Freescale Semiconductor
Application Note

Document Number: AN3020
Rev. 0, 09/2005

Configuring and Using the SPI
MC9328MX1, MC9328MXL, and MC9328MXS

by: David Babin

1 Abstract

The i.MX Serial Peripheral Interface (SPI) is a configurable, synchronous port that is full duplex. This document covers some of the configuration options, information about register programming, and the timing waveforms that result. Packet transmissions of up to 128 bits are described.

Both the i.MX1 and i.MXL have two SPI ports. You can configure SPI-1 as a master or slave, while SPI-2 is always a master. The i.MXS has a single SPI port, which can be either a master or slave.

This document applies to the following devices, collectively called i.MX throughout this document:

- MC9328MX1
- MC9328MXL
- MC9328MXS

Contents

2 Transmitting Data in Packets of 16 Bits or Less

The 8×16-bit FIFO makes it a straightforward operation to transmit data in packets of 16 bits or less. This topic contains examples of data transmissions that consist of a maximum of eight packets, with 16 bits or less in each packet. If you want to send more than eight packets contiguously, use watermarks. Watermarks are used to control the flow of data to the TX FIFO—avoiding FIFO underrun and overrun. The INTREG and DMAREG use watermarks with the *empty*, *half*, and *full* nomenclature.

With SPI configured in master mode, load the TX FIFO with the number of packets to send, then set the XCH bit in the CONTROLREG to a value of 1. On-chip hardware then sends out the FIFO contents of a maximum of eight packets.

Figure 1, Figure 2, and Figure 3 illustrate the results of successively loading the TX FIFO one, two, and eight times before you initiate transmission. Pseudo code examples are included with all three figures. Once the FIFO load is complete, you must set the XCH bit. In the examples, each FIFO load consists of 8 bits. For simplification, Example 3 (the eight-packet pseudo code) loads the TX FIFO by using discrete successive write operations. In practice, you typically use an array to accomplish this goal.

Use the PERIODREG to determine the interval between successive packets. Figure 4 and Figure 5 illustrate the effect of configuring the period counter in the PERIODREG to use the SPI data clock. With an alternate configuration, the counter can use a 32 or 32.768 kHz clock.

In Figure 4 and Figure 5, the TX FIFO is loaded four times, then an exchange command is issued. $\overline{\text{SS}}$ is programmed to toggle between transfers. An exchange command is issued by setting the CONTROLREG XCH bit value to high.

Example 1. Pseudo Code for One Data Transfer of an 8-Bit Packet

```
RESETREG = 0x00000001     //reset SPI
RESETREG = 0
CONTROLREG = 0x00000400   //flush FIFO and configure in master mode
CONTROLREG = 0x0000E647   //set data rate, enable SPI, master, 8-bit transfer
                          //(alter for 1 to 16 bits)
INTREG = 0                //default
TESTREG = 0               //default
PERIODREG = 0             //default
DMAREG = 0                //default
TXDATAREG = (1 to 16 bits of data; 8 bits for the example shown in the following figure)
CONTROLREG = 0x0000E747   //initiate data exchange

// Check that transmission is completed by monitoring TESTREG TXCNT bits
// or CONTROLREG XCH bit.
// Include watchdog in case transmission never completes.
```

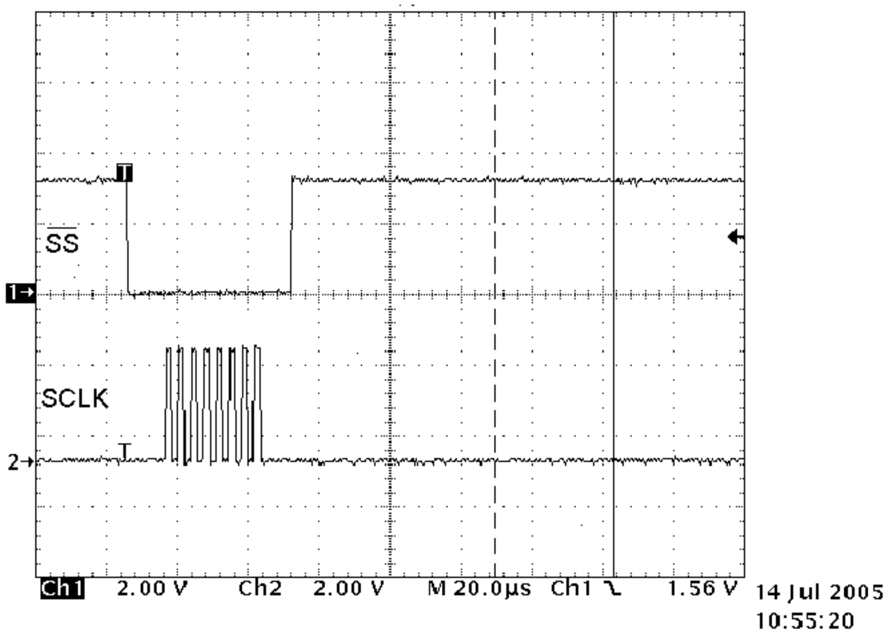

Figure 1. One-Byte Transfer (as described in previous code)

