C28x Serial Peripheral Interface

Introduction

The TMS320C28x contains built-in features that allow several methods of communication and data exchange between the C28x and other devices. This chapter deals with the Serial Peripheral Interface (SPI). Two other interface techniques (SCI and CAN) will be discussed in later chapters.

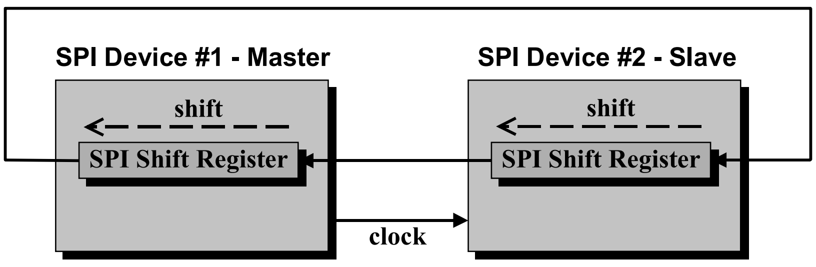
The SPI module is a synchronous serial I/O port that shifts a serial bit stream of variable length and data rate between the 'C28x' and other peripheral devices. Here "synchronous" means that the data transmission is synchronized to a clock signal.

During data transfers, one SPI device must be configured as the transfer MASTER, and all other devices configured as SLAVES. The master drives the transfer clock signal for all SLAVES on the bus. SPI communication can be implemented in any of three different modes:

- MASTER sends data, SLAVES send dummy data
- MASTER sends data, one SLAVE sends data
- MASTER sends dummy data, one SLAVE sends data

SPI Data Flow

- **Simultaneous transmits and receive**
- **SPI Master provides the clock signal**



7 - 2

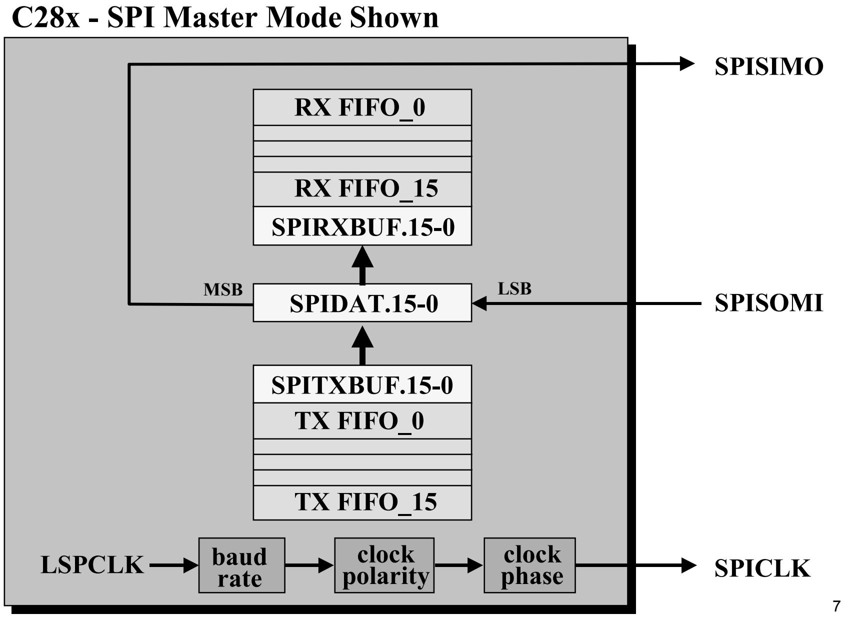
Module Topics

Serial Peripheral Interface (SPI) – Overview

In its simplest form, the SPI can be thought of as a programmable shift register. Data bits are shifted in and out of the SPI through the SPIDAT register. Two more registers set the programming interface. To transmit a data frame, we have to write the 16-bit message into the SPITXBUF buffer. A received frame will be read by the SPI directly into the SPIRXBUF buffer. For our lab exercises, this means we write directly to SPITXBUF and we read from SPIRXBUF.

There are two operating modes for the SPI: “basic mode” and “enhanced FIFO-buffered mode”. In “basic mode”, a receive operation is double-buffered, that is the CPU need not read the current received data from SPIRXBUF before a new receive operation can be started. However, the CPU must read SPIRXBUF before the new operation is complete or a receiver-overrun error will occur. Double-buffered transmit is not supported in this mode; the current transmission must be complete before the next data character is written to SPITXDAT or the current transmission will be corrupted. The Master can initiate a data transfer at any time because it controls the SPICLK signal.

SPI Block Diagram



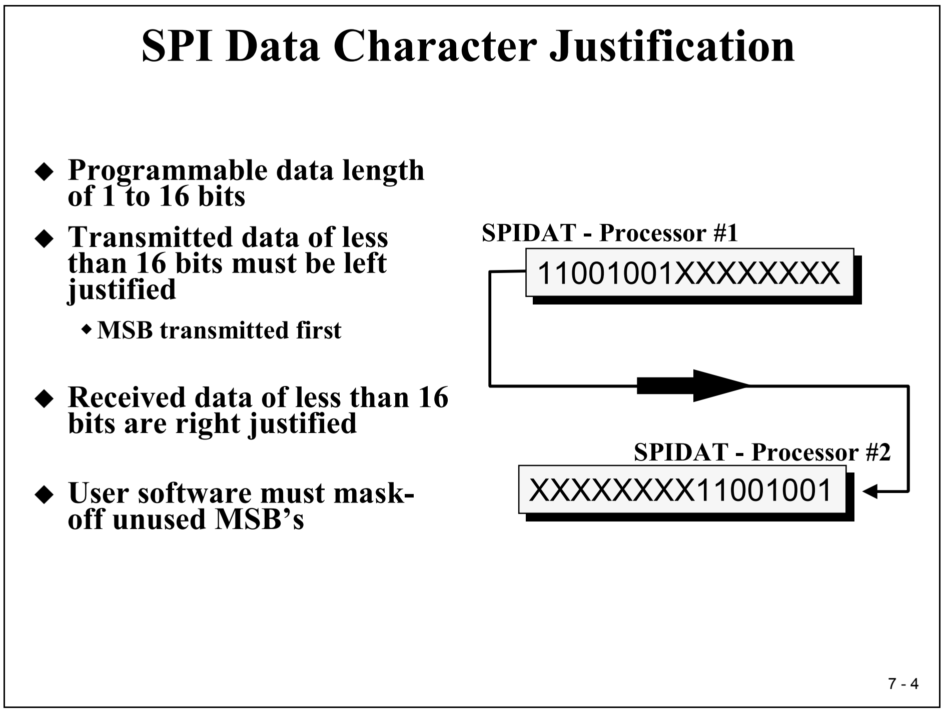
7 - 3

In “enhanced FIFO – buffered mode” we can build up to 16 levels of transmit- and receive FIFO memory. Again, our program interfaces to the SPI unit are the registers SPITXBUF and SPIRXBUF. This will expand the SPI’s buffer capacity for receive and transmit to up to 16 times. In this mode we are also able to specify an interrupt level that depends on the filling state of the two FIFO’s.

SPI Data Transfer

As you can see from the previous slide, the SPI master is responsible for generating the data rate of the communication. Derived from the internal low speed clock prescaler (LSPCLK), we can specify an individual baud rate for the SPI. Because not all SPI devices are interfaced in the same way, we can adjust the shape of the clock signal by two more bits, "clock polarity" and "clock phase". Strictly speaking, the SPI is not a standard; slave devices like EEPROM's, DAC's, ADC's, Real Time Clocks, temperature sensors do have different timing requirements for the interface timing. For this reason TI includes options to adjust the SPI timing.

A transmission always starts with the MSB (most significant bit) out of SPIDAT first and received data will be shifted into the device, also with MSB first. Both transmitter and receiver perform a left shift with every SPI clock period. For frames of less than 16 bits, data to be transmitted must be left justified before transmission starts. Received frames of less than 16 bits must be masked by user software to suppress unused bits.



SPI Data Character Justification

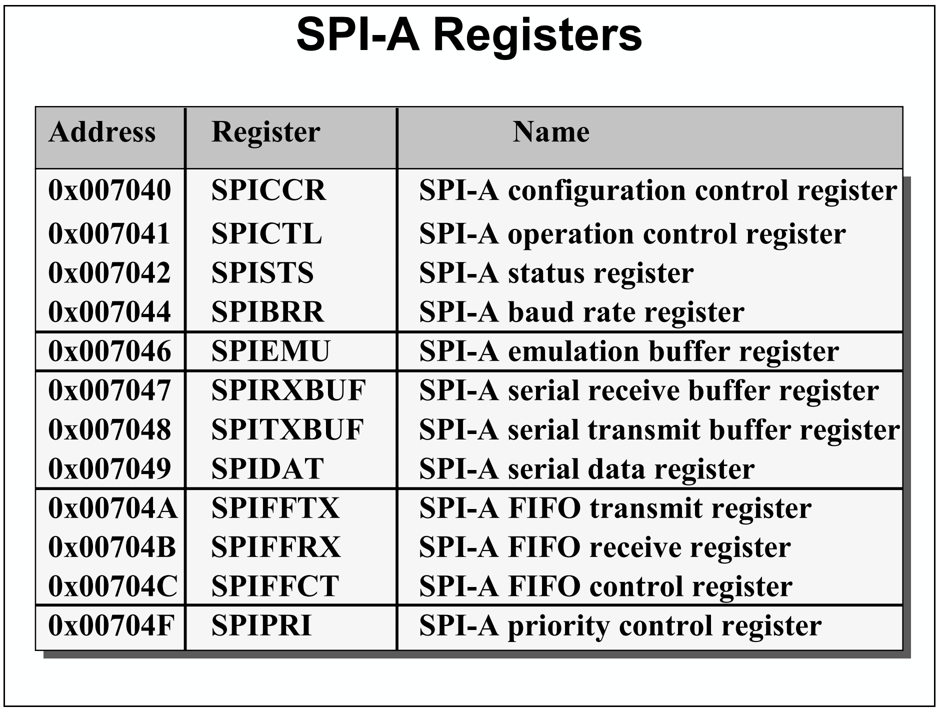
- Programmable data length of 1 to 16 bits
- Transmitted data of less than 16 bits must be left justified
  - MSB transmitted first
- Received data of less than 16 bits are right justified
- User software must mask-off unused MSB's

SPIDAT - Processor #1
11001001XXXXXXXX
SPIDAT - Processor #2
XXXXXXXX11001001

7 - 4

SPI Register Set

The next slide summarizes all SPI control registers. In future devices of the C28x family we will have a second SPI, for this reason the names of the first SPI are expanded with an ‘A’.

