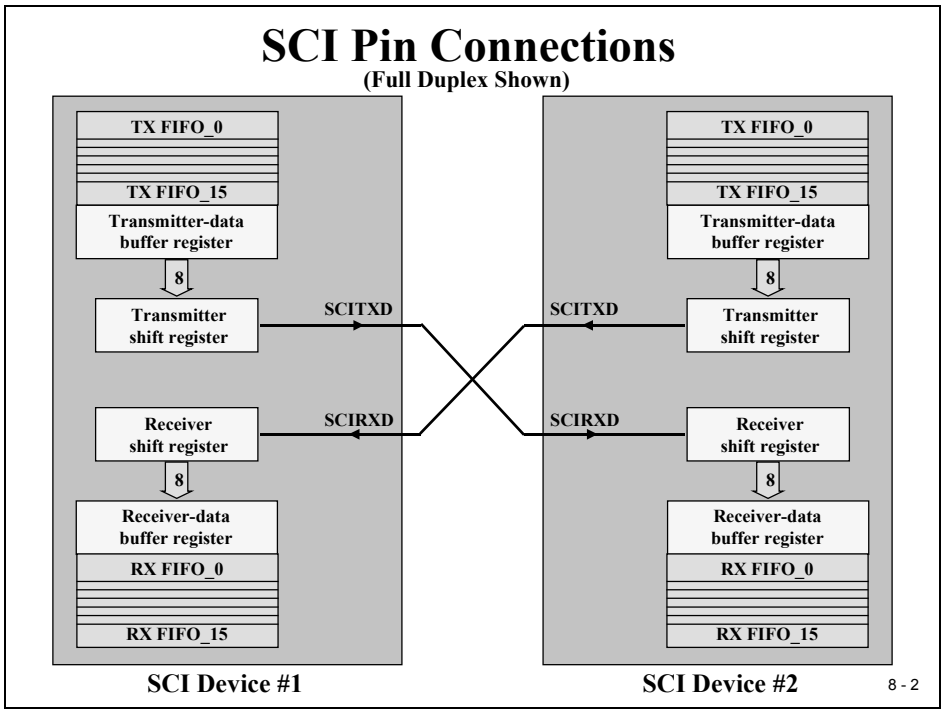
C28x Serial Communication Interface

Introduction

The Serial Communication Interface (SCI) module is a serial I/O port that permits asynchronous communication between the C28x and other peripheral devices. It is usually known as a UART (Universal Asynchronous Receiver Transmitter) and is used according to the RS232 standard. To allow for efficient CPU usage, the SCI transmit and receive registers are both FIFO-buffered to prevent data collisions. In addition, the C28x SCI has a full duplex interface, which provides for simultaneous data transmit and receive. Parity checking and data formatting can also be done by the SCI port hardware, further reducing the software overhead.

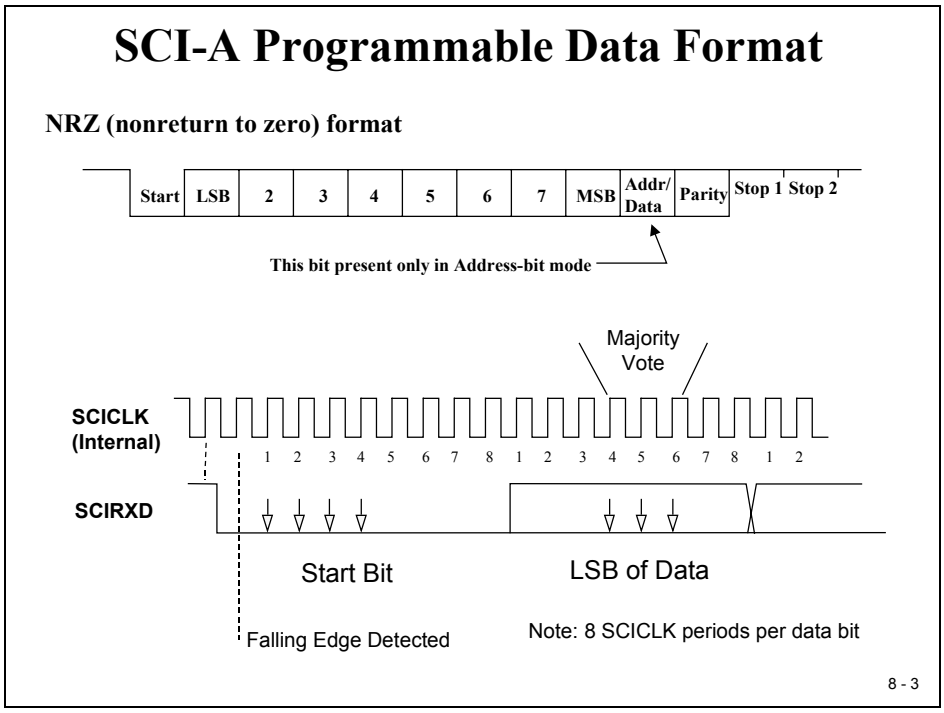


Module Topics

SCI Data Format

The basic unit of data is called a **character** and is 1 to 8 bits in length. Each character of data is formatted with a start bit, 1 or 2 stop bits, an optional parity bit, and an optional address/data bit. A character of data along with its formatting bits is called a **frame**. Frames are organized into groups called blocks. If more than two serial ports exist on the SCI bus, a block of data will usually begin with an address frame, which specifies the destination port of the data as determined by the user's protocol.

The start bit is a low bit at the beginning of each frame, which marks the beginning of a frame. The SCI uses a NRZ (Non-Return-to-Zero) format, which means that in an inactive state the SCIRX and SCITX lines will be held high. Peripherals are expected to pull the SCIRX and SCITX lines to a high level when they are not receiving or transmitting on their respective lines.



SCI Multi Processor Wake Up Modes

Multiprocessor Wake-Up Modes

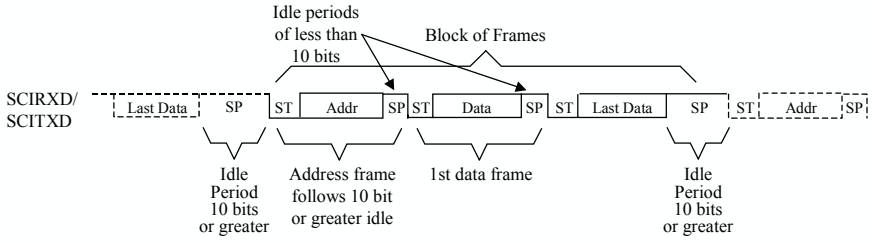
- **Allows numerous processors to be hooked up to the bus, but transmission occurs between only two of them**
- **Idle-line or Address-bit modes**
- **Sequence of Operation**
  1. Potential receivers set SLEEP = 1, which disables RXINT except when an address frame is received
  2. All transmissions begin with an address frame
  3. Incoming address frame temporarily wakes up all SCIs on bus
  4. CPUs compare incoming SCI address to their SCI address
  5. Process following data frames only if address matches

8 - 4

Although a SCI data transfer is usually a point-to-point communication, the C28x SCI interface allows two operation modes to communicate between a master and more than one slave.

Idle-Line Wake-Up Mode

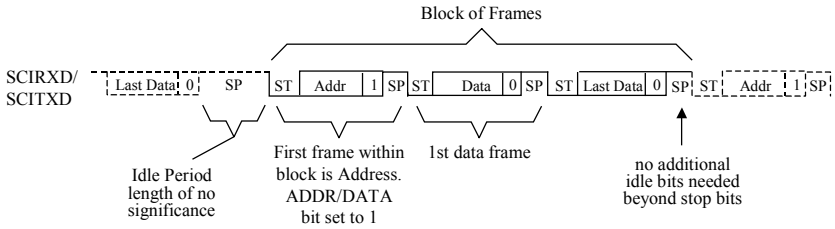
- **Idle time separates blocks of frames**
- **Receiver wakes up with falling edge after SCIRXD was high for 10 or more bit periods**
- **Two transmit address methods**
  - **deliberate software delay of 10 or more bits**
  - **set TXWAKE bit to automatically leave exactly 11 idle bits**



8 - 5

Address-Bit Wake-Up Mode

- All frames contain an extra address bit
- Receiver wakes up when address bit detected
- Automatic setting of Addr/Data bit in frame by setting TXWAKE = 1 prior to writing address to SCITXBUF



8 - 6

SCI Summary

- Asynchronous communications format
- 65,000+ different programmable baud rates
- Two wake-up multiprocessor modes
  - Idle-line wake-up & Address-bit wake-up
- Programmable data word format
  - 1 to 8 bit data word length
  - 1 or 2 stop bits
  - even/odd/no parity
- Error Detection Flags
  - Parity error; Framing error; Overrun error; Break detection
- FIFO-buffered transmit and receive
- Individual interrupts for transmit and receive

8 - 7

SCI Register Set

The next slide summarizes all SCI control registers for SCI channel A. Note that there is a second SCI channel B available in the C28x.

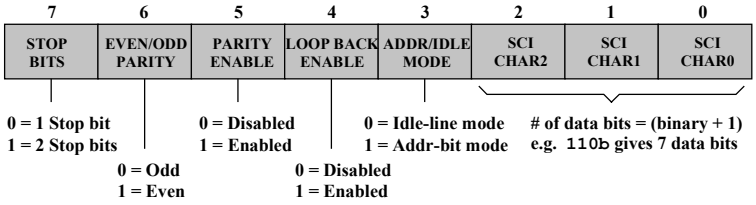
SCI-A Registers

Address	Register	Name
0x007050	SCICCR	SCI-A commun. control register
0x007051	SCICTL1	SCI-A control register 1
0x007052	SCIHBAUD	SCI-A baud register, high byte
0x007053	SCILBAUD	SCI-A baud register, low byte
0x007054	SCICTL2	SCI-A control register 2 register
0x007055	SCIRXST	SCI-A receive status register
0x007056	SCIRXEMU	SCI-A receive emulation data buffer
0x007057	SCIRXBUF	SCI-A receive data buffer register
0x007059	SCITXBUF	SCI-A transmit data buffer register
0x00705A	SCIFFTX	SCI-A FIFO transmit register
0x00705B	SCIFFRX	SCI-A FIFO receive register
0x00705C	SCIFFCT	SCI-A FIFO control register
0x00705F	SCIPRI	SCI-A priority control register

8 - 8

SCI-A Communication Control Register

Communications Control Register (SCICCR) – 0x007050



[SCI-B Communications Control Register (SCICCR) – 0x007750]

8 - 9

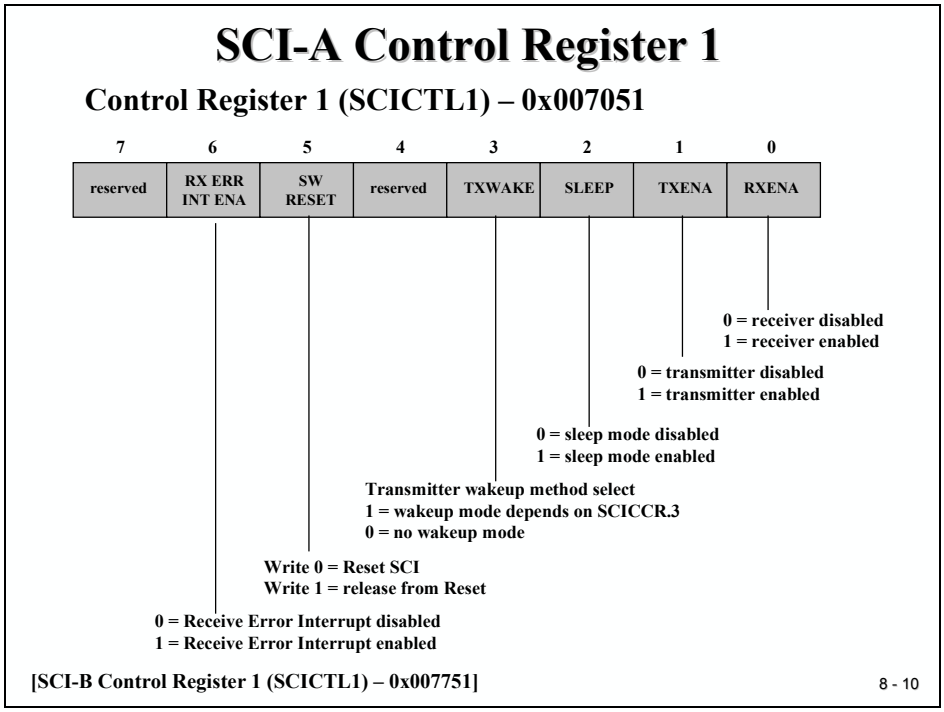
SCI Communications Control Register (SCICCR)

The previous slide explains the setup for the SCI data frame structure. If Multi Processor Wakeup Mode is not used, bit 3 should be cleared. This avoids the generation of an additional address/data selection bit at the end of the data frame (see slide 8-3). Some hosts or other devices are not able to handle this additional bit.

The other bit fields of SCICCR can be initialized, as you like. For our lab exercises in this chapter we will use:

- **8 data bit per character**
- **no parity**
- **1 Stop bit**
- **loop back disabled**

SCI Control Register 1(SCICTL1)



When configuring the SCICCR, the SCI port should first be held in an inactive state. This is done using the SW RESET bit of the SCI Control Register 1 (SCICTL1.5). Writing a 0 to this bit initializes and holds the SCI state machines and operating flags at their reset condition. The SCICCR can then be configured. Afterwards, re-enable the SCI port by writing a 1 to the SW RESET bit. At system reset, the SW RESET bit equals 0.

For our Lab exercises we will not use wakeup or sleep features (SCICTL1.3 = 0 and SCICTL1.2 = 0).

Depending on the direction of the communication we will have to enable the transmitter (SCICTL1.1 = 1) or the receiver (SCICTL1.0 = 1) or both.

For a real project we would have to think about potential communication errors. The receiver error could then be allowed to generate a receiver error interrupt request (SCICTL1.6 = 1). Our first labs will not use this feature.

SCI Baud Rate Register

SCI-A Baud Rate

$$\text{SCI baud rate} = \begin{cases} \dfrac{\text{LSPCLK}}{(\text{BRR} + 1) \times 8}, & \text{BRR} = 1 \text{ to } 65535 \\ \dfrac{\text{LSPCLK}}{16}, & \text{BRR} = 0 \end{cases}$$

Baud-Select MSbyte Register (SCIHBAUD) – 0x007052

7	6	5	4	3	2	1	0
BAUD15 (MSB)	BAUD14	BAUD13	BAUD12	BAUD11	BAUD10	BAUD9	BAUD8

Baud-Select LSbyte Register (SCILBAUD) – 0x007053

7	6	5	4	3	2	1	0
BAUD7	BAUD6	BAUD5	BAUD4	BAUD3	BAUD2	BAUD1	BAUD0 (LSB)

[SCI-B Baud-Select MSbyte Register (SCIHBAUD) – 0x007752]
[SCI-B Baud-Select LSbyte Register (SCILBAUD) – 0x007753]

8 - 11

The baud rate for the SCI is derived from the low speed pre-scaler (LSPCLK).

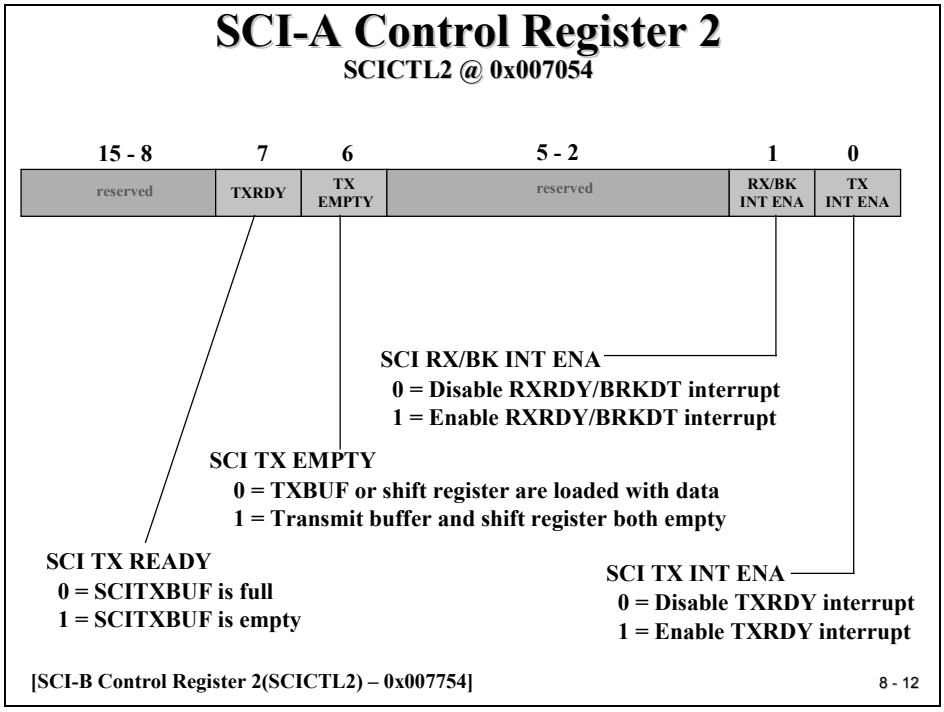
Assuming a SYSCLK frequency of 150MHz and a low speed pre-scaler initialized to “divide by 4” we can calculate the set up for BRR to initialize, let’s say a baud rate of 9600 baud:

$$9.600Hz = \frac{37.5MHz}{(BRR+1)*8}$$

$$BRR = \frac{37.5MHz}{9.600Hz*8} - 1 = 487.2$$

BRR must be an integer, so we have to round the result to 487. The reverse calculation with BRR = 487 leads to the real baud rate of 9605.5 baud (error = 0,06 %).

SCI Control Register 2 – SCICTL2

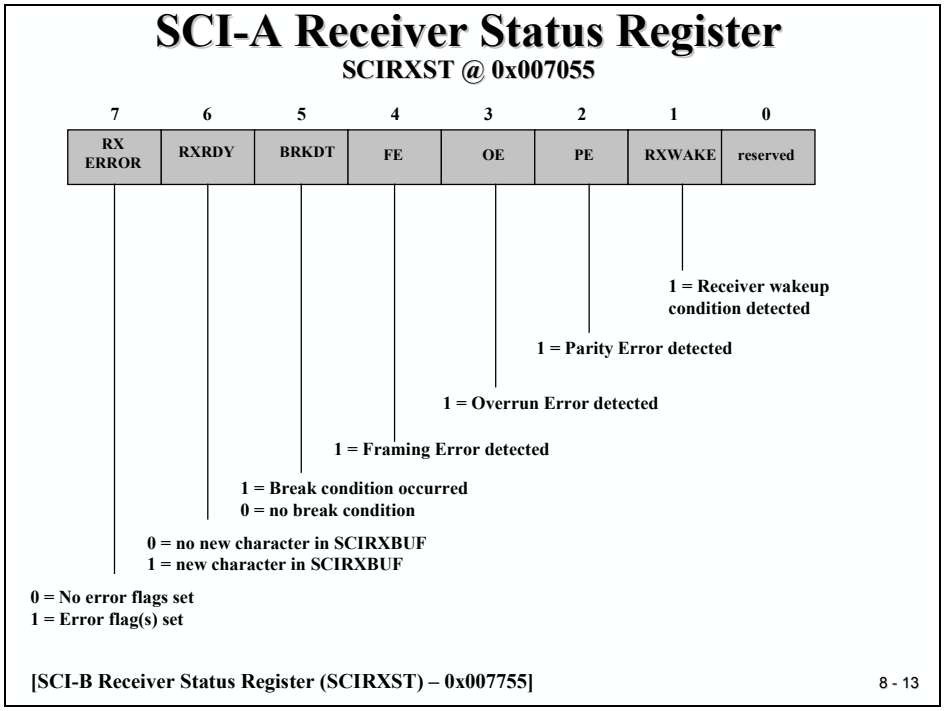


Bit 1 and 0 enable or disable the SCI- transmit and receive interrupts. If interrupts are not used, we can disable this feature by clearing bits 1 and 0. In this case we have to apply a polling scheme to the transmitter status flags (SCICTL2.7 and SCICTL2.6). The flag SCITXEMPTY waits until the whole data frame has left the SCI output, whereas flag SCITXREADY indicates the situation that we can reload the next character into SCITXBUF before the previous character was physically sent.

The status flags for the receiver part can be found in the SCI receiver status register (see next slide).

For the first basic lab exercise we will not use SCI interrupts. This means we have to rely on a polling scheme. Later we will include SCI interrupts in our experiments.

SCI Receiver Status Register – SCIRXST

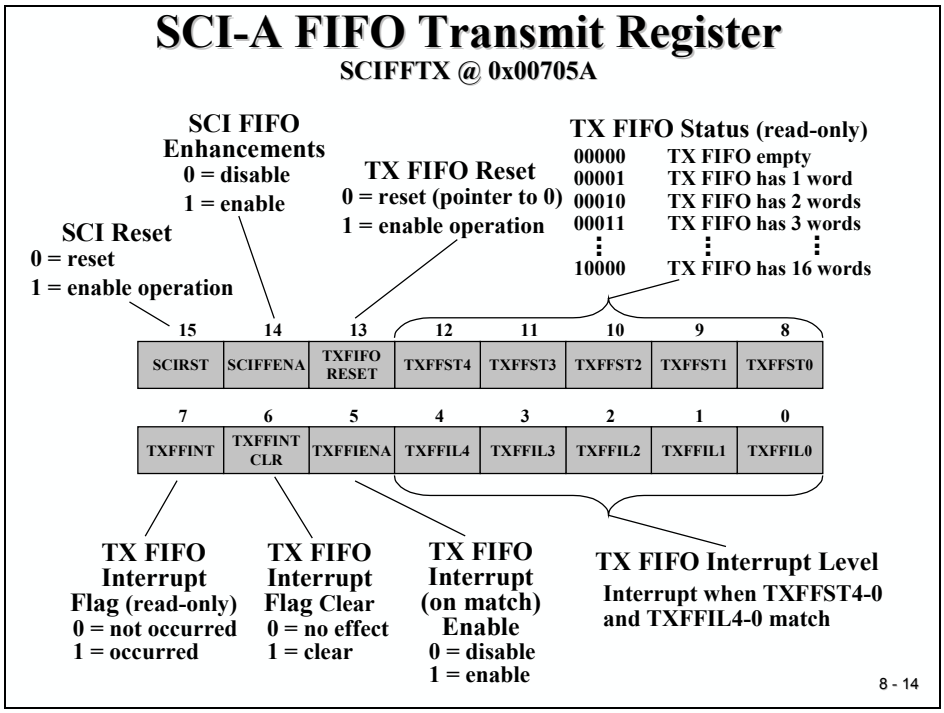


The SCI interrupt logic generates interrupt flags when it receives or transmits a complete character as determined by the SCI character length. This provides a convenient and efficient way of timing and controlling the operation of the SCI transmitter and receiver. The interrupt flag for the transmitter is TXRDY (SCICTL2.7), and for the receiver RXRDY (SCIRXST.6). TXRDY is set when a character is transferred to TXSHF and SCITXBUF is ready to receive the next character. In addition, when both the SCIBUF and TXSHF registers are empty, the TX EMPTY flag (SCICTL2.6) is set.

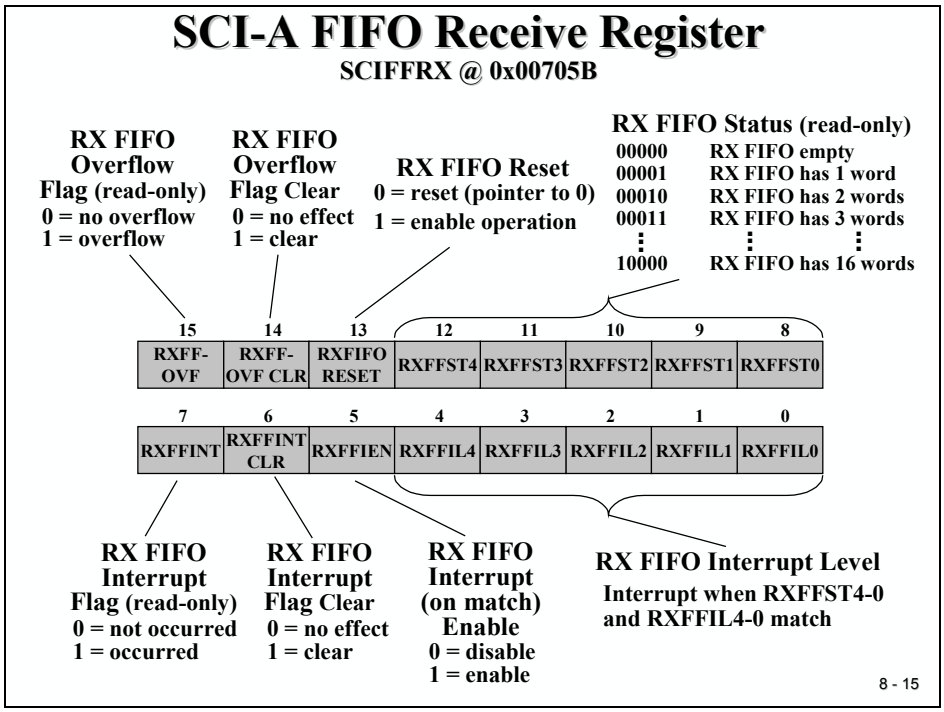
When a new character has been received and shifted into SCIRXBUF, the RXRDY flag is set. In addition, the BRKDT flag is set if a break condition occurs. A break condition is where the SCIRXD line remains continuously low for at least ten bits, beginning after a missing stop bit. The CPU to control SCI operations can poll each of the above flags, or interrupts associated with the flags can be enabled by setting the RX/BK INT ENA (SCICTL2.1) and/or the TX INT ENA (SCICTL2.0) bits active high.

Additional flag and interrupt capability exists for other receiver errors. The RX ERROR flag is the logical OR of the break detect (BRKDT), framing error (FE), receiver overrun (OE), and parity error (PE) bits. RX ERROR high indicates that at least one of these four errors has occurred during transmission. This will also send an interrupt request to the CPU if the RX ERR INT ENA (SCICTL1.6) bit is set.

SCI FIFO Mode Register

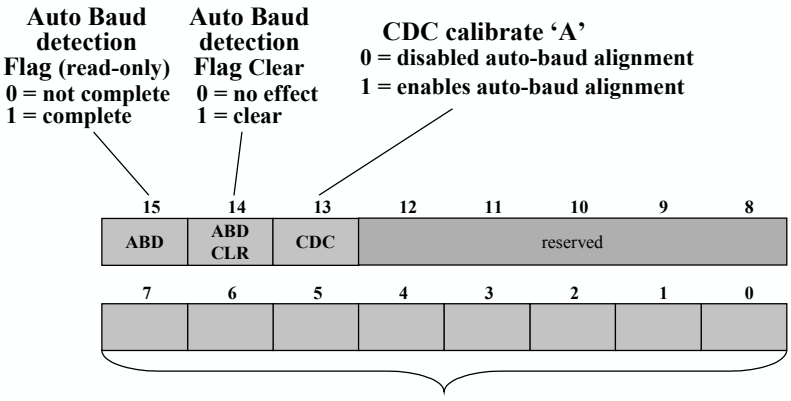


The C28x SCI is equipped with an enhanced buffer mode with 16 levels for the transmitter and receiver. We will use this enhanced mode at the end of the lab exercise series of this chapter.



SCI-A FIFO Control Register

SCIFFCT @ 0x00705C

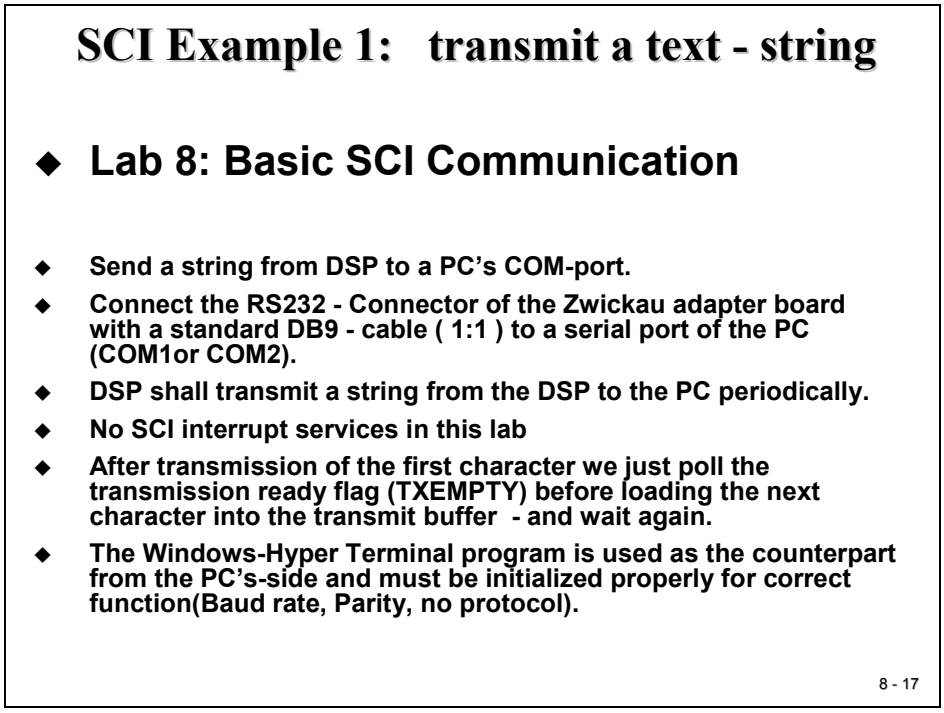


FFTXDLY

Time delay between every transfer from FIFO to transmit shift register in number of SCI baud clock cycles (0 to 255)

8 - 16

Lab 8: Basic SCI – Transmission



Objective

The objective of this lab is to establish an SCI transmission between the C28x and a serial port of a PC. The Zwickau adapter board converts the serial signals of the DSP into a standard RS232 format. The DB9 female connector (X1) has to be connected with a standard serial cable (1:1 connection) to a serial COM-port of the PC.

The DSP shall transmit a string, e.g. "The F2812-UART is fine!\n\r" periodically. No interrupt services are used for this basic test.

As the counterpart at the PC we will use Windows® - hyper terminal program. This program can be found under Windows - OS ➔ start ➔ programs ➔ Accessories ➔ Communication ➔ HyperTerminal. Create a new connection, select a symbol and name it "SCI-Test.

In the "connect to" field select the COM-port of your PC, e.g. COM1.

Setup:

- data rate to 9600,
- 8 bits per character,
- no parity bit,
- 1stopbit,
- no protocol.

Procedure

Open Files, Create Project File

1. Create a new project, called **Lab8.pjt** in E:\C281x\Labs.

2. Open the file Lab2.c from E:\C281x\Labs\Lab2 and save it as Lab8.c in E:\C281x\Labs\Lab8.

3. Add the source code file to your project:
   - **Lab8.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:\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 Lab8.c to edit: double click on "Lab8.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, empty the "while(1)"-loop of main, we will add some more code later:

   ```
   while(1)
   {
   }
   ```

9. Delete the function "delay_loop" and its prototype declaration.

Add the SCI initialization code

10. Inside function "InitSystem()" enable the SCI-A clock unit:

    ```
    SysCtrlRegs.PCLKCR.bit.SCIAENCLK=1;
    ```

11. Inside "Gpio_select()"modify multiplex register GPFMUX to use the 2 SCI-signals:

    ```
    GpioMuxRegs.GPFMUX.bit.SCIRXDA_GPIOF5 = 1;

    GpioMuxRegs.GPFMUX.bit.SCITXDA_GPIOF4 = 1;
    ```

12. At the begin of main define a string variable with the following text in it:

    ```
    "The F2812-UART is fine !\n\r"
    ```

13. In main, just before entering the "while(1)"-loop add a function call to function "SCI_Init()". Also add a function prototype at the start of your code.

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

    Inside this function, include the following steps:

    - SCICCR:
      - 1 stop bit, no loop back, no parity, 8 bits per character
    - SCICTL1:
      - Enable TX, RX -output
      - Disable RXERR INT, SLEEP and TXWAKE

- SCIHBAUD / SCILBAUD:
    - BRR = (LSPCLK/(SCI_Baudrate *8)) – 1
    - Example: assuming LSPCLK = 37.5MHz and SCI_Baudrate = 9600 the SCIBRR must be set to 487.

Finish the main loop

15. Now we can finalize the while(1)-loop of main. Recall, we have to add the following:
    - Load the next character out of the string variable into SCITXBUF
    - Wait (poll) bit TXEMPTY of register SCICTL2. It will be set to 1 when the character has been sent. The bit will be cleared automatically when the next character is written into SCITXBUF.
    - Increment an array pointer to point to the next character of the string. Also include a test if the whole string has been sent. In this case reset the pointer to prepare the next transmission sequence.
    - Include a software loop before the start of the next transmission sequence of approximately 2 seconds:

      for(i=0;i<15000000;i++); // Software - delay approx. 2 sec.

    - Service the watchdog periodically

Build, Load and Run

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

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

17. In the hyper terminal window you should see the received string every 2 seconds.

    If not ➔ Debug!

END of LAB 8

Lab 8A: Interrupt SCI – Transmission

The objective of the next lab exercise is to improve Lab 8 by including both the SCI – Transmit interrupt service to service an empty transmit buffer and the CPU Core Timer 0 interrupt service to trigger the transmission of the first character of the string every 2 seconds.

Use your code from Lab8 as a starting point.

Procedure

Open Files, Create Project File

1. Create a new project, called **Lab8A.pjt** in E:\C281x\Labs.
2. Open the file Lab8.c from E:\C281x\Labs\Lab8 and save it as Lab8A.c in E:\C281x\Labs\Lab8A.
3. Add the source code file to your project:
   - **Lab8A.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_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 Lab8A.c to edit: double click on "Lab8A.c" inside the project window. In main, after the function call "Gpio_select()"add a call of function:

   InitPieCtrl();

   and:

   InitPieVectTable();

   Next, re-map the Interrupt table entry for CPU Core Timer 0 Interrupt

   EALLOW;

   PieVectTable.TINT0 = &cpu_timer0_isr;

   EDIS;

   Initialize the CPU Core Timer group by calling:

   InitCpuTimers();

   and configure CPU-Timer 0 to interrupt every 50 ms:

   ConfigCpuTimer(&CpuTimer0, 150, 50000);

Now enable the CpuTimer0 PIE interrupt line:

```
PieCtrlRegs.PIEIER1.bit.INTx7 = 1;

IER = 1;
```

Finally enable the global interrupt flag and reset start the CPU Core Timer 0:

```
EINT;   // Enable Global interrupt INTM

ERTM;   // Enable Global real-time interrupt DBGM

CpuTimer0Regs.TCR.bit.TSS = 0;
```

8. Add a CPU Core Timer0 Interrupt service routine "cpu_timer0_isr()" at the very end of your source code. You can use the same code that we used in Lab7B.c for this function. Do not forget to add a function prototype at begin of your code.

9. Next, we have to modify the SCI initialization function "SCI_Init()". Not a big change, the only modification is that for this project we have to enable the SCI-Transmit Interrupt:

```
SciaRegs.SCICTL2.bit.TXINTENA = 1;
```

10. The SCI Transmit Interrupt must be also enabled inside the PIE and the address of the interrupt service routine must be written into the PIE vector table. Before the global interrupt enable line "EINT" add the following code:

```
EALLOW;

PieVectTable.TXAINT = &SCI_TX_isr;

EDIS;

PieCtrlRegs.PIEIER9.bit.INTx2 = 1;

IER | = 0x100;
```

11. If the SCI-TX interrupt is enabled we have to provide an interrupt service routine "SCI_TX_isr()". At the top of your code add a function prototype and at the very end of the code add the definition of this function. What should be done inside this function? Answer:

    - Load the next character of the string into SCITXBUF, if the string pointer has not already reached the last character of the string and increment this pointer.

    - If the string pointer points beyond the last character of the string, do NOT load anything into SCITXBUF. Transmission of the string is finished.

- In every single call of this function acknowledge it's call by resetting the PIEACK-register:

```
PieCtrlRegs.PIEACK.all = 0x0100;
```

12. Because the string variable and the variable "index" are now used out of main and "SCI_TX_isr" they must now be declared as global variables. Remove the definition from main and add them as global variables at the begin of your code, outside any function:

```
char message[ ]= {"The F2812-UART is fine !\n\r"};

int index =0;                    // pointer into string
```

13. Now it is time to think about the while(1) – loop of main. Remove everything that is inside this loop. We do not need the old code from Lab8.

    Instead, let's add new code:

    First, we will get a CPU Timer0 Interrupt every 50ms. According to the setup of our interrupt service routine "cpu_timer0_isr()" variable "CpuTimer0.InterruptCount" will be incremented every 50ms. To trigger a SCI Transmission every 2 seconds we just have to wait until this variable has reached 40. If the value is less than 40 we just have to wait and do nothing, right? NO! Our watchdog is alive and we have to serve it! If the first reset key is applied inside function "cpu_timer0_isr()" we have to apply the second key while we wait for variable "CpuTimer0.InterruptCount" to reach 40.

    This could be the portion of code:

```
while(CpuTimer0.InterruptCount < 40)  // wait for 2000ms
{
        EALLOW;
        SysCtrlRegs.WDKEY = 0xAA;    // service watchdog #2
        EDIS;
 }
```

    What's next? Well, these actions are required now:

    - Reset variable "CpuTimer0.InterruptCount" to 0
    - Reset variable "index" to 0
    - Load first character out of message into SCITXBUF and
    - Increment variable index afterwards.

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 & Run

16. Reset the DSP by clicking on:

    Debug → Reset CPU followed by
    Debug ➔ Restart and
    Debug → Go main
    Debug ➔ Run (F5)

    As we've done in Lab8 open a Hyper Terminal Session (use 9600, n, 1 and no protocol as parameters). Every 2 seconds you should receive the string from the DSP.

    If your code does not work try to debug systematically.

    - Does the CPU core timer work?
    - Is the CPU core timer interrupt service called periodically?
    - Is the SCITX interrupt service called?

    Try to watch important variables and set breakpoints as needed.

17. The result of this lab does not differ that much from lab8. All we do is to send a string every 2 seconds to the PC. The big difference however is that the time interval is now generated by a hardware timer instead of a software delay loop. This is a big improvement, because the period is now very precise and the DSP is not overloaded by such a stupid task to count a variable from x to y. For real projects we would gain a lot of CPU time by using a hardware timer.

Optional Exercise

18. Instead of transmitting the string to the PC your task is now to transmit the current status of the input switches (GPIO B15-B8), which is an integer, to the PC. Recall, to use Windows Hyper Terminal to display data, you must transmit ASCII-code characters. To convert a long integer into an ASCII-string we can use function "ltoa"(see help).

END of LAB 8A

Lab 8B: SCI – FIFO Transmission

The objective of this lab is to improve Lab 8A by using the transmit FIFO capabilities of the C28x. Instead of generating a lot of SCI – transmit interrupts to send the whole string we now will use a type of 'burst transmit' technique to fill up to 16 characters into the SCI transmit FIFO. This will reduce the number of SCI-interrupt services from 16 to 1 per string transmission!

Use your code from Lab8A as a starting point.

Procedure

Open Files, Create Project File

1. Create a new project, called **Lab8B.pjt** in E:\C281x\Labs.
2. Open the file Lab8A.c from E:\C281x\Labs\Lab8A and save it as Lab8B.c in E:\C281x\Labs\Lab8B.
3. Add the source code file to your project:
   - **Lab8B.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_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 Lab8B.c to edit: double click on "Lab8B.c" inside the project window.

   Modify the SCI Initialization in function "SCI_Init()". Add the Initialization for register "SCIFFTX". Include the following:

   - Relinquish FIFO unit from reset
   - Enable FIFO- Enhancements
   - Enable TX FIFO Operation
   - Clear TXFFINT-Flag
   - Enable TX FIFO match
   - Set FIFO interrupt level to interrupt, if FIFO is empty (0)

8. Change the content of variable "message[]" from "The F2812-UART is fine !\n\r" into "BURST-Transmit\n\r". The length of the string is now limited to 16 characters and using the TX-FIFO we can transmit the whole string in one single SCI interrupt service routine.

9. Go into function "SCI_TX_isr()" and modify it. Recall that this service will be called when the FIFO interrupt level was hit. Due to our set up of this level to 0 we can load 16 characters into the TX-FIFO:

   for(i=0;i<16;i++) SciaRegs.SCITXBUF = message[i];

   Note: Variable i should be a local variable inside "SCI_TX_isr()". Also, do NOT remove the PIEACK- reset instruction at the end of this function!

10. Go into the while(1)-loop of main. We still will use the CPU Core Timer 0 as our time base. It is still initialized to increment variable "CpuTimer0.InterruptCount" every 50ms. No need to change our wait construction to wait for 40 increments (equals to 2 seconds).

