Table 1: Cross Reference of Applicable Products

PRODUCT NAME	MANUFACTURER PART NUMBER	SMD #	DEVICE TYPE	INTERNAL PIC NUMBER
Arm Cortex M0+	UT32M0R500	5962-17212	SPI Unit	QS30

1.0 Overview

The UT32M0R500 contains one Serial Peripheral Interface (SPI) controller. SPI is a synchronous serial communications interface consisting of four wires. SPI is full-duplex--data can be transmitted and received by using separate lines. The UT32M0R500 SPI only works as SPI master using Motorola SPI protocol. SPI applications range from memory SD cards to 8bit shift registers, i.e., 74HC595. The latter is used for illustration in this application note.

To interface to a slave device, the SPI master controller follows the next steps:

1) Pull chip select (CSn) low, to activate a particular slave device.
2) Drive the clock (SCLK) to synchronize communication.
3) Send data on MOSI and receive data on MISO.
4) Pull CSn high to deactivate particular slave device.

Figure 1 shows the SPI master/slave interface and the 4-wire out/in direction.

Figure 1: SPI master/slave communication

2.0 Application Note Layout

This application note (AN) provides a brief description of the SPI unit's memory map, configuration and programming.

3.0 SPI Unit Hardware

The SPI Unit is mapped to the memory region from 0x4000_6000 to 0x4000_6FFF. It has 22 registers plus 32 data registers. For more information on each register, refer to **Chapter 12** of the UT32R500 Functional Manual.

3.1 SPI Unit Control Register 0

The Control Register 0 (**CTRLR0**) sets the SPI clock polarity (SCPOL), bit [7], to either high or low; the SPI clock phase (SCPH), bit [6], to toggle data in the middle or start of the first data bit; SPI data size to 3-15 (DFS), bits [3:0] +1, with default set to 8 bits. SPI Mode (TMOD), bits [9:8] has four different modes, with Transmit and Receive as default, table 2 list all the modes.

Table 2. SPI modes

MODE	MODE Value
Transmit & Receive	00
Transmit Only	01
Receive Only	10
EEPROM Read	11

3.2 SPI SSI Enable Register

The SSI Enable Register (**SSIENR**), bit [0], enables and disables all SPI master operations. When it is disabled, all serial transfers are halted immediately, transmit and receive FIFO buffers are cleared, and after a short delay, the SCLK is disabled for saving power consumption in the system.

3.3 SPI Slave Enable Register

The Slave Enable Register (**SER**), bits [2:0], corresponds to a slave select line respectively. When the master pulls the corresponding CSn line low, serial data begins to be transferred. When not operating in broadcast mode, only one bit in this field should be set.

3.4 SPI Baud Rate Select Register

The Baud Rate Select Register (**BAUDR**) sets the clock divider value (SCKDV), bits [15:0]. If the value is 0, the serial output clock (SCLK) is disabled. The frequency of the SCLK is derived by dividing SCKDV from the system clock to any even value between 2 and 65534.

3.5 SPI Transmit FIFO Threshold Level Register

SPI Transmit FIFO Threshold Level Register (**TXFTLR**). When the number of transmit FIFO entries is less than or equal to this value, the transmit FIFO empty interrupt is triggered.

3.6 SPI Receive FIFO Threshold Level Register

SPI Receive FIFO Threshold Level Register (**RXFTLR**). When the number of receive FIFO entries is greater than or equal to this value + 1, the receive FIFO full interrupt is triggered.

3.7 SPI Transmit FIFO Level Register

SPI Transmit FIFO Level Register (**TXFLR**) contains the number of valid data entries in the transmit FIFO.

3.8 SPI Receive FIFO Level Register

SPI Transmit FIFO Level Register (**RXFLR**) contains the number of valid data entries in the receive FIFO memory. This register can be read at any time.

3.9 SPI Interrupt Mask Register

SPI Interrupt Mask Register (**IMR**) is read/write and masks or enables all interrupts generated by the SPI master. Masking is accomplished by clearing a bit to value '0'.

3.10 SPI Raw Interrupt Status Register

SPI Interrupt Mask Register (**RISR**) is read-only for reading the status of the mask interrupts; bit values of 1 are for active interrupts.

3.11 SPI Transmit FIFO Overflow Interrupt Clear Register

SPI Clear Transmit FIFO Overflow Interrupt (**TXOICR**) reflects the status of the interrupt. A read from this register clears the ssi_txo_intr interrupt; writing has no effect.

