F2833x Serial Communication Interface

Introduction

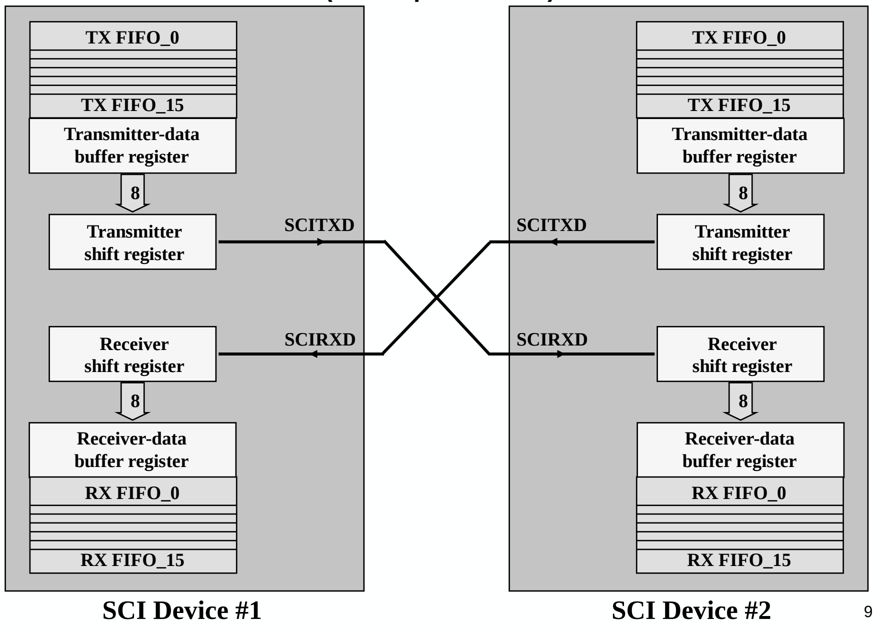
The Serial Communication Interface (SCI) module is a serial I/O port that permits asynchronous communication between the F2833x and other peripheral devices. It is usually known as a UART (Universal Asynchronous Receiver Transmitter) and is often used according to the RS232 standard.

The SCI receiver and transmitter each have a 16-deep FIFO for reducing servicing overhead, each with its own separate enable and interrupt bits. Both can be operated independently for half-duplex communication, or simultaneously for full-duplex communication. To maintain data integrity, the SCI checks received data for break detection, parity, overrun, and framing errors. The bit rate is programmable for different communication speeds through a 16-bit baud-select register.

Parity checking and data formatting can also be done by the SCI port hardware, further reducing the software overhead.

SCI Pin Connections

(Full Duplex Shown)



9 - 2

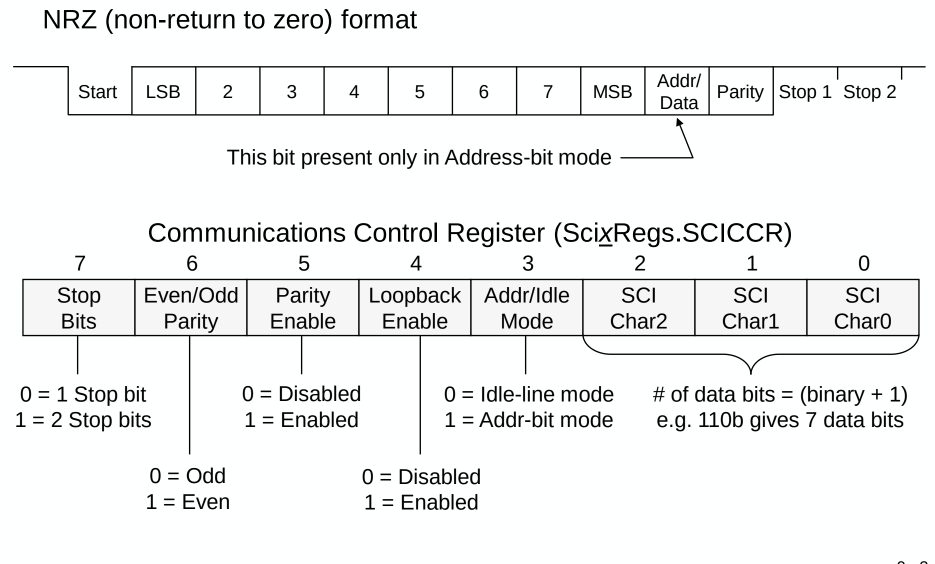
Module Topics

SCI Data Format

The basic unit of data is called a **character** and is 1 bit 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 start 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 Data Format