Example 2. Pseudo Code for Two Data Transfers of 8 Bits per Packet

```
RESETREG = 0x00000001     //reset SPI
RESETREG = 0
CONTROLREG = 0x00000400   //flush FIFO and configure in master mode
CONTROLREG = 0x0000E647   //set data rate, enable SPI, master, 8-bit transfer
                          //(alter for 1 to 16 bits)
INTREG = 0                //default
TESTREG = 0               //default
PERIODREG = 0             //default
DMAREG = 0                //default
TXDATAREG = (first 1-bit to 16-bit data packet; 8 bits for the example shown in Figure 2)
TXDATAREG = (second 1-bit to 16-bit data packet; 8 bits for the example)
CONTROLREG = 0x0000E747   //initiate data exchange

// Check that transmission is completed by monitoring TESTREG TXCNT bits or
// CONTROLREG XCH bit.
// Include watchdog in case transmission never completes.
```

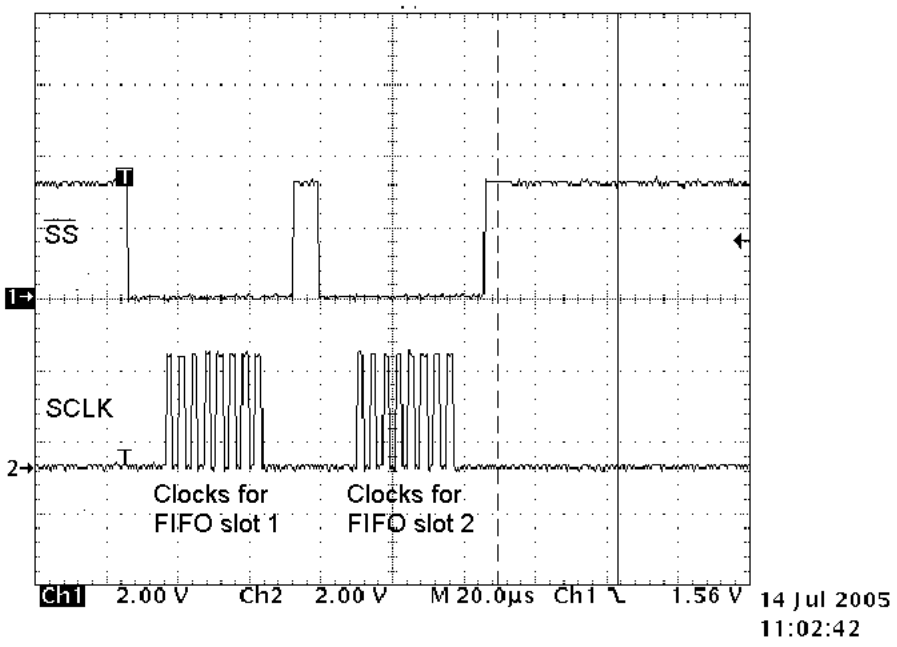

Figure 2. Two-Byte Transfer (as described in previous code)

Example 3. Pseudo Code for Eight Data Transfers of 8 Bits per Packet

```
RESETREG = 0x00000001     //reset SPI
RESETREG = 0
CONTROLREG = 0x00000400   //flush FIFO and configure in master mode
CONTROLREG = 0x0000E647   //set data rate, enable SPI, master, 8-bit transfer
                          //(alter for 1 to 16 bits)
INTREG = 0                //default
TESTREG = 0               //default
PERIODREG = 0             //default
DMAREG = 0                //default
TXDATAREG = (first 1-bit to 16-bit data packet; 8 bits for the example shown in Figure 3)
TXDATAREG = (second 1-bit to 16-bit data packet; 8 bits for the example)
TXDATAREG = (third 1-bit to 16-bit data packet; 8 bits for the example)
TXDATAREG = (fourth 1-bit to 16-bit data packet; 8 bits for the example)
TXDATAREG = (fifth 1-bit to 16-bit data packet; 8 bits for the example)
TXDATAREG = (sixth 1-bit to 16-bit data packet; 8 bits for the example)
TXDATAREG = (seventh 1- to 16-bit data packet; 8 bits for the example)
TXDATAREG = (eighth 1-bit to 16-bit data packet; 8 bits for the example)
CONTROLREG = 0x0000E747   //initiate data exchange

// Check that transmission is completed by monitoring TESTREG TXCNT bits
// or CONTROLREG XCH bit.
// Include watchdog in case transmission never completes.
```

