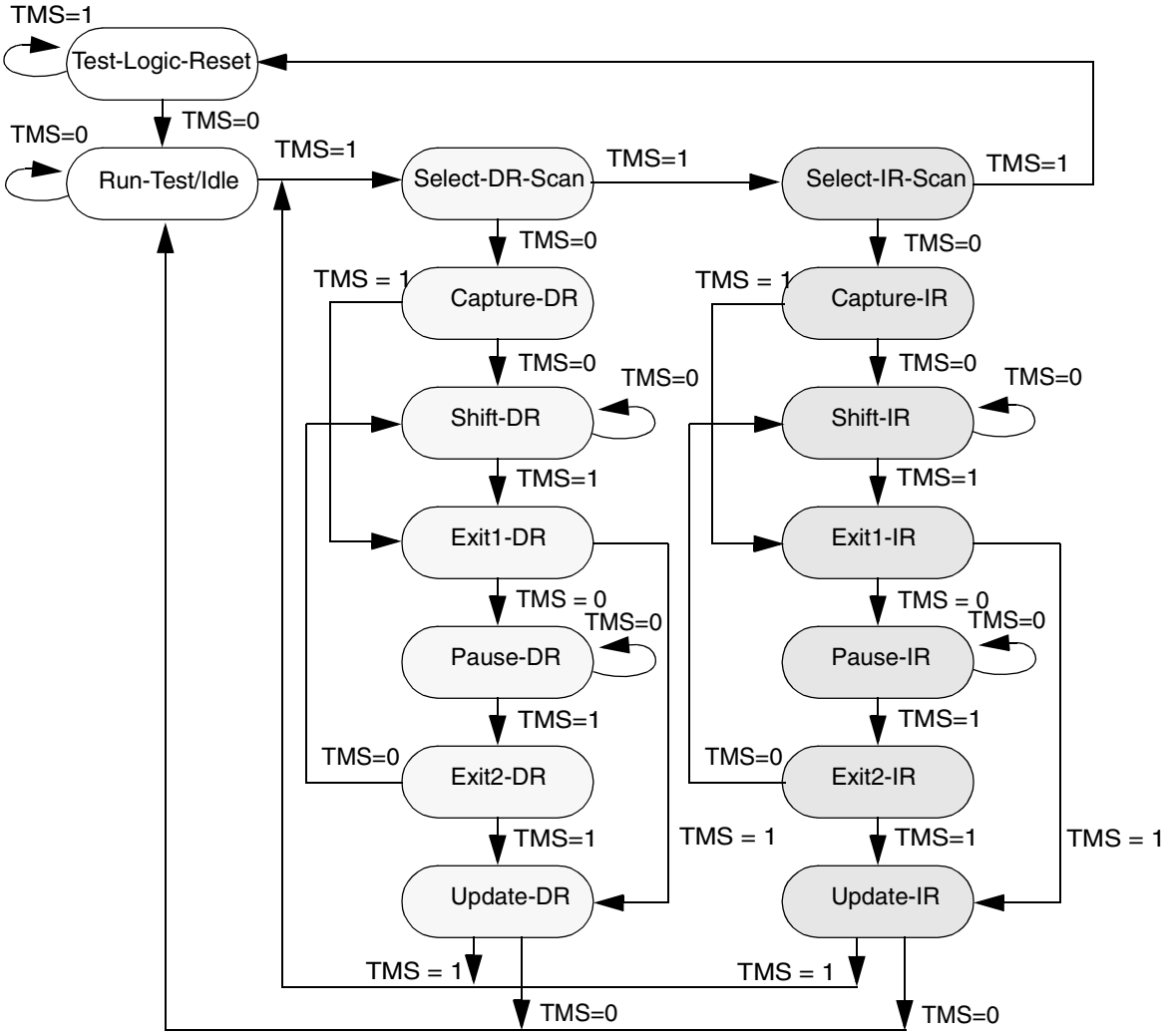


Figure 12-1. TAP Controller State Machine

Table 12-1. JTAG Scan Paths

Select-DR Scan Path	Select-IR Scan Path
Select-DR_SCAN	Select-IR_SCAN
Capture-DR	Capture-IR
Shift-DR	Shift-IR

Table 12-1. JTAG Scan Paths

Select-DR Scan Path	Select-IR Scan Path
Exit1-DR	Exit1-IR
Update-DR	Update-IR

12.1.1 Instructions

The host sends JTAG instructions to the MSC8101 least significant bit first. As **Figure 12-2** shows, the TDI pin inputs the instruction into the MSC8101 and is sampled on the rising edge of TCK.

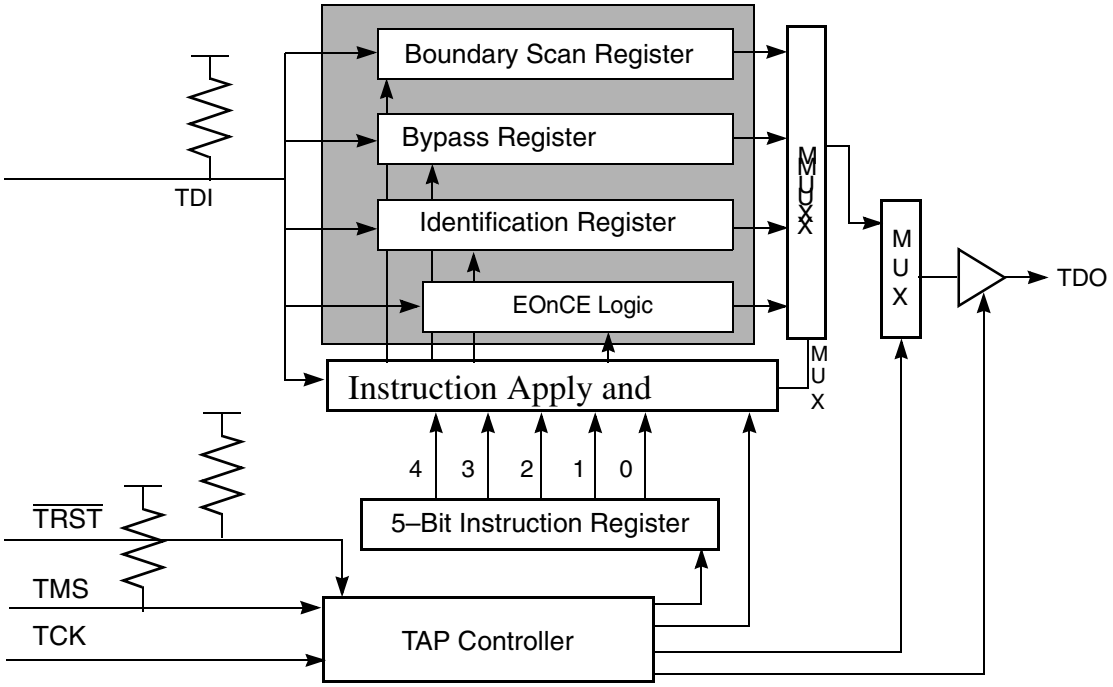


Figure 12-2. Test Logic Diagram Showing the Five-Bit Instruction Register

Table 12-2 describes the JTAG instructions and lists the bit values of the five-bit instruction register for each instruction (B0–B4, with B0 as the least significant bit). In the MSC8101, there is only one EOnCE module.

Table 12-2. JTAG Instructions

B4	B3	B2	B1	B0	Instruction	Description
0	0	0	0	0	EXTEST	Selects the Boundary Scan Register. Forces a predictable internal state while performing external boundary scan operations.
0	0	0	0	1	SAMPLE/PRELOAD	Selects the Boundary Scan Register. Provides a snapshot of system data and control signals on the rising edge of TCK in the Capture-DR controller state. Initializes the BSR output cells prior to selection of EXTEST or CLAMP.
0	0	0	1	0	IDCODE	Selects the ID Register. Allows the manufacturer, part number and version of a component to be identified.
0	0	0	1	1	CLAMP	Selects the Bypass Register. Allows signals driven from the component pins to be determined from the Boundary Scan Register.
0	0	1	0	0	HIGHZ	Selects the Bypass Register. Disables all device output drivers and forces the output to high impedance (tri-state) as per the IEEE specification.
0	0	1	1	0	ENABLE_EONCE	Selects the EOnCE registers. Allows you to perform system debug functions. Before this instruction is selected, the CHOOSE_EONCE instruction is activated to define which EOnCE is going to be activated.
0	0	1	1	1	DEBUG_REQUEST	Selects the EOnCE registers. Forces the MSC8101 into the debug mode of operation. In addition, ENABLE_EONCE is active to allow system debug functions to be performed. Before this instruction is selected, the CHOOSE_EONCE instruction is activated to define which EOnCE is going to request debug mode in a system with multiple MSC8101 devices.
0	1	0	0	0	RUNBIST	Selects the BIST registers. Allows you to generate a built-in self-test for checking the system circuitry.
0	1	0	0	1	CHOOSE_EONCE	Selects the EOnCE registers. Allows you to operate multiple devices. This instruction is activated before the ENABLE_EONCE and DEBUG_REQUEST instructions.
0	1	1	0	0	ENABLE_SCAN	Selects the DFT registers. Allows the DFT chain registers to be loaded by a known value or examined in the Shift_DR controller state.
0	1	1	0	1	LOAD_GPR	Allows the component manufacturer to gain access to test features of the device.
0	1	1	1	0	LOAD_SPR	Allows the component manufacturer to gain access to test features of the device.
1	1	1	1	1	BYPASS	Selects the Bypass register. Creates a shift register path from TDI to the Bypass Register and to TDO. Enhances test efficiency when a component other than the MSC8101 becomes the device under test.

12.1.2 Executing a JTAG Instruction

This section presents an example of how the host takes the instruction register scan path to send the JTAG instruction `DEBUG_REQUEST` (`00111`) to the MSC8101. JTAG instructions are sent

least significant bit first on TDI. If the TAP controller is in the Run-Test/Idle state, DEBUG_REQUEST is issued via JTAG, as shown in **Figure 12-3**.

