1.0 Overview

This application note explains the UT700 LEON 3FT SPARC™ V8 Microprocessor Serial Peripheral Interface (SPI) Controller with particular emphasis on the SPI configuration for the different modes of operation.

With this information, the programmer can apply the appropriate SPI configuration to their project needs.

Note: The description in this application note describes how to directly use the memory mapped interface of a specific hardware peripheral. If you are using an operating system such as RTEMS, Linux, and VxWorks or an environment such as BCC then it is recommended to use the infrastructure provided by those environments instead of accessing the peripheral directly as described in this application note.

2.0 Application Note Layout

This application note starts by providing a brief description of the SPI memory map, the SPI associated registers, and the SPI different bus modes of operation. These descriptions fall under the SPI Hardware sections.

After the SPI hardware sections, this application note provides several SPI use cases configuration options. These configuration options fall under the SPI Configuration sections.
Finally, we apply this knowledge using C programming code to configure the SPI. The C code programming examples fall under the SPI Programming sections.

These subsections are described in detail below:

- SPI Hardware
- SPI Configuration
- SPI Programming

### 3.0 SPI Hardware

The SPI Controller is mapped to the memory region from 0x8010_0100 to 0x80100134. It has seven registers; the MODE register provides a FACT bit, when set, reconfigures the SPI core registers that are compatible with the register interface found in the MPC83XX SoCs. If the FACT bit is clear (LEON SPI), the core can generate an SPI clock (SPICLK) with higher frequency.

We will only discuss the SPI configuration with the FACT bit clear. For SPI configuration with FACT bit set, refer to the MPC83XX reference manual. For more information about the seven registers, see the UT700 LEON Functional Manual, Chapter 19.

The SPI is a master centric interface that is a master device initiating the transaction by providing the clock (SPICLK) for data transmission and slave select (SPISLVSEL) to select the target slave device. The UT700’s SPI Controller provides one master device with full-duplex synchronous serial bus capable of operating in four different bus modes. Let’s explore the SPI different bus modes of operation, Receive and Transmit Queues, and Clock Generation in the following sections.

**Note:** The SPI can only work as a master device.

### 3.1 SPI Bus Modes of Operation

The SPI controller provides four bus modes for data transmission to cater for wider communication needs. These four bus modes of operation are the combination of the clock polarity (CPOL) and the clock phase (CPHA) that determine how data are transferred over the SPI bus, see the UT700 LEON Functional Manual, Chapter 19 and section 4.1 for more information.

### 3.2 SPI Receive and Transmit Queues
The SPI core provides transmission queue control logics and word FIFOs. The total number of words that can exist in each queue is the FIFO depth plus one. The FIFO depth (FDEPTH) is defined in the SPI Controller Capability Register (SPICPR).

3.3 **SPI Clock Generation**

The SPI core provides the clock (SPICLK) for transmission. The clock frequency is derived from the equations as shown in the UT700 LEON Functional Manual, Chapter 19. The numerator, AMBAClockFrequency, of the equations depends on the system clock and the \texttt{NODIV} input pin.

**Note:** See section 4.3.

4.0 **SPI Configuration**

Similar to all communication devices, a proper initialized communication interface ensures the data throughput. In this section, we show how to configure the SPI registers for specific operation and collectively using these registers to initialize the SPI for different modes of operation.

4.1 **SPI Bus Modes**

The SPI Controller provides four bus modes of operation as shown in Table 2.

<table>
<thead>
<tr>
<th>CPOL</th>
<th>CPHA</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Refer to the UT700 LEON Functional Manual, Chapter 19 for the bus waveforms.

4.2 **SPI Slave Select**

The SPI Controller provides one slave select output pin (SPISLVSEL); if the application requires more slave select output pins, user can use the GPIO, a binary-to-decimal decoder, and the SPISLVSEL as an enable signal to the decoder to achieve the required slave select output pins.