    Delete the next two lines of the old code:

    index = 0;

    SciaRegs.SCITXBUF = message[index++];

    The difference between Lab8A.c and Lab8B.c is the initialization of the SCI-unit. In this lab we enabled the TX-FIFO interrupt to request a service when the FIFO-level is zero. This will be true immediately after the initialization of the SCI-unit and will cause the first TX-interrupt! The next TX-interrupt will be called only after setting the TX FIFO INT CLR – bit to 1, clears the TX FIFO INT FLAG. If we execute this clear instruction every 2 seconds we will allow the next TX FIFO transmission to take place at this very moment. To do so, add the following instruction:

    SciaRegs.SCIFFTX.bit.TXINTCLR = 1;

    That's it.

Build, Load and Test

11. Apply all the commands needed to translate and debug your project. Meanwhile you should be familiar with the individual steps to do so; therefore we skip a detailed procedure. If you are successful, you should receive the string every 2 seconds at the hyper terminal window. If not – debug!

12. Again, the big improvement of this Lab8B is that we reduced the number of interrupt services to transmit a 16-character string from 16 services to 1 service. This adds up to a considerable amount of time that can be saved! The exercise has shown the advantage of the C28x SCI-transmit FIFO enhancement compared to a standard UART interface.

END of LAB 8B

Lab 8C: SCI – Receive & Transmit

The final project in this module asks you to include the SCI receiver in out lab exercises. The objective of this lab is to receive a string "Texas" from the PC and to answer it by transmitting "Instruments" back to the PC.

Use your code from Lab8B as a starting point.

Procedure

Open Files, Create Project File

1. Create a new project, called **Lab8C.pjt** in E:\C281x\Labs.

2. Open the file Lab8B.c from E:\C281x\Labs\Lab8B and save it as Lab8C.c in E:\C281x\Labs\Lab8C.

3. Add the source code file to your project:
   - **Lab8C.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**

   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 Lab8C.c to edit.

   First we have to remove everything that deals with CPU Core Timer0 – we do not need a timer for this exercise.

   - Remove prototype and definition of function "cpu_timer0_isr".
   - In main, remove the overload instruction:

     ```
     EALLOW;
     PieVectTable.TINT0 = &cpu_timer0_isr;
     EDIS;
     ```

   - Remove the function calls:

     ```
     InitCpuTimers();
     ConfigCpuTimer(&CpuTimer0, 150, 50000);
     ```

     and the interrupt enable lines:

     ```
     PieCtrlRegs.PIEIER1.bit.INTx7 = 1;
     IER = 1;
     ```

   - Remove the start instruction for CpuTimer0:

     ```
     CpuTimer0Regs.TCR.bit.TSS = 0;
     ```

- In the while(1)-loop of main remove everything. Replace it by:

```
while(1)
        {
        EALLOW;
        SysCtrlRegs.WDKEY = 0x55;     // service watchdog #1
        SysCtrlRegs.WDKEY = 0xAA;     // service watchdog #2
        EDIS;
        }
```

  All activities will be done by interrupt service routines, nothing to do in the main loop but to service the watchdog!

8. Change the content of variable "message[ ]" into " Instruments\n\r".

9. Now we need to introduce a new interrupt service routine for the SCI receiver, called "SCI_RX_isr". Declare it's prototype at the begin of your code:

```
interrupt void SCI_RX_isr(void);
```

10. Add an overload instruction for this function inside the PIE vector table. Add this line directly after the overload for TXAINT:

```
PieVectTable.RXAINT  = &SCI_RX_isr;
```

11. Enable the PIE interrupt for RXAINT:

```
PieCtrlRegs.PIEIER9.bit.INTx1  = 1;
```

12. Modify the initialization function for the SCI: "SCI_Init()".

    - Inside register "SCICTL2" set bit "RXBKINTENA" to 1 to enable the receiver interrupt.

    - For register "SCIFFTX" do NOT enable the TX FIFO operation (bit 13) yet. It will be enabled later, when we have something to transmit.

    - Add the initialization for register "SCIFFRX". Recall, we wait for 5 characters "Texas", so why not initialize the FIFO receive interrupt level to 5? This setup will cause the RX interrupt when at least 5 characters have been received.

13. At the end of your source code add interrupt function "SCI_RX_isr()".

    - What should be done inside? Well, this interrupt service will be requested if 5 characters have been received. First we need to verify that the 5 characters match the string "Texas".

    - With five consecutive read instructions of register "SCIRXBUF" you can empty the FIFO into a local variable "buffer[16]".

- The C standard function “strncmp” can be used to compare two strings of a fixed length. The lines:

```
if( strncmp(buffer, “Texas” , 5) == 0)
{
SciaRegs.SCIFFTX.bit.TXFIFORESET = 1;
SciaRegs.SCIFFTX.bit.TXINTCLR = 1;
}
```

  will compare the first 5 characters of “buffer” with “Texas”. If they match the two next instructions will start the SCI Transmission of “ Instruments\n\r” with the help of the TX-interrupt service.

- At the end of interrupt service routine we need to reset the RX FIFO, clear the RX FIFO Interrupt flag and acknowledge the PIE interrupt:

```
SciaRegs.SCIFFRX.bit.RXFIFORESET = 0; // reset pointer
SciaRegs.SCIFFRX.bit.RXFIFORESET = 1; // enable op.
SciaRegs.SCIFFRX.bit.RXFFINTCLR = 1; // reset RX int
PieCtrlRegs.PIEACK.all = 0x0100; // acknowledge PIE
```

- That’s it.

Build, Load and Test

14. Apply all the commands needed to translate and debug your project.

15. Start Windows Hyper Terminal and type in the text “Texas”. The C28x will respond with the string “ Instruments\n\r”. If not ➔ debug!

Optional Exercise

16. DSP – Junkies only! ➔ Remote Control of the C28x by a PC!

    Try to combine the “Knight-Rider” exercise (Lab4) with Lab8C. Let the PC send a string with a numerical value and use this value to control the speed of the “Knight-Rider”!

    If a new value was received by the DSP it should answer back to the PC with a text like “control value xxx received”.

    Note: The C standard function “atoi” can be used to convert an ASCII-string into a numerical value (see Code Composer Studio Help for details).