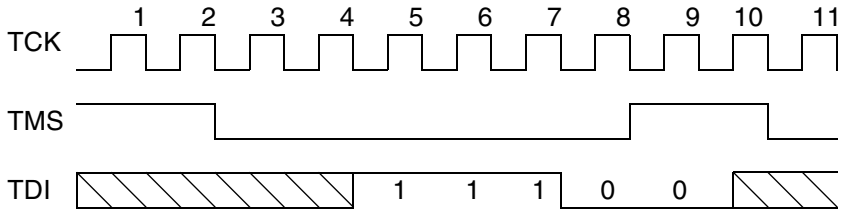


Figure 12-3. Executing DEBUG_REQUEST

The following sequence occurs at the rising edge of each TCK cycle:

1. TMS = 1 to enter the Select-DR state.
2. TMS = 1 to enter the Select-IR state.
3. TMS = 0 to enter the Capture-IR state.
4. TMS = 0 to enter the Shift-IR state.
5. TMS = 0 to stay in the Shift-IR state and TDI = 1.
6. TMS = 0 to stay in the Shift-IR state and TDI = 1.
7. TMS = 0 to stay in the Shift-IR state and TDI = 1.
8. TMS = 0 to stay in the Shift-IR state and TDI = 0.
9. TMS = 1 to enter the Exit1-IR state and TDI = 0.
10. TMS = 1 to enter the Update-IR state.
11. TMS = 0 to return to the Run-Test/Idle state.

12.1.3 Registers

Two registers of special concern to the EOnCE/JTAG programmer are the EOnCE Control Register (ECR) and the Core Command Register (CORE_CMD). The 16-bit write-only ECR receives its serial data from the TDI input signal. It is accessible only via JTAG. The host writes to the ECR to specify the direction of the data transfer with the selected register and additional control bits. **Table 12-3** shows the ECR bit definitions, and **Table 12-4** summarizes the EOnCE registers, which are either a source or destination for a read or write operation.

Table 12-3. EOnce Control Register (ECR) Bits

Name	Description	Settings
15–10	Reserved. Write to zero for future compatibility.	

Table 12-3. EOnce Control Register (ECR) Bits

Name		Description	Settings
9	R/W	Specifies the direction of a data transfer.	0 Write the data into the register specified by REGSEL
			1 Read the data in the register specified by REGSEL
8	GO	An instruction written to the CORE_CMD register executes and the core remains in Debug mode unless the EX bit is set. If EX is set, the system exits Debug mode after the instruction executes. When a register other than the CORE_CMD register is written or read, the next instruction in the pipeline executes.	0 Inactive
			1 Execute one instruction
7	EX	When EX is set, the SC140 core leaves Debug mode and resumes normal operation after executing the read or write command.	0 Remain in debug mode
			1 Exit debug mode
6–0	REGSEL	Defines which register is the source or destination for the read or write operation.	See **Table 12-4** for a the EOnCE registers' address offsets.

All the EOnCE registers are accessible from the core and are memory-mapped. Therefore, each register has its own address in the memory space. The memory address of an EOnCE register is defined by adding four times the register address offset from the address offset shown in **Table 12-4** to the EOnCE register base address defined for each SOC derivative. The address offset is from the EOnCE base address 0x00EFFE00. For example, the memory address for the LSB part of register ERCV is `58 + rba_via`, where `rba_via` is the derivative dependent register base address. For details, consult the *SC140 DSP Core Reference Manual*.

Table 12-4. EOnce Register Summary

Address Offset	Mnemonic	Register	Width
00	ESR	EOnCE Status Register	32
01	EMCR	EOnCE Monitor and Control Register	32
02	ERCV	EOnCE Receive Register LSB	64
03		EOnCE Receive Register MSB	
04	ETRSMT	EOnCE Transmit Register LSB	64
05		EOnCE Transmit Register MSB	
06	EE_CTRL	EOnCE Pins Control Register	16
07	PC_EXCP	Exception PC Register	32
08	PC_NEXT	PC of next execution set	32
09	PC_LAST	PC of last execution set	32
0A	PC_DETECT	PC Breakpoint Detection Register	32
...	Reserved		

Table 12-4. EOnce Register Summary (Continued)

Address Offset	Mnemonic	Register	Width
10	EDCA0_CTRL	EDCA 0 Control Register	16
11	EDCA1_CTRL	EDCA 1 Control Register	16
12	EDCA2_CTRL	EDCA 2 Control Register	16
13	EDCA3_CTRL	EDCA 3 Control Register	16
14	EDCA4_CTRL	EDCA 4 Control Register	16
15	EDCA5_CTRL	EDCA 5 Control Register	16
...	Reserved		
18	EDCA0_REFA	EDCA 0 Reference Value A	32
19	EDCA1_REFA	EDCA 1 Reference Value A	32
1A	EDCA2_REFA	EDCA 2 Reference Value A	32
1B	EDCA3_REFA	EDCA 3 Reference Value A	32
1C	EDCA4_REFA	EDCA 4 Reference Value A	32
1D	EDCA5_REFA	EDCA 5 Reference Value A	32
...	Reserved		
20	EDCA0_REFB	EDCA 0 Reference Value B	32
21	EDCA0_REFB	EDCA 0 Reference Value B	32
22	EDCA0_REFB	EDCA 0 Reference Value B	32
23	EDCA0_REFB	EDCA 0 Reference Value B	32
24	EDCA0_REFB	EDCA 0 Reference Value B	32
25	EDCA0_REFB	EDCA 0 Reference Value B	32
...	Reserved		
30	EDCA0_MASK	EDCA 0 Mask Register	32
31	EDCA0_MASK	EDCA 1 Mask Register	32
32	EDCA0_MASK	EDCA 2 Mask Register	32
33	EDCA0_MASK	EDCA 3 Mask Register	32
34	EDCA0_MASK	EDCA 4 Mask Register	32
35	EDCA0_MASK	EDCA 5 Mask Register	32
...	Reserved		
38	EDCD_CTRL	EDCD Control Register	16
39	EDCD_REF	EDCD Reference Value	32
3A	EDCD_MASK	EDCD Mask Register	32
...	Reserved		
40	ECNT_CNTRL	Counter Control Register	16
41	ECNT_VAL	Counter Value Register	32
42	ECNT_EXT	Extension Counter Value	32
...	Reserved		

Table 12-4. EOnce Register Summary (Continued)

Address Offset	Mnemonic	Register	Width
48	ESEL_CTRL	Selector Control Register	8
49	ESEL_DM	Selector DM Mask	16
4A	ESEL_DI	Selector DI Mask	16
4B	Reserved		
4C	ESEL_ETB	Selector Enable TB Mask	16
4D	ESEL_DTB	Selector Disable TB Mask	16
...	Reserved		
50	TB_CTRL	Trace Buffer Control Register	8
51	TB_RD	Trace Buffer Read Pointer	16
52	TB_WR	Trace Buffer Write Pointer	16
53	TB_BUFF	Trace Buffer	32
...	Reserved		
7E	CORE_CMD	Core Command Register	48
7F	NOREG	No register selected	

The external host writes the instruction to be executed by the SC140 core into the CORE_CMD register. The SC140 core executes the instruction without leaving Debug mode unless ECR[EXIT] = 1, in which case the SC140 core exits Debug mode after the instruction executes. **Figure 12-4** shows the instruction format of the 48-bit CORE_CMD register. The bits in this format are defined as follows:

- *Word Length* (bits 1–0). Specify the instruction word length. Valid CORE_CMD word lengths are one, two, or three words, as shown here:

Table 12-5.

Word Length Bits		Description
0	0	Not supported
0	1	One-word instruction
1	0	Two-word instruction
1	1	Three-word instruction

- *Prefix Bits* (bits 3–2). Derive from bits 5 and 7 of the Prefix1 word. The CORE_CMD register does not use the other bits of the Prefix1 word.
- *Opcode* (bits 19–4). Derive from the instruction opcode. These bits are reversed in order from the instruction opcode value. That is, bits 15–0 of the instruction opcode are reversed as bits 0–15 of the CORE_CMD register.

- *Immediate A* (bits 33–20). Derive from the instruction immediate A value. These bits are reversed in order from the instruction immediate A value. The two most significant bits of the instruction immediate A value are not used. Therefore, bits 15–0 of the instruction immediate A are reversed as bits 0–13 of the CORE_CMD register.
- *Immediate B* (bits 47–34). Derive from the instruction immediate B value. These bits are reversed in order from the instruction immediate B value. The two most significant bits of the instruction immediate B value are not used. Therefore, bits 15–0 of the instruction immediate B are reversed as bits 0–13 of the CORE_CMD register.