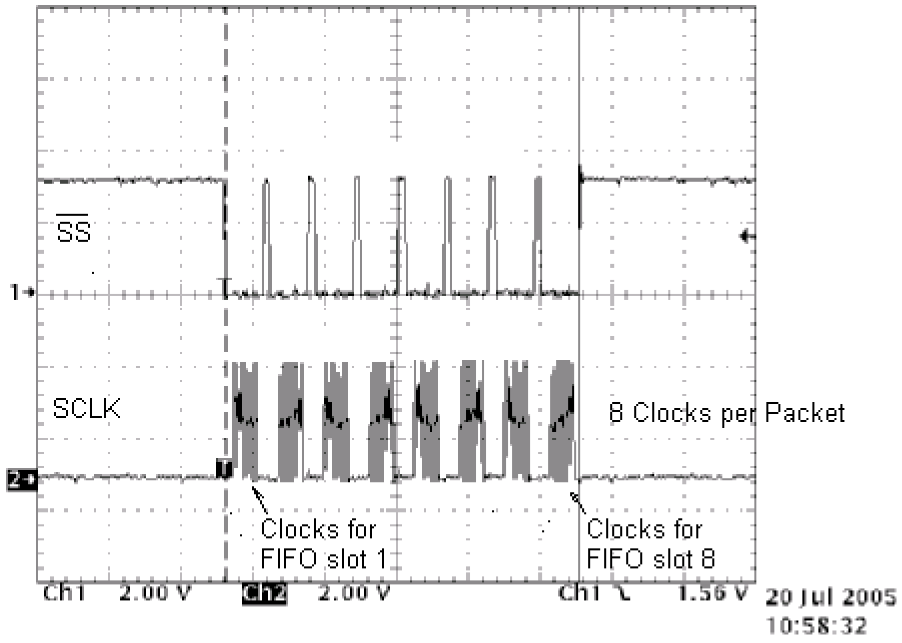

Figure 3. Eight-Byte Transfer (as described in previous code)