**Note:** To enable the SPISLVSEL, set the SSEN bit in the SPICCR register.

4.3 **SPI Clock Generation with FACT=0**
The SPI core clock (SPICLK) is programmable; see the equations as shown in the UT700 LEON Functional Manual, Chapter 19. In this application, we discuss the SPI configuration with FACT bit clear in the SPICCR; therefore, the equations are as follows:

With SPIMR register field DIV16 clear:

\[
SCKFrequency = \frac{AMBAClockFrequency}{4 \times (PM + 1)}
\]

With SPIMR register field DIV16 set:

\[
SCKFrequency = \frac{AMBAClockFrequency}{(16 \times 4) \times (PM + 1)}
\]

The PM (Prescaler Modulus) is a four bit field in the SPIMD register.

Note: If the NODIV input pin is pulled high, the AMBAClockFrequency is equal to the system clock frequency. If the NODIV input pin is pulled low, the AMBAClockFrequency is equal to half the system clock frequency.

4.4 SPI Transfer Word Length

The SPI core provides a LEN field in the SPIMD register to program the different word lengths. These words length are shown in Table 3:

<table>
<thead>
<tr>
<th>LEN Field</th>
<th>Word Length (Bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>32</td>
</tr>
<tr>
<td>0b0001</td>
<td>illegal</td>
</tr>
<tr>
<td>0b0010</td>
<td>illegal</td>
</tr>
<tr>
<td>0b0011</td>
<td>4</td>
</tr>
<tr>
<td>0b0100</td>
<td>5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0b1110</td>
<td>15</td>
</tr>
<tr>
<td>0b1111</td>
<td>16</td>
</tr>
</tbody>
</table>

LEN field values 0b0001 and 0b0010 are illegal values.

Note: The value of the LEN field must never specify a word length that is greater than the maximum allowed word length specified by the MXL field in the SPICPR register.
4.5 **SPI Data Endianness**

Besides the word length, the SPI core also provides configurable dataendianness transmission. This allows data transmission starting with the LSB or MSB bit to cater for different communication requirements. This feature is defined in the SPIMD register REVD field.

4.6 **SPI Interrupt**

The SPI core provides an interrupt for asynchronous data retrievable; see the Enable the Interrupt Controller application note for more information.

*Note:* See section 5.6.

5.0 **SPI Programming**

We learned from section 4.0 that the SPI Controller is highly configurable. The SPI Controller has no specific order in configuring its registers. However, there is only one restriction, when all the required options are configured, only then can the SPI core be enabled.

In the following sections, we provide programming examples to configure and initialize the SPI Controller.

*Note:* When the EN bit in the SPIMD register is set to ‘1’ the core is enabled. No fields in the SPIMD register should be changed while the core is enabled.

5.1 **Setup Bus Modes**

From section 4.1, we know the SPI Controller offers four bus modes for data transmission. The following code examples show how to set the four bus modes.

**Mode 0:**

```c
SPICTRL.SPIMD.B.CPOL = 0;  // set clock polarity
SPICTRL.SPIMD.B.CPHA = 0;  // set clock phase
```

**Mode 1:**

```c
SPICTRL.SPIMD.B.CPOL = 0;  // set clock polarity
SPICTRL.SPIMD.B.CPHA = 1;  // set clock phase
```

**Mode 2:**

```c
SPICTRL.SPIMD.B.CPOL = 1;  // set clock polarity
```
SPICCTRL.SPIMD.B.CPHA = 0; // set clock phase

**Mode 3:**

SPICCTRL.SPIMD.B.CPOL = 1; // set clock polarity
SPICCTRL.SPIMD.B.CPHA = 1; // set clock phase

5.2 **Enable Slave Select**

When **SSEN** bit is set in the **SPICCR** register, the **SPIISLVSEL** goes low when SPI data transaction is active.

SPICCTRL.SPICCR.B.SSEN = 1; // Slave select enable

5.3 **Baud Rate**

In this application note, we only configure the SPI baud rate with the **FACT** bit clear, see section 4.3.