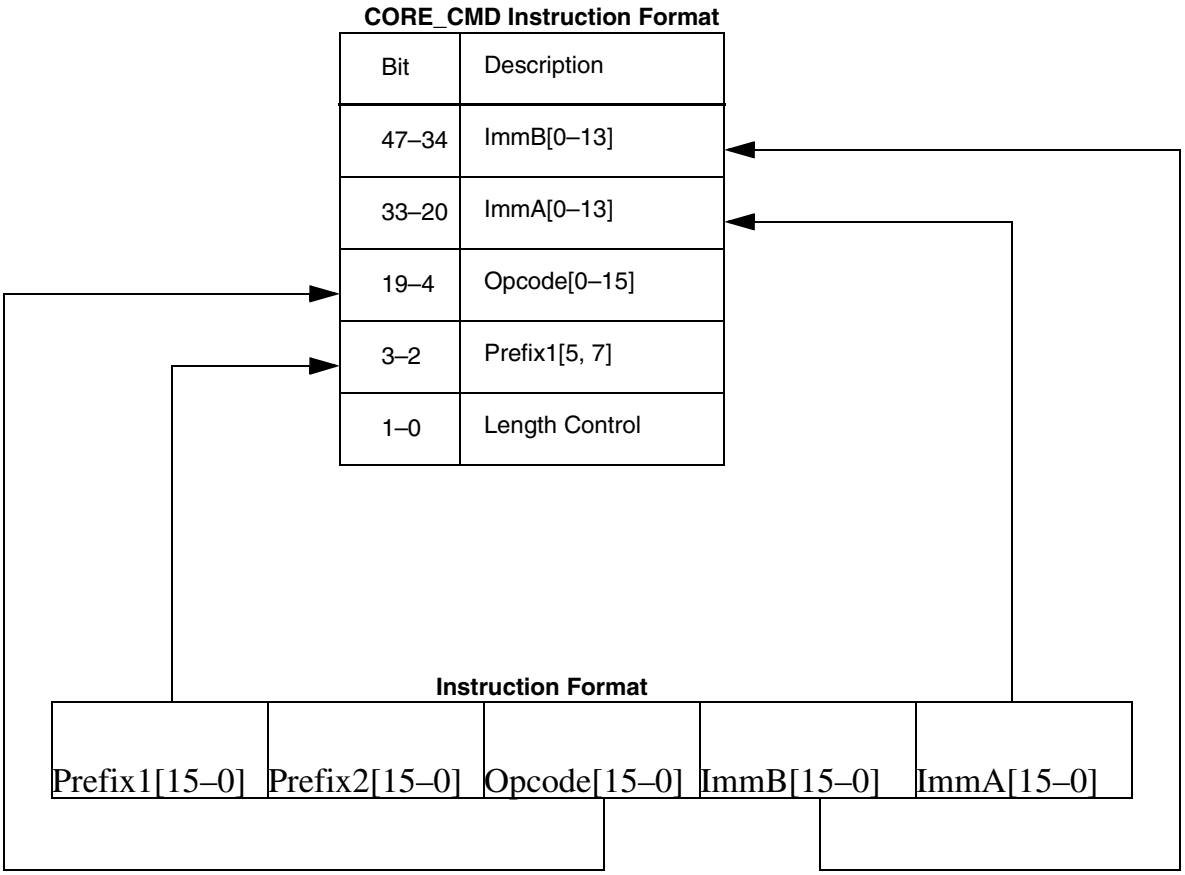


Figure 12-4. CORE_CMD Instruction Format

12.1.3.1 CORE_CMD Example 1

Instruction: `move.l #0xdead,d0`

Opcode: `0x30C0 3EAD 8000`

CORE_CMD: `0x0002 D5F0 30C3`

Table 12-6.

ImmA	ImmB	Opcode	Prefix1[5, 7]	Length
0x8000	0x3EAD	0x30C0		3 words
ImmA[15:0] 1000 0000 0000 0000	ImmB[15–0] 0011 1110 1010 1101	Opcode[15–0] 0011 0000 1100 0000		
ImmA[0:13] **0000 0000 0000 00**	ImmB[0–13] **1011 0101 0111 11**	Opcode[0–15] **0000 0011 0000 1100**	**00**	**11**
Note: The 48-bit CORE_CMD register is the concatenation of the bits in boldface.				

12.1.3.2 CORE_CMD Example 2

Instruction: `move.l (r1)+,d1`

Opcode: `0x5199`

CORE_CMD: `0x0000 0009 98A1`

Table 12-7.

ImmA	ImmB	Opcode	Prefix1[5, 7]	Length
		0x5199		1 word
		Opcode[15–0] 0101 0001 1001 1001		
ImmA[0–13] **0000 0000 0000 00**	ImmB[0–13] **0000 0000 0000 00**	Opcode[0–15] **1001 1001 1000 1010**	**00**	**01**
Note: The 48-bit CORE_CMD register is the concatenation of the bits in boldface.				

12.1.3.3 CORE_CMD Example 3

Instruction: `move.2l d0:d1,(r0)+`

Opcode: `0xC018`

CORE_CMD: `0x0000 0001 8031`

Table 12-8.

ImmA	ImmB	Opcode	Prefix1[5, 7]	Length
		0xC018		1 word
		Opcode[15–0] 1100 0000 0001 1000		
ImmA[0–13] **0000 0000 0000 00**	ImmB[0–13] **0000 0000 0000 00**	Opcode[0–15] 0001 1000 0000 0011	**00**	**01**
Note: The 48-bit CORE_CMD register is the concatenation of the bits in boldface.				

12.1.3.4 CORE_CMD Example 4

Instruction: `move.l #$c0ffee,d8`

Opcode: `0x3820 A000 30E0 3FEE 80C0`

CORE_CMD: `0x0301 DFF0 70CB`

Table 12-9.

ImmA	ImmB	Opcode	Prefix1	Length
0x80C0	0x3FEE	0x30E0	0x3820	3 words
ImmA[15–0] 1000 0000 1100 0000	ImmB[15–0] 0011 1111 1110 1110	Opcode[15–0] 0011 0000 1110 0000	Prefix1[5] 1 Prefix1[7] 0	
ImmA[0–13] **0000 0011 0000 00**	ImmB[0–13] **0111 0111 1111 11**	Opcode[0–15] **0000 0111 0000 1100**	Prefix1[5, [7] **10**	**11**
Note: The 48-bit CORE_CMD register is the concatenation of the bits in boldface.				

12.2 Writing EOnCE Registers Through JTAG

This section presents an example of how the host writes to the SC140 core 32-bit Event Counter Value Register (ECNT_VAL) via JTAG. This example shows how an EOnCE register can be written via JTAG. This general procedure applies to writing all the writable EOnCE registers:

1. Select-IR: `CHOOSE_EONCE` instruction to select EOnCE device.
2. Select-DR: '`1`' since the MSC8101 has only one EOnCE device.
3. Select-IR: `ENABLE_EONCE` instruction to allow you to perform system debug functions.
4. Select-DR: Write `0x0041` into the ECR to perform the following operation:

   ECR[R/W] = `0` to perform a write access.

   ECR[GO] = `0` to remain inactive.

   ECR[REGSEL] = `1000001` to select the ECNT_VAL register.
5. Select-DR: Write the 32-bit ECNT_VAL data on TDI.

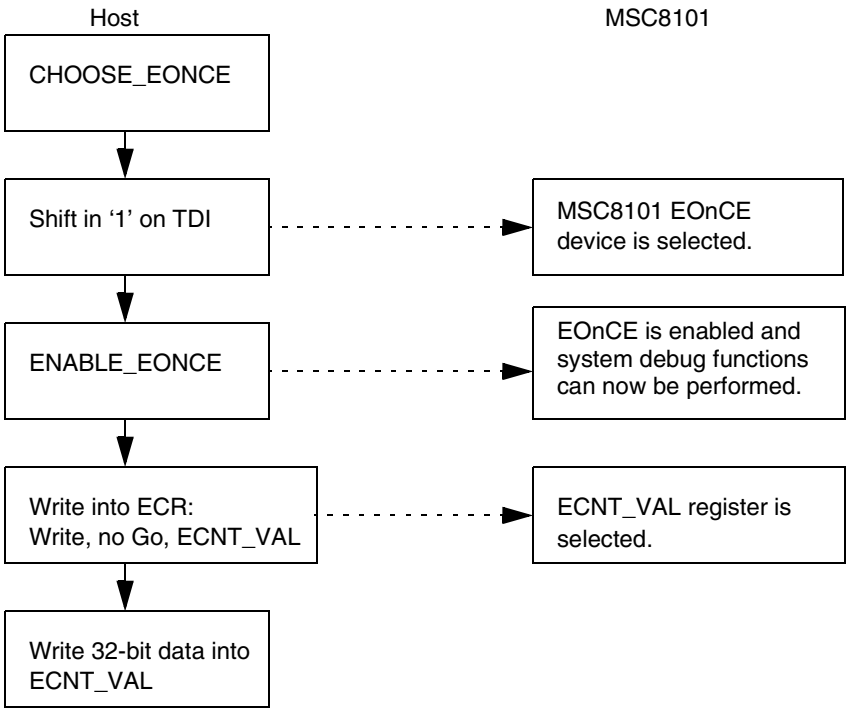


Figure 12-5. Writing EOnCE Registers

12.3 Reading EOnCE Registers Through JTAG

This section presents an example of the host reads from the SC140 core 32-bit Event Counter Value Register (ECNT_VAL) via JTAG. This example shows how an EOnCE register is read through JTAG. This general procedure applies to reading all the readable EOnCE registers:

1. Select-IR: `CHOOSE_EONCE` instruction to select EOnCE device.
2. Select-DR: '`1`' since the MSC8101 has only one EOnCE device.
3. Select-IR: `ENABLE_EONCE` instruction to allow you to perform system debug functions.
4. Select-DR: Write `0x0241` into the ECR to perform the following operation:

   ECR[R/W] = `1` to perform a read access.

   ECR[GO] = `0` to remain inactive.

   ECR[REGSEL] = `1000001` to select the ECNT_VAL register.
5. Select-DR: Read the 32-bit ECNT_VAL data on TDO.

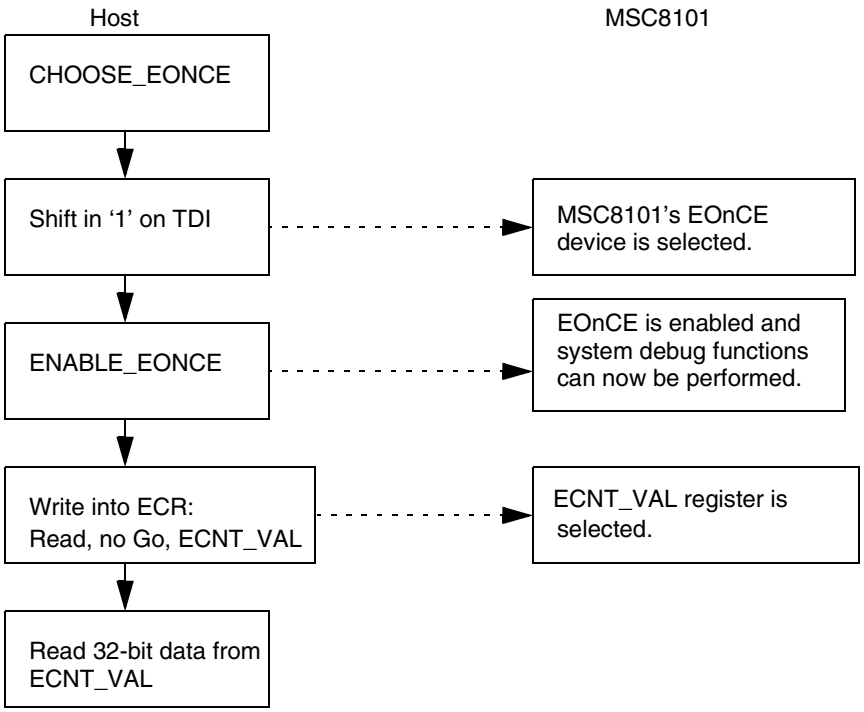


Figure 12-6. Reading EOnCE Registers

12.4 Executing a Single Instruction Through JTAG

This section presents an example of how the host writes an instruction into the CORE_CMD register for the SC140 core to execute.