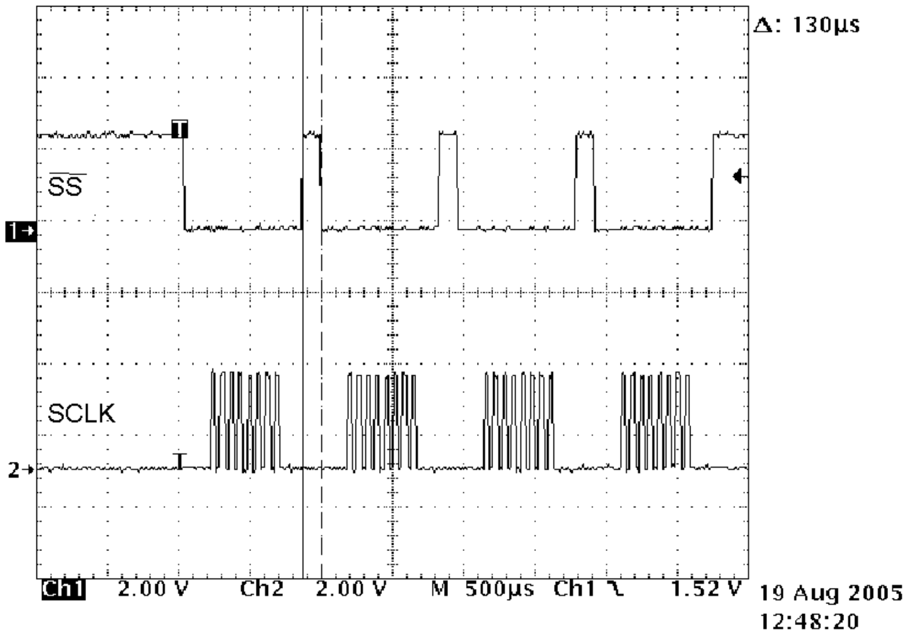

Figure 4. Four Bytes with PERIODREG Wait = 0

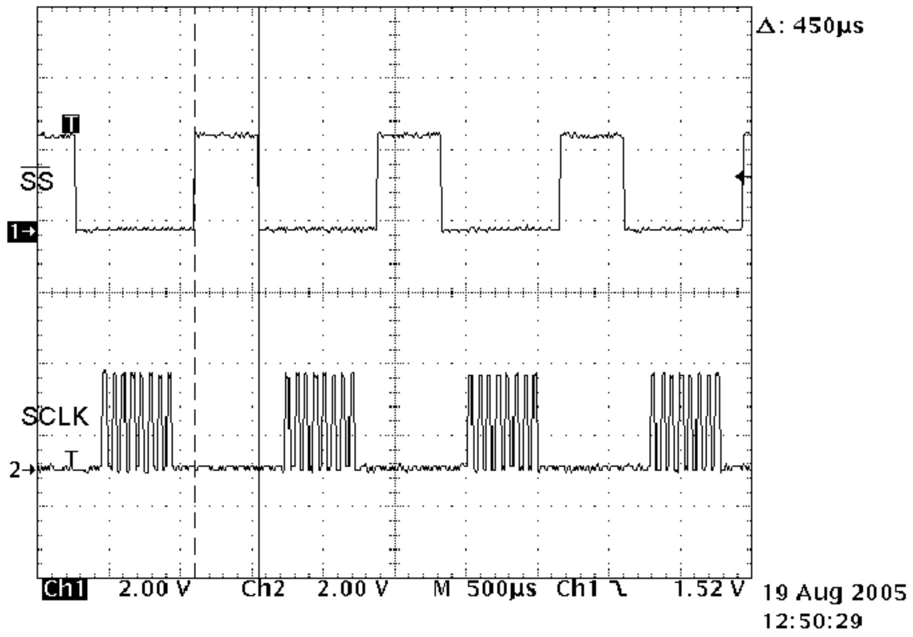

Figure 5. Four Bytes with PERIODREG Wait = 5

3 Transmitting Data in Packets of More Than 16 Bits

To handle transactions of more than 16 bits, you must properly configure the SPI module and load the TX FIFO with the number of bits to transfer in a single packet. This requires more extensive software than handling lower bit payloads because, in most cases, you cannot use a single exchange instruction after the TX FIFO is completely filled. The only exception to this general rule is the transmission of a 128-bit packet. (The FIFO is 8×16 bits = 128 bits.)

The code in Example 4 shows pseudo software for four 32-bit packet transmissions. Data is parsed from an array of four 32-bit words. Parsing is required because the TX FIFO is 8×16 bits.

The routine in Example 4 loads the TX FIFO with two parsed half words, then sends the FIFO contents. This process is repeated four times to send the four packets. Associated $\overline{\text{SS}}$ (Slave Select *bar*) and SCLK (SPI Clock) waveforms are shown in Figure 6 and Figure 7.

Configurations for data transfers of 20–128 bits are shown in Table 1.

Example 4. Pseudo Code for Four Data Transfers of 32 Bits per Packet

```
RESETREG = 0x00000001     //reset SPI
RESETREG = 0
CONTROLREG = 0            //flush FIFO
INTREG = 0                //default
TESTREG = 0               //default
CONTROLREG = 0x0000060F   //SCLK = PERCLK2 divided by 4, enable SPI, master,
                          //“16-bit” transfer
PERIODREG = 0             //no pause between 16-bit transfers of a 32-bit transmission
DMAREG = 0                //default; not used
for (count = 0; count < 4; count++)
{
TXDATAREG = xmitdata[count]>>16    //parse upper 16 bits; store to TX FIFO
TXDATAREG = xmitdata[count]        //lower 16 bits of transmission sent to FIFO
                                   //(upper 16 bits automatically ignored by SPI)
CONTROLREG |= 0x100                //initiate data exchange
int txcnt = 9                      //initialize txcnt to value unequal to zero
while (txcnt != 0) txcnt = TESTREG & 0xF    //wait for 32-bit transmission to complete
}

// Include watchdog in case transmission never completes.
```

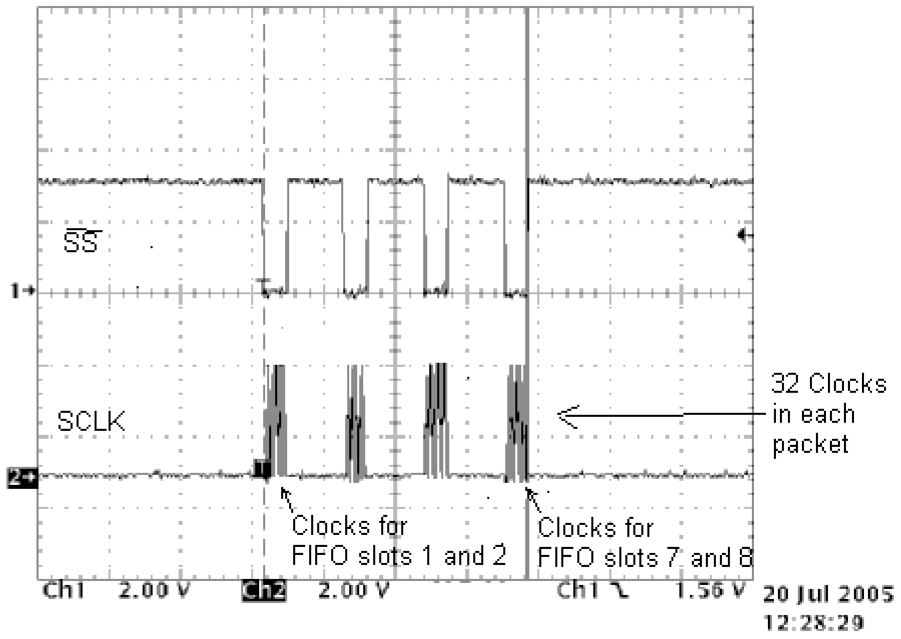

Figure 6. Four 32-Bit Transfers (as described in Example 4 code)

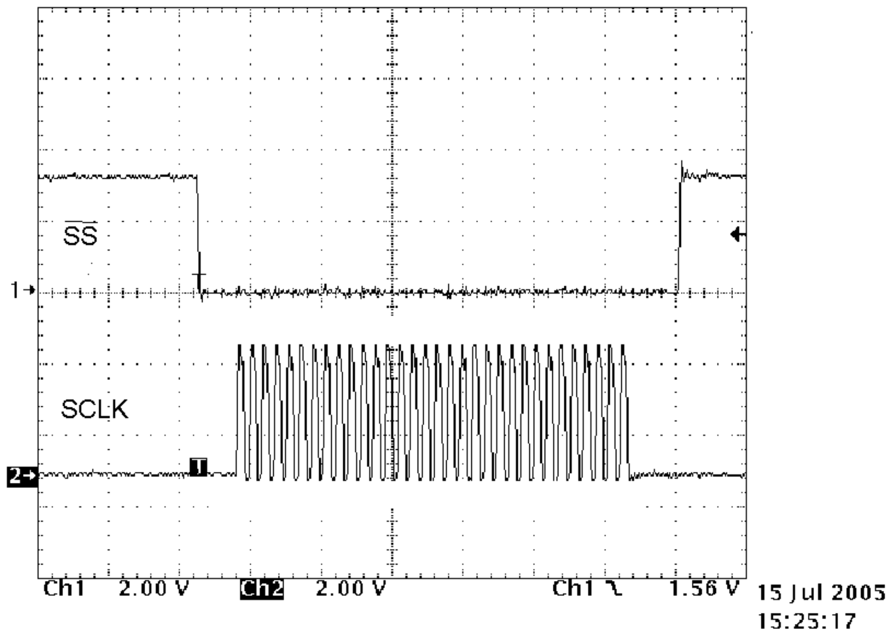

Figure 7. Enlarged View of One-of-Four 32-Bit Packets (as described in Example 4 code)

Table 1. Example Configurations for Packets Over 16 Bits

Packet Size (bits)	SPI CONTROLREG BIT_COUNT Bits [3:0]	Required Consecutive TX FIFO Loads
20	1001 → 10-bit transfer	2
24	1011 → 12-bit transfer	2
32	1111 → 16-bit transfer	2
48	1111 → 16-bit transfer	3
64	1111 → 16-bit transfer	4
128	1111 → 16-bit transfer	8

3.1 Using the TX FIFO with Larger Data Packets

To handle larger data packets, the FIFO restriction of being 16 bits wide demands a special approach. In the preceding pseudo code example, each of four 32-bit data packets is parsed into 16-bit groups. Next, two 16-bit groups are loaded into the TX FIFO, the transmission is initiated, the next two 16-bit groups are loaded, and the next transmission is initiated. The process continues until all four 32-bit packets are sent. The array can be expanded to fit the needs of the application.

3.2 Configuring Slave Select Output for Larger Data Packets

To send more than 16 bits in a packet, you must configure the Slave Select output so it does not toggle between transfers. To prevent toggling between transfers, set the CONTROLREG bit 6 to a value of zero (0). If bit 6 does not have a value of 0, the $\overline{\text{SS}}$ toggles each time a FIFO location is transmitted.

As previously described, the code in Example 4 transfers four data packets of 32 bits each. For this situation, set the bit count to 16 bits. Note that 32 bits are sent in each packet because the TX FIFO is loaded with 32 bits before transmission is initiated.

3.3 Setting the Transmit Wait Interval Between FIFO Bursts

Figure 8, Figure 9, Figure 10, and Figure 11 show the effect of setting the Wait bit values in the PERIODREG to values of 0, 1, 2, and 3, respectively. For these examples, $\overline{SS}$ bar is programmed to not toggle between FIFO slot transfers. For 32-bit packets, you typically program the PERIODREG bit to a value of 0 (zero).

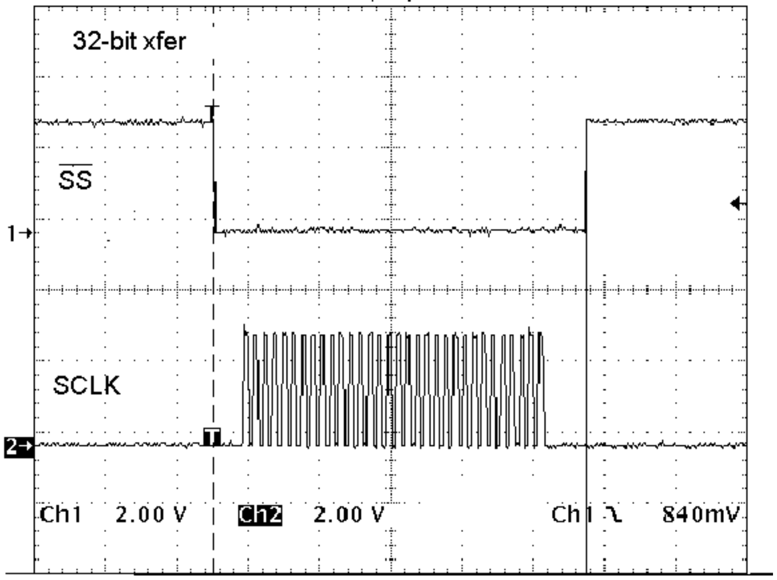

Figure 8. PERIODREG Wait = 0

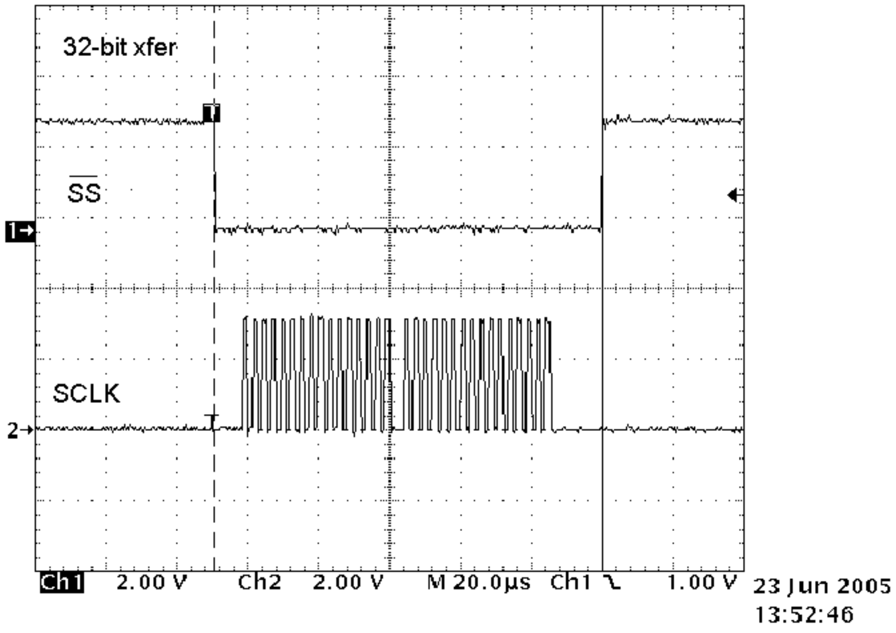

Figure 9. PERIODREG Wait = 1

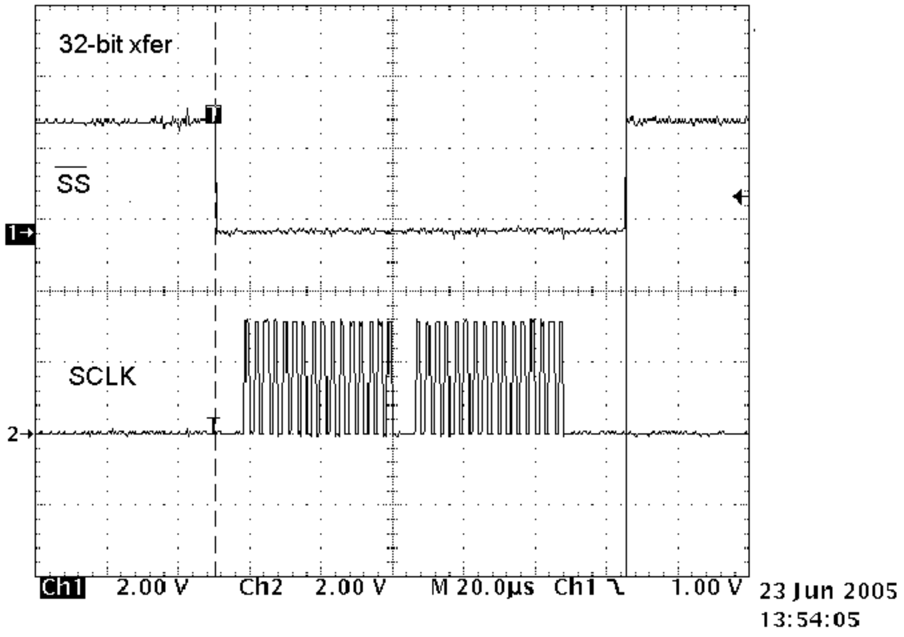

Figure 10. PERIODREG Wait = 2

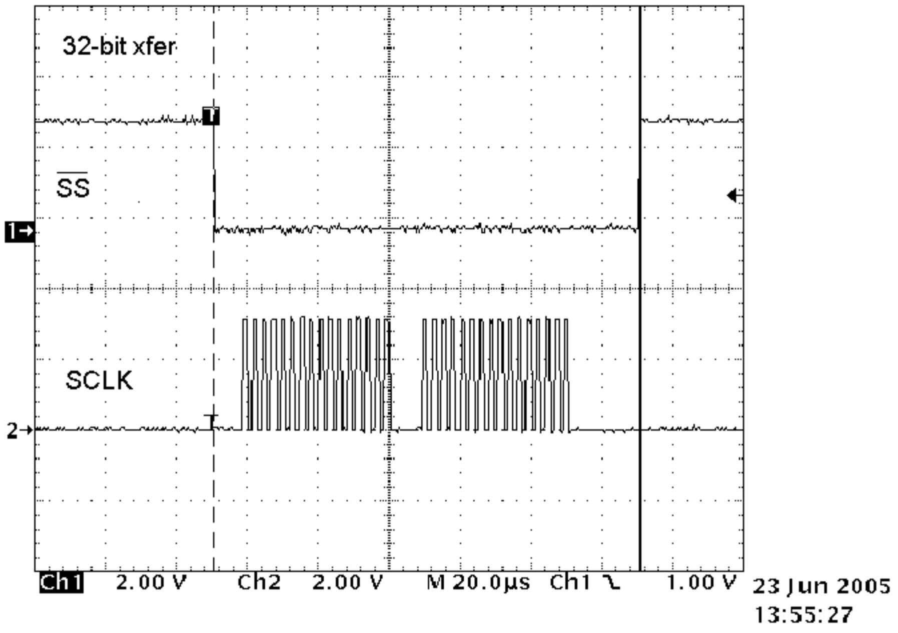

Figure 11. PERIODREG Wait = 3

4 Setting the Transmit Clock Rate

The TX SCLK rate is determined by the System PLL frequency and the programming of associated dividers. The clock rate directly determines the data rate because the SPI is synchronous. The drive strength setting must be commensurate with the SPI output loading and data rate.

4.1 Setting Up Dividers

The interaction between the PLL and the SPI module dividers is illustrated in Figure 12. Table 2 shows several divider programming combinations and the SCLK speeds these combinations produce.

PERCLK2, which supplies the master clock to the SPI module, also clocks the LCD Controller (for i.MX1, i.MXL, and i.MXS) and the MMC/SD module (for i.MX1and i.MXL). It is essential to select a frequency that satisfies the requirements of these modules as well as the SPI's requirements. Each module contains separate dividers, which allow some flexibility for selecting the PERCLK2 frequency.

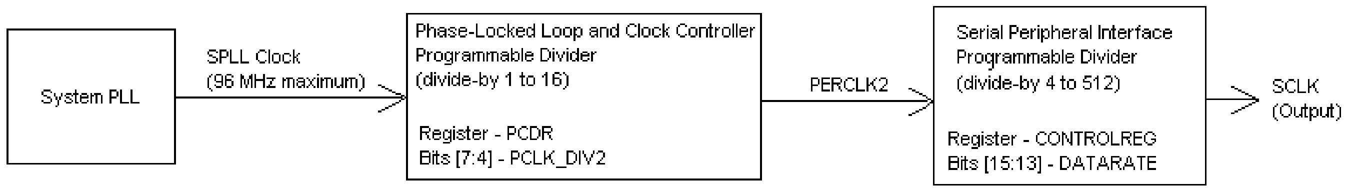

Figure 12. SPI SCLK Derivation

Table 2. Divider Programming and SCLK Frequency Examples

System PLL Output Frequency	Phase-Locked Loop and Clock Controller: Register – PCDR Bits [7:4] – PCLK_DIV2	PERCLK2 Frequency (MHz)	Serial Peripheral Interface: Register – CONTROLREG Bits [15:13] – DATARATE	Resulting SCLK Frequency
48 MHz	1001 → divide-by 10	4.8 MHz	011 → divide-by 32	150 kHz
96 MHz	0011 → divide-by 4	24 MHz	010 → divide-by 16	1.5 MHz
96 MHz	0001 → divide-by 2	48 MHz	001 → divide-by 8	6 MHz
80 MHz	0000 → divide-by 1	80 MHz	001 → divide-by 8	10 MHz

4.2 Setting the Drive Strength

Drive strength is the amount of current the i.MX SPI outputs can sink or source. The SPI output loading and data rate determine the drive strength required. To control the drive strength, use the values of bits 10 and 11 in the Global Peripheral Control Register (GPCR).