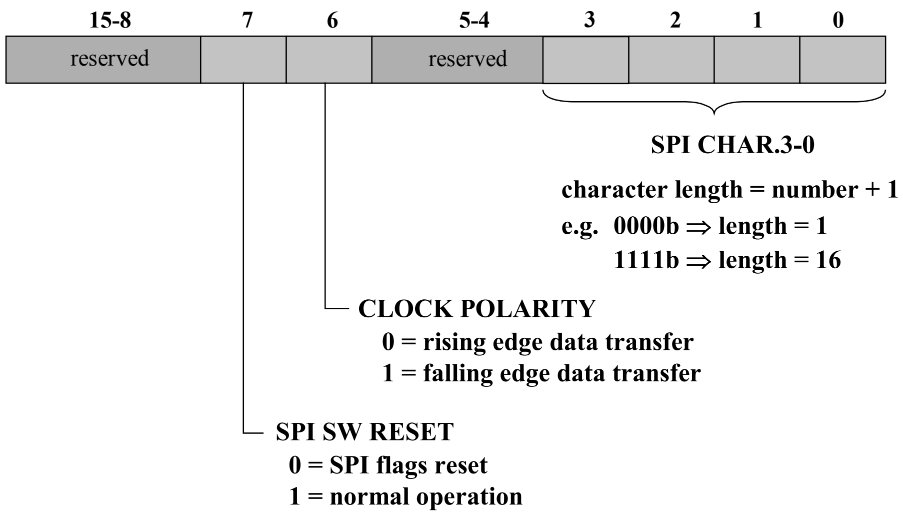


SPI-A Registers

Address	Register	Name
0x007040	SPICCR	SPI-A configuration control register
0x007041	SPICTL	SPI-A operation control register
0x007042	SPISTS	SPI-A status register
0x007044	SPIBRR	SPI-A baud rate register
0x007046	SPIEMU	SPI-A emulation buffer register
0x007047	SPIRXBUF	SPI-A serial receive buffer register
0x007048	SPITXBUF	SPI-A serial transmit buffer register
0x007049	SPIDAT	SPI-A serial data register
0x00704A	SPIFFTX	SPI-A FIFO transmit register
0x00704B	SPIFFRX	SPI-A FIFO receive register
0x00704C	SPIFFCT	SPI-A FIFO control register
0x00704F	SPIPRI	SPI-A priority control register

SPI-A Configuration Control Register

SPICCR @ 0x007040

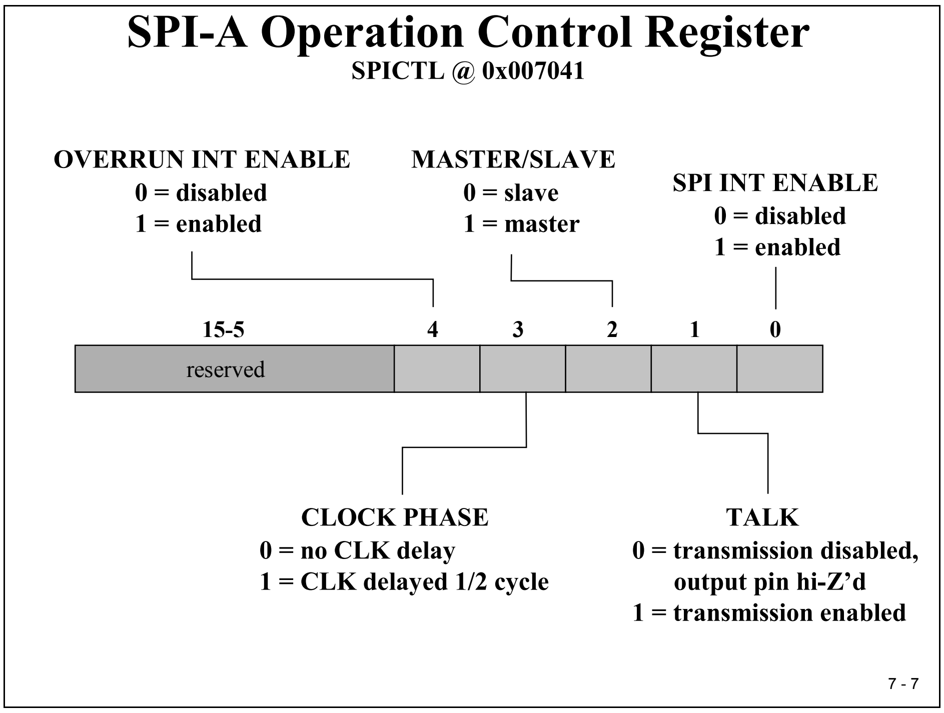


7 - 6

SPI Confguration Control Register - SPICCR

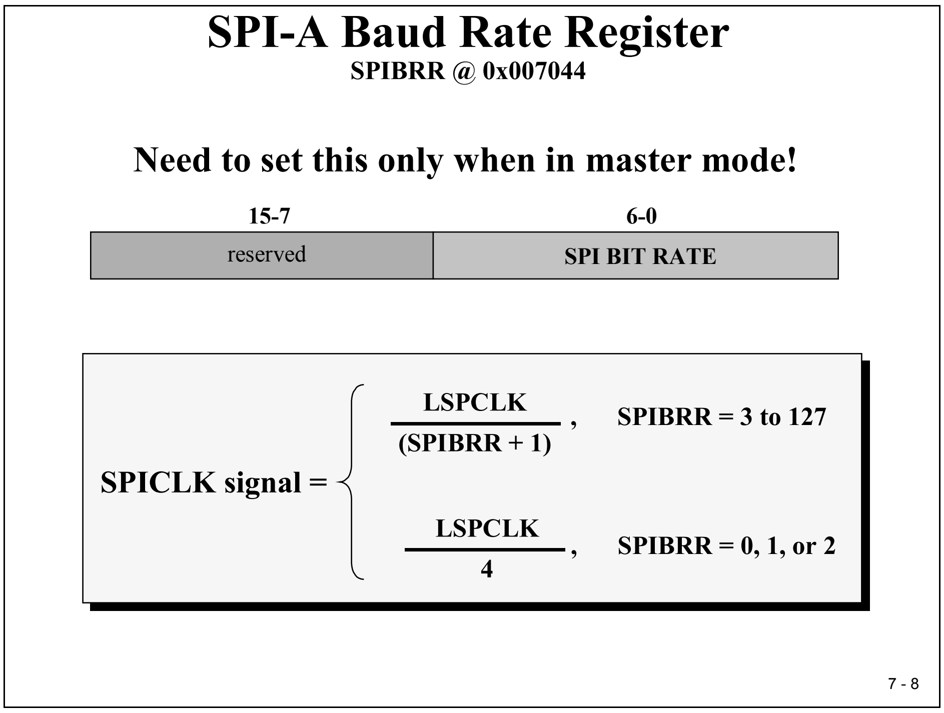
It is good practice to RESET the SPI unit at begin of the initialization procedure. This is done by clearing Bit 7 (SPI SW RESET) to 0 followed by setting it to 1. Bit 6 selects the active clock edge to declare the data as valid. This selection depends on the particular SPI – device (see the two examples used in the labs of this chapter). Bits 3...0 define the character length of the SPI-frame.

SPI Operation Control Register – SPICTL



Bit 4 and 0 enable or disable the SPI- interrupts; Bit 4 enables the receivers overflow interrupt. Bit 2 defines the operating mode for the C28x to be master or slave of the SPI-chain. With the help of bit 3 we can implement another half clock cycle delay between the active clock edge and the point of time, when data are valid. Again, this bit depends on the particular SPI-device. Bit 1 controls whether the C28x listens only (Bit 1 = 0) or if the C28x is initialized as receiver and transmitter (Bit 1 = 1).

SPI Baud Rate Register – SPIBRR



SPI-A Baud Rate Register

SPIBRR @ 0x007044

Need to set this only when in master mode!

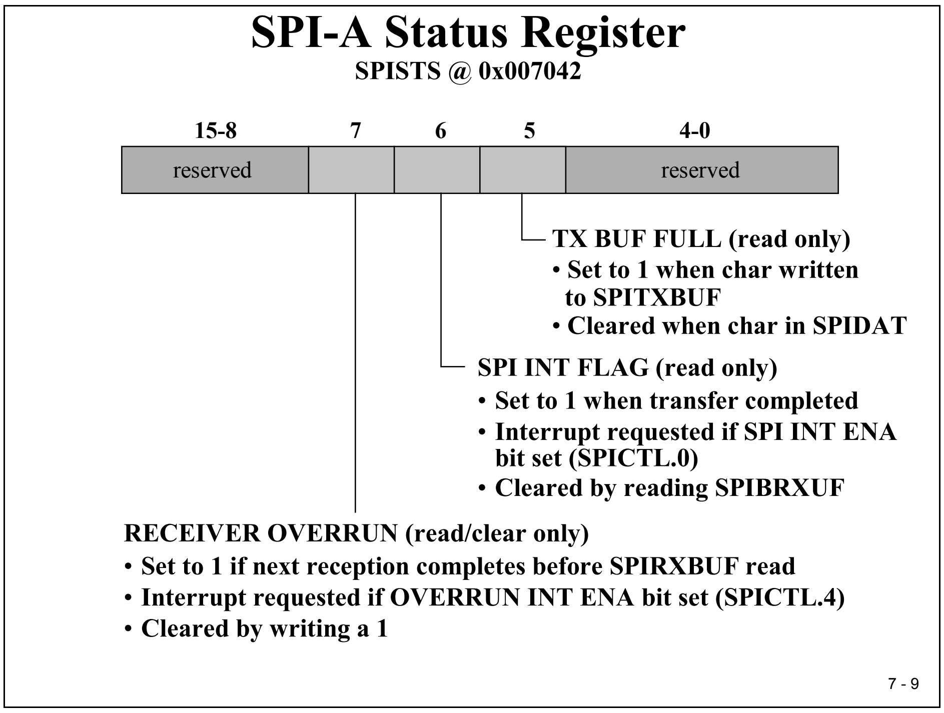
15-7	6-0
reserved	SPI BIT RATE

$$\text{SPICLK signal} = \begin{cases} \dfrac{\text{LSPCLK}}{(\text{SPIBRR} + 1)}, & \text{SPIBRR} = 3 \text{ to } 127 \\ \dfrac{\text{LSPCLK}}{4}, & \text{SPIBRR} = 0, 1, \text{ or } 2 \end{cases}$$

7 - 8

Clock base for the SPI baud rate selection is the Low speed Clock Prescaler (LSPCLK).

SPI Status Register – SPISTS



SPI-A Status Register

SPISTS @ 0x007042

15-8	7	6	5	4-0
reserved				reserved

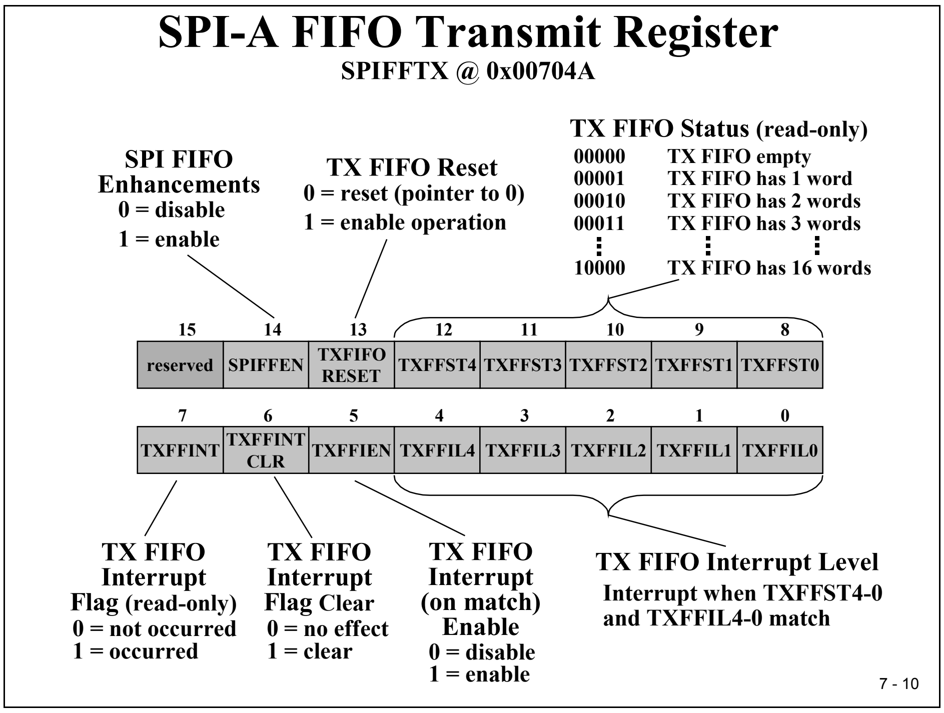
TX BUF FULL (read only)
- **Set to 1 when char written to SPITXBUF**
- **Cleared when char in SPIDAT**