1. Select-IR: `CHOOSE_EONCE` instruction to select EOnCE device.
2. Select-DR: '`1`' since the MSC8101 has only one EOnCE device.
3. Select-IR: `DEBUG_REQUEST` instruction to generate a debug request to the MSC8101 and to allow the user to perform system debug functions.
4. Select-DR: Write `0x017E` into the ECR to perform the following operation:

   ECR[R/W] = `0` to perform a write access.

   ECR[GO] = `1` to execute the instruction.

   ECR[REGSEL] = `111110` to select the CORE_CMD register.
5. Select-DR: Write 48-bit CORE_CMD data to move `0xdead` into data register d0.

   ```
   move.l #$dead,d0
   ```

   The CORE_CMD value is `0x0002 D5F0 30C3`. See **Section 12.1.3.1**, *CORE_CMD Example 1*, on page 12-9 for information on calculating this value.

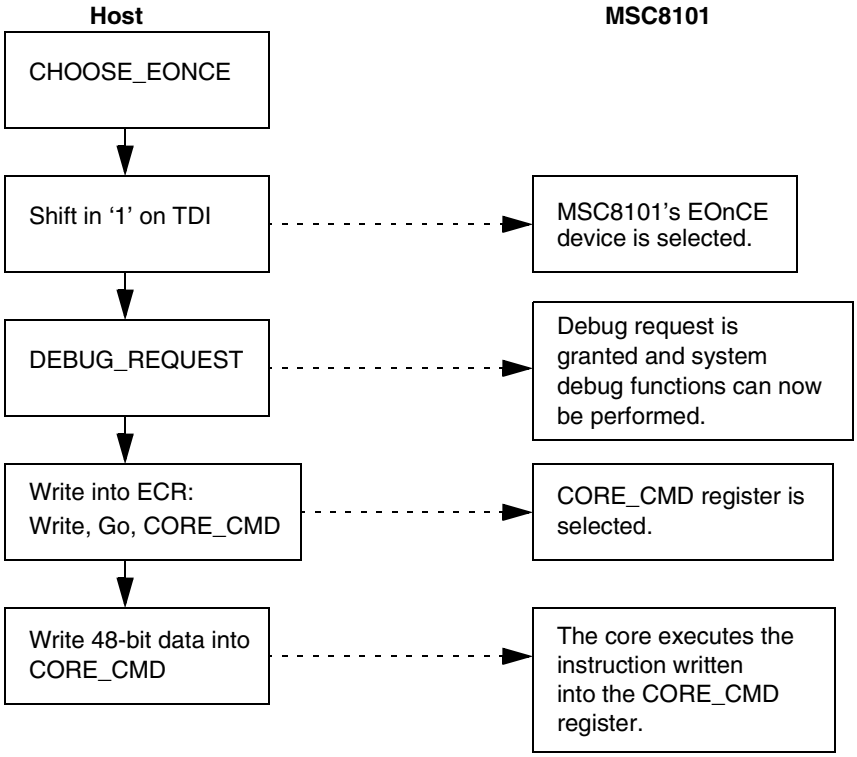


Figure 12-7. Executing a Single Instruction Through JTAG

12.5 Writing to the EOnCE Receive Register (ERCV)

This section presents an example of how to write to the ERCV register through JTAG.

1. Select-IR: `CHOOSE_EONCE` instruction to select EOnCE device.
2. Select-DR: '`1`' since the MSC8101 has only one EOnCE device.
3. Select-IR: `ENABLE_EONCE` instruction to enable the EOnCE registers.
4. Select-DR: Write `0x0002` into the ECR to perform the following operation:

   ECR[R/W] = `0` to perform a write access.

   ECR[GO] = `0` to remain inactive.

   ECR[REGSEL] = `0000010` to select the ERCV register.
5. Select-DR: Write the 64-bit ERCV data on TDI. After the most significant bit of the ERCV is written, the ESR[RCV] bit is set to indicate that the host has finished writing to the ERCV. The MSC8101 can now access the ERCV. The ESR[RCV] bit is cleared when the MSC8101 reads the most significant bit.
6. The host can poll the EE3 pin when EE_CTRL[EE3DEF] = `01` to indicate that the SC140 core has read the ERCV register.

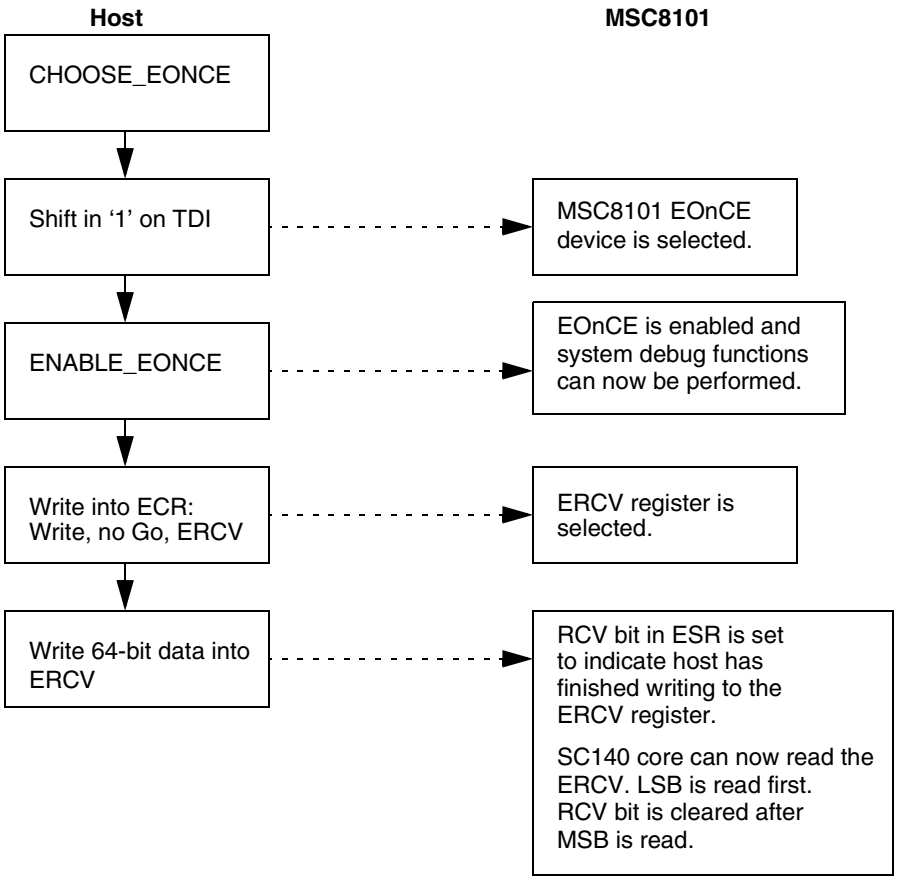


Figure 12-8. Writing to ERCV

12.6 Reading From the EOnCE Transmit Register (ETRSMT)

This section presents an example of how to read from the ETRSMT register through JTAG.

1. Select-IR: `CHOOSE_EONCE` instruction to select EOnCE device.
2. Select-DR: '`1`' since the MSC8101 has only one EOnCE device.
3. Select-IR: `ENABLE_EONCE` instruction to enable the EOnCE registers.
4. The host can poll the EE4 pin when EE_CTRL[EE4DEF] = `01` to indicate that the SC140 core has written the ETRSMT register.
5. Select-DR: Write `0x0204` into the ECR to perform the following operation:

   ECR[R/W] = `1` to perform a read access.

   ECR[GO] = `0` to remain inactive.

   ECR[REGSEL] = `0000100` to select the ETRSMT register.
6. Select-DR: Read the 64-bit ETRSMT data on TDO.