3.12 SPI Receive FIFO Overflow Interrupt Clear Register

SPI Clear Receive FIFO Overflow Interrupt (**RXOICR**) reflects the status of the interrupt. A read from this register clears the ssi_rxo_intr interrupt; writing has no effect.

3.13 SPI Receive FIFO Underflow Interrupt Clear Register

SPI Clear Receive FIFO Underflow Interrupt (**RXUICR**) reflects the status of the interrupt. A read from this register clears the ssi_rxu_intr interrupt; writing has no effect.

3.14 SPI Multi-Master Interrupt Clear Register

SPI Clear Multi-Master Contention Interrupt (**MSTICR**) reflects the status of the interrupt. A read from this register clears the ssi_mst_intr interrupt; writing has no effect.

3.15 SPI Interrupt Clear Register

SPI Clear Register (**ICR**) is set if any of the interrupts are active. A read clears the ssi_txo_intr, ssi_rxu_intr, ssi_rxo_intr, and the ssi_mst_intr interrupts. Writing to this register has no effect.

4.0 SPI Unit Initialization

The SCLK clock frequency is derived by the following equation:

$$fSCLK = \frac{fCLK(system\ clock)}{Clock\ Divider(SCKDV)}$$

Once the frequency is set, there are four possible clock modes available for programming the clock edge used for data sampling and data toggling.

Figure 2 shows the four different modes.

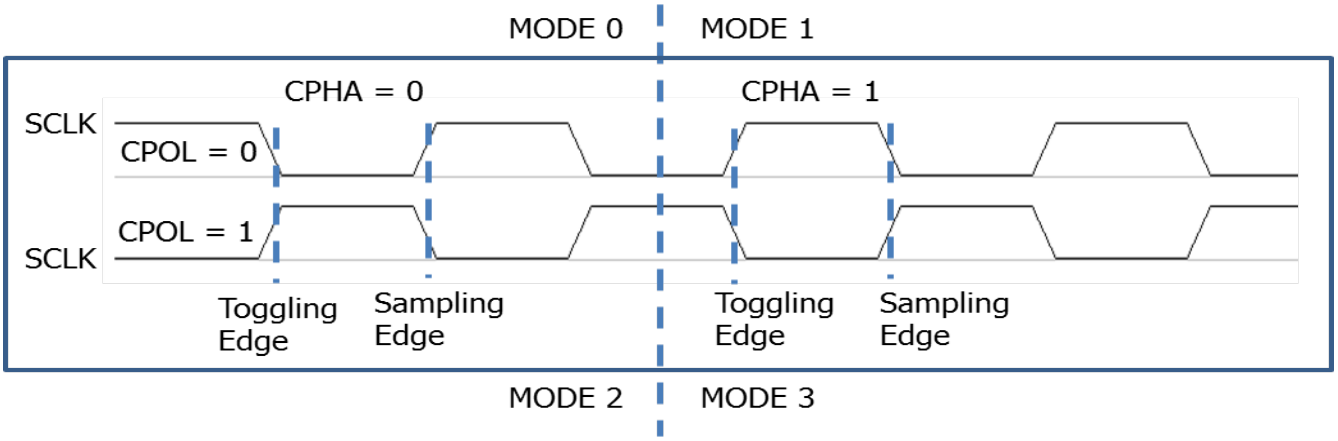

Figure 2: Clock Polarity and Clock Phase Configuration

Figure 3 and Figure 4 show scope plots of the 4 different SPI modes.