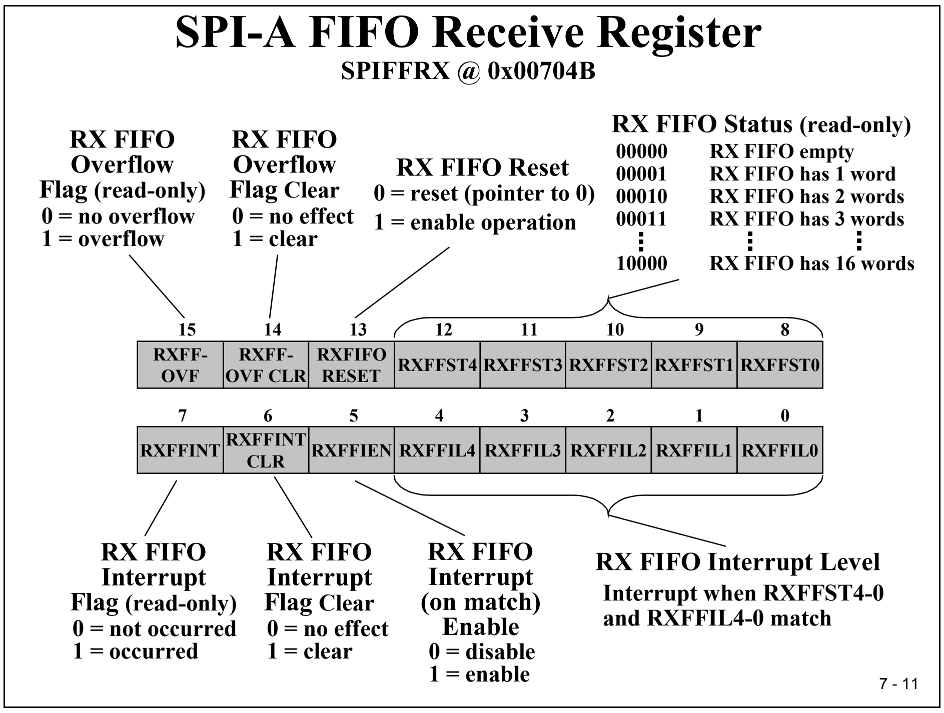
SPI INT FLAG (read only)
- **Set to 1 when transfer completed**
- **Interrupt requested if SPI INT ENA bit set (SPICTL.0)**
- **Cleared by reading SPIBRXUF**

RECEIVER OVERRUN (read/clear only)
- **Set to 1 if next reception completes before SPIRXBUF read**
- **Interrupt requested if OVERRUN INT ENA bit set (SPICTL.4)**
- **Cleared by writing a 1**

7 - 9

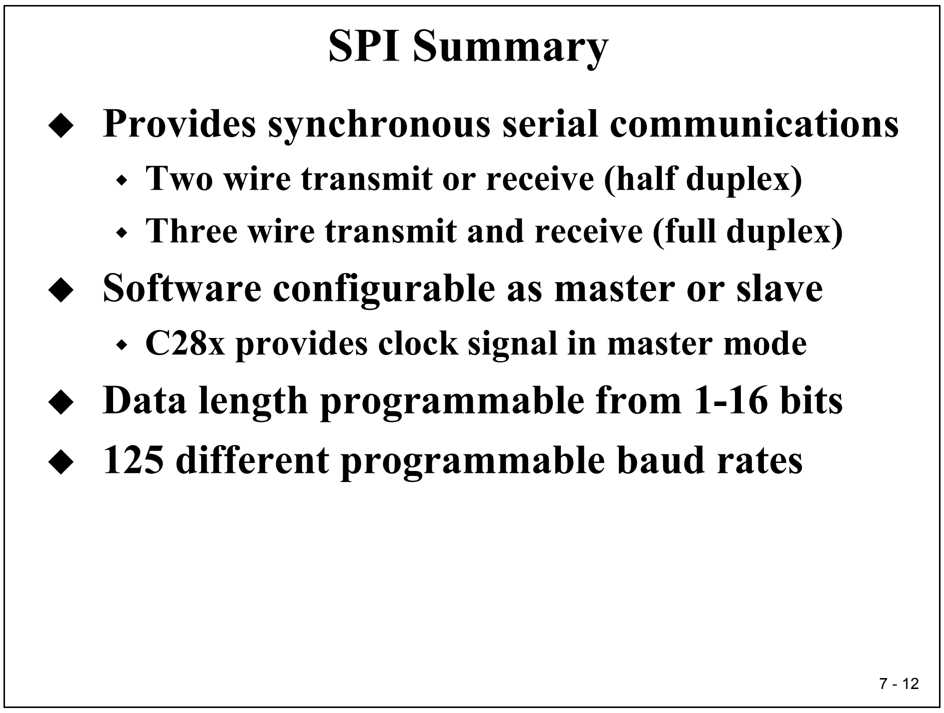
SPI FIFO Transmit Register


SPI-A FIFO Transmit Register
SPIFFTX @ 0x00704A
SPI FIFO Enhancements
0 = disable
1 = enable
TX FIFO Reset
0 = reset (pointer to 0)
1 = enable operation
TX FIFO Status (read-only)
00000 TX FIFO empty
00001 TX FIFO has 1 word
00010 TX FIFO has 2 words
00011 TX FIFO has 3 words
10000 TX FIFO has 16 words
15 14 13 12 11 10 9 8
reserved SPIFFEN TXFIFO RESET TXFFST4 TXFFST3 TXFFST2 TXFFST1 TXFFST0
7 6 5 4 3 2 1 0
TXFFINT TXFFINT CLR TXFFIEN TXFFIL4 TXFFIL3 TXFFIL2 TXFFIL1 TXFFIL0
TX FIFO Interrupt Flag (read-only)
0 = not occurred
1 = occurred
TX FIFO Interrupt Flag Clear
0 = no effect
1 = clear
TX FIFO Interrupt (on match) Enable
0 = disable
1 = enable
TX FIFO Interrupt Level
Interrupt when TXFFST4-0 and TXFFIL4-0 match
7 - 10


SPI-A FIFO Receive Register
SPIFFRX @ 0x00704B
RX FIFO Overflow Flag (read-only)
0 = no overflow
1 = overflow
RX FIFO Overflow Flag Clear
0 = no effect
1 = clear
RX FIFO Reset
0 = reset (pointer to 0)
1 = enable operation
RX FIFO Status (read-only)
00000 RX FIFO empty
00001 RX FIFO has 1 word
00010 RX FIFO has 2 words
00011 RX FIFO has 3 words
10000 RX FIFO has 16 words
15 14 13 12 11 10 9 8
RXFF-OVF RXFF-OVF CLR RXFIFO RESET RXFFST4 RXFFST3 RXFFST2 RXFFST1 RXFFST0
7 6 5 4 3 2 1 0
RXFFINT RXFFINT CLR RXFFIEN RXFFIL4 RXFFIL3 RXFFIL2 RXFFIL1 RXFFIL0
RX FIFO Interrupt Flag (read-only)
0 = not occurred
1 = occurred
RX FIFO Interrupt Flag Clear
0 = no effect
1 = clear
RX FIFO Interrupt (on match) Enable
0 = disable
1 = enable
RX FIFO Interrupt Level
Interrupt when RXFFST4-0 and RXFFIL4-0 match
7 - 11

The FIFO operation of the SPI is controlled with Bit 14 as a master switch. The SPI-Transmit FIFO interrupt service call depends now on the match between TX FIFO Status and TX FIFO Interrupt Level. The TX FIFO Reset can be used to reset the FIFO state machine (Bit13= 0) and to re-enable it (Bit 13=1).

SPI Summary

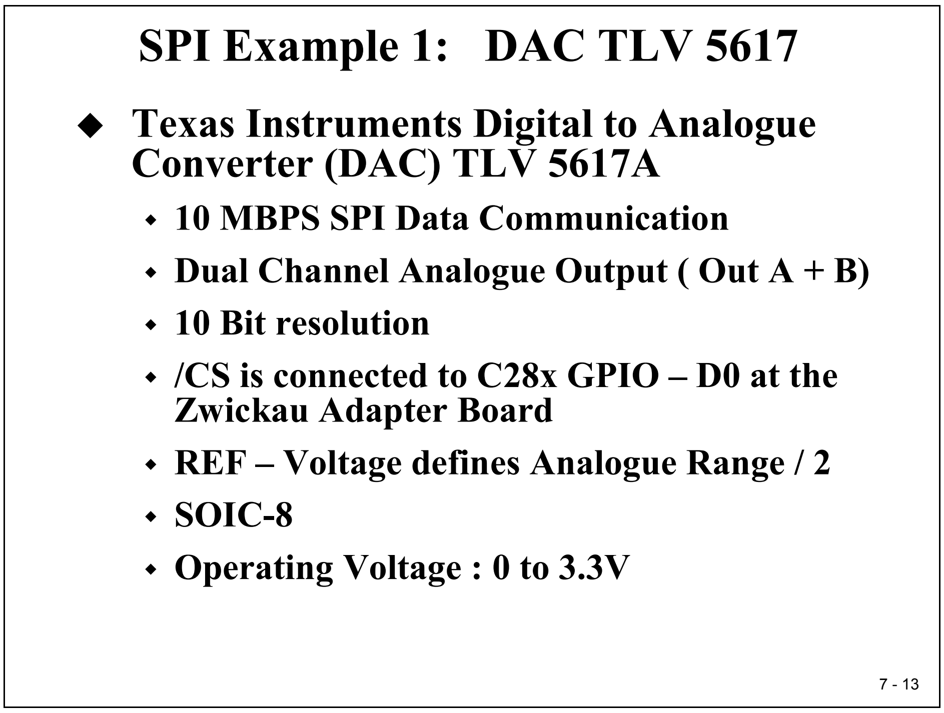


SPI Summary

- Provides synchronous serial communications
  - Two wire transmit or receive (half duplex)
  - Three wire transmit and receive (full duplex)
- Software configurable as master or slave
  - C28x provides clock signal in master mode
- Data length programmable from 1-16 bits
- 125 different programmable baud rates

7 - 12

Lab 7: SPI – Dual DAC Texas Instruments TLV5617A



SPI Example 1: DAC TLV 5617

- **Texas Instruments Digital to Analogue Converter (DAC) TLV 5617A**
  - **10 MBPS SPI Data Communication**
  - **Dual Channel Analogue Output (Out A + B)**
  - **10 Bit resolution**
  - **/CS is connected to C28x GPIO – D0 at the Zwickau Adapter Board**
  - **REF – Voltage defines Analogue Range / 2**
  - **SOIC-8**
  - **Operating Voltage : 0 to 3.3V**

7 - 13

Objective

The objective of this lab is to establish an SPI communication between the C28x and a serial DAC TLV 5617. This DAC has two output channels and a resolution of 10 bits. The datasheet of this device is available from TI's website, search for document number 'SLAS234'. The interface between the C28x and the DAC is defined as follows:

TLV5617 – Signal Name	Pin – No.	Description	Connected to…
AGND	5	Ground	AGND
/CS	3	Chip - Select	C28x- GPIO D0
DIN	1	Input Data	C28x-SPISIMO
SCLK	2	SPI Clock	C28x-SPICLK
REF	6	Analogue Reference input	3.3V
VDD	8	Power supply	3.3V
OUT A	4	DAC output A	JP7 – 1
OUT B	7	DAC output B	JP8 – 1

The chip-select (/CS) of the DAC is connected to the GPIO – D0. The TLV5617 is a 'listen only' SPI device; it sends no data back to the C28x. The REF voltage defines the full-scale value for the analogue output voltages. It is connected to 3.3V. Although the TLV5617 is a 10 Bit DAC it uses an internal multiplier by 2, therefore the digital input values lie in the range 0 to 511.

The TLV 5617 has the following timing requirements:

SPI Example : DAC TLV 5617

- **Timing Diagram:**

timing requirements

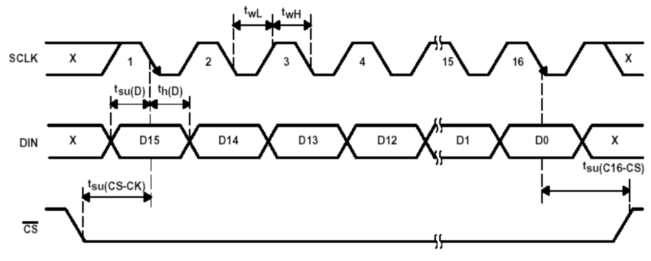


Figure 1. Timing Diagram

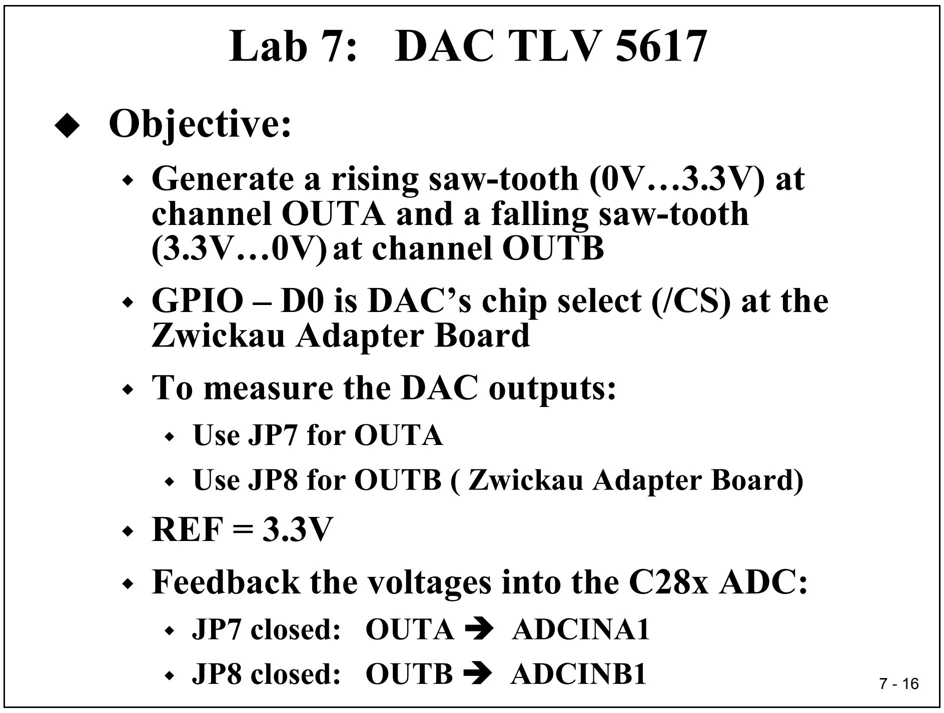
© Texas Instruments SLAS234F – JULY 1999 – REVISED JULY 2002 ; page 6

7 - 14

The active frame is covered with an active /CS-Signal, MSB comes first, the data bit leads the falling edge of the SCLK-Signal by half of a clock period. The DAC's internal operation starts with the rising edge of /CS.

Aim of Lab7:

The Aim of the Lab7 is to generate two saw tooth signals at the outputs of the SPI-DAC: DAC-A - "rising saw tooth" and DAC-B" - falling saw tooth between 0V and 3.3V. With the help of a scope meter, if available, we should be able to measure the two voltages. In an optional exercise ("Lab7A") we also will make use of the graphical tool of Code Composer Studio to visualize the two voltages. To do this we have to reconnect the DAC signals back to two ADC-Input lines. This is done by closing the jumpers JP7 and JP8 on the Zwickau Adapter board.

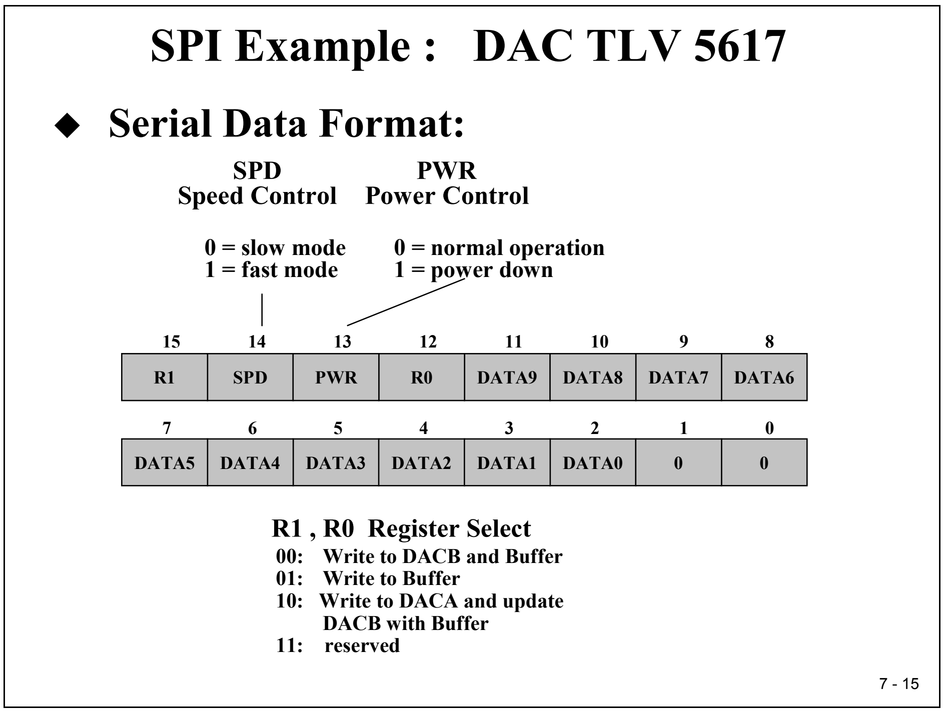


Lab 7: DAC TLV 5617

- **Objective:**
  - **Generate a rising saw-tooth (0V…3.3V) at channel OUTA and a falling saw-tooth (3.3V…0V) at channel OUTB**
  - **GPIO – D0 is DAC's chip select (/CS) at the Zwickau Adapter Board**
  - **To measure the DAC outputs:**
    - **Use JP7 for OUTA**
    - **Use JP8 for OUTB (Zwickau Adapter Board)**
  - **REF = 3.3V**
  - **Feedback the voltages into the C28x ADC:**
    - **JP7 closed: OUTA ➔ ADCINA1**
    - **JP8 closed: OUTB ➔ ADCINB1**

7 - 16

DAC TLV5617A Data Format



SPI Example : DAC TLV 5617

- **Serial Data Format:**

SPD
Speed Control

0 = slow mode
1 = fast mode

PWR
Power Control

0 = normal operation
1 = power down

15	14	13	12	11	10	9	8
R1	SPD	PWR	R0	DATA9	DATA8	DATA7	DATA6

7	6	5	4	3	2	1	0
DATA5	DATA4	DATA3	DATA2	DATA1	DATA0	0	0

R1 , R0 Register Select

00:	Write to DACB and Buffer
01:	Write to Buffer
10:	Write to DACA and update DACB with Buffer
11:	reserved

7 - 15

Procedure

Open Files, Create Project File

1. Create a new project, called **Lab7.pjt** in E:\C281x\Labs.

2. Open the file Lab4.c from E:\C281x\Labs\Lab4 and save it as Lab7.c in E:\C281x\Labs\Lab7.

3. Add the source code file to your project:
   - **Lab7.c**

4. From *C:\tidcs\c28\dsp281x\v100\DSP281x_headers\source* add:

   - **DSP281x_GlobalVariableDefs.c**

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_headers\cmd* add:

   - **F2812_Headers_nonBIOS.cmd**

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\cmd* add:

   - **F2812_EzDSP_RAM_lnk.cmd**

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\source* add to project:

   - **DSP281x_PieCtrl.c**
   - **DSP281x_PieVect.c**
   - **DSP281x_DefaultIsr.c**
   - **DSP281x_CpuTimers.c**

   From *C:\ti\c2000\cgtoolslib* add:

   - **rts2800_ml.lib**

Project Build Options

5. Setup the search path to include the peripheral register header files. Click:

   Project → Build Options

   Select the Compiler tab. In the preprocessor Category, find the Include Search Path (-i) box and enter:

C:\tidcs\C28\dsp281x\v100\DSP281x_headers\include;
..\include

6. Setup the stack size: Inside Build Options select the Linker tab and enter in the Stack Size (-stack) box:

   400

   Close the Build Options Menu by Clicking <**OK**>.

Modify Source Code

7. Open Lab7.c to edit: double click on "Lab7.c" inside the project window. First we have to cancel the parts of the code that we do not need any longer. We will not use the main variables "LED[8]" and "i" for this exercise:

   At the beginning of main, delete the lines:

```
unsigned int i;
unsigned int LED[8]= {0x0001,0x0002,0x0004,0x0008,
                      0x0010,0x0020,0x0040,0x0080};
```

8. Next, inside the "while(1)"-loop of main reduce the code to just the following lines (we will add some more code later):

```
while(1)
{
        while(CpuTimer0.InterruptCount < 3); // wait for Timer 0
        CpuTimer0.InterruptCount = 0;
        EALLOW;
        SysCtrlRegs.WDKEY = 0xAA;       // and service watchdog #2
        EDIS;
}
```

9. Before we continue to add the SPI modifications lets test if the project in its preliminary stage runs as expected. Recall, with the start code of Lab4 we initialized the CPU core timer 0 to generate an interrupt request every 50ms. The interrupt service routine "cpu_timer0_isr()" increments a global variable"CpuTimer0.InterruptCount" with every hit. If everything works as expected the DSP should hit the line

```
        CpuTimer0.InterruptCount = 0;
```

   in the while(1) – loop (procedure step 8) every 3*50ms = 150 ms.

Build and Load

10. Click the "Rebuild All" button or perform:

    Project → Build

and watch the tools run in the build window. If you get syntax errors or warnings debug as necessary.

11. Load the output file down to the DSP. Click:

    File → Load Program and choose the desired output file.

Test

12. Reset the DSP by clicking on:

    Debug → Reset CPU followed by
    Debug ➔ Restart and
    Debug → Go main.

13. In the “while(1)”-loop of main set a breakpoint at line:
    CpuTimer0.InterruptCount = 0;

Run

14. Verify, that the breakpoint is hit periodically when you start the DSP by:

    Debug → Run (F5).

    Remove the breakpoint when you are done.