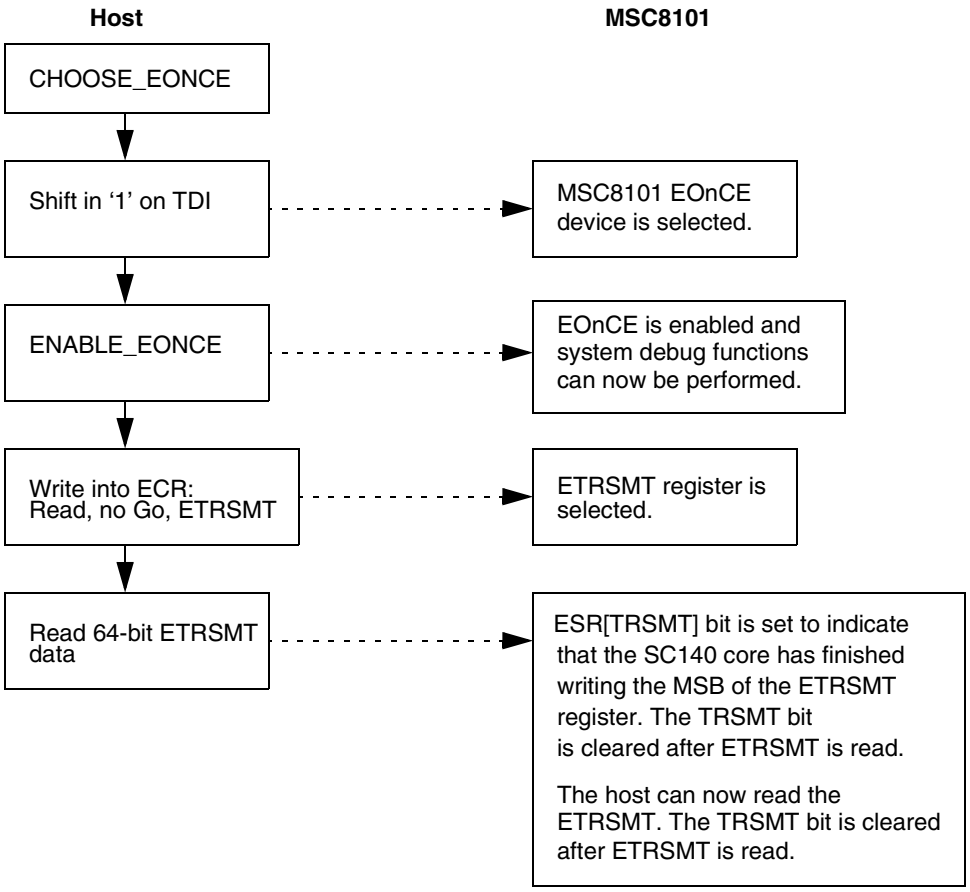


Figure 12-9. Reading From ETRSMT

12.7 Downloading Software

This section presents an example showing how software is downloaded from the host to the DSP via JTAG.

1. Select-IR: `CHOOSE_EONCE` instruction to select EOnCE device.
2. Select-DR: '`1`' since the MSC8101 has only one EOnCE device.
3. Select-IR: `DEBUG_REQUEST` instruction to generate a debug request to the MSC8101 and to allow you to perform system debug functions.
4. Select-DR: Write `0x0002` into the ECR to perform the following operation:

   ECR[R/W] = `0` to perform a write access.

   ECR[GO] = `0` to remain inactive.

   ECR[REGSEL] = `0000010` to select the ERCV register.
5. Select-DR: Write the 64-bit ERCV data on TDI.
6. Select-DR: Write `0x017E` into the ECR to perform the following operation:

   ECR[R/W] = `0` to perform a write access.

ECR[GO] = 1 to execute the instruction.

ECR[REGSEL] = 1111110 to select the CORE_CMD register.

7. Select-DR: Write 48-bit CORE_CMD data to move data from the lower 32-bits of the ERCV to an internal register. Assuming that address register r1 points to 0xEFFE08, the ERCV address, the following command moves the ERCV data into data register d1:

```
move.l (r1)+,d1
```

The CORE_CMD value is 0x0000 0009 98A1. See **Section 12.1.3.2**, *CORE_CMD Example 2,* on page 12-10 for information on calculating this value.

8. Select-DR: Write 0x017E into the ECR to perform the following operation:

ECR[R/W] = 0 to perform a write access.

ECR[GO] = 1 to execute the instruction.

ECR[REGSEL] = 1111110 to select the CORE_CMD register.

9. Select-DR: Write 48-bit CORE_CMD data to move data from the upper 32-bits of the ERCV to an internal register. The following command moves the upper 32-bits of ERCV data into data register d0:

```
move.l (r1)+,d0
```

The CORE_CMD value is 0x0000 0009 90A1.

After the move operation executes, the r1 pointer must be reinitialized to 0xEFFE08 so that it points to ERCV before the next iteration.

10. Select-DR: Write 0x017E into the ECR to perform the following operation:

ECR[R/W] = 0 to perform a write access.

ECR[GO] = 1 to execute the instruction.

ECR[REGSEL] = 1111110 to select the CORE_CMD register.

11. Select-DR: Write 48-bit CORE_CMD data to move data from address registers d0 and d1 to memory. Assuming that address register r0 points to the starting memory address, the following command moves the ERCV data into memory:

```
move.2l d0:d1,(r0)+
```

The CORE_CMD value is 0x0000 0001 8031. See **Section 12.1.3.3**, *CORE_CMD Example 3,* on page 12-10 for information on calculating this value.

12. Repeat steps 4 – 11 until all data and code are downloaded.

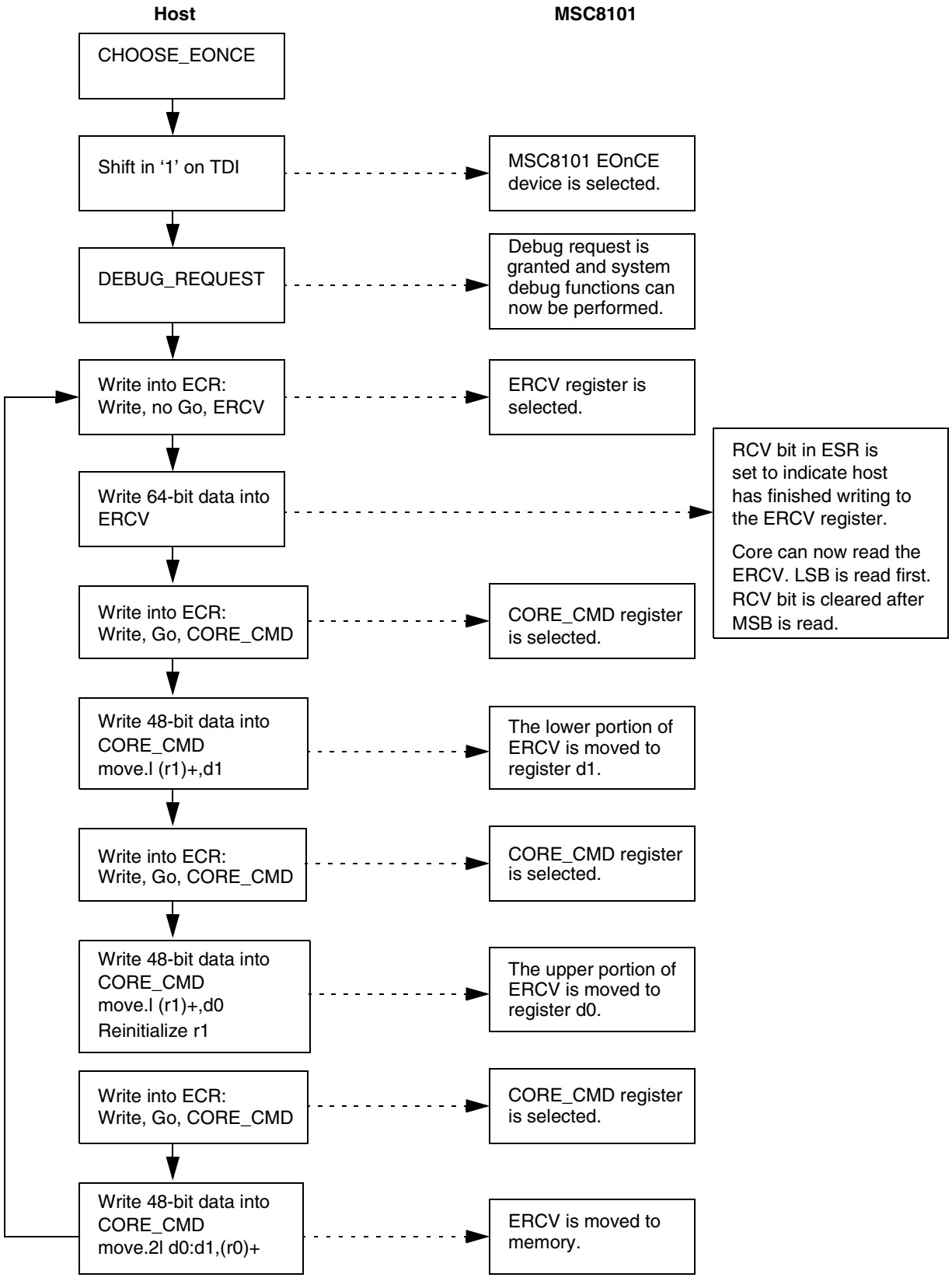


Figure 12-10. Software Downloading

12.8 Writing and Reading the Trace Buffer

This section presents an example that shows how the trace buffer is written and read. **Table 12-5** shows the trace buffer register set:

Table 12-5. Trace Buffer Register Set

Register	Description
TB_CTRL	Trace Buffer Control Register
TB_RD	Trace Buffer Read Pointer
TB_WR	Trace Buffer Write Pointer
TB_BUFF	Trace Buffer Virtual Register

1. Enable the trace buffer by setting TB_CTRL[TEN] = 1. Both TB_WR and TB_RD are cleared when the trace buffer is enabled.
2. Select trace mode. Possible trace modes are:
   - trace change-of-flow instructions
   - trace addresses of interrupt vectors
   - trace issue of execution sets
   - trace MARK instruction
   - trace hardware loops

   For this example, setting TB_CTRL[TEXEC] = 1 traces any execution set.
3. The trace buffer is written.

   The addresses of every issued execution set is written to TB_BUFF, and TB_WR increments after every trace.
4. Read TB_BUFF when the trace is buffer is full or disabled.

   The ESR[TBFULL] flag is set when the trace buffer is full or when 2033 entries (2 KB entry size minus 15) are written. Each entry is 32-bits long. When the end of memory is reached, the trace buffer wraps around to address zero and continues unless EMCR[TBFDM] is set. When this bit is set, the system enters Debug mode when the trace buffer is full. Disabling the trace buffer by clearing TB_CTRL[TEN] allows you to read the contents of the TB_BUFF.
5. Wait three cycles before reading the trace buffer.

Because of the pre-fetch mechanism, a three-cycle delay must occur from the time the trace buffer is disabled until the first read access to the trace buffer is issued.

12.9 Using EE0 to Enter Debug Mode

In the previous examples, the JTAG instruction DEBUG_REQUEST is used to enter Debug mode. Another method of entering Debug mode is to program the EE0 pin to cause the SC140 core to enter Debug mode after core reset. Holding EE0 at logic 1 during and after core reset forces the SC140 core to enter Debug mode.