Be sure the drive strength is not set higher than necessary. Overly high drive strength results in higher power consumption and can cause excessive ringing on the SPI lines. A good method of checking waveform integrity is to examine the SPI lines in a system with an oscilloscope.

5 Using the Handshaking Option

Master mode has a provision for using an external signal to start a data transfer. Example 5 shows code for SPI_RDY to trigger such a transaction, and the transaction that results is illustrated in Figure 13. The selected configuration for SPI_RDY is “active low-level triggers input.” (The edge-triggered mode is an alternative.) A one-byte transfer is shown in the example, but SPI_RDY can handle any SPI transaction.

SPI_RDY is an asynchronous signal, so its response time varies. Figure 13 shows a response time of 600 ns. Figure 14 shows response times for several SPI_RDY-triggered events with the same setup used for each event. The configuration used to produce the results in Figure 14 is the code in Example 5 with a system PLL frequency of 96 MHz.

Example 5. Pseudo Code for One Data Transfer of 8 bits Initiated by SPI_RDY

```
RESETREG = 0x00000001     //reset SPI
RESETREG = 0
CONTROLREG = 0x00000400   //flush FIFO and configure in master mode
CONTROLREG = 0x00009607   //set data rate, allow level-triggered SPI_RDY, enable SPI, master,
                          //8-bit transfer
INTREG = 0                //default
TESTREG = 0               //default
PERIODREG = 0             //default
DMAREG = 0                //default
TXDATAREG = (data word, 8 bits for this example; upper 24 bits ignored by SPI)
CONTROLREG = 0x00009707   //SPI is now ready for data exchange; wait for SPI_RDY

// Check that transmission is completed by monitoring TESTREG TXCNT bits or
// CONTROLREG XCH bit.
// Include watchdog in case transmission never completes.
```