SPICCTRL.SPIMD.B.FACT = 0; // LEON SPI

**For example:**

| System Clock | = 160 MHz |
| NODIV         | = 1       |
| SPI baud Rate | = 4.0 MHz/ 250 KHz |
| DIV16         | = 0 / 1   |
| AMBAClockFrequency | = 160 MHz |

SPICCTRL.SPIMD.B.DIV16 = 0; // divide by 1
SPICCTRL.SPIMD.B.PM = 9; // SPI baud rate = 4.0 MHz

SPICCTRL.SPIMD.B.DIV16 = 1; // divide by 16
SPICCTRL.SPIMD.B.PM = 9; // SPI baud rate = 250 KHz

| System Clock | = 160 MHz |
| NODIV         | = 0       |
| SPI baud Rate | = 2.0 MHz/ 125 KHz |
| DIV16         | = 0       |
| AMBAClockFrequency | = 80 MHz |

SPICCTRL.SPIMD.B.DIV16 = 0; // divide by 1
SPICCTRL.SPIMD.B.PM = 9; // SPI baud rate = 2.0 MHz
SPICCTRL.SPIMD.B.DIV16 = 1;  // divide by 16
SPICCTRL.SPIMD.B = 9;  // SPI baud rate = 125 KHz

5.4 Transfer Word Length and Endianness

The following codes show how to configure the word length and endianness, see sections 4.4 and 4.5.

SPICCTRL.SPICPR.B.MXL = 0;  // set the max word length, see Table 3

SPICCTRL.SPIMD.B.LEN = 0;  // 32-bit word length
SPICCTRL.SPIMD.B.REVD = 0;  // transmit LSB first

SPICCTRL.SPIMD.B.LEN = 3;  // 4-bit word length
SPICCTRL.SPIMD.B.REVD = 1;  // transmit MSB first

5.5 Data Transmission

The following codes show the proper way to handle data transmission.

Transmit:

while (SPICCTRL.SPIER.B.NF == 0);  // make sure transmit FIFO has room
SPICCTRL.SPITR.R = data;  // write data to transmit FIFO

Receive:

while (SPICCTRL.SPIER.B.NE == 0);  // make sure data in receive FIFO
data = SPICCTRL.SPIRR.R;  // read data from receive FIFO

5.6 Interrupt Sub Routine

The SPI uses the extended interrupt ID, number 18 to signal an SPI interrupt has occurred via the interrupt number 9, see Enable the Interrupt Controller application note for more information.

5.7 Enable the SPI
The SPI EN bit is enabled only when all the required SPI registers are configured. Once the EN bit is set, no fields in the SPIMD register should be changed.

```
SPICTRL.SPIMD.B.EN = 1; // enable SPI core
```

**Note:** When the EN bit is set to ‘1’ the core is enabled. No fields in the SPIMD register should be changed while the core is enabled.

### 6.0 Summary and Conclusion

After going through this AN, the reader should know how to configure the SPI for the different modes of operation.

For more information about our UT700 LEON 3FT/SPARC™ V8 Microprocessor and other products please visit our website, [www.cobham.com/HiRel](http://www.cobham.com/HiRel) or email us at info-ams@cobham.com.

---

**Appendix A: Header File**

This header file is designed for this application note purpose only.

```
/****************************************************************************
* MODULE: SPI Controller (SPICTRL) *
****************************************************************************/
```
#include <stdint.h>
#define SPICTRL_BASEADDR 0x80100100

typedef volatile uint32_t vuint32_t;

struct SPICTRL_TAG {
    union {
        vuint32_t R;
        struct {
            vuint32_t SSSZ: 8; // Slave select register size
            vuint32_t MXL: 4;  // Max word length
            vuint32_t REV19_17: 3; // Reserved
            vuint32_t SSEN: 1;  // Slave select enable
            vuint32_t FDEPTH: 8; // FIFO Depth
            vuint32_t SR: 1;    // SYNCRAM
            vuint32_t SFT: 2;   // Fault tolerance
            vuint32_t REV: 5;   // Core revision
        } B;
    } SPICPR;