12.10 Counting Core Cycles

Core cycles are counted using the event counter, event detection unit, and event selector. This example shows how you can use the EOnCE port to perform program profiling. The program executes and when the start address is detected, the counter is enabled and the core clocks are counted. When the final address is detected, a debug exception is generated. The interrupt service routine disables the counter and calculates the number of clocks between the start and final addresses. The event counter, event detection, and event selector register sets are shown in **Table 12-6**.

Table 12-6. Event Register Sets

	Name	Description
Event Counter	ECNT_CTRL	Event Counter Register
	ECNT_VAL	Event Counter Value Register
	ECNT_EXT	Extension Counter Value Register
Event Detection Channel Address	EDCA*i*_CTRL	EDCA Control Register
	EDCA*i*_REFA	EDCA Reference Value Register A
	EDCA*i*_REFB	EDCA Reference Value Register B
	EDCA*i*_MASK	EDCA Mask Register
Event Selector	ESEL_CTRL	Event Selector Control Register
	ESEL_DM	Event Selector Mask Debug Mode Register
	ESEL_DI	Event Selector Mask Debug Exception Register
	ESEL_ETBL	Event Selector Mask Enable Trace Register
	ESEL_DTB	Event Selector Mask Disable Trace Register

1. Initialize the event counter value. Set ECNT_VAL to an initial value of `0xFFFFFFFF`.
2. Specify what needs to be counted. Configure the event counter to count core clocks by setting ECNT_CTRL[ECNTWHAT] = `1100`.
3. Enable the event counter. The event counter is disabled but hardware enables it when EDCA #0 detects an event because ECNT_CTRL[ECNTEN] = `0001`.

In this example, the event counter is enabled when EDCA #0 detects the starting address.

4. Enable the event detection channels. Set EDCA0_CTRL[EDCAEN] = 1111 and EDCA1_CTRL[EDCAEN] = 1111 to enable the EDCA.
5. Set the reference values to be compared by the event detection channel comparators. Set EDCA0_REFA to the start address and EDCA1_REFA to the final address of the program.
6. Specify which condition generates an event detection. Set EDCA0_CTRL[CS] = 00 so only comparator A condition is detected and EDCA0_CTRL[CACS] = 00 to select equal to EDCA0_REFA.

   Set EDCA1_CTRL[CS] = 00 so only the comparator A condition is detected and EDCA1_CTRL[CACS] = 00 to select equal to EDCA1_REFA.
7. Specify the access type and the bus to be sampled for comparison by comparator A. Read accesses are detected by setting EDCA0_CTRL[ATS] = 00 and EDCA1_CTRL[ATS] = 00. When EDCA0_CTRL[BS] = 11 and EDCA1_CTRL[BS] = 11, the program counter is compared to reference registers at every execution of an execution set.
8. Specify which source causes a debug exception. Set ESEL_DI[EDCA1] = 1 so that the detection of the final address causes a debug exception.
9. Configure the event selector for debug exception.

   A debug exception is reached upon detection of the event by any one of the sources selected on the ESEL_DI register by setting ESEL_CTRL[SELDI] = 0. In this example, a debug exception is reached when EDCA #1 reads the final address of the program.
10. Service the debug exception. The interrupt service routine for the debug exception disables the event counter by setting ECNT_CTRL[ECNTEN] = 0000, reads the ECNT_VAL register, and subtracts the number of cycles of the interrupt service routine overhead. When the event counter counts the core clock, the memory contention and external wait state clocks are not counted.

12.11 Related Reading

StarCore SC140 Core Reference Manual

Chapter 4, *Emulation and Debug (EOnCE)*

MSC8101 Reference Manual

Chapter 17, *JTAG and IEEE 1149.1 Test Access Port*

Glossary A

This glossary presents an alphabetical list of terms, phrases, and abbreviations that are used in this manual. Many of the terms are defined in the context of how they are used in this manual—that is, in the context of the MSC8101. Some of the definitions are derived from *Newton's Telecom Dictionary: The Official Dictionary of Telecommunications*, © 1998 by Harry Newton.

AAL — ATM adaptation layer. This layer of the ATM Protocol Reference Model is divided into the convergence sublayer (CS) and the segmentation and reassembly (SAR) sublayer. The AAL accomplishes conversion from the higher-layer, native data format and service specifications of the user data into the ATM layer.

AAU — Address arithmetic unit. On the SC140 core, there are two identical AAUs. Each contains a 32-bit full adder called an offset adder that can add or subtract two AGU registers, add immediate value, increment or decrement an AGU register, add PC, or add with reverse-carry. The offset adder also performs compare or test operations and arithmetic and logical shifts. The offset values added in this adder are pre-shifted by 1, 2, or 3, according to the access width. In reverse-carry mode, the carry propagates in the opposite direction. A second full adder, called a modulo adder, adds the summed result of the first full adder to a modulo value, M or minus M, where M is stored in the selected modifier register. In modulo mode, the modulo comparator tests whether the result is inside the buffer, by comparing the results to the B register, and chooses the correct result from between the offset adder and the modulo adder.

ABI — Application binary interface.

ALU — Arithmetic logic unit. The part of the CPU that performs the arithmetic and logical operations. The SC140 is the four-ALU version of the StarCore SC100 DSP core family.

anti-aliasing filter — Band limits the input, eliminating out-of-band signals that can be aliased back into the pass band due to the sampling filter. The output of the line driver probably has a low pass filter to remove the effects of digitizing.

ASIC Application-specific integrated circuit. An integrated circuit that performs a particular function by defining the interconnection of a set of basic circuit building blocks taken from a library provided by a circuit manufacturer.

ATM Asynchronous transfer mode. A high-speed transmission technology. ATM is a high bandwidth, low-delay connection-oriented, packet-like switching and multiplexing technique.

atomic A bus access that attempts to be part of a read-write operation to the same address uninterrupted by any other access to that address. The MSC8101 initiates the read and write separately, but it signals the memory system that it is attempting an atomic operation. If the operation fails, status is kept so that MSC8101 can try again.

AUI Attachment unit interface. A 15-pin connection interface for ethernet connections defined in IEEE 802.3.

bandwidth A measure of the carrying capacity, or size, of a communications channel. For an analog circuit, the bandwidth is the difference between the highest and lowest frequencies that a medium can transmit and is expressed in hertz (Hz). Hz is equal to one cycle per second.

baseband The original band of frequencies of a signal before it is modulated for transmission at a higher frequency. The signal is typically multiplexed and sent on a carrier with other signals at the same time.

beat A single state on the MSC8101 interface that may extend across multiple bus cycles. An MSC8101 transaction can be composed of multiple address or data beats.

big-endian For big-endian scalars, the most significant byte (MSB) is stored at the lowest, or starting, address while the least significant byte (LSB) is stored at the highest, or ending, address. This memory structure is called “big-endian” because the big end of the scalar comes first in memory. The MSC8101 supports big-endian. *See also* little-endian.

BMU Bit mask unit. On the SC140 core, performs bit mask operations, such as setting, clearing, changing, or testing a destination, according to an immediate mask operand. All bit mask instructions typically execute in two cycles and work on 16-bit data. This data can be a memory location, or a portion (high or low) of a register. Only a single bit mask instruction is allowed in any single execution set, since only one execution unit exists for these instructions.

bootloader Loads and executes source code that initializes the after it completes a reset sequence and programs its registers for the required mode of operation. The bootloader program, which is provided in the internal ROM of the MSC8101, loads and executes source programs received from a host processor, an EPROM, or a standard memory device.

BRG Baud-rate generator. The CPM contains eight independent, identical BRGs for use with the FCCs, SCCs, and SMCs. The clocks produced by the BRGs are sent to the bank-of-clocks selection logic, where they can be routed to the controllers. In addition, the output of a BRG can be routed to a pin for external use. *See also* FCC, SCC, SMC.

broadband Also called wideband. A type of data transmission in which a single medium (wire) can carry several channels at once. Cable TV, for example, uses broadband transmission. In contrast, baseband transmission allows only one signal at a time. Most communications between computers, including the majority of local-area networks, use baseband communications. An exception is B-ISDN networks, which employ broadband transmission.

buffer descriptor (BD) The MSC8101 has two types of buffer descriptors. One type is associated with the CPM, while the second defines action by the DMA controller. For the CPM buffer descriptors, data associated with each communications controller channel is stored in buffers and each buffer is referenced by a BD that can reside anywhere in dual-port RAM. The total number of 8-byte BDs is limited only by the size of the dual-port RAM. These BDs are shared among all communications controllers—FCCs, SCCs, SMCs, SPI, and I^2C. The user defines how the BDs are allocated among the controllers.

Each DMA channel uses the specifications in its associated buffer descriptors to define operation of the channel.

burst A multiple-beat data transfer in the MSC8101 whose total size is equal to 32 bytes or 4 data beats at 8 bytes per beat.

CD Carrier detect.

CDMA Code division multiple access. A spread spectrum method of allowing multiple users to share the radio frequency spectrum by assigning each active user an individual code. In a CDMA system, each voice circuit is labeled with a unique code and transmitted on a single channel simultaneously along with many other coded voice circuits. The only distinctions between the multiple voice circuits are the assigned codes. The channel is typically very wide with each voice circuit occupying the entire channel bandwidth.

CP Communications processor. *See* CPM.

CPM Communications processor module. One of three internal modules in the MSC8101: CPM, SIU, and SC140 extended core. The CPM consists of much more than PIO ports. The brains of the CPM is the communications processor (CP), a 32-bit RISC microcontroller that resides on a separate bus from the SC140 core and performs tasks independently of the SC140 core. The CP handles lower-layer communications tasks and DMA control, freeing the SC140 core to handle higher-layer activities. The CP works with the peripheral controllers, serial DMA (SDMA) channels, timers, baud-rate generators, and PIO ports.

CTS Clear to send.