9 - 3

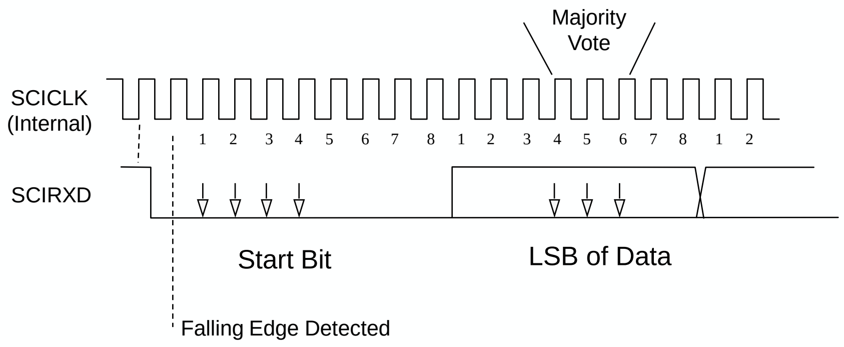
Note: If you are working on a RS232 – Interface, then all voltage-levels at the serial lines are driven by external interface circuits, such as Texas Instruments MAX3221. A logical '0' is transmitted as a voltage between +5 and +15V, a logical '1' as a negative Voltage between -5 and -15V. On the receiver side, a voltage above +3V will be recognized as a valid '0', a voltage below -3V as a logical '1'.

SCI Data Timing

The SCI asynchronous communication format uses either single line (one way) or two line (two ways) communications. In this mode, the frame consists of a start bit, one to eight data bits, an optional even/odd parity bit, and one or two stop bits (shown in Slide 9-3). There are eight SCICLK periods per data bit.

SCI Data Timing

- Start bit valid if 4 consecutive SCICLK periods of zero bits after falling edge
- Majority vote taken on 4th, 5th, and 6th SCICLK cycles



Note: 8 SCICLK periods per data bit

9 - 4

The receiver begins operation on receipt of a valid start bit. A valid start bit is identified by four consecutive internal SCICLK periods of zero bits as shown in Slide 9-4. If any bit is not zero, then the processor starts over and begins looking for another start bit.

For the bits following the start bit, the processor determines the bit value by making three samples in the middle of the bits. These samples occur on the fourth, fifth, and sixth SCICLK periods, and bit-value determination is on a majority (two out of three) basis. Slide 9-4 illustrates the asynchronous communication format for this with a start bit showing where a majority vote is taken. Since the receiver synchronizes itself to frames, the external transmitting and receiving devices do not have to use a synchronized serial clock. The clock can be generated locally.

SCI Multi Processor Wake Up Modes

Multiprocessor Wake-Up Modes

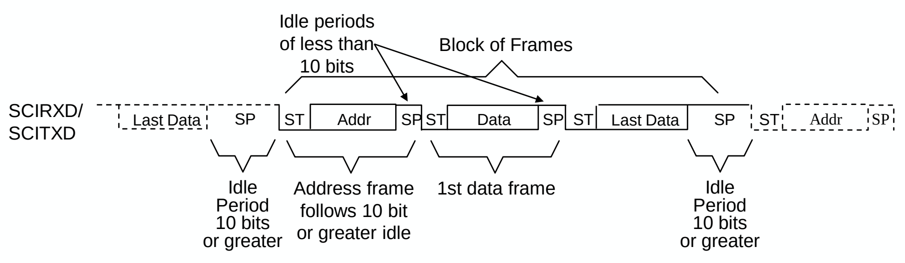
- **Allows numerous processors to be hooked up to the bus, but transmission occurs between only two of them**
- **Idle-line or Address-bit modes**
- **Sequence of Operation**
  1. Potential receivers set SLEEP = 1, which disables RXINT except when an address frame is received
  2. All transmissions begin 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

9 - 5

Although a SCI data transfer is usually a point-to-point communication, the F2833x SCI interface allows two operation modes to communicate between a master and more than one slave.