    vuint32_t PADDING3[3]; // padding

    union {
        vuint32_t R;
        struct {
            vuint32_t REV31: 1; // Reserved
            vuint32_t LOOP: 1;  // Loop
            vuint32_t CPOL: 1;  // Clock polarity
            vuint32_t CPHA: 1;  // Clock Phase
            vuint32_t DIV16: 1; // Divide by 16
            vuint32_t REVD: 1;  // Reverse data
            vuint32_t MS: 1;    // Master Mode
            vuint32_t EN: 1;    // Enable core
            vuint32_t LEN: 4;   // Word
            vuint32_t PM: 4;    // Prescaler modulus
            vuint32_t TW: 1;    // Three-wire mode
            vuint32_t ASEL: 1;  // Automatic slave select
            vuint32_t FACT: 1;  // PM factor
            vuint32_t OD: 1;    // Open drain mode
            vuint32_t CG: 5;    // Clock gap
            vuint32_t ASELDEL: 2; // Automatic slave select delay
            vuint32_t RES4_0: 5; // Reserved
        } B;
    } SPIMD;
union {
  vuint32_t R; // Table 19.4: Description of SPI Controller Event Register
  struct {
    // Transfer in progress LF
    vuint32_t TIP: 1;
    // Reserved
    vuint32_t RES30_15: 16;
    // Last character LF
    vuint32_t LT: 1;
    // Reserved
    vuint32_t RES13: 1;
    // Overrun LF
    vuint32_t OV: 1;
    // Underrun
    vuint32_t UN: 1;
    // Not Empty LF
    vuint32_t NE: 1;
    // Not full LF
    vuint32_t NF: 1;
  } B;
} SPIER;

union {
  vuint32_t R; // Table 19.5: Description of SPI Controller Mask Register
  struct {
    // Transfer in progress enable
    vuint32_t TIPE: 1;
    // Reserved
    vuint32_t REV30_15: 16;
    // Last character enable
    vuint32_t LTE: 1;
    // Reserved
    vuint32_t REV13: 1;
    // Overrun Enable
    vuint32_t OVE: 1;
    // Underrun Enable
    vuint32_t UNE: 1;
    // Multiple Master error enable
    vuint32_t MMEE: 1;
    // Not empty enable
    vuint32_t NEE: 1;
    // Not full enable
    vuint32_t NFE: 1;
    // Reserved
    vuint32_t REV7_0: 8;
  } B;
} SPIMR;

union {
  vuint32_t R; // Table 19.6: Description of SPI Controller Command Register
  struct {
    // Reserved
    vuint32_t REV31_23: 9;
    // Last
    vuint32_t LST: 1;
    // Reserved
    vuint32_t REV20_0: 22;
  } B;
} SPICCR;

union {
  vuint32_t R; // Table 19.7: Description of SPI Controller Transmit Register
  struct {
    // Transmit data
    vuint32_t TDATA: 32;
  } B;
}
union {
    vuint32_t R;
    struct {
        vuint32_t RDATA: 32;
    } B;
} SPIRR;

#define SPICTRL (*((volatile struct SPICTRL_TAG*) SPICTRL_BASEADDR))
## REVISION HISTORY

<table>
<thead>
<tr>
<th>Date</th>
<th>Rev. #</th>
<th>Author</th>
<th>Change Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>43/25/2017</td>
<td>1.0.0</td>
<td>MTS</td>
<td>Initial Release</td>
</tr>
<tr>
<td>09/29/2017</td>
<td>1.0.1</td>
<td>MTS</td>
<td>Add note, page 1</td>
</tr>
</tbody>
</table>
This product is controlled for export under the U.S. Department of Commerce (DoC). A license may be required prior to the export of this product from the United States.