DALU Data ALU. Performs arithmetic and logical operations on data operands in the MSC8101. The source operands for the Data ALU, which may be 16, 32, or 40 bits, originate either from data registers or from immediate data. The results of all Data ALU operations are stored in the data registers. All Data ALU operations are performed in one clock cycle. Up to four parallel arithmetic operations can be performed in each cycle. The destination of every arithmetic operation can be used as a source operand for the operation immediately following, without any time penalty.

Debug mode On the MSC8101, the JTAG and IEEE 1149.1 Test Access Port gives entry to the debug mode of operation. With the EOnCE real-time debugging capability, users can read the chip's internal resources without having to stop the device and go into debug mode. The benefits range from faster debugging to reduced system development costs and improved field diagnostics. The SC140 core has a debug mode that is enabled at reset by pulling up the DBREQ/EE0 pin. *See also* EOnCE.

DMA

Direct memory access. A fast method of moving data from a storage device to RAM, which speeds up processing. The MSC8101 multi-channel DMA controller supports up to 16 time-multiplexed channels and buffer alignment by hardware. The DMA controller connects to both the 60x-compatible system bus and the local bus and can function as a bridge between both buses. The MSC8101 DMA controller supports flyby transactions on either bus. The DMA controller enables hot swap between channels, by time-multiplexed channels with no cost in clock cycles. Sixteen priority levels support synchronous and asynchronous transfers on the bus and give a varying bus bandwidth per channel. The DMA controller can service multiple requestors. A requestor can be any one of four external peripherals, two internal peripherals, or sixteen internal requests generated by the DMA FIFO itself. *See also* flyby transfer.

double word

For the 16-bit SC140 core, a double word is 32 bits. For the CPM and 60x-compatible bus, a double word is 64 bits.

DRAM

Dynamic random-access memory. Dynamic memory is solid-state memory in which the stored information decays over a period of time. The decay time can range from milliseconds to seconds depending on the device and its physical environment. The memory cells must undergo refresh operations often enough to maintain the integrity of the stored information. The dynamic nature of the circuits for DRAM require data to be written back after being read, hence the difference between access time and cycle time. DRAM memory is organized as a rectangular matrix addressed by rows and columns.

DSP MIPs

At its initial clock speed of 300 MHz, the SC140 core can execute 1,200 true DSP MIPS—1.2 billion multiply-accumulate operations, together with associated data movement functions and pointer updates—per second. One such DSP MIPS is the equivalent of several RISC MIPS, the performance measure used by some other DSPs. For purposes of comparison, the SC140 core can be said to perform 3000 RISC MIPS—ten RISC operations per cycle at 300 MHz. Moreover, the MSC8101 enhanced filter coprocessor (EFCOP) performs filtering operations at a 70 percent usage rate—a typical average for EFCOP utilization in DSP applications—the EFCOP provides 210 MIPS above the SC140 core's 1,200-MIPS performance.

E1

The European equivalent of the North American T1, except that E1 carries information at the rate of 2.048 Mbps. This is a telephony standard. Its size is based on the number of channels, each of which carries 64 Kbps. *See also* T1.

EFCOP

Enhanced filter coprocessor. A peripheral MSC8101 module that functions as a general-purpose, fully programmable complex filter. Its optimized modes of operation perform complex finite impulse response (FIR) filtering, infinite impulse response (IIR) filtering, adaptive FIR filtering, and multichannel filtering. The EFCOP allows filter operations to be completed concurrently with the SC140 core operations with minimal CPU intervention. It has dedicated modes of operation optimized for cellular base station applications. In a transceiver base station, the EFCOP can be used for complex matched filtering to maximize the signal-to-noise ratio (SNR) within an equalizer. In a transcoder base station or a mobile switching center, the EFCOP can be used for all types of FIR and IIR filtering within a vocoder, as well as for LMS-type echo cancellation.

EOnCE

Enhanced On-Chip Emulation. Allows nonintrusive interaction with the MSC8101 and its peripherals so that a user can examine registers, memory, or internal peripherals, define various breakpoints, and read the trace-FIFO. These interactions facilitate hardware and software development on the MSC8101 processor. The EOnCE module interfaces with the debugging system through the internal JTAG TAP controller pins.

FCC

Fast communications controllers. A type of serial communications controller optimized for synchronous high-rate protocols. MSC8101 FCCs can be configured independently to implement different protocols. Together, they can implement bridging functions, routers, and gateways; they can interface with a wide variety of standard WANs, LANs, and proprietary networks. FCCs have many physical interface options, such as interfacing to TDM buses, ISDN buses, standard modem interfaces, fast Ethernet interface (MII), and ATM interfaces (UTOPIA). The FCCs are independent from the physical interface, but FCC logic formats and manipulates data from the physical interface. The FCC is described in terms of the protocol that it runs. When an FCC is programmed to a certain protocol, it implements a certain level of functionality associated with that protocol. For most protocols, this corresponds to portions of the link layer (layer 2 of the seven-layer OSI model). Many FCC functions are common to all protocols.

FC-PBGA package — Flip Chip-Plastic Ball Grid Array. The MSC8101 FC-PBGA package has 332 pins.

FDMA — Frequency division multiple access. A method of allowing multiple users to share the radio frequency spectrum by assigning each active user an individual frequency channel. In this practice, users are dynamically allocated a group of frequencies so that the apparent availability is greater than the number of channels.

FIR — Finite impulse response. A type of filter. FIR filters are characterized by transfer functions that are polynomials, where the coefficients are directly the impulse response of the filter. The form of an FIR filter gives rise to the terminology of tapped delay line and the coefficients as tap weights. The length of an FIR filter is the number of taps, N, and thus the convention of using indices from 0 through (N-1) for the coefficients.

flyby transfer — Also known as a “single access transaction.” The data path is between a peripheral and memory with the same port size, located on the same bus. On the MSC8101, flyby transactions can occur only between external peripherals and external memories located on the 60x-compatible system bus, or between internal peripherals and internal SRAM located on the local bus. Flyby operations do not require access to the DMA FIFO. *See also* DMA.

full duplex — Transmission in two directions simultaneously—that is, simultaneous two-way communications. Such communications occur on four-wire circuits. In contrast, half duplex communications occur in only one direction at one time.

GPCM — General-purpose chip-select machine. Part of the MSC8101 memory controller. The GPCM provides interfacing for simpler, lower-performance memory resources and memory-mapped devices. The GPCM has inherently lower performance because it does not support bursting. For this reason, GPCM-controlled banks are used primarily for boot-loading and access to low-performance memory-mapped peripherals. The MSC8101 GPCM controls Bank 11, which is assigned for DSP peripherals. Banks 0–7 can be assigned to the GPCM as well.

half-word — For the 16-bit SC140 core, a half-word is 8 bits. For the CPM and 60x-compatible bus, a half-word is 16 bits.

HDLC High-level data link control. An ITU-TSS link layer protocol standard for point-to-point and multi-point communications. In HDLC, control information is always placed in the same position. Specific bit patterns used for control differ dramatically from those used in representing data so that errors are less likely to occur. On the MSC8101, the SCCs can run in HDLC mode. *See also* SCC.

I^2C Inter-integrated circuit, a simple and low-cost mechanism for connecting multiple devices. I^2C is more flexible than SPI for multi-masters in handling collisions. I^2C has a synchronous two-wire interface (clock and data). It features bidirectional operation, master/slave modes, and multi-master modes. Clock rates run up to 520 kHz@25 MHz system clock. Developed by Phillips.

IIR Infinite impulse response. A type of filter. *See* FIR.

ISR Interrupt service routine. In the MSC8101, the SC140 core handles pending unmasked interrupts in order of priority. The interrupt controller passes an interrupt vector corresponding to the highest-priority, unmasked, pending interrupt.

lane A sub-grouping of signals within a bus. An 8-bit section of the address or data bus may be referred to as a byte lane for that bus.

little-endian For little-endian scalars, the least-significant byte (LSB) is stored at the lowest (or starting) address. This is called “little-endian” because the little end of the scalar comes first in memory. *See also* big-endian.

MAC Multiply and accumulate. On the SC140 core, the MAC unit is the main arithmetic processing unit. It performs all the calculations on data operands. The MAC unit outputs one 40-bit result in the form of [Extension:Most Significant Portion:Least Significant Portion] (EXT:MSP:LSP). The multiplier executes 16-bit x 16-bit fractional or integer multiplication between two’s complement signed, unsigned, or mixed operands. The 32-bit product is right-justified and added to the 40-bit contents of one of the 16 data registers.

maskable interrupt A hardware interrupt that can be enabled or disabled through software.

master The device that owns the address or data bus, the device that initiates or requests the transaction.

MCC

Multi-channel controller. The two MSC8101 MCCs (MCC1 and MCC2) each handle up to 128 serial, full-duplex data channels. The 128 channels are divided into four subgroups (of 32 channels each). One or more subgroups can be multiplexed through SIx time-division multiplexing (TDM) channels. Multiplexed in this way, the MCCs can support a total of four T1 or E1 lines. MCC1 connects through SI1, and MCC2 uses SI2. Each channel can be programmed separately either to perform high-level data link control (HDLC) formatting/deformatting or to act as a transparent channel. *See also* SI, TDM, T1, and E1.

memory controller

A unit whose main function is to control the bus memories and I/O devices. The MSC8101 memory controller is located in the SIU portion of the MSC8101. It controls a maximum of 10 memory banks shared by a high-performance SDRAM machine, a general-purpose chip-select machine (GPCM), and three user-programmable machines (UPMs). It supports a glueless interface to synchronous DRAM (SDRAM), SRAM, EPROM, flash EPROM, burstable RAM, regular DRAM devices, extended data output DRAM devices, and other peripherals.

MII

Media Independent Interface. Part of the Fast Ethernet specification, the MII replaces 10BaseT AUI (Attachment Unit Interface) and connects the MAC layer to the physical layer. The MII is the standard for all three 100Base-T specifications: 100Base-TX, 100Base-T4, and 100Base0Fx. The MII interface can be used for both 100Base-T and 10Base-T. MSC8101 MII support includes four bits of data for transmit and four bits of data for receive.