Idle-Line Wake-Up Mode

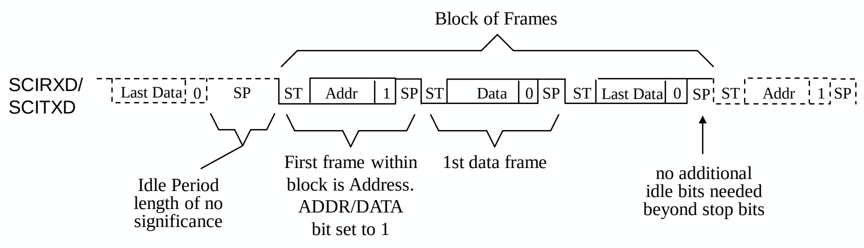
- **Idle time separates blocks of frames**
- **Receiver wakes up with falling edge after SCIRXD was high for 10 or more bit periods**
- **Two transmit address methods**
  - **deliberate software delay of 10 or more bits**
  - **set TXWAKE bit to automatically leave exactly 11 idle bits**


Idle periods of less than 10 bits
Block of Frames
SCIRXD/ SCITXD
Last Data
SP
ST
Addr
SP
ST
Data
SP
ST
Last Data
SP
ST
Addr
SP
Idle Period 10 bits or greater
Address frame follows 10 bit or greater idle
1st data frame
Idle Period 10 bits or greater

9 - 6

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


Block of Frames
SCIRXD/
SCITXD
Last Data
0
SP
ST
Addr
1
SP
ST
Data
0
SP
ST
Last Data
0
SP
ST
Addr
1
SP
Idle Period length of no significance
First frame within block is Address. ADDR/DATA bit set to 1
1st data frame
no additional idle bits needed beyond stop bits

9 - 7

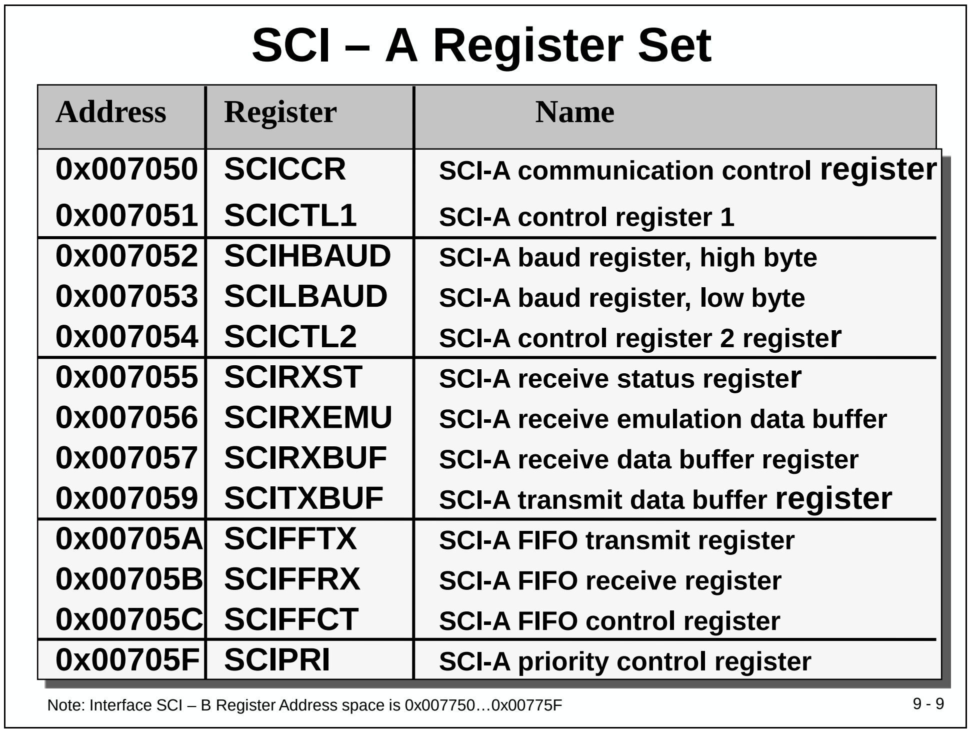
SCI Summary

- 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
- 28335 include channel SCI-A and SCI-B

9 - 8

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 F2833x.



SCI – A Register Set

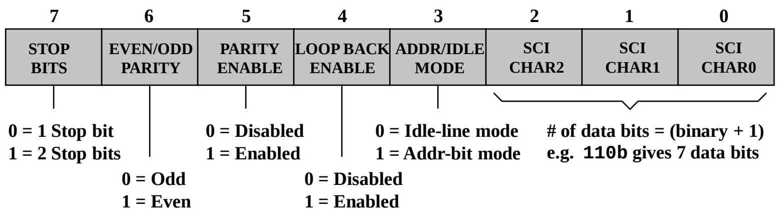
Address	Register	Name
0x007050	SCICCR	SCI-A communication control register
0x007051	SCICTL1	SCI-A control register 1
0x007052	SCIHBAUD	SCI-A baud register, high byte
0x007053	SCILBAUD	SCI-A baud register, low byte
0x007054	SCICTL2	SCI-A control register 2 register
0x007055	SCIRXST	SCI-A receive status 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

Note: Interface SCI – B Register Address space is 0x007750…0x00775F

9 - 9

SCI-A Communication Control Register

Communications Control Register (SCICCR) – 0x007050



9 - 10

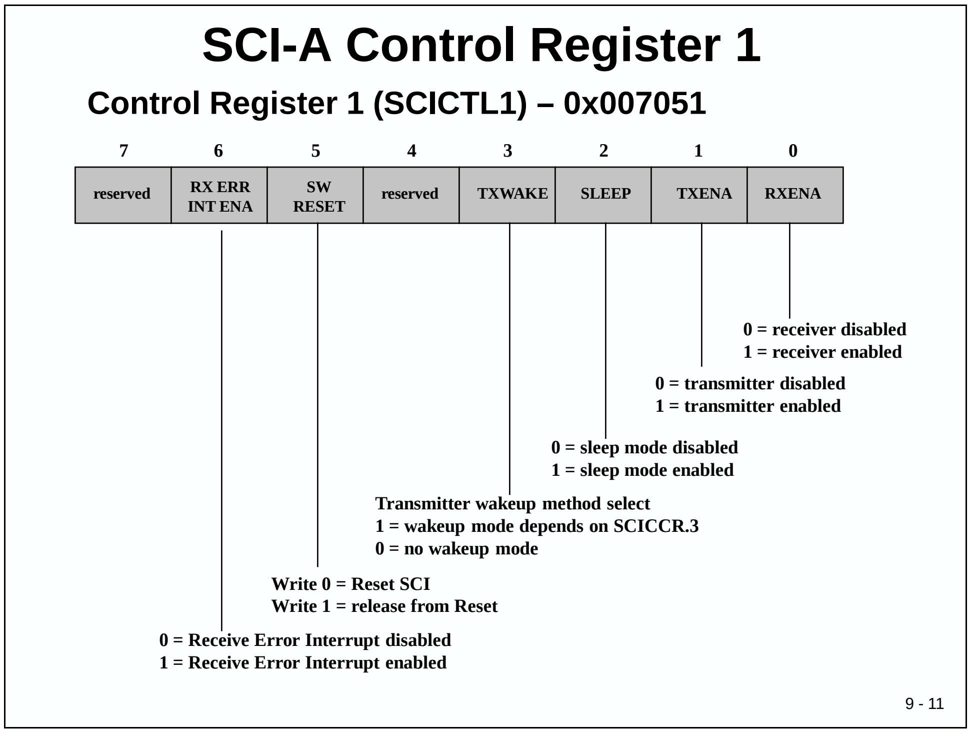
SCI Communications Control Register (SCICCR)

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 9-3). 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**
- **odd parity**
- **1 Stop bit**
- **loop back disabled**

SCI Control Register 1(SCICTL1)



SCI-A Control Register 1

Control Register 1 (SCICTL1) – 0x007051

7	6	5	4	3	2	1	0
reserved	RX ERR INT ENA	SW RESET	reserved	TXWAKE	SLEEP	TXENA	RXENA

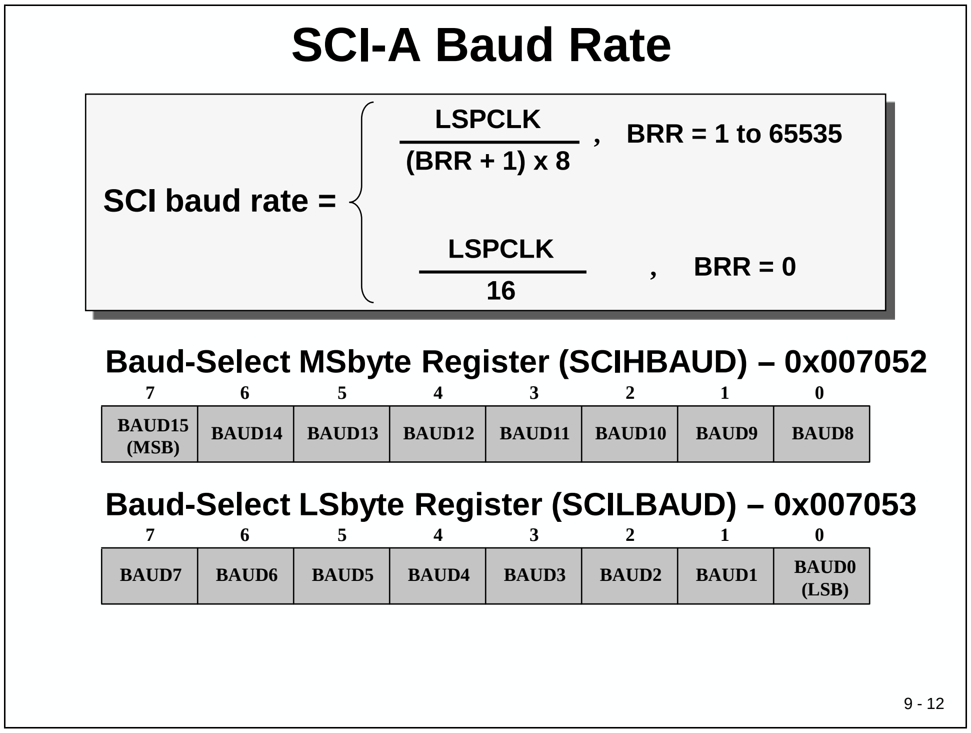
When configuring the SCICCR register, the SCI port should first be held in an inactive state. This is done using the SW RESET bit of the SCI Control Register 1 (SCICTL1.5). Writing a 0 to this bit initializes and holds the SCI state machines and operating flags at their reset condition. The SCICCR can 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 need to take precautions to handle possible communication errors. The receiver error could then be allowed to generate a receiver error interrupt request (SCICTL1.6 = 1). To simplify our first labs, we will not use this feature. However, for a real-world project, do NOT skip this part!

SCI Baud Rate Register



SCI-A Baud Rate

$$\text{SCI baud rate} = \begin{cases} \dfrac{\text{LSPCLK}}{(\text{BRR} + 1) \times 8}, & \text{BRR} = 1 \text{ to } 65535 \\[2ex] \dfrac{\text{LSPCLK}}{16}, & \text{BRR} = 0 \end{cases}$$

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)

9 - 12

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

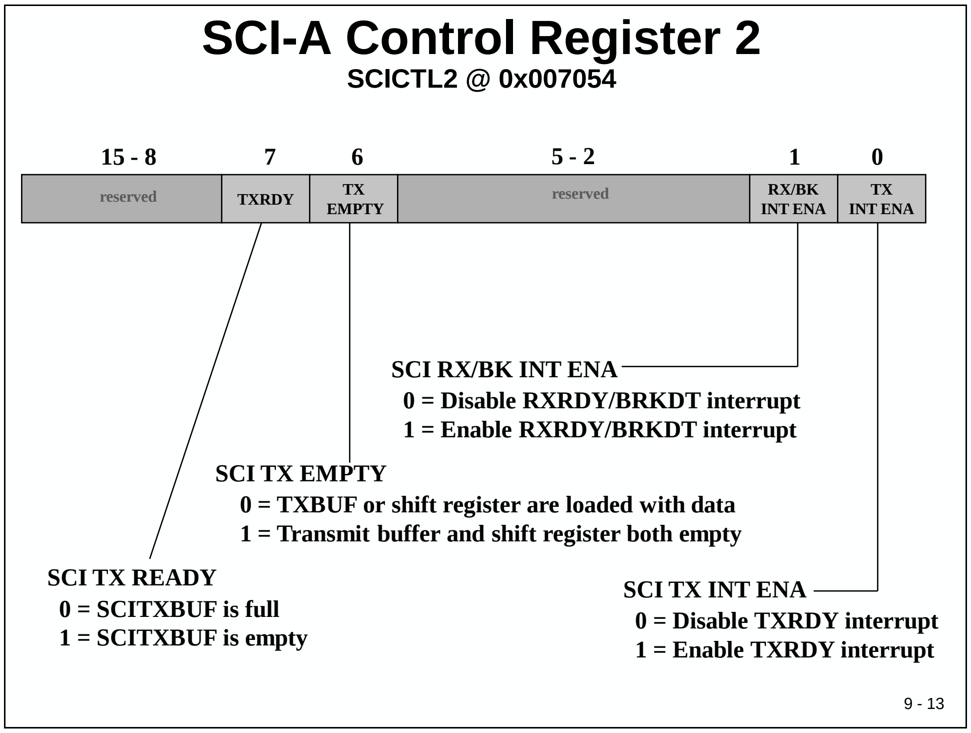
Assuming a SYSCLK frequency of 150MHz and a low speed pre-scaler initialized to “divide by 4”, we can calculate the value for the BRR, let us say for a data rate of 9600 baud:

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

$$BRR = \frac{37.5MHz}{9.600Hz*8} - 1 = 487.28$$

BRR must be an integer, so we have to round the result to 487. The reverse calculation with BRR = 487 leads to the real data rate of 9605 bits/second (error = 0.05 %).

SCI Control Register 2 – SCICTL2


SCI-A Control Register 2

SCICTL2 @ 0x007054

15 - 8	7	6	5 - 2	1	0
reserved	TXRDY	TX EMPTY	reserved	RX/BK INT ENA	TX INT ENA

SCI RX/BK INT ENA
- **0 = Disable RXRDY/BRKDT interrupt**
- **1 = Enable RXRDY/BRKDT interrupt**

SCI TX EMPTY
- **0 = TXBUF or shift register are loaded with data**
- **1 = Transmit buffer and shift register both empty**

SCI TX READY
- **0 = SCITXBUF is full**
- **1 = SCITXBUF is empty**

SCI TX INT ENA
- **0 = Disable TXRDY interrupt**
- **1 = Enable TXRDY interrupt**

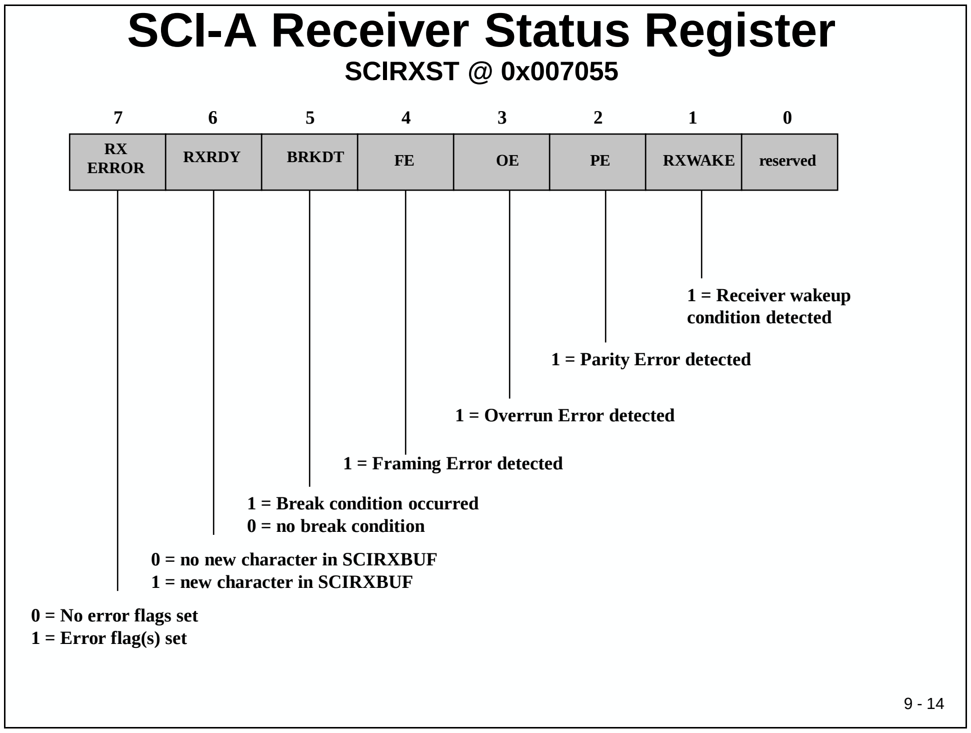
9 - 13

Bit 1 and bit 0 enable or disable the SCI- transmit and receive interrupts. If interrupts are not used, this feature can be disabled by clearing bit 1 and bit 0. In this case, we need to apply a polling method 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 the polling method described above. Later of course, we will include SCI interrupts in our experiments.

SCI Receiver Status Register – SCIRXST

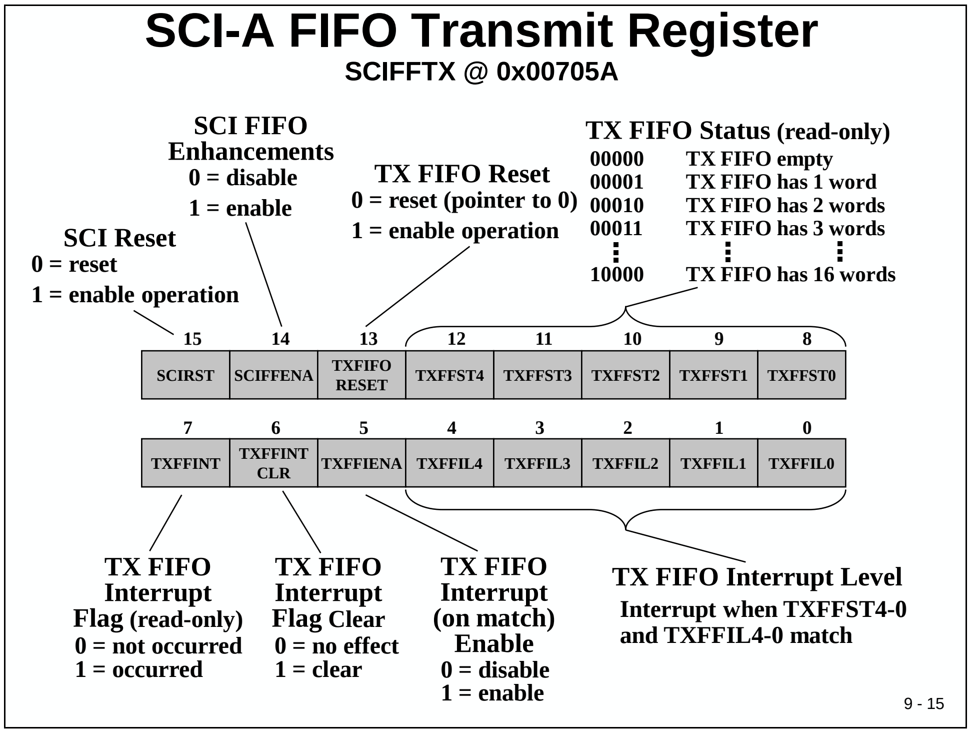

SCI-A Receiver Status Register
SCIRXST @ 0x007055
7
6
5
4
3
2
1
0
RX ERROR
RXRDY
BRKDT
FE
OE
PE
RXWAKE
reserved
1 = Receiver wakeup condition detected
1 = Parity Error detected
1 = Overrun Error detected
1 = