

B ADSP-2191 PERIPHERALS USED WITH THE SIMULATOR

In This Appendix

This appendix contains the following topics:

- [“General-Purpose IO \(GPIO\) or Flag IO \(FIO\) Peripheral” on page B-2](#)
- [“Host Port Interface \(HPI\) Peripheral” on page B-8](#)
- [“Serial Peripheral Interface \(SPI\)” on page B-17](#)
- [“Serial Port \(SPT\) Peripheral” on page B-33](#)
- [“Serial Port \(UART\) Peripheral” on page B-43](#)
- [“Timer \(TMR\) Peripheral” on page B-53](#)
- [“Memory DMA \(MEMDMA\) Peripheral” on page B-61](#)

The example programs in this appendix are located in the following folder:

`VisualDSP\219x\Examples\Simulator Peripheral Examples`

General-Purpose IO (GPIO) or Flag IO (FIO) Peripheral

Overview of GPIO in the Simulator

The ADSP-2191 GPIO peripheral provides flag IO functionality. The sixteen flag IO bits are grouped as a 16-bit register. The behavior of the bits is controlled with ten, 16-bit registers. Note that each pin/bit is independent of the others.

The simulator supports all the GPIO registers, described in the following table:

Register	Address	Description
Direction Register	0x1800	Controls whether bits are input or output (output = 1) You can alter this value while the program is running.
Control and Status Register Pair	0x1802	Write 1 to clear. Clears the value of output bits.
	0x1803	Write 1 to set. Sets the value of output bits.
	Note: Input bits are read-only. Both registers can be read for current status.	
Interrupt Mask A Register Pair	0x1804	Write 1 to clear. Clears bits in Interrupt Mask A.
	0x1805	Write 1 to set. Sets bits in Interrupt Mask A.
	Note: Both registers can be read for the current mask A value.	

ADSP-2191 Peripherals Used with the Simulator

Register	Address	Description
Interrupt Mask B Register Pair	0x1806	Write 1 to clear. Clears bits in Interrupt Mask B.
	0x1807	Write 1 to set. Sets bits in Interrupt Mask B.
	Note: Both registers can be read for current mask B value.	
Polarity Register	0x1808	Relation between GPIO bits and chip pins (reverse = 1)
Interrupt Sensitivity Register	0x180a	Level or edge sensitive interrupts (edge = 1)
Interrupt Edge Sensitivity Register	0x180c	Double- or single-edge sensitive (double = 1)

The reset value of all GPIO registers is 0 (zero).

Input and Output Handling

You can simulate GPIO flags by using streams. Input values are read from stream GPIO.RX, and output values are written to stream GPIO.TX. Only bits currently set as input bits are changed from the input stream, and only bits currently set as output bits are written to the output stream.

You can individually set the GPIO flags by using the signal interface, which overrides the streams for the bits in question. Any input bit that has no signal or stream input value is either set to 0 (zero) initially, or is returned to its previous value.

NB refers to signals not implemented in the initial release.

GPIO Window in VisualDSP++

You can view the GPIO register from the GPIO window.

To open the GPIO window:

1. From the Registers menu, choose Peripherals.
2. Choose GPIO.

When you set registers from the window, their values are updated immediately.

Example – GPIO Test

This example GPIO test sets up all flags as outputs, and then writes each one in turn to cause an interrupt, whose handling routine clears the GPIO status register. The test is not complete, as an LDF file is required to put the sections in the correct place.

```
// reset interrupt vector
.section/code IVreset;
                JUMP START
// interrupt vector 15 (peripheral 11)
// default for gpio
.section/code IVint15;
                jump IVSR0;

.section/code program;
START:
    // set up gpio registers
    iopg=0x06;
    srl=  0xffff ;
    sr0=  0x0000 ;
    // direction all output
    io( 0x0000 )=srl ;
    // edge sensitive
    io( 0x000a )=srl ;
    // single edge
    io( 0x000c )=sr0 ;
    // interrupt mask A
    io( 0x0005 )=srl ;
    // polarity normal
    io( 0x0008 )=sr0 ;
    // set interrupt mask bit
    ax0=imask;
    ay0= 0x8000 ;
    ar=ax0 or ay0;
    imask=ar;
```

General-Purpose IO (GPIO) or Flag IO (FIO) Peripheral

```
// set peripheral interrupt priority
iopg=1;
sr0 = 0xbbbb;
io(0x0203) = sr0;
// enable interrupts
ena int;
// set up circular buffer
ax0=DATA_SRC1;
i0 =ax0;
m0 =1;
reg(b0) =ax0;
l0 =32;
// loop: write each bit of gpio status
iopg = 0x6;
cntr=0x0020;
DO LOOP0-1 UNTIL ce;
    nop;
    sr0 = dm(i0, m0);
    io(0x0003) = sr0;
    nop;
    nop;
    nop;
    nop;
    nop;
LOOP0:
END:
    jump END;

// interrupt handling routine
IVSR0:
    iopg=0x06;
    // clear all gpio status bits
    sr1 = 0xffff;
    io( 0x0002 ) = sr1;
    rti;

.section/data data1;
```

ADSP-2191 Peripherals Used with the Simulator

```
DATA_SRC1:  
.var= 0x0001;  
.var= 0x0002;  
.var= 0x0004;  
.var= 0x0008;  
.var= 0x0010;  
.var= 0x0020;  
.var= 0x0040;  
.var= 0x0080;  
.var= 0x0100;  
.var= 0x0200;  
.var= 0x0400;  
.var= 0x0800;  
.var= 0x1000;  
.var= 0x2000;  
.var= 0x4000;  
.var= 0x8000;
```

Host Port Interface (HPI) Peripheral

Overview of HPI in the Simulator

The ADSP-2191 Host Port Interface peripheral enables you to simulate host-port functionality. The peripheral supports DMA-controlled accesses and external host-initiated accesses.

The Host Port has ten registers to control its operation:

- Host Port Configuration Register (0x1C01)
- Host Port Page Register (0x1C02)
- Host Port Status Register (0x1C03)

The DMA Control Register conforms to the standard ADSP-2191 DMA descriptors:

- DMA Descriptor Pointer Register (0x1D00)
- DMA Configuration Register (0x1D01)
- DMA Page Register (0x1D02)
- DMA Address Register (0x1D03)
- DMA Count Register (0x1D04)
- DMA Next Descriptor Pointer Register (0x1D05)
- DMA Descriptor Ready (0x1D06)

Input and Output

The Host Port has three associated streams to handle input, output, and external control:

- HOST.RX

Data input is from this stream. Data for DMA and external host-initiated writes to memory is read from the input stream.

- HOST.TX

Data output goes to this stream. Data from DMA and external host-initiated reads from memory is written to this output stream.

- HOST.CONTROL

External host-initiated operations are controlled from this control stream. Commands are read from this file and carried out until the end of the file is reached or the STOP command is found.

DMA operations have priority over external initiated operations, and, consequently, force the suspension of any external initiated operations until the DMA operation is finished. Processing of the external initiated operations then resumes from the point at which they were suspended.

Commands for the External Initiated Control File

The control file is a stream file, which is a list of numbers. You must construct this file if you have to simulate external host initiated operations.

The C header file `hostcommands.h`, used in the VisualDSP++ build, is included to ensure that the exact definitions are available. They are documented in [“Command Bit Definitions” on page B-11](#).

In the control file, any line that is blank or zero is skipped. On the same peripheral cycle, the control file is read until a command is reached. Arguments, however, are expected on the line immediately following their command. Therefore, they can be zero.

Processing of the control file continues during DMA operations until any of the following occur:

- External initiated direct operation

This operation is delayed until the DMA operation is finished.

- Stop command

This command stops processing.

- Wait for DMA command

This command suspends the control file until a DMA interrupt occurs.

Command Bit Definitions

The following bit definitions are used to construct the commands. The word is read from the control file and decoded. If the word requires arguments, subsequent words are read to provide the arguments (such as counts for repeated operations).

- `#define HOST_DIRECT (0x100)`

An external host-initiated direct operation

The particular direct operation is specified in the lower 8 bits.

In the case of a burst operation, the number of operations is given as an argument.

For extra bit definitions, see [“External Initiated Direct Operation Bit Definitions” on page B-12](#).

- `#define HOST_DELAY (0x200)`

Provides a delay

The number of peripheral cycles is given as an argument.

- `#define HOST_REPEAT (0x300)`

The Next command is repeated a specified number of times.

The number of repetitions is given as an argument.

- `#define HOST_NULL (0x400)`

No operation

Provides a one peripheral cycle delay.

No argument

Host Port Interface (HPI) Peripheral

- `#define HOST_STOP (0x500)`

Terminates processing of the control file until a reset operation occurs

No argument

- `#define HOST_WAIT_FOR_DMA (0x600)`

Suspends processing of the control file until a host DMA interrupt is generated

No argument

External Initiated Direct Operation Bit Definitions

For external initiated direct operations, the following bits further define the operation:

- `#define HOST_READ (0x00)`

A host-port read, memory-write operation

- `#define HOST_WRITE (0x01)`

A host port write, memory read operation

Data memory (HSC0) or IO memory (HSC1)

- `#define HOST_HSC0 (0x02)`

Simulates the external pin HSC0 being high

- `#define HOST_HSC1 (0x04)`

Simulates the external pin HSC1 being high

- `#define HOST_MEM               HOST_HSC0`

- `#define HOST_ONCHIP_IO HOST_HSC1`

Provided for convenience

- `#define HOST_SLAVE (0x00)`

A single direct operation

- `#define HOST_BURST (0x08)`

A burst operation

The number of accesses is specified by an argument.

- `#define HOST_LATCH (0x00)`

The operation is conducted with Address Latch Enable.

- `#define HOST_CYCLE (0x10)`

The operation is conducted with Address Cycle Control.

Currently, the simulator sees no difference between Address Latch Enable and Address Cycle Control.

Host Port Window in VisualDSP++

You can view the host port registers from the Host Port window.

To open the Host Port window:

1. From the **Registers** menu, choose **Peripherals**.
2. Choose **Host**.

Host Port Interface (HPI) Peripheral

Unsupported Features

Packing is not simulated. Words are read from and written to stream files as complete words.

Example – DMA Transfer to the Host Port

This example sets up a DMA transfer to the host port and generates an interrupt upon completion.

Set up a stream file to accept the output from the stream HOST.TX.

```
// Reset interrupt
.section/code IVreset;
        JUMP START;

// Interrupt 4 (Peripheral 0)
// default interrupt for Host Port
.section/code IVint4;
        jump IVSR0;

.section/code program;
START:
    // clear DAG
    ax0=0;
    i3 =ax0;
    m3 = 1;
    l3 = ax0;
    reg(b3) =ax0;
    // set interrupt mask bit
    ax0=imask;
    ay0= 0x0010 ;
    ar=ax0 or ay0;
    imask=ar;
    // set peripheral interrupt priority
    iopg=0x1;
    ax0 = 0x7654 ;
```

ADSP-2191 Peripherals Used with the Simulator

```
io(0x201) = ax0;
// enable interrupts
ena int;
// set host port bus width
iopg=0x07;
sr0= 0x0001 ;
io( 0x0001 )=sr0 ;

// trigger start of DMA
iopg=0x07;
sr0= DESCRIPTOR ;
io( 0x0105 )=sr0 ;
sr0= 0x0001 ;
io( 0x0106 )=sr0 ;
sr0= 0x8001 ;
io( 0x0101 )=sr0 ;
nop;
idle(0);
nop;
END:
    jump END;

ERR:
    jump ERR;

IVSR0:
    // interrupt service routine reached!
    nop;
    rti;

.section/data data1;
DESCRIPTOR:
.var = 0x8005;    // control word
.var = 0x0000;    // page
.var = DATA_SRC; // address
.var = 0x0004;    // count
.var = DESCRIPTOR; // next pointer
```

Host Port Interface (HPI) Peripheral

```
.var = 0x0000;  
.var = 0x0000;  
.var = 0x0000;  
DATA_SRC:  
.var = 0x0001;  
.var = 0x0002;  
.var = 0x0003;  
.var = 0x0004;
```

Serial Peripheral Interface (SPI)

Overview of SPI in the Simulator

The ADSP-2191 SPI peripherals provide industry-standard synchronous serial link functionality. Two SPI peripherals are on the ADSP-2191 DSP.

In the external four-wire interface, the `MOSI` and `MISO` pins are simulated, but the `SPICLK` and `PIO_nSPISSIN` pins are not. You can use the stream interface with the `MOSI` and `MISO` pins only.

The simulator supports DMA and non-DMA modes as both master and slave, which takes into account baud rates but not clock phase and polarity.

The simulator supports the following SPI registers:

- Control [15:0] – Control Register
- Flag [15:0] – Flag Register (content is ignored)
- Status [6:0] – Status Register
- TDBR [15:0] – Transmit Data Buffer Register
- RDBR [15:0] – Receive Data Buffer Register
- Baud [15:0] – Baud rate Register
- RDBR_SHADOW [15:0] – Receive Data Buffer Register (Shadow)
- DMA Current Pointer [15:0]
- DMA Configuration [15:0]
- DMA Start Page [8:0]
- DMA Start Address [15:0]

Serial Peripheral Interface (SPI)

- DMA Word Count [15:0]
- DMA Next Descriptor [15:0]
- DMA Descriptor Ready [0]
- DMA Interrupt [0]

Global Status and Control

The following table describes the bits in the Status and Control register:

Bit	Status[6:0]	Global Status and Control
0	SPIF	Single-word transfer complete
1	MODF	Mode fault error for master device (not supported)
2	TXE	Transmission error: transmission occurred with no new data in TDBR
3	TXS	TDBR status: 0 = empty, 1 = full
4	RBSY	Receive error: data is received with RDBR full
5	RXS	RBDR status: 0 = empty, 1 = full
6	TXCOL	Transmit collision error: possible corrupted data transmitted

The mode fault error bit in the SPIFLG register is not supported.

The following configuration bits do not affect the simulator:

Bit	Config[15:0]	SPI Configuration Register
4	PSSE	PIO_nSPISSIN input for master
5	EMISO	MISO pin as output
10	CPHA	Clock phase
11	CPOL	Clock polarity
13	WOM	Open drain data output

SPI Signal Usage

The following signals are not recognized:

- PIO_nSPISSIN
- SPICLK

Modes of Operation

Master Mode Operation (no DMA)

The core writes to `SPICTL` and `SPIBAUD` to configure the device as master and to set the word length and baud rate. Transfer starts upon a write to `TDBR` or a read from `RDBR`, depending on the configuration of the `TIMOD` bits in `SPICTL`. Data is read from the `MISO` pin into `RDBR` and sent out to the `MOSI` pin from `TDBR`.

If `TDBR` remains empty or `RDBR` remains full, the SPI operates according to the `SZ` and `GM` bits in the `SPICTL` register. The simulator does not generate the programmed clock pulses on `SPICLK` during transmission.

Slave Mode Operation (no DMA)

Slave mode operation is summarized as follows:

1. The core writes to `SPICTL` to configure the device as slave and to set the word length.
2. In hardware, the transfer starts once a falling edge of the `PIO_nSPISSIN` is detected. Note that the simulator does not simulate transitions of the `PIO_nSPISSIN` pin. Therefore, transmission starts as soon as slave mode is properly set and the SPI device is enabled.
3. Transmission ends after the proper number of clock cycles but not when `PIO_nSPISSIN` is released.
4. Data is read from the `MOSI` pin into `RDBR`, and is sent out to the `MISO` pin from `TDBR`.

If `TDBR` remains empty or `RDBR` remains full, the SPI operates according to the `SZ` and `GM` bits in the `SPICTL` register. The simulator does not synchronize transmission on `SPICLK` active edges.

Master Mode DMA Operation

Master mode DMA operation is summarized as follows:

1. The core writes to `SPICTL` and `SPIBAUD` to configure the device as master and to set the word length and baud rate. `TIMOD` bits are set to the “Transmit or Receive with DMA” mode.
2. The core generates one or more descriptor blocks in page 0 of the memory space.
3. The core writes to the DMA Configuration register to enable the DMA engine.
4. If the device is configured for transmit operation, the transfer starts when the DMA engine reads data from memory into the DMA buffer. If the device is configured for receive operation, the transfer starts when the DMA engine reads data from DMA buffer into memory.
5. Data is read from the `MISO` pin into `RDBR` and is sent out to the `MOSI` pin from `TDBR` until the DMA Word Count register reaches 0.

If `TDBR` remains empty or `RDBR` remains full, the SPI operates according to the `SZ` and `GM` bits in the `SPICTL` register.

The simulator does not generate the programmed clock pulses on `SPICLK` during transmission.

Serial Peripheral Interface (SPI)

Slave Mode DMA Operation

Slave mode DMA operation is summarized as follows:

1. The core writes to `SPICTL` to configure the device as slave and to set the word length. `TIMOD` bits are set to the “Transmit or Receive with DMA” mode.
2. The core generates a DMA receive descriptor block in page 0 of the memory space.
3. The core writes to the DMA Configuration register to enable the DMA engine. The head of the descriptor block is written to the DMA Next Descriptor register.
4. In hardware the transfer starts once a falling edge of the `PIO_nSPISSIN` is detected. Note that the simulator does not simulate transitions of the `PIO_nSPISSIN` pin. Therefore, transmission starts as soon as the slave mode DMA is properly set and the SPI device is enabled.
5. Transmission ends when the DMA Word Count register reaches 0.
6. The core can continue by queuing the next command buffer descriptor block.
7. Data is read from the `MOSI` pin into `RDBR` and is sent to the `MISO` pin from `TDBR`.

If `TDBR` remains empty or `RDBR` remains full, the SPI operates according to the `SZ` and `GM` bits in the `SPICTL` register. The simulator does not synchronize transmission on `SPICLK` active edges.

SPI with Streams

The SPI can read and write from the `MISO` and `MOSI` pins by using the Streams functionality in VisualDSP++.

You can attach a file to the following device names:

`SPI0.MISO`

`SPI0.MOSI`

`SPI1.MISO`

`SPI1.MOSI`

The format of the input file is as follows:

`Data0`

`Data1`

`...`

`DataN`

Data can be either 8 or 16 bits long, depending on the word length set in the `SPICTL` register.

Note that the `MISO` and `MOSI` pins are used as input or output pins, depending on whether the SPI device is configured as master or slave.

Serial Peripheral Interface (SPI)

Master Mode Example (No DMA)

From the Tools menu, choose Streams to attach the `SPI0.MOSI` pin to a file as output. This example writes data from a buffer in memory to SPI0's MOSI pin by using the interrupt mode.

```
// Set the Page to the SPI page
iopg=0x04;

// Write to SPI Configuration register
// TIMOD [1:0] = 01 : Initiate transfer by writing to TDBR.
// Interrupt active when // TDBR is empty
// SIZE [8] = 1 : Word length is 16 bits
// MSTR [12] = 1 : Device is master
// SPE [14] = 1 : SPI module is enabled
sr0= 0x5101 ;
io( 0x0000 )=sr0 ;

// Mask the SPI interrupt
ax0=imask;
ay0= 0x0010 ;
ar=ax0 or ay0;
imask=ar;

// Write to SPI Baud rate register
sr0= 0x0004 ;
io( 0x0005 )=sr0 ;
nop;
nop;
```

ADSP-2191 Peripherals Used with the Simulator

```
nop;
// Set up buffer
i1 = buffer;
m1 = 1;

// Trigger transmission by writing to TDBR
sr0 = dm(i1, m1) ;
io( 0x0003 )=sr0;
nop;
nop;
nop;
nop;
nop;
cntr = 0x10;
Do LOOP0 UNTIL ce
nop;
nop;
// Wait for interrupt to occur
idle(0);
LOOP0: nop;

// Interrupt Service Routine
sr0 = dm(i1, m1); // read next data from buffer
iopg=0x04;        // set SPI memory page
io( 0x0003 ) = sr0; // transmit next data
nop;
nop;
nop;
rti;
```

Serial Peripheral Interface (SPI)

Slave DMA Mode Example

From the Tools menu, choose Streams to attach the `SPI0.MOSI` pin to the following input file:

0x0001

0x0002

0x0004

0x0008

0x0010

0x0020

0x0040

0x0080

0x0100

0x0200

0x0400

0x0800

0x1000

0x2000

0x4000

0x8000

This example reads data from the SPI0.MOSI pin to a memory buffer by using DMA, and compares the input with internal values.

```
// Mask the SPI interrupt
ax0=imask;
ay0= 0x0010 ;
ar=ax0 or ay0;
imask=ar;

// Set dm page to 0 to write the descriptor block
dmpg1=0x00;
my0 = i3 ;
i3 = 0x1000 ;
nop ;

// Prepares a DMA descriptor block in memory
ax0= 0x8057 ;
dm(i3 ,m3 )=ax0 ;
ax0= 0x0000 ;
dm(i3 ,m3 )=ax0 ;
ax0= 0x1020 ;
dm(i3 ,m3 )=ax0 ;
ax0= 0x0010 ;
dm(i3 ,m3 )=ax0 ;
ax0= 0x1000 ;
dm(i3 ,m3 )=ax0 ;
i3 = my0 ;
```

Serial Peripheral Interface (SPI)

```
// Set the Page to the SPI page
iopg=0x04;

// Programs the SPI for DMA tranfer, slave mode
sr0= 0x4102 ;
io( 0x0000 )=sr0 ;
sr0= 0x0002 ;
io( 0x0005 )=sr0 ;

// Writes to SPI DMA registers
sr0= 0x1000 ;
io( 0x0105 )=sr0 ;
sr0= 0x0005 ;
io( 0x0101 )=sr0 ;
sr0= 0x0001 ;
io( 0x0107 )=sr0 ;
sr0= 0x0001 ;
io( 0x0106 )=sr0 ;
sr0= 0x0010 ;
io( 0x0104 )=sr0 ;
nop;
nop;
nop;

// Wait for interrupt to occur (on DMA completion)
nop;
idle(0);
nop;
```

ADSP-2191 Peripherals Used with the Simulator

```
// Check the buffer filled by the DMA engine (starts at
0x1020) against
// internal values
my0 = i3 ;
i3 = 0x1020 ;
nop ;
ax0=dm(i3 ,m3 );
ay0= 0x0001 ;
call ALU_COMP1 ;
ax0=dm(i3 ,m3 );
ay0= 0x0002 ;
call ALU_COMP1 ;
ax0=dm(i3 ,m3 );
ay0= 0x0004 ;
call ALU_COMP1 ;
ax0=dm(i3 ,m3 );
ay0= 0x0008 ;
call ALU_COMP1 ;
ax0=dm(i3 ,m3 );
ay0= 0x0010 ;
call ALU_COMP1 ;
ax0=dm(i3 ,m3 );
ay0= 0x0020 ;
call ALU_COMP1 ;
ax0=dm(i3 ,m3 );
ay0= 0x0040 ;
call ALU_COMP1 ;
ax0=dm(i3 ,m3 );
ay0= 0x0080 ;
```

Serial Peripheral Interface (SPI)

```
call ALU_COMP1 ;
ax0=dm(i3 ,m3 );
ay0= 0x0100 ;
call ALU_COMP1 ;
ax0=dm(i3 ,m3 );
ay0= 0x0200 ;
call ALU_COMP1 ;
ax0=dm(i3 ,m3 );
ay0= 0x0400 ;
call ALU_COMP1 ;
ax0=dm(i3 ,m3 );
ay0= 0x0800 ;
call ALU_COMP1 ;
ax0=dm(i3 ,m3 );
ay0= 0x1000 ;
call ALU_COMP1 ;
ax0=dm(i3 ,m3 );
ay0= 0x2000 ;
call ALU_COMP1 ;
ax0=dm(i3 ,m3 );
ay0= 0x4000 ;
call ALU_COMP1 ;
ax0=dm(i3 ,m3 );
ay0= 0x8000 ;
call ALU_COMP1 ;
i3 = my0 ;
nop;
nop;
nop;
```

ADSP-2191 Peripherals Used with the Simulator

```
nop;
```

```
END:
```

```
    nop;
```

```
    nop;
```

```
    jump END;
```

```
    nop;
```

```
    nop;
```

```
    nop;
```

```
ALU_COMP1:
```

```
    ar=ax0 xor ay0;
```

```
    if eq rts;
```

```
ERR:
```

```
    nop;
```

```
    nop;
```

```
    nop;
```

```
    nop;
```

```
    nop;
```

```
    jump ERR;
```

```
    nop;
```

```
    nop;
```

```
// Interrupt Service Routine
```

```
IVSR0:
```

```
    iopg=0x04;    // Set the Page to the SPI page
```

```
    sr0= 0x0001 ;    // Reset DMA interrupt status bit  
                    // by writing to DMA register.
```

Serial Peripheral Interface (SPI)

```
io( 0x0107 )=sr0 ;  
nop;  
rti;  
nop;
```

Serial Port (SPT) Peripheral

Overview of SPT in the Simulator

The ADSP-2191 Serial Port peripheral provides a simulation of serial port functionality. The module supports all modes of operation, including Direct Memory Access (DMA), Multi-Channel Mode (MCM), and interrupt-driven modes for both transmit (TX) and receive (RX). Additionally, the zero-fill, sign-extend, and A-Law μ -Law companding data types are supported on transmit and receive. The ADSP-2191 has three serial ports.

The Serial Port has the following registers to control its operation:

Configuration and Status Registers	SPT0	SPT1	SPT2
Serial Port TX Configuration Register	0x0A00	0x0C00	0x0E00
Serial Port RX Configuration Register	0x0A01	0x0C01	0x0E01
Serial Port TX Buffer	0x0A02	0x0C02	0x0E02
Serial Port RX Buffer	0x0A03	0x0C03	0x0E03
Serial Port TX Serial Clock Divisor	0x0A04	0x0C04	0x0E04
Serial Port RX Serial Clock Divisor	0x0A05	0x0C05	0x0E05
Serial Port TX Frame Synch Divisor	0x0A06	0x0C06	0x0E06
Serial Port RX Frame Synch Divisor	0x0A07	0x0C07	0x0E07
Serial Port Status Register	0x0A08	0x0C08	0x0E08

Serial Port (SPT) Peripheral

MCM Select and Configuration Registers	SPT0	SPT1	SPT2
TX Channel Select MTCS[0:7]	0x0A09 –10	0x0C09-10	0x0E09-10
RX Channel Select MRCS[0:7]	0x0A11 –18	0x0C11-18	0x0E11-18
Multi-Channel Configuration Register 1	0x0A19	0x0C19	0x0E19
Multi-Channel Configuration Register 2	0x0A1A	0x0C1A	0x0E1A

DMA Control Registers (these conform to the standard ADSP-2191 DMA descriptors)	SPT0	SPT1	SPT2
RX DMA Descriptor Pointer Register	0x0B00	0x0D00	0x0F00
RX DMA Configuration Register	0x0B01	0x0D01	0x0F01
RX DMA Page Register	0x0B02	0x0D02	0x0F02
RX DMA Address Register	0x0B03	0x0D03	0x0F03
RX DMA Count Register	0x0B04	0x0D04	0x0F04
RX DMA Next Descriptor Pointer Register	0x0B05	0x0D05	0x0F05
RX DMA Descriptor Ready	0x0B06	0x0D06	0x0F06
RX DMA IRQ Status	0x0B07	0x0D07	0x0F07
TX DMA Descriptor Pointer Register	0x0B80	0x0D80	0x0F80
TX DMA Configuration Register	0x0B81	0x0D81	0x0F81
TX DMA Page Register	0x0B82	0x0D82	0x0F82

ADSP-2191 Peripherals Used with the Simulator

DMA Control Registers (these conform to the standard ADSP-2191 DMA descriptors)	SPT0	SPT1	SPT2
TX DMA Address Register	0x0B83	0x0D83	0x0F83
TX DMA Count Register	0x0B84	0x0D84	0x0F84
TX DMA Next Descriptor Pointer Register	0x0B85	0x0D85	0x0F85
TX DMA Descriptor Ready	0x0B86	0x0D86	0x0F86
TX DMA IRQ Status	0x0B87	0x0D87	0x0F87

Input and Output

Each of the three serial ports has two standard VisualDSP++ data streams.

- SPT0: SPT0.TX, SPT0.RX
- SPT1: SPT1.TX, SPT1.RX
- SPT2: SPT2.TX, SPT2.RX

Serial Port (SPT) Peripheral

Serial Port Windows in VisualDSP++

The serial port registers appear in the Serial Port window.

To view these registers:

1. From the Registers menu, choose Peripherals.
2. Choose spt0, spt1, or spt2.

These windows provide a compact view of all the registers for each SPORT. You can access individual status and configuration register fields by creating custom register windows.

Unsupported Features

You can manipulate all the serial port configuration bits. Configuration related to frame synchs does not always have a significant impact on the simulation. For example, the FSR bits are used. The following, however, are not simulated:

- Whether the synch is internal or external
- Synch polarity
- Whether the synch is early or late

Serial clock and frame synch divisor registers are simulated. Therefore, they impact the duration required to transfer words.

Example

```
/*-----*/
/* The example performs a 32 word DMA receive after configuring */
/* the interrupt controller, the sport, and the dma engine.      */
/*-----*/

#define IRQCTL_PAGE      1

#define IPR0              0x200
#define IPR1              0x201
#define IPR2              0x202
#define IPR3              0x203

#define RSPEN             1          /* rx enable */
#define IRFS              0x200     /* use internal frame sync */
#define RFSR              0x400     /* rx frame sync required */
#define BITS16            0x1E0     /* slen of 16 bits */

#define SPO_IOPAGE        0x2

#define SPO_TX_CONFIG     0x200
#define SPO_RX_CONFIG     0x201
#define SPO_TX            0x202
#define SPO_RX            0x203
#define SPO_TSCLKDIV      0x204
#define SPO_RSCLKDIV      0x205
#define SPO_TFSDIV        0x206
#define SPO_RFSDIV        0x207
#define SPO_STAT          0x208

#define SPO_CURR_PTR_RX   0x300
#define SPO_CONFIG_DMA_RX 0x301
#define SPO_START_PG_RX   0x302
#define SPO_START_ADDR_RX 0x303
#define SPO_COUNT_RX      0x304
#define SPO_NEXT_DESCR_RX 0x305
```

Serial Port (SPT) Peripheral

```
#define SPO_DESCR_RDY_RX      0x306
#define SPO_IRQSTAT_RX       0x307

#define SPO_CONFIG_DMA_TX    0x381
#define SPO_MCMC1            0x219
#define SPO_MCMC2            0x21A

.GLOBAL      _main;
.GLOBAL      Start;
.GLOBAL      rcv_ddb;

/*****
/*          DMA Descriptor and Input Buffer Allocation/Init      */
*****/
.SECTION/data data1;
.var rcv_ddb[5]      = 0,0,0,0,0;          /* DMA Descriptor */

.var Input_Buffer[32] = 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;

/*****
/*          Interrupt Vector Table Entries                      */
*****/
.section/code IVreset;
        jump START;

.section/code IVint5;
        jump IVSR0;
```

ADSP-2191 Peripherals Used with the Simulator

```

/*****
/*          Main Routine                      */
*****/
.SECTION/code program;
START:
_main:
    call Setup_Interrupts;
    call Setup_SPORT0;
    call Setup_DMA;

wait_forever:
    jump wait_forever;

/*****
/*          Programming Interrupts          */
*****/
Setup_Interrupts:
    IRPTL = 0x0;
    ICNTL = 0x0;
    IMASK = 0x0;

    iopg=0x0;
    ax0=0x0008;
    io(0x80)=ax0;
    iopg=0x6;
    ax0=0x0040;
    io(0x202)=ax0;
    io(0x203)=ax0;
    io(0x204)=ax0;
    io(0x205)=ax0;
    io(0x206)=ax0;
    io(0x207)=ax0;
    IOPG = IRQCTL_PAGE;
    ar = 0xbb0b;
    io(IPR0) = ar;

```

Serial Port (SPT) Peripheral

```
    ar = 0xbbbb;
    io(IPR1) = ar;
    io(IPR2) = ar;
    io(IPR3) = ar;
    ENA int;                /* Enable Interrupts */
    rts;

/*****
/*          Programming For SPORT0          */
*****/
Setup_SPORT0:
/* RESET SPORT0 & DMA */
    IOPG = SPO_IOPAGE;
    ar = 0x0000;
    io(SPO_TX_CONFIG) = ar;
    io(SPO_RX_CONFIG) = ar;
    io(SPO_TX_CONFIG) = ar;
    io(SPO_RX_CONFIG) = ar;
    io(SPO_CONFIG_DMA_RX) = ar;    /* stop DMA */
    io(SPO_CONFIG_DMA_TX) = ar;
    io(SPO_MCMC1) = ar;
    ar = 3;
    io(SPO_IRQSTAT_RX) = ar;

/* Set SPORT0 frame sync divisor */
    ar = 0x0010;
    io(SPO_RFSDIV) = ar;

/* Set RFS required, Internal RFS, 16 bits receive */
    ar = (RFSR + IRFS + BITS16);
    io(SPO_RX_CONFIG) = ar;

/* SPORT0 interrupts enabled */
    AY0=IMASK;
    AY1=0x0010;
    AR = AY0 or AY1;            /* Enable sport0 rcv */
```

ADSP-2191 Peripherals Used with the Simulator

```
IMASK=AR;

ar = 0x0000;
io(SPO_MCMC1) = ar;

ar = 0x0000;
io(SPO_MCMC2) = ar;

RTS;

/*****
/*          DMA Controller Programming For SPORT0          */
*****/
Setup_DMA:

/* SPORT0 DMA DESCRIPTOR BLOCK RX */
    ar = 0x0082;                /* Set Direction*/
    io(SPO_CONFIG_DMA_RX) = ar;

    ar = rcv_ddb;                /* Next Descriptor Pntr Reg
*/
    dm(rcv_ddb + 4) = ar;

    ar = LENGTH(Input_Buffer);    /* DMA Memory Count */
    dm(rcv_ddb + 3) = ar;

    ar = Input_Buffer;            /* DMA Memory Address */
    dm(rcv_ddb + 2) = ar;

    ar = 0x0;                    /* DMA Memory Page */
    dm(rcv_ddb + 1) = ar;

    ar = 0x8007;                /* Enable DMA, interrupt */
    dm(rcv_ddb) = ar;

/* DMA CONFIG */
```

Serial Port (SPT) Peripheral

```
        ar = rcv_ddb;                                /*NEXT Descriptor */
        io(SPO_NEXT_DESCR_RX)=ar;

/* Signify DMA Descriptor Ready */
        ar=0x0001;
        io(SPO_DESCR_RDY_RX) = ar;                /* DMA Descriptor Ready */

        ar = 0x0003;
        io(SPO_CONFIG_DMA_RX) = ar;

        ax0 = io(SPO_RX_CONFIG);
        ar = setbit 0 of ax0;
        io(SPO_RX_CONFIG) = ar;

        RTS;

/*****
/*          Interrupt Service Routine for the DMA Receive          */
*****/
IVSR0:
        ay1 = IOPG;
        IOPG = SPO_IOPAGE;

        ar = 3;                                /* retire interrupts */
        io(SPO_IRQSTAT_RX)=ar;

        ar = 0x8007;                            /* Enable DMA, interrupt */
        dm(rcv_ddb) = ar;

        ar=0x0001;
        io(SPO_DESCR_RDY_RX) = ar;

        IOPG = ay1;
        rti;
```

Serial Port (UART) Peripheral

Overview of UART in the Simulator

The ADSP-2191 UART Port peripheral provides a simulation of UART port functionality. It supports all modes of operation, including Direct Memory Access (DMA) and interrupt driven modes for both transmit (TX) and receive (RX).

The Serial Port has the following registers to control its operation:

Configuration and Status	Real	Phantom (if different)
UART Transmit Hold Register (THR)	0x1400	
UART Divisor Latch (Low-Byte) (DLL)	0x1400	0x1408
UART Receive Buffer Register (RBR)	0x1400	0x1409
UART Interrupt Enable Register (IER)	0x1401	
UART Divisor Latch (High-Byte) (DLH)	0x1401	0x140A
UART Interrupt Identification Register (IIR)	0x1402	
UART Line Control Register (LCR)	0x1403	
UART Modem Control Register (MCR)	0x1404	
UART Line Status Register (LSR)	0x1405	
UART Modem Status Register (MSR)	0x1406	
UART Scratch Register (SCR)	0x1407	

Serial Port (UART) Peripheral

Note: The *Phantom addresses* appear in the VisualDSP++ IOM display. The registers are **not** addressable by user code at those locations.

DMA Control Registers (these conform to the standard 2191 DMA descriptors)	SPT0
RX DMA Descriptor Pointer Register	0x1500
RX DMA Configuration Register	0x1501
RX DMA Page Register	0x1502
RX DMA Address Register	0x1503
RX DMA Count Register	0x1504
RX DMA Next Descriptor Pointer Register	0x1505
RX DMA Descriptor Ready	0x1506
RX DMA IRQ Status	0x1507
TX DMA Descriptor Pointer Register	0x1580
TX DMA Configuration Register	0x1581
TX DMA Page Register	0x1582
TX DMA Address Register	0x1583
TX DMA Count Register	0x1584
TX DMA Next Descriptor Pointer Register	0x1585
TX DMA Descriptor Ready	0x1586
TX DMA IRQ Status	0x1587

Input and Output

The UART has two standard VisualDSP++ data streams:

- UART.TX
- UART.RX

UART Window in VisualDSP++

The serial port registers appear in the UART window.

To view these registers:

1. From the Registers menu, choose Peripherals.
2. Choose UART to open the UART window.

This window provides a compact view of all the registers for the UART. You can access individual status and configuration register fields by creating a custom register window.

Unsupported Features

You can manipulate all the UART configuration bits. Currently, you cannot simulate the data error (Framing Error, Parity Error, Break Interrupt) conditions or the Modem Status status bits (Data Carrier Detect, Ring Indicator, Data Set Ready, Clear To Send). You can specify Set Break in the Line Control Register, but this setting has no effect.

Serial Port (UART) Peripheral

Example

```
/*-----*/
/* The example performs a 32 word DMA receive after configuring */
/* the interrupt controller, the sport, and the dma engine.      */
/*-----*/

#define IRQCTL_PAGE      1

#define IPR0              0x200
#define IPR1              0x201
#define IPR2              0x202
#define IPR3              0x203

#define UART_IOPAGE       0x5
#define UART_BASE         0x1400

#define UART_THR           (0x1400-UART_BASE)
#define UART_RBR           (0x1400-UART_BASE)
#define UART_DLL           (0x1400-UART_BASE)
#define UART_IER           (0x1401-UART_BASE)
#define UART_DLH           (0x1401-UART_BASE)
#define UART_IIR           (0x1402-UART_BASE)
#define UART_LCR           (0x1403-UART_BASE)
#define UART_MCR           (0x1404-UART_BASE)
#define UART_LSR           (0x1405-UART_BASE)
#define UART_MSR           (0x1406-UART_BASE)
#define UART_SCR           (0x1407-UART_BASE)
#define UART_DMA_CURR_PTR_RX (0x1500-UART_BASE)
#define UART_DMA_CONFIG_RX  (0x1501-UART_BASE)
#define UART_DMA_START_PG_RX (0x1502-UART_BASE)
#define UART_DMA_START_ADDR_RX (0x1503-UART_BASE)
#define UART_DMA_COUNT_RX   (0x1504-UART_BASE)
#define UART_DMA_NEXT_DESCR_RX (0x1505-UART_BASE)
#define UART_DMA_DESCR_RDY_RX (0x1506-UART_BASE)
#define UART_DMA_IRQSTAT_RX (0x1507-UART_BASE)
#define UART_DMA_CURR_PTR_TX (0x1580-UART_BASE)
#define UART_DMA_CONFIG_TX  (0x1581-UART_BASE)
```

ADSP-2191 Peripherals Used with the Simulator

```
#define UART_DMA_START_PG_TX      (0x1582-UART_BASE)
#define UART_DMA_START_ADDR_TX    (0x1583-UART_BASE)
#define UART_DMA_COUNT_TX         (0x1584-UART_BASE)
#define UART_DMA_NEXT_DESCR_TX    (0x1585-UART_BASE)
#define UART_DMA_DESCR_RDY_TX     (0x1586-UART_BASE)
#define UART_DMA_IRQSTAT_TX       (0x1587-UART_BASE)

#define UART_LCR_DLAB              0x80
#define UART_LCR_BRK               0x40
#define UART_LCR_StickyParity      0x20
#define UART_LCR_EPS               0x10
#define UART_LCR_PEN               0x08
#define UART_LCR_STOP              0x04
#define UART_LCR_WLS_8             0x03
#define UART_LCR_WLS_7             0x02
#define UART_LCR_WLS_6             0x01
#define UART_LCR_WLS_5             0x00

.GLOBAL      _main;
.GLOBAL      Start;
.GLOBAL      rcv_ddb;

/*****
/*          DMA Descriptor and Input Buffer Allocation/Init          */
*****/
.SECTION/data data1;
.var rcv_ddb[5]      = 0,0,0,0,0;          /* DMA Descriptor */

.var Input_Buffer[32] = 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;

/*****
/*          Interrupt Vector Table Entries                          */
*****/
.section/code IVreset;
        jump START;
```

Serial Port (UART) Peripheral

```
.section/code IVint5;
    jump IVSR0;

/*****
/*      Main Routine                               */
*****/
.SECTION/code program;
START:
_main:
    call Setup_Interrupts;
    call Setup_UART;
    call Setup_DMA;

wait_forever:
    idle;
    jump wait_forever;
```

ADSP-2191 Peripherals Used with the Simulator

```

/*****
/*          Programming Interrupts          */
*****/
Setup_Interrupts:
    IRPTL = 0x0;
    ICNTL = 0x0;
    IMASK = 0x0;

    iopg=0x0;
    ax0=0x0008;
    io(0x80)=ax0;
    iopg=0x6;
    ax0=0x0040;
    io(0x202)=ax0;
    io(0x203)=ax0;
    io(0x204)=ax0;
    io(0x205)=ax0;
    io(0x206)=ax0;
    io(0x207)=ax0;
    IOPG = IRQCTL_PAGE;
    ar = 0xbbbb;
    ar = 0x0;
    io(IPR0) = ar;
    io(IPR1) = ar;
    io(IPR2) = ar;
    io(IPR3) = ar;
    ENA int;                      /* Enable Interrupts */
    rts;

```

Serial Port (UART) Peripheral

```

/*****
/*          Programming For UART          */
*****/
Setup_UART:
/* RESET UART & DMA */
    IOPG = UART_IOPAGE;
    /* Set LCR to enable access to the Divisor Latch registers */
    sr0=  UART_LCR_DLAB | UART_LCR_WLS_8 ;
    io( UART_LCR )=sr0 ;
    nop;
    sr0=  0x0001 ; // Set Baud rate to fastest available
    io( UART_DLL )=sr0 ;
    sr0=  0x0000 ;
    io( UART_DLH )=sr0 ;
    sr0=  UART_LCR_WLS_8 ; // Clear DLAB, keeping width at 8
    io( UART_LCR )=sr0 ;
    nop;
    sr0=  0x0000 ;
    io( UART_IER )=sr0 ; // Ensure IER (Interrupt Enable Register) is
clear

    // setup send and receive interrupt from core
    ax0=imask;
    ay0= 0x0010 ;
    ar=ax0 or ay0;
    imask=ar;
    ax0=imask;

    ay0= 0x8020 ;
    ar=ax0 or ay0;
    imask=ar;

    RTS;

```

ADSP-2191 Peripherals Used with the Simulator

```

/*****
/*          DMA Controller Programming For UART          */
*****/
Setup_DMA:

/* UART DMA DESCRIPTOR BLOCK RX */
    ar= 0x0082;                /* Set Direction*/
    io(UART_DMA_CONFIG_RX) = ar;

    ar = rcv_ddb;              /* Next Descriptor Pntr Reg
*/
    dm(rcv_ddb + 4) = ar;

    ar = LENGTH(Input_Buffer); /* DMA Memory Count */
    dm(rcv_ddb + 3) = ar;

    ar = Input_Buffer;         /* DMA Memory Address */
    dm(rcv_ddb + 2) = ar;

    ar = 0x0;                  /* DMA Memory Page */
    dm(rcv_ddb + 1) = ar;

    ar = 0x8007;               /* Enable DMA, interrupt */
    dm(rcv_ddb) = ar;

/* DMA CONFIG */
    ar = rcv_ddb;              /*NEXT Descriptor */
    io(UART_DMA_NEXT_DESCR_RX)=ar;

/* Signify DMA Descriptor Ready */
    ar=0x0001;
    io(UART_DMA_DESCR_RDY_RX) = ar;    /* DMA Descriptor
Ready */
    nop;

    ar= 0x0083;                /* Set Direction*/
    io(UART_DMA_CONFIG_RX) = ar;

```

Serial Port (UART) Peripheral

```
RTS;

/*****
/*      Interrupt Service Routine for the DMA Receive      */
*****/
IVSR0:
    ay1 = IOPG;
    IOPG = UART_IOPAGE;

    ar = 3;                      /* retire interrupts */
    io(UART_DMA_IRQSTAT_RX)=ar;

    ar = 0x8007;                 /* Enable DMA, interrupt */
    dm(rcv_ddb) = ar;

    ar=0x0001;
    io(UART_DMA_DESCR_RDY_RX) = ar;

    IOPG = ay1;

    rti;
```

Timer (TMR) Peripheral

Overview of TMR in the Simulator

The ADSP-2191 Timer peripheral provides general-purpose timer functionality. The ADSP-2191 has three Timer peripherals.

The simulator supports each of the three functional modes:

- PWM_OUT Mode – Pulse Width Modulation
- WDT_CAP Mode – Pulse Width and Period Capture
- EXT_CLK Mode – External Event Watchdog

Each Timer has one bi-directional pin, TMR_PIN, and one auxiliary module input pin, AUX_IN. The simulator does not, however, distinguish between these two pins (as described below).

The simulator supports the architected registers for each timer. These registers are:

- Config [15:0] – Configuration Register
- Width [31:0] – Pulse Width Register
- Period [31:0] – Pulse Period Register
- Counter [31:0] – Timer Counter

Global Status and Control

The simulator supports all statuses, described in the following table:

Bit	Status[15:0]	Global Status and Control
0	IRQ0	Timer 0 IRQ “Write one to clear” (also an output)
1	IRQ1	Timer 1 IRQ “Write one to clear” (also an output)
2	IRQ2	Timer 2 IRQ “Write one to clear” (also an output)
3	Reserved	Timer 3 (reserved)
4	OVF_ERR0	Timer 0 Counter overflow
5	OVF_ERR1	Timer 1 Counter overflow
6	OVF_ERR2	Timer 2 Counter overflow
7	Reserved	Timer 3 (reserved)
8	TIMEN0	Timer 0 – “Write one to set”
9	TIMEN0	Timer 0 – “Write one to clear”
10	TIMEN1	Timer 1 – “Write one to set”
11	TIMEN1	Timer 1 – “Write one to clear”
12	TIMEN2	Timer 2 – “Write one to set”
13	TIMEN2	Timer 2 – “Write one to clear”
14	Reserved	Timer 3 (reserved)
15	Reserved	Timer 3 (reserved)

ADSP-2191 Peripherals Used with the Simulator

In the following table, configuration bit 5 does not affect the simulator:

Bit	Config[15:0]	Timer Configuration Register	Simulator Semantics
1:0	MODE_FIELD	00 – Reset State - unused 01 – PWM_OUT Mode 10 – WIDTH_CAP Mode 11 – EXT_CLK Mode	
2	PULSE_HI	1 – Positive Active Pulse 0 – Negative Active Pulse	
3	PERIOD_CNT	1 – Count to end of Period 0 – Count to end of Width	
4	IRQ_ENA	1 – Interrupt Request Enable 0 – Interrupt Request Disable	
5	AUX_IN_SEL	1 – AUX_IN Select 0 – TMR_PIN Select	For input the simulator uses “Streams” as described below. When using Streams as input to the timers there is no distinction between AUX_IN and TMR_PIN
15:6	Unused		

Timer (TMR) Peripheral

Timer Signal Usage

The following signals are not recognized:

REGISTER	PWM_OUT Mode	WDTH_CAP Mode	EXT_CLK Mode
AUX_IN_SEL	Unused	1 – Select AUX_IN Input 0 – Select TMR_PIN input	Unused
TMR_AUX_IN	Unused	Auxiliary Input waveform	Unused
WAKE_UP			

Modes of Operation

The modes of operation are:

- PWM (Pulse Width Modulation)
- Width Capture
- External Clock

Timer with Streams Usage

WIDTH_CAP Mode

In Width Capture mode the Timer counts the number of clocks in both the width and period. The waveform that the timer reads is attached via the Streams dialog box in VisualDSP++.

You can attach a file to the following device names:

- `TIMER0_WDTH_CAP`
- `TIMER1_WDTH_CAP`
- `TIMER2_WDTH_CAP`

The format of the input file is as follows:

`PERIOD_COUNT`

`WIDTH_COUNT`

`PERIOD_COUNT`

`WIDTH_COUNT`

Whenever a timer is enabled in `WIDTH_CAP` mode, it reads two 32-bit values from the input file. The first is the number of pulses (clocks) in the period. The second is the number of pulses in the width.

If `PULSE_HI` is set, the timer delivers high widths and low periods. If `PULSE_HI` is not set, the timer delivers low widths and high periods.

Timer (TMR) Peripheral

Example – WDT_CAP Mode with Streams

Streams data file attached to TIMER0_WDT_CAP

0x2	← First period value
0x1	← First width value
0x4	← Second period value
0x2	← Second width value
0x8	← Third period value
0x4	← Third width value

Resulting waveforms, given:

MODE = WDT_CAP

PULSE_HI = 0

0 1	← First pulse chain
0 0 1 1	← Second pulse chain
0 0 0 0 1 1 1 1	← Third period value

ADSP-2191 Peripherals Used with the Simulator

```
// Set the Page to the TIMER page
iopg=0x05;

// Write to TIMERO config register
// IRQ_ENA      = 1
// PERIOD_CNT = 0
// PULSE_HI     = 0
// MODE_FIELD = 0x10 = WDT_CAP mode
sr0= 0x0012 ;
io( 0x0201 )=sr0 ;
nop;
cntr=0x0003;
DO LOOP0-1 UNTIL ce;
    nop;
    nop;

// Write to the global status register
// Enable TIMERO
sr0= 0x0100 ;
io( 0x0200 )=sr0 ;

// TIMERO, if connected via the STREAM interface
to the data file,
// will now deliver the pulses shown above

nop;
nop;

// Wait for interrupt
```

Timer (TMR) Peripheral

```
        idle(0);  
        nop;  
LOOP0:
```

External Clock Mode

The external clock is limited to 25MHz. Therefore, the simulator automatically divides the peripheral clock when a timer is enabled in external clock mode.

Memory DMA (MEMDMA) Peripheral

Overview of MEMDMA in the Simulator

The Memory DMA (MDMA) peripheral enables you to move data and instructions between internal memory and off-chip memory. The MDMA uses the ADSP-219x style of distributed DMA, whereby each DMA channel is located in the peripheral itself. The MDMA has dedicated Write and Read DMA channels. This DMA scheme is further based on *descriptors*, which describe the type of transfer to be carried out. These descriptors are managed in internal memory, and are downloaded to the MDMA.

For details about DMA in the ADSP-2191, refer to the *ADSP-219x/2191 Hardware Reference*.

Modes

Of the two DMA modes, Autobuf and Descriptor block, only Descriptor block is supported in the MEMDMA peripheral. This mode is set up as follows:

HEAD refers to the value of the current descriptor.

1. The DM memory writes HEAD+1,+2,+3,+4 to memory page 0.
2. The DM memory writes HEAD+0 to memory page 0. Bit 15 (ownership) must be set to 1.
3. The I/O memory writes the DescReady register of the respective DMA channel (if necessary).
4. The I/O memory writes the NXPTR register of the DMA channel (only to start the first descriptor).
5. The I/O memory writes the Config register setting DMAEn high (only to start the first descriptor).

Memory DMA (MEMDMA) Peripheral

Note:

- Burst mode is not simulated in the simulator. Therefore, values in memory show up sooner than they would appear in the hardware.
- Simulation of the MEMDMA is not cycle accurate.

Registers

The following table describes the Write Channel registers:

Address	Name	Description
0x900	Current PTR	Pointer of current descriptor being worked on
0x901	DMA Config DMA Enable (bit 0/1 bit) Direction (bit 1/1 bit) Interrupt On Completion (bit 2/1 bit) Data Type (bit 3/1 bit) DMA Buffer Clear (bit 7/1 bit) DMA Buffer Status (bit 12/2 bit) Ownership (bit 15/1 bit)	Enable Bit - Start Transfer Locked as 1 for Write Channel Generate Interrupt on Completion of transfer 16- or 24-bit transfer Clear Write channel FIFO DMA Buffer Status – 00 empty Descriptor Ownership 0 = DSP 1 = DMA
0x902	Start Page Start Page (bit 0/8 bits) Space (bit 8/1 bit)	Page of transfer 0 = mem 1 = boot
0x903	Start Address	Current Address of memory writes
0x904	DMA Count	Current Count of transfer
0x905	Next Descriptor	Location of next descriptor

ADSP-2191 Peripherals Used with the Simulator

0x906	Descriptor Ready	Indicates Descriptor is ready for loading
0x907	IRQSTAT	Status of Interrupt

The following table describes the Read Channel registers:

Address	Name	Description
0x980	Current PTR	Pointer of current descriptor being worked on.
0x981	DMA Config DMA Enable (bit 0/1 bit) Direction (bit 1/1 bit) Interrupt On Completion (bit 2/1 bit) Data Type (bit 3/1 bit) DMA Buffer Clear (bit 7/1 bit) DMA Buffer Status (bit 12/2 bit) Ownership (bit 15/1 bit)	Enable Bit - Start Transfer Locked as 0 for Read Channel Generate Interrupt on Completion of transfer 16- or 24-bit transfer Clear Write channel FIFO DMA Buffer Status – 00 empty Descriptor Ownership 0 = DSP 1 = DMA
0x982	Start Page Start Page (bit 0/8 bits) Space (bit 8/1 bit)	Page of transfer 0 = mem 1 = boot
0x983	Start Address	Current Address of memory writes
0x984	DMA Count	Current Count of transfer
0x985	Next Descriptor	Location of next descriptor
0x986	Descriptor Ready	Indicates Descriptor is ready for loading
0x987	IRQSTAT	Status of Interrupt

Memory DMA (MEMDMA) Peripheral

Note: MEMDMA does not generate errors in the transfer. No errors are generated as long as the registers are programmed correctly. Refer to the *ADSP-219x/2191 Hardware Reference* for detailed information.

Example – MEMDMA Transfer

```
////////////////////////////////////
////////////////////////////////////

// Analog Devices, Inc.                      All rights
reserved

// File:                                     run at 9:03 on 2/20/2001
////////////////////////////////////
////////////////////////////////////

.section/code IVreset;

JUMP start;

.section/code IVint4;
JUMP IVSR0;
.section/code IVint5;
JUMP ERR;
.section/code IVint6;
JUMP ERR;
.section/code IVint7;
JUMP ERR;
.section/code IVint9;
JUMP ERR;
.section/code IVint10;
JUMP ERR;
.section/code IVint11;
```

```
JUMP ERR;
.section/code IVint12;
JUMP ERR;
.section/code IVint13;
JUMP ERR;
.section/code IVint14;
JUMP ERR;
.section/code IVint15;
JUMP ERR;

/* main code begins here */

.section/code program;          !PROG
start: nop(0x70) ;
ax1 = iopg;
// Program the Interrupt Controller
//to send all interrupts to Interrupt 4 Vector 0x80
iopg=0x1;
ax0 = 0xbbbb ; io(0x200) = ax0;
ax0 = 0xbbbb ; io(0x201) = ax0;
ax0 = 0xbbbb ; io(0x202) = ax0;
ax0 = 0xb0bb ; io(0x203) = ax0;
iopg = ax1;
ena int;
ax0=0;
i3 =ax0; m3 =1; l3 =ax0; reg(b3) =ax0; my0 =ax0;
i7 =ax0; m7 =1; l7 =ax0; reg(b7) =ax0; mx0 =ax0;
nop;
nop;
```

Memory DMA (MEMDMA) Peripheral

```
nop;
nop;
nop;
dmpg1=0x01; //Write to Page 1
my0 = i3 ; i3 = 0x0000 ; nop ;
//Fill Mem with Values
ax0= 0x0001 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0002 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0004 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0008 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0010 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0020 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0040 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0080 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0100 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0200 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0400 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0800 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x1000 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x2000 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x4000 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x8000 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x7fff ; dm(i3 ,m3 )=ax0 ;
ax0= 0xbfff ; dm(i3 ,m3 )=ax0 ;
ax0= 0xdfff ; dm(i3 ,m3 )=ax0 ;
ax0= 0xefff ; dm(i3 ,m3 )=ax0 ;
ax0= 0xf7ff ; dm(i3 ,m3 )=ax0 ;
ax0= 0xfbff ; dm(i3 ,m3 )=ax0 ;
ax0= 0xfdff ; dm(i3 ,m3 )=ax0 ;
```

ADSP-2191 Peripherals Used with the Simulator

```
ax0= 0xfeff ; dm(i3 ,m3 )=ax0 ;
ax0= 0xff7f ; dm(i3 ,m3 )=ax0 ;
ax0= 0xffbf ; dm(i3 ,m3 )=ax0 ;
ax0= 0xffdf ; dm(i3 ,m3 )=ax0 ;
ax0= 0xffef ; dm(i3 ,m3 )=ax0 ;
ax0= 0xffff7 ; dm(i3 ,m3 )=ax0 ;
ax0= 0xffffb ; dm(i3 ,m3 )=ax0 ;
ax0= 0xffffd ; dm(i3 ,m3 )=ax0 ;
ax0= 0xffffe ; dm(i3 ,m3 )=ax0 ;
ax0= 0xa5a5 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x5a5a ; dm(i3 ,m3 )=ax0 ;
ax0= 0xc6c6 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0e0f ; dm(i3 ,m3 )=ax0 ;
dmpg1=0x00; //Write to Page 0
my0 = i3 ; i3 = 0x1000 ; nop ;
//Fill Descriptor block Values - Read
ax0= 0x8001 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0001 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0000 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0001 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x1005 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x8001 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0001 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0001 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0002 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x100a ; dm(i3 ,m3 )=ax0 ;
ax0= 0x8001 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0001 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0003 ; dm(i3 ,m3 )=ax0 ;
```

Memory DMA (MEMDMA) Peripheral

```
ax0= 0x0003 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x1010 ; dm(i3 ,m3 )=ax0 ;
my0 = i3 ; i3 = 0x1010 ; nop ;
//Fill Descriptor block Values - Read
ax0= 0x8001 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0001 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0006 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0004 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x1015 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x8001 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0001 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x000a ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0005 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x101a ; dm(i3 ,m3 )=ax0 ;
ax0= 0x8001 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0001 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x000f ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0006 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x1020 ; dm(i3 ,m3 )=ax0 ;
my0 = i3 ; i3 = 0x1020 ; nop ;
//Fill Descriptor block Values - Read
ax0= 0x8001 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0001 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0015 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0007 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x1025 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x8001 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0001 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x001c ; dm(i3 ,m3 )=ax0 ;
```

ADSP-2191 Peripherals Used with the Simulator

```
ax0= 0x0008 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x1000 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x8007 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0000 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x1060 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0001 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x1030 ; dm(i3 ,m3 )=ax0 ;
my0 = i3 ; i3 = 0x1030 ; nop ;
//Fill Descriptor block Values - Write
ax0= 0x8007 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0000 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x1061 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0002 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x1035 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x8007 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0000 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x1063 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0003 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x103a ; dm(i3 ,m3 )=ax0 ;
ax0= 0x8007 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0000 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x1066 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0004 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x1040 ; dm(i3 ,m3 )=ax0 ;
my0 = i3 ; i3 = 0x1040 ; nop ;
//Fill Descriptor block Values - Write
ax0= 0x8007 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0000 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x106a ; dm(i3 ,m3 )=ax0 ;
```

Memory DMA (MEMDMA) Peripheral

```
ax0= 0x0005 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x1045 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x8007 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0000 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x106f ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0006 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x104a ; dm(i3 ,m3 )=ax0 ;
ax0= 0x8007 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0000 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x1075 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0007 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x1050 ; dm(i3 ,m3 )=ax0 ;
my0 = i3 ; i3 = 0x1050 ; nop ;
//Fill Descriptor block Values - Write
ax0= 0x8007 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0000 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x107c ; dm(i3 ,m3 )=ax0 ;
ax0= 0x0008 ; dm(i3 ,m3 )=ax0 ;
ax0= 0x102a ; dm(i3 ,m3 )=ax0 ;
i3 = my0 ;
iopg=0x02;
//Give Location of Read Descriptor Block
sr0= 0x1000 ; io( 0x0185 )=sr0 ;
//Give Location of Write Descriptor Block
sr0= 0x102a ; io( 0x0105 )=sr0 ;
//Setup Interrupt
ax0=imask;
ay0= 0x0010 ;
ar=ax0 or ay0;
```

ADSP-2191 Peripherals Used with the Simulator

```
imask=ar;
iopg=0x02;
//Enable Read Channel
sr0= 0x0101 ; io( 0x0181 )=sr0 ;
//Enable Write Channel
sr0= 0x0103 ; io( 0x0101 )=sr0 ;
//Wait for Interrupt
JUMP_LABEL170:
jump JUMP_LABEL170;
nop;
jump EOF ;
EOF: nop;nop;nop;nop;nop;nop;
//Reached here everything is OK
END: nop(0xfe);nop(0x7f) ;
jump END;
nop;nop;nop;
ALU_COMP2:
ar=ax1 xor ay1;
if ne jump ERR;
ALU_COMP1:
ar=ax0 xor ay0;
if eq rts;
//Failure
ERR: nop;nop;nop;nop;nop;nop;
nop(0xff);nop(0x7f) ;
jump ERR;
nop;nop;nop;
nop;nop;nop;
```

Memory DMA (MEMDMA) Peripheral

```
//Test All Values of Transfer
IVSR0:
iopg=0x02;
sr0= 0x0001 ; io( 0x0107 )=sr0 ;
sr0= 0x0001 ; io( 0x0187 )=sr0 ;
dmpg1=0x00;
my0 = i3 ; i3 = 0x1083 ; nop ;
ax0=dm(i3 ,m3 ) ;
i3 = my0 ;
ay0= 0x0e0f ; ar=ax0 xor ay0; ax0=ar ;
ar=pass ax0; if ne jump JUMP_LABEL5 ;
my0 = i3 ; i3 = 0x1060 ; nop ;
ax0=dm(i3 ,m3 ); ay0= 0x0001 ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0x0002 ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0x0004 ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0x0008 ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0x0010 ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0x0020 ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0x0040 ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0x0080 ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0x0100 ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0x0200 ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0x0400 ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0x0800 ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0x1000 ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0x2000 ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0x4000 ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0x8000 ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0x7fff ; call ALU_COMP1 ;
```

ADSP-2191 Peripherals Used with the Simulator

```
ax0=dm(i3 ,m3 ); ay0= 0xbfff ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0xdfff ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0xefff ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0xf7ff ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0xfbff ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0xfdff ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0xfeff ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0xff7f ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0xffbf ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0xffdf ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0xffef ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0xffff7 ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0xffffb ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0xffffd ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0xffffe ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0xa5a5 ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0x5a5a ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0xc6c6 ; call ALU_COMP1 ;
ax0=dm(i3 ,m3 ); ay0= 0x0e0f ; call ALU_COMP1 ;
i3 = my0 ;
jump EOF ;
JUMP_LABEL5:
rti;
nop;nop;nop;
nop;nop;nop;
```

Memory DMA (MEMDMA) Peripheral