MIPS

Millions of instructions per second. A rough measure of processor performance, measuring the number of instructions that can be executed in one second. However, different instructions require more or less time than others, and performance can be limited by other factors, such as memory and I/O speed.

modulo

An arithmetic term to designate an operation that uses the remainder value from a division operation.

Multi-Master Bus Mode

The multi-master bus mode can include one or more potential bus masters external to the MSC8101. The other bus masters can, for example, be ASIC DMAs, high-end PowerQUICC IIs, or other MSC8101 devices. Also see Single Master Bus Mode.

multiplexing

A method by which two or more signals share a physical pin connection or a single data stream.

operation code Also known as “opcode.” The command part of a machine instruction. That is, in most cases, the first byte of the machine code that describes the type of operation and combination of operands to the central processing unit (CPU).

parameter RAM The CPM maintains a section of RAM called the parameter RAM, which contains many parameters for the operation of the FCCs, SCCs, SMCs, SPI, and I^2C channels. The exact definition of the parameter RAM is contained in each protocol subsection describing a device that uses a parameter RAM.

parking Granting potential bus mastership without requiring a prior bus request from that device. This eliminates the arbitration delay associated with the bus request.

PIC Programmable interrupt controller. A peripheral module to serve all the interrupt requests ($\overline{\text{IRQ}}$s) and non-maskable interrupts ($\overline{\text{NMI}}$s) received from MSC8101 peripherals and I/O pins. The PIC is memory mapped to the SC140 core and is accessed via the SC140 core QBus. The PIC not only handles incoming interrupts from internal and external devices, but also generates interrupts to other devices. This capability enables the MSC8101 to be used as a companion chip complementing an external CPU such as a 60x-compatible processor. For example, the MSC8101 might be used to provide protocol handling services, sending an interrupt to notify the central processor each time it finishes processing a batch of data.

pipelining Initiating a bus transaction before the current one finishes. This involves running an address tenure for a new bus transaction before the data tenure for a current bus transaction completes.

PLL Phase lock loop. An electronic circuit that controls an oscillator so that it maintains a constant phase angle relative to a reference signal. A PLL can be used to multiply or divide an input clock frequency to generate a different output frequency.

QFP Quad flat pack. A flat, rectangular package that holds an integrated circuit. The electrical leads, or pins, project from all four sides of the package. These packages are usually made of ceramic materials. When such a package is made of plastic, it is called a PQFP.

quad word For the 16-bit SC140 core, a quad word is 64 bits. For the CPM and 60x-compatible bus, a quad word is 128 bits.

requestor An external peripheral, an internal peripheral, or an internal request generated by the DMA FIFO. A peripheral interfaces with the DMA by placing a request for service. The request can be external or internal, depending on its origin.

reset A means to bring a device and its components to a known state by setting the registers and control bits to predetermined values and signaling execution to start at a specified address.

RS232 with modem controls An ANSI standard specifying three interfaces: electrical, functional, and mechanical. The RS232 standard is typically used to communicate between computers, terminals, and modems. All PCs support RS-232, typically through a DB9 connector, as specified by ANSI. The MSC8101 supports TXD, RXD, $\overline{\text{RTS}}$, $\overline{\text{CTS}}$, and $\overline{\text{CD}}$ signals.

RS232 with no modem controls The MSC8101 SMC1-2 interface supports TXD, RXD, and $\overline{\text{SMSYN}}$ signals. The SMC interface does not support modem control signals, and the data rates are not as fast as those of the SCC interface.

SCC Serial communications controller. The MSC8101 has four SCCs, which can be configured independently to implement different protocols for bridging functions, routers, and gateways and to interface with a wide variety of standard WANs, LANs, and proprietary networks. An SCC offers many physical interface options, such as interfacing to TDM buses, ISDN buses, and standard modem interfaces. The SCCs are independent from the physical interface, but SCC logic formats and manipulates data from the physical interface. Furthermore, the choice of protocol is independent from the choice of interface. An SCC is described in terms of the protocol it runs. When an SCC is programmed to a certain protocol or mode, it implements functionality that corresponds to parts of the protocol's link layer (layer 2 of the OSI reference model). *See also* CPM and FCC.

SDMA Serial DMA. The MSC8101 has two physical SDMA channels. The CP implements many dedicated virtual SDMA channels for each FCC, MCC, SCC, SMC, SPI, and I^2C—one for each transmitter and receiver. Each channel is permanently assigned to service either the receive or transmit operation of an FCC, MCC, SCC, SMC, SPI, or I^2C. On the MSC8101, the SDMA makes it possible for local bus transfers to occur at the same time as other operations on the external 60x-compatible system bus. *See also* CPM.

SDRAM A type of DRAM that can deliver bursts of data at very high speeds using a synchronous interface.

set To write a non-zero value to a bit or bit field; the opposite of *clear*. The term *set* can also more generally describe the updating of a bit or bit field.

SI Serial interface. On the MSC8101, there are actually two serial interfaces (SI1 and SI2). The “1” and “2” at the end of the TDM names indicates which serial interface (SI) they belong to: SI1 or SI2. The MSC8101 has one TDM from SI1 (TDMA1) and three from SI2 (TDMB2, TDMC2, TDMD2). The MSC8101 TDM interfaces are split up between SI1 and SI2 to allow for higher system performance. *See also* TDM.

Single-Master Bus Mode This mode uses the MSC8101 memory controller as the only 60x-compatible bus master to connect external devices to the bus. In single-master bus mode, the MSC8101 uses the address bus as a memory address bus. Slaves cannot use the 60x-compatible system bus signals because the addresses use memory timing, not address tenure timing. Also see Multi-Master Bus Mode.

SIU System interface unit. Controls system start-up and initialization, as well as operation, protection, and the external system bus. The system configuration and protection functions provide various monitors and timers, including the bus monitor, software watchdog timer, periodic interrupt timer, and time counter. The clock synthesizer generates the clock signals for the SIU and other MSC8101 modules.

slave The device addressed by the master. The slave is identified in the address tenure and is responsible for sourcing or sinking the requested data for the master during the data tenure.

SMC	Serial management controller. The CPM has two SMCs, which are full-duplex ports that can be configured independently to support one of three protocols or modes: UART mode, Transparent mode, and General-Circuit Interface (CGI) mode. UART operation provides a debug/monitor port in an application, freeing the SCCs for other uses. In totally Transparent mode, the SMC can connect to a TDM channel (such as an E1 line) or directly to its own set of signals. This mode can be used for a fast connection between MSC8101s.
snooping	Monitoring addresses driven by a bus master to detect the need for coherency actions.
SPI	Serial peripheral interface. A full-duplex, synchronous, character-oriented channel that supports a four-wire interface (receive, transmit, clock, and slave select). SPI supports master/slave and multi-master modes. Character size is programmable. Developed by Freescale Semiconductor.
split-transaction	A transaction with separate request and response tenures.
SRAM	Static random access memory. Contrast with dynamic random access memory (DRAM). The dynamic nature of the circuits for DRAM require data to be written back after being read, hence the difference between the access time and the cycle time and also the need to refresh. SRAMs use more circuits per bit to prevent the information from being disturbed when read. Thus, unlike DRAMs, there is no difference between access time and cycle time, and there is no need to refresh SRAM. In DRAM designs, the emphasis is on capacity, while SRAM designs are concerned with both capacity and speed.
super channel	The super channel table entry redirects an MCC slot to a different channel number. Therefore, the transmitter super channel uses more FIFO in the MCC hardware (2 bytes—half of a single-channel transmitter FIFO—multiplied by the number of the channels in the super channel). On the transmitter side, super channels must be defined in the SI RAM, and a super-channel table must be created. On the receiver side, the transparent super channels that require slot synchronization must be programmed in the SI RAM as super channels. The slot synchronization ensures that the data is aligned in the receiver buffer starting from the first time slot after a sync pulse. *See also* MCC, HDLC, and SI.
T1	Digital transmission link with a capacity of 1.544 Mbps. *See also* E1.

TDM Time-division multiplexing. The external interface to the MSC8101 and the MSC8101 time-slot assigner (TSA) block. The MSC8101 supports four TDM interfaces: TDMA1, TDMB2, TDMC2, and TDMD2.

tenure The period of bus mastership. For MSC8101, there can be separate address bus tenures and data bus tenures.

transaction A complete exchange between two bus devices. A typical transaction is composed of an address tenure and a data tenure, which may overlap or occur separately from the address tenure. A transaction can minimally consist of an address tenure alone.

TSA Time-slot assigner. A functional block within the MSC8101 CPM that connects the time-division multiplexing (TDM) interfaces to selected communications controllers inside the MSC8101.

UART Universal asynchronous receiver/transmitter. A serial communications interface.

UPM User-programmable machine. The MSC8101 memory controller has three UPMs. The UPMs support address multiplexing of the 60x-compatible system bus, refresh timers, and generation of programmable control signals for row address and column address strobes to allow for a glueless interface to DRAMs, burstable SRAMs, and almost any other kind of peripheral. The UPM can generate different timing patterns for the control signals that govern a memory device. These patterns define how the external control signals behave during a read, write, burst-read, or burst-write access request. Refresh timers are also available to periodically generate user-defined refresh cycles.

UTOPIA Universal Test and Operations Interface for ATM. UTOPIA is the interface to an ATM network. It is defined by the ATM Forum in the UTOPIA Specification Level 1 and UTOPIA Specification Level 2 documents. The Level 2 specification is a continuation of the Level 1 document. The MSC8101 is Level-1 and Level-2 compliant.

wait state A period of time when a bus does nothing but wait. Wait states are used to synchronize circuitry or devices operating at different speeds so that they seem to be operating at the same speed.

word The MSC8101 DSP core is a 16-bit processor, so a word in the core portion of the MSC8101 is 16 bits. The SIU and CPM portions of the MSC8101 consider a word equal to 32 bits.

Index

Numerics

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W