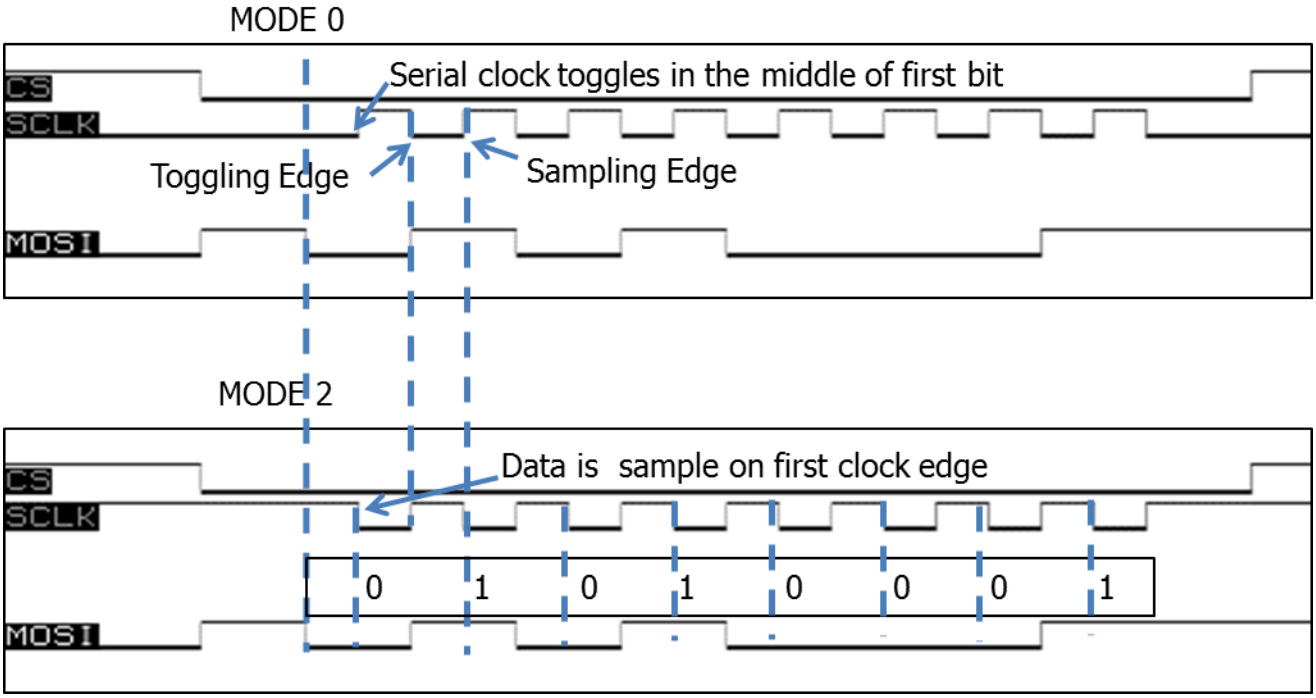

Figure 3: Mode 0 and Mode 2

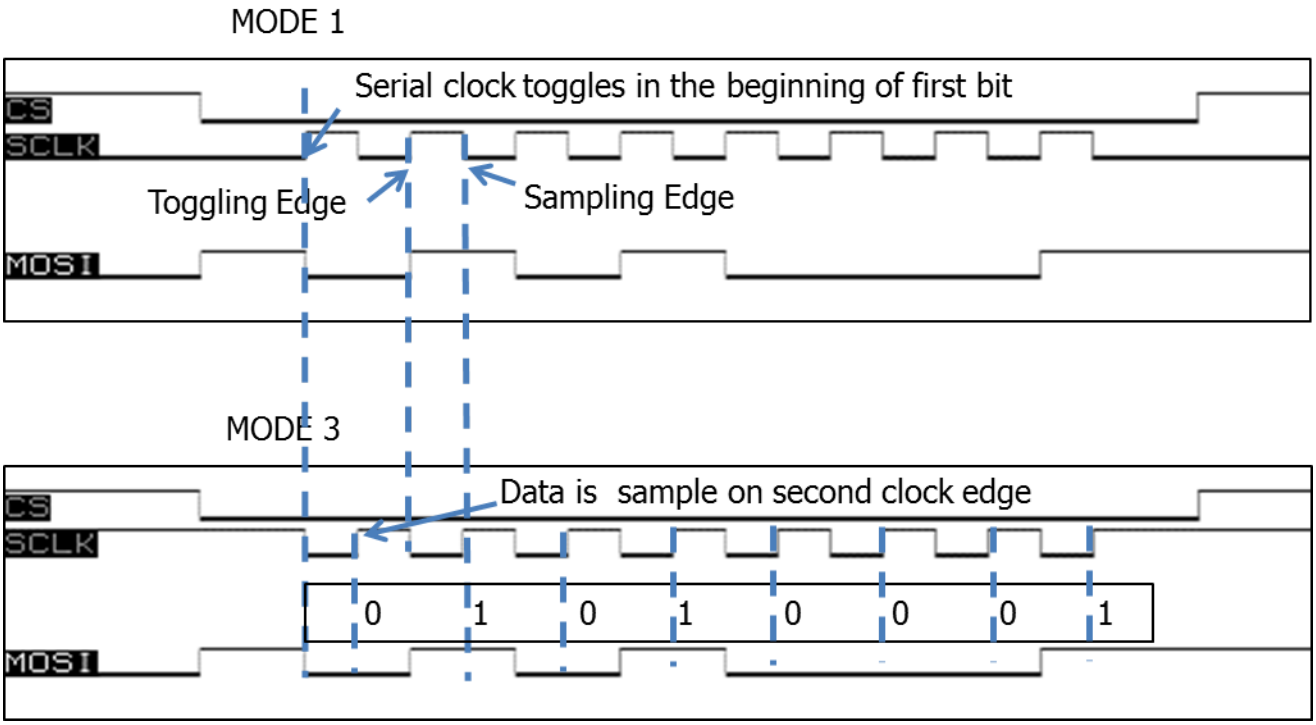

Figure 4: Mode 1 and Mode 3

Code 1 initializes the SPI master control unit, and for specifics on the API's, refer to https://ams.aeroflex.com/pagesproduct/software/access/default.cfm.

```
 // Enables a device specific interrupt in the NVIC interrupt controller.
NVIC_EnableIRQ (SPI_IRQn);

// Set SPI Init structure defualts:
//CTRLR0=0x00000007:
// SRL=0,Normal operation; TMOD=0,Transmit and receive; SCPOL=0, Toggling edge;
//SCPH=0, Sampling edge; FRF=0,Motorola SPI; DFS=0x7, Data transfer is 8 bits
// TXFTLR=0x00000008: TFT=0x8, Tx FIFO threshold
// RXFTLR=0x00000008: RFT=0x8, Rx FIFO threshold
SPI_StructInit (&SPI_InitStruct);

// Set SPI baud rate
// BAUDR=0x000000C8: SCKDV=0xC8, SPI clock divider
SPI_InitStruct.SPI_BaudRate = 125000;

// Set CS to CS0
// SER=0x00000001: SER=001, CS0 asserted upon transfer
SPI_SER_NDF_Config (SPI, 1, 0);

SPI->IMR=0; // Mask all interrupts
SPI_IntConfig (SPI, SPI_ISR_RXFIM_MASK,  ENABLE);// Enable RX FIFO full interrupt

// Initialize and enable SPI
// SSIENR=0x00000001: SSI_EN=1, SPI enabled
SPI_Init (SPI, &SPI_InitStruct);
```

Code 1: SPI Initialization

5.0 SPI Unit Programming