Add the SPI initialization code

15. So far we verified that the CPU Core Timer0 generates a period of 50 ms and that our main-loop waits for 3 periods of Timer0 before it moves to the next instruction. Now we need to add the code for the SPI to control the DAC TLV 5617A. First function that is called is “InitSystem()”. Do we have to adjust this function? YES! We have to enable the SPI clock unit! Inside “InitSystem()” do:

    SysCtrlRegs.PCLKCR.bit.SPIENCLK=1;

16. The next function that is called in main is “Gpio_select()”. Inside this function we have to modify the multiplex register to use the four SPI-signals:

    GpioMuxRegs.GPFMUX.all = 0xF;

    Bits 3...0 control the setup for SPISTEA, SPICLKA, SPISOMIA and SPISIMOA. We also prepare GPIO-signal D0 and D5 to be digital outputs. D0 will be used as chip-select for the TLV5617A and D5 for the EEPROM M95080 (used in Lab7B). To do this, setup GPDDIR register:

GpioMuxRegs.GPDDIR.all=0;

GpioMuxRegs.GPDDIR.bit.GPIOD0 = 1; // /CS for DAC TLV5617A

GpioMuxRegs.GPDDIR.bit.GPIOD5 = 1; // /CS for EEPROM

As an initial state we should switch off both /CS-signals as well as the LED's at B7...B0:

GpioDataRegs.GPBDAT.all = 0x0000; // Switch off LED's (B7...B0)

GpioDataRegs.GPDDAT.bit.GPIOD0 = 1; // /CS for DAC off

GpioDataRegs.GPDDAT.bit.GPIOD5 = 1; // /CS for EEPROM off

17. In main, just before we enter the "while(1)"-loop add a function call to function "SPI_Init()". Also add a function prototype at the start of your code.

    At the end of your code, add the definition of function "SPI_Init()".

    Inside this function include the following steps:

    - SPICCR:
        - Clock polarity =1: data output at falling edge of clock
        - 16 bit per data frame
    - SPICTL:
        - No SPI interrupts for this exercise
        - Master on
        - Talk enabled
        - Clock phase = 1: one half cycle delay
    - SPIBRR:
        - BRR = LSPCLK/SPI_Baudrate – 1
        - Example: assuming LSPCLK = 37.5MHz and SPIBRR = 124 the SPI-Baud rate is 300 kBPS.

Add the DAC – update code

18. Now we can add code to update the DAC-outputs with every loop of our main code. Recall, the objective of this lab was to generate a rising saw tooth voltage at DAC-output A and a falling saw tooth at DAC-output B.

    Obviously we need two integer variables “Voltage_A” and “Voltage_B” to store the current digital value for the two DAC-output lines. “Voltage_A” (rising saw tooth) starts with initial value 0, “Voltage_B” (falling saw tooth) with the maximum digital value 511. Add the two lines at begin of “main”:

    ```
    int Voltage_A = 0;

    int Voltage_B = 511;
    ```

    To update the DAC it would be a good solution to write a function called “DAC_Update” with two input parameters (channel_number and value). Add a prototype “void DAC_Update (char channel, int value)”at begin of the code and the definition at the end of your code:

    ```
    void DAC_Update(char channel, int value);
    ```

    In main, call the function “DAC_Update (channel, value) inside the “while(1)”-loop of main twice, just after the line:

    ```
    CpuTimer0.InterruptCount = 0;
    DAC_Update(‘B’,Voltage_B);
    DAC_Update(‘A’,Voltage_A);
    ```

    After the calls we need to increment “Voltage_A” and decrement “Voltage_B” and we have to reset “Voltage_A” if it exceeds the maximum of 511. Same if “Voltage_B” is decremented below 0:

    ```
    if (Voltage_A++ > 511) Voltage_A = 0;
    if (Voltage_B-- < 0) Voltage_B = 511;
    ```

19. Now, what should be done inside “DAC_Update(char channel, int value)”?

    Obviously, the activity depends on the selected channel. If channel == ‘B’ we have to load “value” into DAC-Buffer (see Slide 7-15, command R1, R0 = 01), if channel == ‘A’ we have to load “value direct on DAC-output channel A and update DAC-output channel B with value out of buffer (slide 7-15, command R1, R0 = 10). By doing so, we can make sure that both outputs are updated synchronously.

    Before we can load SPITXBUF with a data frame we have to enable the DAC’s chip select. For this purpose we defined port GPIO-D0. After the end of the transmission we have to disable it again.

    How can we find out if the transmission from the SPI into the DAC is completed? We did not enable any SPI-interrupts, so all we can do is to poll the SPI-interrupt flag to check if the SPI communication has finished. Note, that there will be still one bit to be transmitted after the SPI-interrupt has been set; therefore it is recommended to add another small wait loop before we switch off the Chip-Select signal of the DAC.

To reset the SPI-Interrupt Flag we have to do a dummy-read from SPIRXBUF.

Adding all the tiny bits of procedure step 19 together, your function "DAC_Update" should include this sequence:

```
int i;
GpioDataRegs.GPDDAT.bit.GPIOD0 = 0;        // activate /CS
if (channel == 'B')
    SpiaRegs.SPITXBUF =  0x1000 + (value<<2);
    // transmit data to DAC-Buffer
if (channel == 'A')
    SpiaRegs.SPITXBUF =  0x8000 + (value<<2);
    // transmit data to DAC-A and update DAC-B with Buffer
while (SpiaRegs.SPISTS.bit.INT_FLAG == 0) ;
// wait for end of transmission
for (i=0;i<100;i++);                       // wait for DAC to finish off
GpioDataRegs.GPDDAT.bit.GPIOD0 = 1;        // deactivate /CS
i = SpiaRegs.SPIRXBUF;                     // read to reset SPI-INT
```

Build, Load and Run

20. Click the "Rebuild All" button or perform:

    Project → Build
    File → Load Program
    Debug → Reset CPU
    Debug → Restart
    Debug → Go main
    Debug → Run(F5)

21. With the help of a scope meter or a scope you should use jumper JP7 and JP8 at the Zwickau Adapter board to verify the two saw tooth voltages. Recall that we used CPU Core Timer0, initialized to a time base of 50ms. In main, we wait for 3 increments of CpuTimer0.InterruptCount (150ms). We have 511 steps of increments for the DAC; therefore the period of the whole saw tooth is 511 * 150ms = 76.65s.

22. If you like it faster, modify the frequency of core timer0!

END of LAB 7

Lab 7A: Code Composer Studio Graph Tool

Objective

At the Zwickau Adapter Board we can re-connect the two saw tooth voltages of the SPI-DAC into the internal ADC. Doing so, we can verify the results of our DAC exercise (Lab7). Code Composer Studio has the ability to show the values of a memory area as a graphical image. The objective of exercise 7A is to use this feature.

The SPI-DAC channel A is connected to ADCINA1 and SPI-DAC channel B to ADCINB1 (Zwickau Adapter Board JP7 and JP8 closed). Note that any voltage above 3.0 to 3.3V will be saturated inside the ADC.

Use your code from Lab7 as a starting point.

Procedure

Open Files, Create Project File

1. Create a new project, called **Lab7A.pjt** in E:\C281x\Labs.

2. Open the file Lab7.c from E:\C281x\Labs\Lab7 and save it as Lab7A.c in E:\C281x\Labs\Lab7A.

3. Add the source code file to your project:
   - **Lab7A.c**

4. From *C:\tidcs\c28\dsp281x\v100\DSP281x_headers\source* add:
   - **DSP281x_GlobalVariableDefs.c**

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\cmd* add:
   - **F2812_EzDSP_RAM_lnk.cmd**

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_headers\cmd* add:
   - **F2812_Headers_nonBIOS.cmd**

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\source* add to project:
   - **DSP281x_PieCtrl.c**
   - **DSP281x_PieVect.c**
   - **DSP281x_DefaultIsr.c**

- **DSP281x_Adc.c**
- **DSP281x_CpuTimers.c**
- **DSP281x_usDelay.asm**

From *C:\ti\c2000\cgtoolslib* add:

- **rts2800_ml.lib**

Project Build Options

5. Setup the search path to include the peripheral register header files. Click:

   Project → Build Options

   Select the Compiler tab. In the preprocessor Category, find the Include Search Path (-i) box and enter:

   C:\tidcs\C28\dsp281x\v100\DSP281x_headers\include;
   ..\include

6. Setup the stack size: Inside Build Options select the Linker tab and enter in the Stack Size (-stack) box:

   400

   Close the Build Options Menu by Clicking <**OK**>.

Modify Source Code

7. Open Lab7A.c to edit: double click on “Lab7A.c” inside the project window. Inside function “InitSystem” enable the ADC clock system:

   SysCtrlRegs.PCLKCR.bit.ADCENCLK=1;

8. In main, just after the function call of “SPI_Init()” call the initialization for the ADC:

   InitAdc();

   This will initialize the ADC’s internal voltages.

9. Direct after the call of “InitAdc” add the remaining steps to initialize the ADC. Eventually you can use your ADC – setup from lab exercise 6. Take into account:

   - Setup Dual Sequencer Mode
   - No Continuous run
   - Adc - Prescaler = 1 (CPS)

- 2 conversions per start of conversion
- Channels ADCINA1 & ACINB1
- Disable EVASOC to start SEQ1
- Enable SEQ1 interrupt every End of Sequence
- ADCCLKPS = 2 (Divide HSPCLK by 4)

10. Next, add an overload for the PIE - vector table (ADCINT) to point to a new function "ADC_ISR()":

    EALLOW;

    PieVectTable.ADCINT = &ADC_ISR;

    EDIS;

    Also, add the enable instruction for the ADC-Interrupt:

    PieCtrlRegs.PIEIER1.bit.INTx6 = 1;

11. Add the provided source code for function "ADC_ISR" to your project. From *E:\C281x\Labs\Lab7A* add to project:

    - **ADC_ISR.c**

12. Add a function prototype for interrupt service routine "ADC_ISR" at the begin of your code in "Lab7A.c":

    interrupt void ADC_ISR(void);

13. Inside the while(1)-loop of main, just after the line "CpuTimer0.InterruptCount = 0;" add the instruction to start a conversion of the ADC:

    AdcRegs.ADCTRL2.bit.SOC_SEQ1 = 1;

Build and Load

14. Click the "Rebuild All" button or perform:

    Project → Build

    and watch the tools run in the build window. If you get syntax errors or warnings debug as necessary.

15. Load the output file down to the DSP. Click:

    File → Load Program and choose the desired output file.

Test

16. Reset the DSP by clicking on:

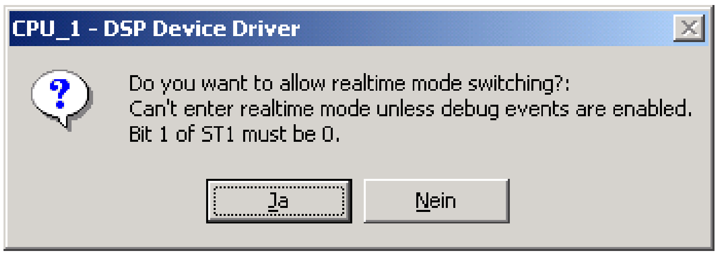
Debug → Reset CPU	followed by
Debug ➔ Restart	and
Debug → Go main.	

Run

17. Enable Real Time Debug. Click on

    ➔ Debug ➔ Real Time mode

    Answer the following window with: Yes



CPU_1 - DSP Device Driver

Do you want to allow realtime mode switching?:
Can't enter realtime mode unless debug events are enabled.
Bit 1 of ST1 must be 0.

18. Open the Watch – Window and add the ADC – Result Register “ADCRESULT0” and “ADCRESULT1” to it. Note: you can do this manually, or (recommended) use the GEL-Menu:

    ➔ GEL ➔ Watch ADC Registers ➔ ADCRESULT_0_to_3

    Inside the Watch Window Click right mouse and select “Continuous Refresh”.

19. Run the Code:

    ➔ Debug ➔ Run (F5)

    ADCRESULT0 should show rising values, ADCRESULT 1 falling values. Stop the DSP.

Add a graphical window

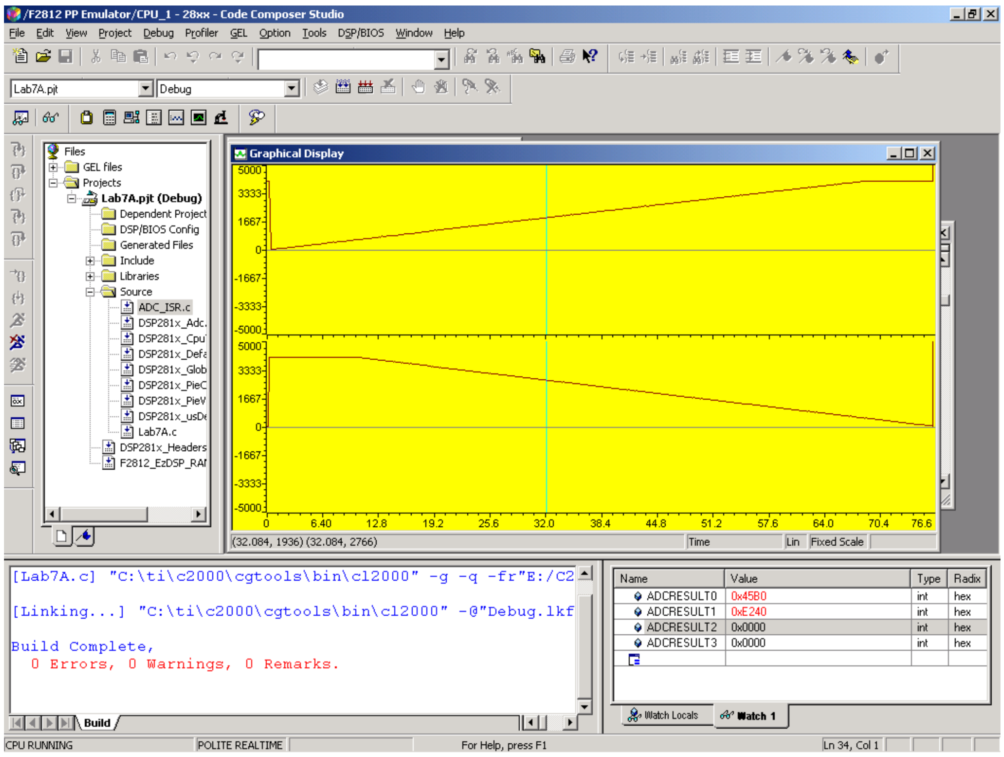
20. To visualize the measured saw tooth voltages open a graph window:

    ➔ View ➔ Graph ➔ Time/Frequency

    and enter the following properties:

Display Type	Dual Time
Start Address Upper Display	AdcBuf_A
Start Address Lower Display	AdcBuf_B
Acquisition Buffer Size	512
Display Data Size	512
Sampling Rate (Hz)	6.67
DSP Data Type	16-Bit unsigned integer
Auto scale	OFF
Time Display Unit	s

Click right mouse inside the graph window and select "Continuous Refresh" and run the code again.

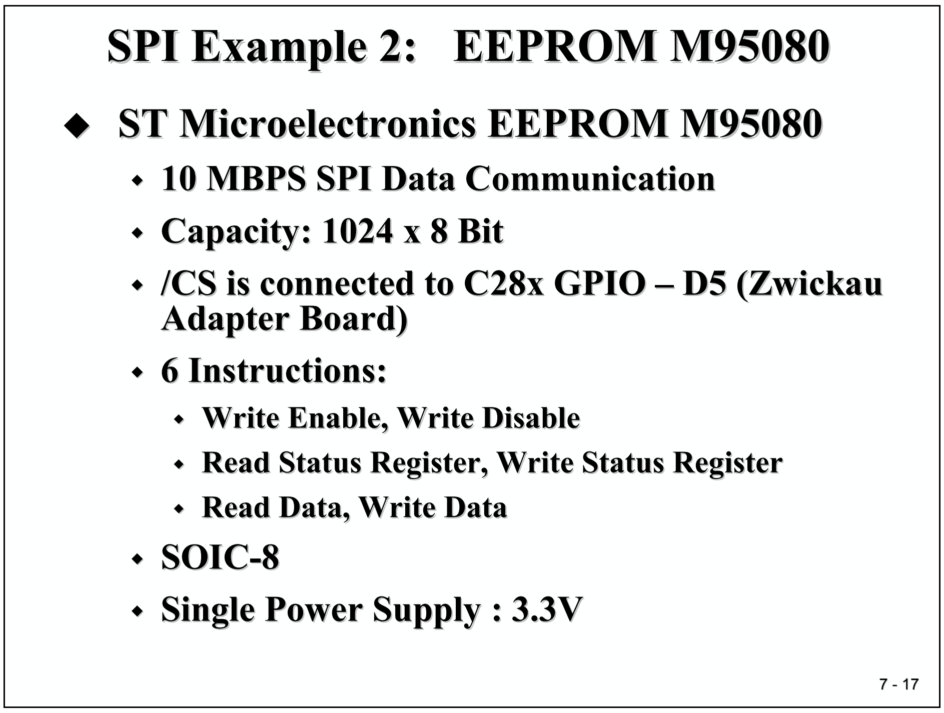


```
[Lab7A.c] "C:\ti\c2000\cgtools\bin\cl2000" -g -q -fr"E:/C2

[Linking...] "C:\ti\c2000\cgtools\bin\cl2000" -@"Debug.lkf

Build Complete,
  0 Errors, 0 Warnings, 0 Remarks.
```

Name	Value	Type	Radix
ADCRESULT0	0x45B0	int	hex
ADCRESULT1	0xE240	int	hex
ADCRESULT2	0x0000	int	hex
ADCRESULT3	0x0000	int	hex

END of LAB 7A

Lab 7B: SPI – EEPROM M95080



SPI Example 2: EEPROM M95080

- ST Microelectronics EEPROM M95080
  - 10 MBPS SPI Data Communication
  - Capacity: 1024 x 8 Bit
  - /CS is connected to C28x GPIO – D5 (Zwickau Adapter Board)
  - 6 Instructions:
    - Write Enable, Write Disable
    - Read Status Register, Write Status Register
    - Read Data, Write Data
  - SOIC-8
  - Single Power Supply : 3.3V

7 - 17

Objective

The objective of this lab is to establish an SPI communication between the C28x and a serial EEPROM ST M95080. The interface between the C28x and this 1024 x 8 Bit - EEPROM is the standard SPI and uses the following connections:

M95080 – Signal	Pin #	Description	Connected to...
VSS	4	Ground	GND
VCC	8	3.3V	3.3V
C	6	SPI Clock	C28x - SPICLK
D	5	SPI Data In	C28x –SPISIMO
Q	2	SPI Data OUT	C28x - SPISOMI
/S	1	Chip Select	C28x – GPIO D5
/W	3	Write protect	3.3V
/HOLD	7	Hold Communication	3.3V

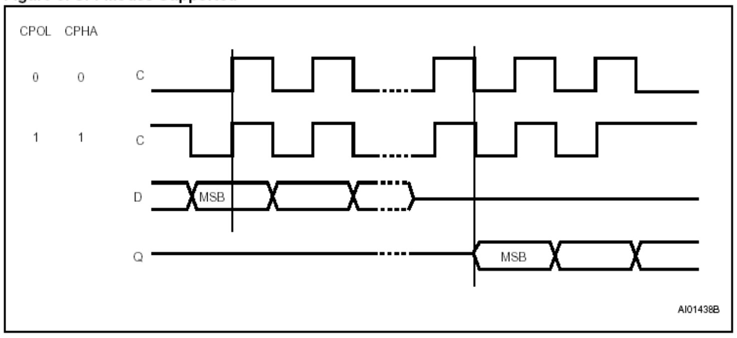
The chip-select (/CS) of the EEPROM is connected to the GPIO – D5. The EEPROM is able to store data non-volatile. This means we need to setup a closed SPI – loop to write and to read data. The EEPROM data input 'D' is connected to the DSP's 'SIMO' (Slave In - Master Out) and the EEPROM's output line 'Q' drives serial signals to 'SOMI' (Slave Out – Master In). The signals 'Write Protect' and '/HOLD' are not used for this experiment.

The M95080 has the following timing requirements:

SPI Example : EEPROM M95080

◆ **Timing Diagram:**

Figure 5. SPI Modes Supported



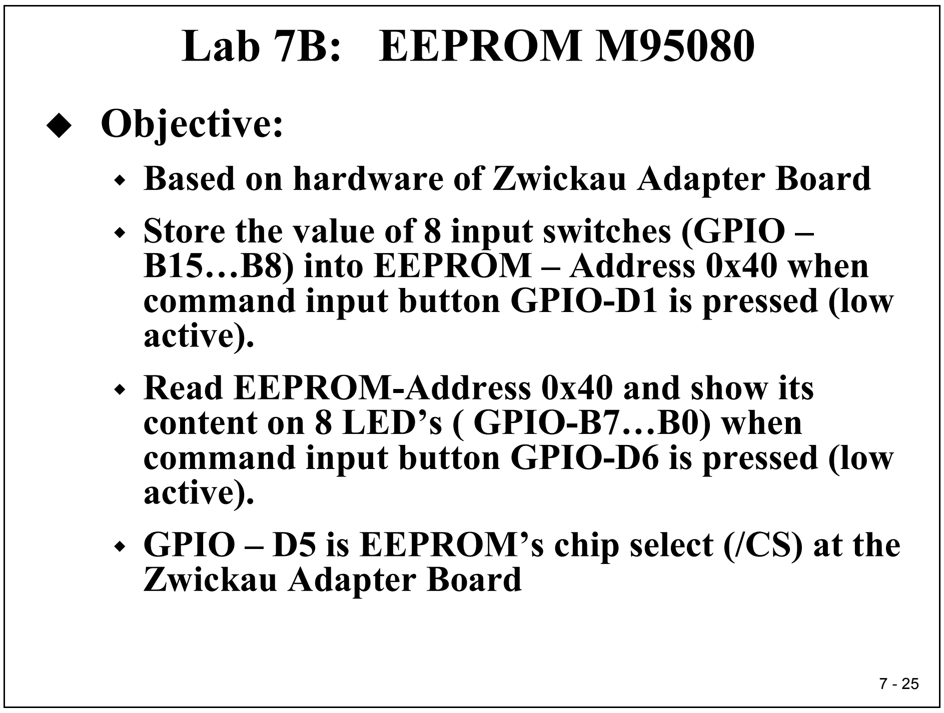
© ST Microelectronics Datasheet M95080 – November 2002, page 4

7 - 18

To write data into the EEPROM the DSP has to generate the data bit first; with a clock delay of ½ cycles the rising edge is the strobe pulse for the EEPROM to store the data. When reading the EEPROM the falling clock edge causes the EEPROM to send out data. With the rising clock edge the DSP can read the valid data bit. The passive state for the clock line is selectable to be high or low.

Aim of Lab 7B:

The Aim of the Lab 7B is to store the data byte derived from the 8 input switches (GPIO B15…B8) in EEPROM address 0x40 when button GPIO-D1 (yellow) is pushed. When button GPIO-D6(red) is pushed the data byte from EEPROM address 0x40 should be read back and shown at GPIO B7…B0 (8 LED's). The program should sample the two command buttons D1 and D6 every 200 ms, forced by CPU Timer0.



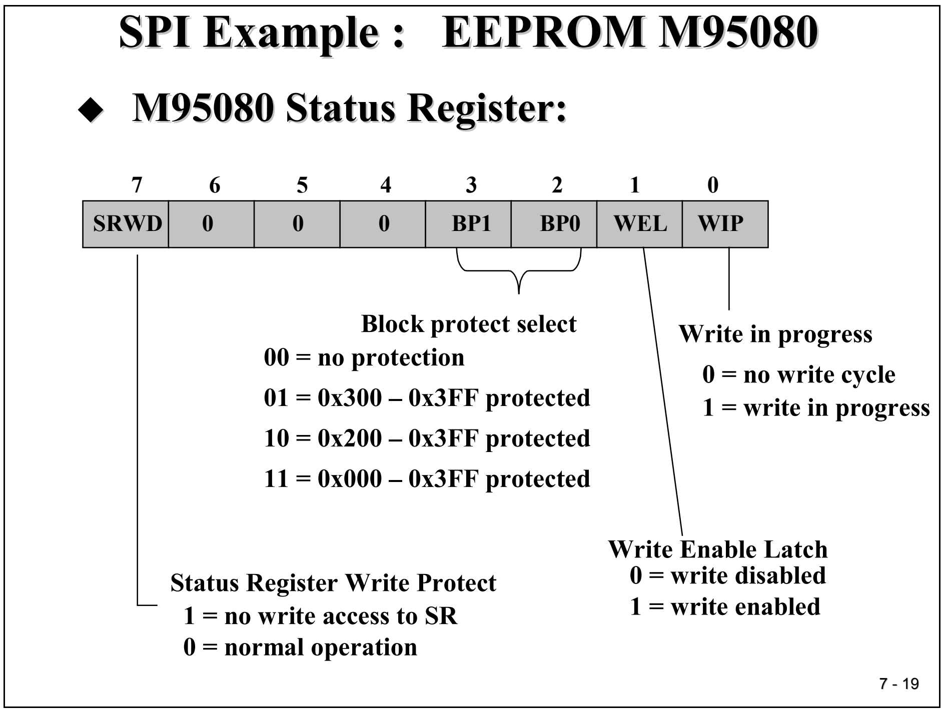
Lab 7B: EEPROM M95080

- **Objective:**
  - **Based on hardware of Zwickau Adapter Board**
  - **Store the value of 8 input switches (GPIO – B15…B8) into EEPROM – Address 0x40 when command input button GPIO-D1 is pressed (low active).**
  - **Read EEPROM-Address 0x40 and show its content on 8 LED's (GPIO-B7…B0) when command input button GPIO-D6 is pressed (low active).**
  - **GPIO – D5 is EEPROM's chip select (/CS) at the Zwickau Adapter Board**

7 - 25

M95080 Status Register

The M95080 Status Register controls write accesses to the internal memory.



SPI Example : EEPROM M95080

- **M95080 Status Register:**

7	6	5	4	3	2	1	0
SRWD	0	0	0	BP1	BP0	WEL	WIP

7 - 19

It also flags the current status of the EEPROM. Bit 0 (“WIP’) flags whether an internal write cycle is in progress or not. Internal write cycles are started at the end of a command sequence and last quite long (maximum 10ms). To avoid the interruption of a write cycle in progress any other write access should be delayed as long as WIP=1.

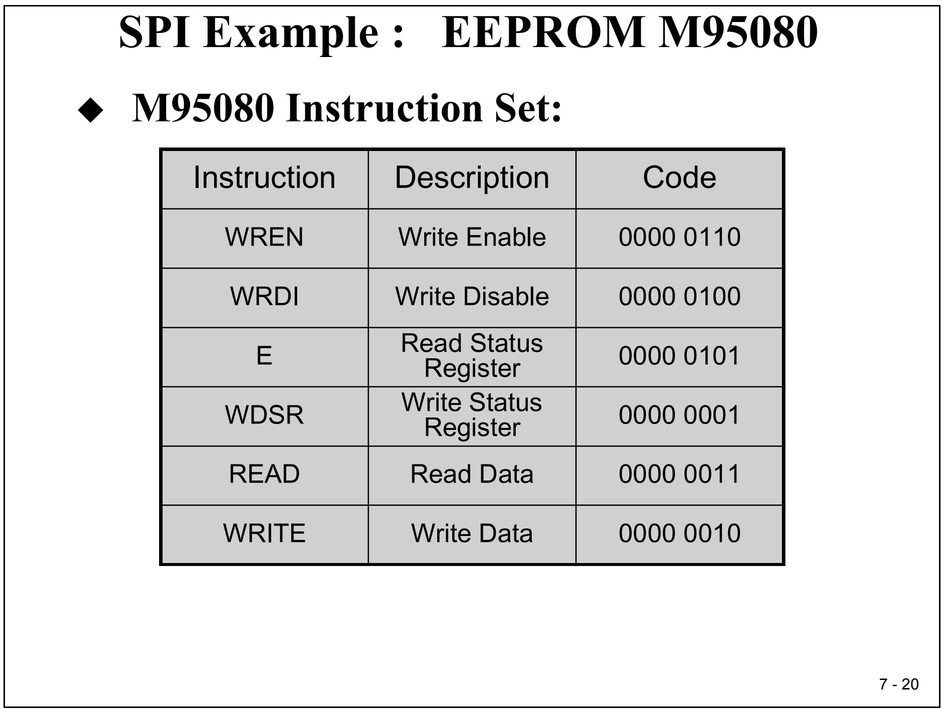
Bit 1(“Write Enable Latch”) is a control bit that must be set to 1 for every write access to the EEPROM. After a successful write cycle this bit is cleared by the EEPROM.

Bits 3 and 2(“Block Protect Select”) are used to define the area of memory that should be protected against any write access. We will not use any protection in our lab, so just set the two bits to ‘00’.

Bit 7 (“Status Register Write Protect”) allows us to disable any write access into the Status Register. For our Lab we will leave this bit cleared all the time (normal operation).

M95080 Instruction Set

To communicate with the M95080 we have to use the following table of instructions. An instruction is the first part of the serial sequence of data between the DSP and the EEPROM.



SPI Example : EEPROM M95080

- M95080 Instruction Set:

Instruction	Description	Code
WREN	Write Enable	0000 0110
WRDI	Write Disable	0000 0100
E	Read Status Register	0000 0101
WDSR	Write Status Register	0000 0001
READ	Read Data	0000 0011
WRITE	Write Data	0000 0010

7 - 20

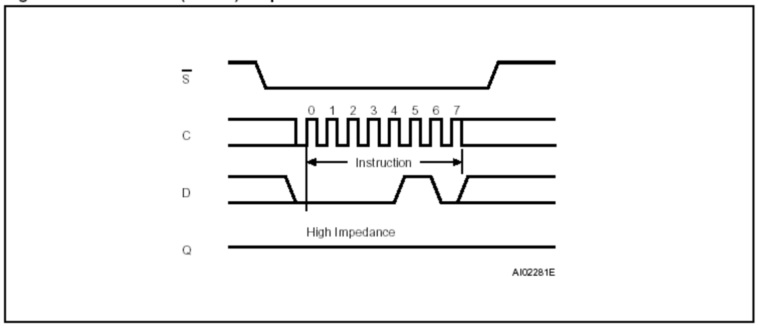
Before we can start our Lab procedure we have to discuss these instructions a little bit more in detail.

The **WREN** command must be applied to the EEPROM to the Write Enable Latch (WEL) **prior to each** WRITE and WRSR instruction. The command is an 8-clock SPI- sequence shown on the next slide:

SPI Example : EEPROM M95080

- **Timing Diagram WREN:**

Figure 7. Write Enable (WREN) Sequence



© ST Microelectronics ; Datasheet (8028.pdf) – November 2002; Page 8

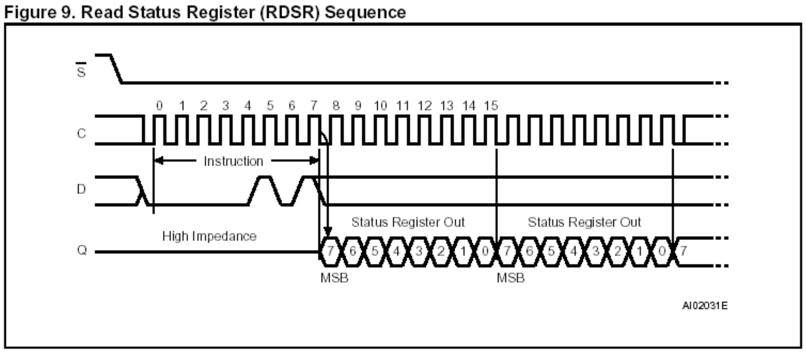
7 - 21

The **RDSR** instruction allows the Status Register to be read. The Status Register may be read any time. It is recommended to use this instruction to check the "Write In Progress" (WIP) bit **before** sending a new instruction to the EEPROM. This is also possible to read the Status Register continuously.

SPI Example : EEPROM M95080

- **Timing Diagram RDSR:**

Figure 9. Read Status Register (RDSR) Sequence



© ST Microelectronics ; Datasheet (8028.pdf) – November 2002; Page 10

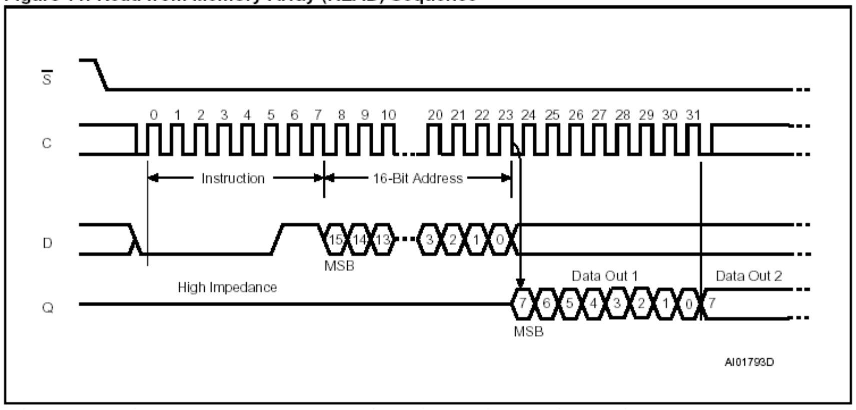
7 - 22

The **READ** instruction is used to read data out of the EEPROM. The address range of the M95080 is from 0 to 1023. The address is given as a full 16 bit address; bits A15 to A10 are don't cares. As shown with the next figure, an internal address counter is incremented with each READ instruction, if chip select (/S) continues to be driven low:

SPI Example : EEPROM M95080

- **Timing Diagram READ:**

Figure 11. Read from Memory Array (READ) Sequence



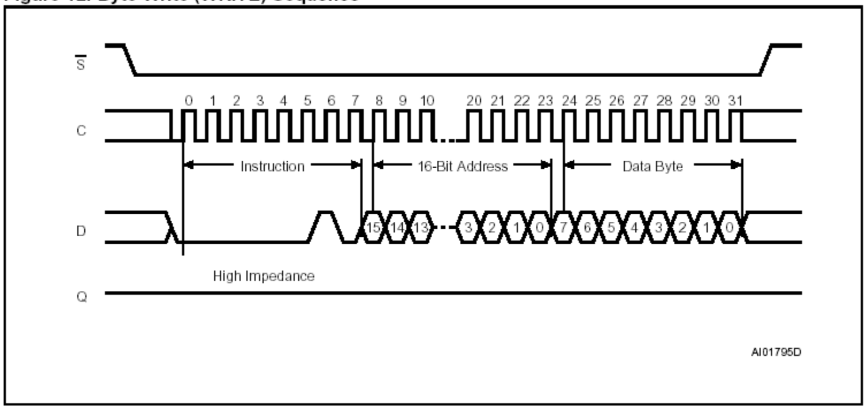
© ST Microelectronics ; Datasheet (8028.pdf) – November 2002; Page 13

7 - 23

SPI Example : EEPROM M95080

- **Timing Diagram WRITE:**

Figure 12. Byte Write (WRITE) Sequence



© ST Microelectronics ; Datasheet (8028.pdf) – November 2002; Page 14

7 - 24

The **WRITE** instruction is used to write data into the EEPROM. The instruction is terminated by driving Chip Select (/S) high. At this point the internal self timed write cycle actually starts, at the end of which the "Write In Progress "(WIP) bit of the Status Register is reset to 0.

Procedure Lab7B

Open Files, Create Project File

1. Create a new project, called **Lab7B.pjt** in E:\C281x\Labs.

2. Open the file Lab4.c from E:\C281x\Labs\Lab4 and save it as Lab7B.c in E:\C281x\Labs\Lab7B.

3. Add the source code file to your project:
   - **Lab7B.c**

4. From *C:\tidcs\c28\dsp281x\v100\DSP281x_headers\source* add:

   - **DSP281x_GlobalVariableDefs.c**

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_headers\cmd* add:

   - **F2812_Headers_nonBIOS.cmd**

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\cmd* add:

   - **F2812_EzDSP_RAM_lnk.cmd**

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\source* add to project:

   - **DSP281x_PieCtrl.c**
   - **DSP281x_PieVect.c**
   - **DSP281x_DefaultIsr.c**
   - **DSP281x_CpuTimers.c**

   From *C:\ti\c2000\cgtoolslib* add:

   - **rts2800_ml.lib**

Project Build Options

5. Setup the search path to include the peripheral register header files. Click:

   Project → Build Options

Select the Compiler tab. In the preprocessor Category, find the Include Search Path (-i) box and enter:

```
C:\tidcs\C28\dsp281x\v100\DSP281x_headers\include;
..\include
```

6. Setup the stack size: Inside Build Options select the Linker tab and enter in the Stack Size (-stack) box:

   400

   Close the Build Options Menu by Clicking <**OK**>.

Modify Source Code

7. Open Lab7B.c to edit: double click on "Lab7B.c" inside the project window. First we have to cancel the parts of the code that we do not need any longer. We will not use the main variables "LED[8]" and "i" for this exercise:

   At beginning of main, delete the lines:

```
unsigned int I;
unsigned int LED[8]= {0x0001,0x0002,0x0004,0x0008,
                      0x0010,0x0020,0x0040,0x0080};
```

8. Next, inside the "while(1)"-loop of main reduce the code to just the following lines (we will add some more code later):

```
while(1)
{
        while(CpuTimer0.InterruptCount < 3); // wait for Timer 0
        CpuTimer0.InterruptCount = 0;
        EALLOW;
        SysCtrlRegs.WDKEY = 0xAA;       // and service watchdog #2
        EDIS;
}
```

9. Before we continue to add the SPI modifications lets test if the project in its preliminary stage runs as expected. Recall, with the start code of Lab4 we initialized the CPU core timer 0 to generate an interrupt request every 50ms. The interrupt service routine "cpu_timer0_isr()" increments a global variable"CpuTimer0.InterruptCount" with every hit. If everything works as expected the DSP should hit the line

```
        CpuTimer0.InterruptCount = 0;
```

   in the while(1) – loop (procedure step 8) every 3*50ms = 150 ms.

Build and Load

10. Click the "Rebuild All" button or perform:

    Project → Build

    and watch the tools run in the build window. If you get syntax errors or warnings debug as necessary.

11. Load the output file down to the DSP Click:

    File → Load Program and choose the desired output file.

Test

12. Reset the DSP by clicking on:

    Debug → Reset CPU followed by
    Debug → Restart and
    Debug → Go main.

13. In the "while(1)"-loop of main set a breakpoint at line:
    CpuTimer0.InterruptCount = 0;

Run

14. Verify, that the breakpoint is hit periodically when you start the DSP by:

    Debug → Run (F5).

    Remove the breakpoint when you are done.

Add the SPI initialization code

15. So far we verified that the CPU Core Timer0 generates a period of 50 ms and that our main-loop waits for 3 periods of Timer0 before it moves to the next instruction. Now we need to add the code for the SPI to control the EEPROM M95080. First function that is called is "InitSystem()". Again we have to enable the SPI clock unit. Inside "InitSystem()" set:

    SysCtrlRegs.PCLKCR.bit.SPIENCLK=1;

16. The next function that is called in main is "Gpio_select()". Inside this function we have to modify the multiplex register to use the four SPI-signals:

    GpioMuxRegs.GPFMUX.all = 0xF;

Bits 3...0 control the setup for SPISTEA, SPICLKA, SPISOMIA and SPISIMOA. We also prepare GPIO-signal D0 and D5 to be digital outputs. D0 will be used as chip-select for the TLV5617A and D5 for the EEPROM M95080 (used in Lab7B). To do this, setup GPDDIR register:

GpioMuxRegs.GPDDIR.all=0;

GpioMuxRegs.GPDDIR.bit.GPIOD0 = 1; // /CS for DAC TLV5617A

GpioMuxRegs.GPDDIR.bit.GPIOD5 = 1; // /CS for EEPROM

As an initial state we should switch off both /CS-signals as well as the LED's at B7...B0:

GpioDataRegs.GPBDAT.all = 0x0000; // Switch off LED's (B7...B0)

GpioDataRegs.GPDDAT.bit.GPIOD0 = 1; // /CS for DAC off

GpioDataRegs.GPDDAT.bit.GPIOD5 = 1; // /CS for EEPROM off

17. In main, just before we enter the "while(1)"-loop add a function call to function "SPI_Init()". Also add a function prototype at the start of your code.

At the end of your code, add the definition of function "SPI_Init()".

Inside this function, include the following steps:

- SPICCR:
  - Clock polarity =1: data output at falling edge of clock
  - 16 bit per data frame
- SPICTL:
  - No SPI interrupts for this exercise
  - Master on
  - Talk enabled
  - Clock phase = 0: no phase shift
- SPIBRR:
  - BRR = LSPCLK/SPI_Baudrate – 1
  - Example: assuming LSPCLK = 37.5MHz and SPIBRR = 124 the SPI-Baud rate is 300 kBPS.

Create EEPROM access functions

18. Now we have to develop the code to access the SPI – EEPROM. A good method would be to write four specific functions to:

    - Read the EEPROM Status Register:
        - int SPI_EEPROM_Read_Status(void)
    - Set the Write Enable Latch:
        - void SPI_EEPROM_Write_Enable(void)
    - Write 8 bit into the EEPROM:
        - void SPI_EEPROM_Write(int address, int data)
    - Read 8 bit out of the EEPROM:
        - int SPI_EEPROM_Read(int address)

19. Function "**SPI_EEPROM_Read_Status**"

    - At the begin of the function activate the chip select (D5) signal for the EEPROM:

      GpioDataRegs.GPDDAT.bit.GPIOD5 = 0;

    - Next, load the code for "Read Status Register" into the SPI-Transmit buffer "SPITXBUF". Take care of the correct alignment when loading this 8 bit code into "SPITXBUF".

    - Before we can continue we will have to wait for the end of the SPI transmission. Because we did disable all SPI interrupts we can't use an interrupt driven synchronization, all we can do now is to 'poll' the SPI INT flag:

      while(SpiaRegs.SPISTS.bit.INT_FLAG = = 0);

    - Now we can read the status out of SPIRXBUF and return it to the calling function.

    - At the end of the function we have to deactivate the EEPROM's chip select (D5).

20. Function "**SPI_EEPROM_Write_Enable** "

    - This function is used to set the Write Enable Latch of the EEPROM and must be called before every write access.

    - At begin of the function activate the chip select (D5) signal for the EEPROM.

- Next, load the code for "Write Enable" into the SPI-Transmit buffer "SPITXBUF". Take care of the correct alignment when loading this 8 bit code into "SPITXBUF".

- 'Poll' the SPI INT flag to wait for the end of SPI transmission.

- To reset the SPI INT flag we have to execute a "dummy"-read from SPIRXBUF:

  i = SpiaRegs.SPIRXBUF;

- Switch off the EEPROM's chip select (D5).

21. Function "**SPI_EEPROM_Write**"

- This function is used to write a character of 8 bit into the EEPROM. The input parameters are (1) the EEPROM – Address (16 bit integer) and (2) the data (16-bit integer – only the 8 least significant bits (LSB's) are used)
- At the beginning of the function, activate the chip select (D5) signal for the EEPROM.
- Next, load the code for "Write" and the upper half of the address parameter into the SPI-Transmit buffer "SPITXBUF".
- 'Poll' the SPI INT flag to wait for the end of SPI transmission.
- To reset the SPI INT flag do a "dummy"-read from SPIRXBUF.
- Next, load the lower half of the address parameter and the lower half of the data parameter into the SPI-Transmit buffer "SPITXBUF".
- 'Poll' the SPI INT flag to wait for the end of SPI transmission.
- To reset the SPI INT flag once more, do a "dummy"-read from SPIRXBUF.
- Switch off the EEPROM's chip select (D5).

22. Function "**SPI_EEPROM_Read**"

- This function returns a 16-bit integer (only lower 8 bits are used) to the calling function. The input parameter is the EEPROM address to be read (16-bit integer).
- At the beginning of the function, activate the chip select (D5) signal for the EEPROM.
- Next, load the code for "Read" and the upper half of the address parameter into the SPI-Transmit buffer "SPITXBUF".
- 'Poll' the SPI INT flag to wait for the end of SPI transmission.
- Reset the SPI INT flag by a "dummy"-read from SPIRXBUF.
- Next, load the lower half of the address parameter into the SPI-Transmit buffer "SPITXBUF" (take care of correct alignment).
- 'Poll' the SPI INT flag to wait for the end of SPI transmission.
- Read SPIRXBUF and return its value to the calling function
- Switch off the EEPROM's chip select (D5).

Finalize the main loop

23. Finally we have to add some function calls to our main loop to use the EEPROM. The objective of this lab exercise is to store the status of the 8 input switches (GPIO

B15-B8) into EEPROM address 0x40 when a first command button (GPIO D1) is pushed(D1 = 0). If a second command button (GPIO D6) is pushed(D6 = 0) the program should read EEPROM address 0x40 and load the value onto 8 LED's (GPIO B7-B0).

- To write into address 0x40:
  - Call function "SPI_EEPROM_Write_Enable"
  - Verify that flag "WEL" is set (= 1) by calling function "SPI_EEPROM_Read_Status"
  - Call function "SPI_EEPROM_Write"
  - Wait for the end of write by polling flag "WIP". It will be cleared at the very end of an internal EEPROM write sequence. Use function "SPI_EEPROM_Read_Status" to poll "WIP" periodically.
- To read from address 0x40:
  - Call function "SPI_EEPROM_Read()" and transfer the return value onto GPIO B7-B0 (8 LED's).

Build, Load and Run

24. Click the "Rebuild All" button or perform:

    Project → Build
    File → Load Program
    Debug → Reset CPU
    Debug ➔ Restart
    Debug → Go main
    Debug ➔ Run (F5)

25. Modify the input switches and push GPIO-D1. Next, push GPIO-D6. The LED's should mirror the last state of the input switches.

26. Close Code Composer Studio and switch off the eZdsp. If you restart the board and Code Composer Studio after a few seconds to start another debug session begin with the read button D6. Now the LED's should again show the last value that was stored inside the EEPROM (it's a non volatile memory, keeps its information also when power supply has been switched off).

END of LAB 7B

This page was intentionally left blank.