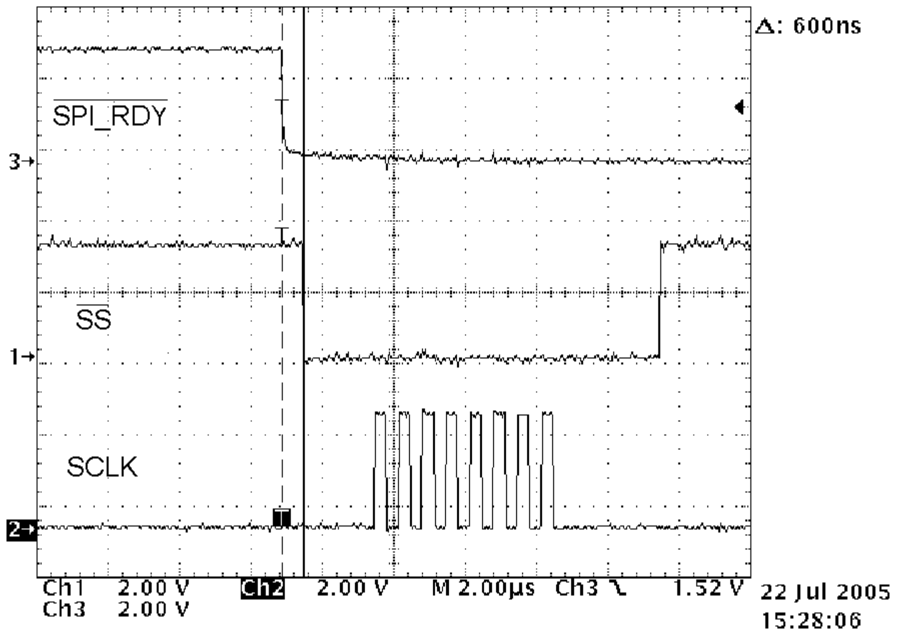

Figure 13. SPI _RDY Used to Initiate a Transfer

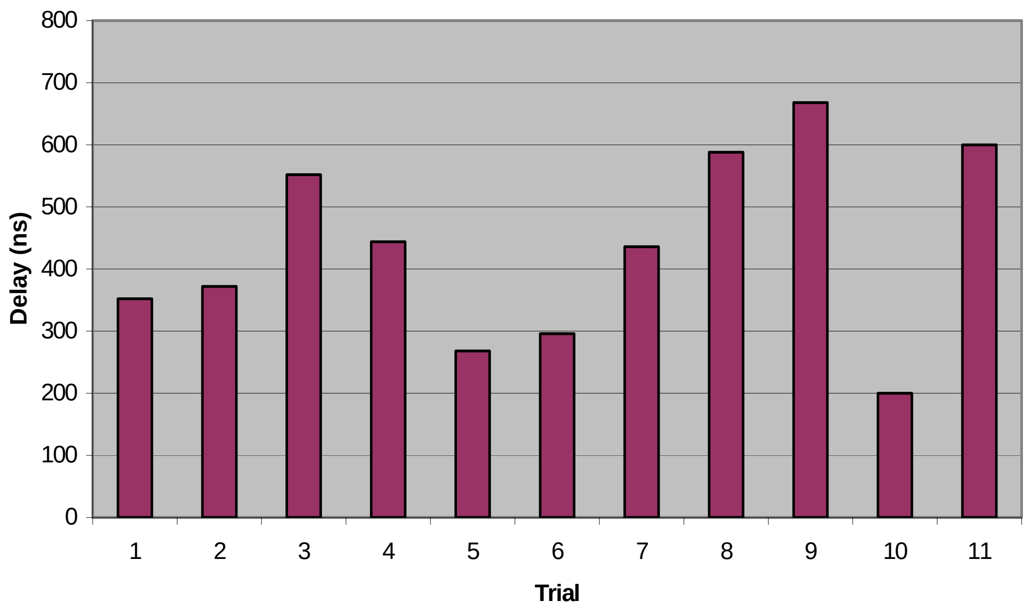

Figure 14. Variation of Transmission Delay for Several Events Triggered by SPI_RDY

6 Conclusion

The i.MX SPI module is a versatile synchronous serial port with a programmable clocking rate. The on-chip hardware makes it easy to handle data transfers of 1–16 bits per packet. Packets that are larger than 16 bits can be handled with a minimal amount of additional software.

7 References

For more information about the i.MX devices, refer to the following documents:

- Data Sheet MC9328MXL
- Data Sheet MC9328MX1
- Data Sheet MC9328MXS
- Reference Manual MC9328MXLRM
- Reference Manual MC9328MX1RM
- Reference Manual MC9328MXSRM
- ADS Schematic M9328MX1_L_ADS_V2_0

To find the latest technical documentation about i.MX on the Web, go to:

www.Freescale.com → Wireless (under Applications) → Apps Processors

Click the link for the device that interests you, then click the link for the i.MX Product Summary Page.

NOTES

Document Number: AN3020
Rev. 0
09/2005