Section 3.0 presented some of the basic configurations for the SPI core. The following sections show programming examples by making use of Cobham API's for the UT32RM0R500.

5.1 SPI Send Data

The API provides a function for sending 4-16 bits to the slave device. The function in Code 2 references the SPI structure and for a single write, sends the byte to the specific slave device.

```
// Send one SPI byte
SPI_SendData (SPI, 0x55);
```

Code 2: Single byte SPI write

Data exchange between master and slave happens concurrently; each device sends and receives data at the same time. When the SPI master exchanges data with the slave, it must pull CSn low and drive the clock SCLK, with the most significant bit of both data registers sent out first, see Figure 5.

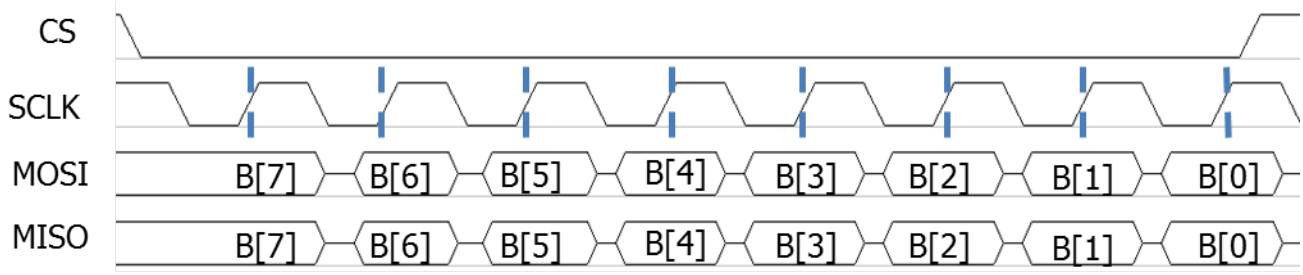

Figure 5: Master and Slave Communication with Mode 0

5.1.1 SPI Receive Data

The API provides a function for receiving data from the specific slave device. The function in Code 3 references the SPI structure and if the flag RX_FIFO_NOT_EMPTY is set, it reads the data.

```
// read SPI byte(s)
SPI_ReceiveData (SPI);
```

Code 3: SPI read byte(s)

Figure 6 shows the Oscilloscope timing diagram for sending and receiving data between SPI master and 74HC595 slave.

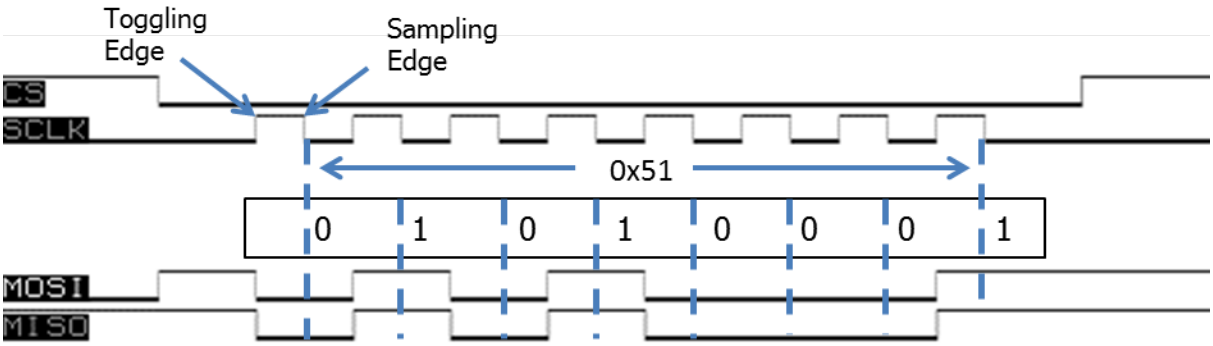

Figure 6: SPI Master/74HC595 Slave Mode 1 Configuration

5.1.2 SPI Interrupt

All SPI interrupts are share to one interrupt (IRQ), which is mapped to number 20 in the Interrupt Vector Table. The address of interrupt 20 in the Interrupt Vector Table is mapped to the SPI_IRQHandler which is the interrupt service routine (ISR) for all SPI interrupts. In the ISR, software must check for which interrupt happened.

```
// Enable RX FIFO full interrupt
SPI_IntConfig (SPI, SPI_ISR_RXFIM_MASK,  ENABLE);

//Enables a device specific interrupt in the NVIC interrupt controller
NVIC_EnableIRQ (SPI_IRQn);
```

Code 4: SPI RX FIFO full Interrupt settings

Putting it all together, Code 5 shows the main subroutine for sending and receiving SPI data in an endless loop. Code 6 shows The SPI_IRQHandler, which is the interrupt service routine for handling the particular SPI interrupt.

```
int main (void){
        // Initialization and settings from previous sections go here.
        for(;;)
        {
                if(SysTickExpired)
                {
                SysTickExpired=0;
                do{
                                SPI_SendData (SPI, 0x51);
                }while(SPI_GetFlagStatus (SPI, SPI_FLAG_TX_FIFO_NOT_FULL));
                }

                if(RxFifoFull)
                {
                        RxFifoFull=0;
                        do{
                                SPI_ReceiveData (SPI);
                        }while(SPI_GetIntStatus (SPI, SPI_ISR_RXOIM_MASK));
                }
         }

}
void SysTick_Handler(void)
 {
    SysTickExpired=1;
}
```

Code 5: Sample program for HC595 shift register

```
void ISR_IRQHandler (void)
{
        // Check to see if the interrupt is SPI RX FIFO full
        if(SPI_GetIntStatus (SPI, SPI_INT_RX_FIFO_FULL)){
                // Clear the flag
                SPI_ClearIntPendingBit (SPI, SPI_INT_RX_FIFO_FULL);

                RxFifoFull=1;

                // set GPIO 45
                GPIO_WriteOutputDataBit (GPIO2, GPIO2_PIN_GPIO45_OUTPUT, SET);
        }
        //GPIO_WriteOutputDataBit (GPIO2, GPIO2_PIN_GPIO45_OUTPUT, SET);
        __ASM volatile ("nop");
        // clear GPIO 45
        GPIO_WriteOutputDataBit (GPIO2, GPIO2_PIN_GPIO45_OUTPUT, RESET);

}
```

Code 6: Sample program SPI RX FIFO full interrupt

6.0 Summary and Conclusion

Since data can be transmitted and received by using separate lines, the synchronous serial communications makes for a fast interface between the UT32M0R500 master and external slaves devices.

For more information about our UT32M0R500 microcontroller and other products, please visit our website: www.cobham.com/HiRel or email us at info-ams@cobham.com.

REVISION HISTORY

Date	Rev. #	Author	Change Description
12/06/18	1.0.0	JA	Initial Release

The following United States (U.S.) Department of Commerce statement shall be applicable if these commodities, technology, or software are exported from the U.S.: These commodities, technology, or software were exported from the United States in accordance with the Export Administration Regulations. Diversion contrary to U.S. law is prohibited.

Cobham Semiconductor Solutions
4350 Centennial Blvd
Colorado Springs, CO 80907

E: info-ams@aeroflex.com
T: 800 645 8862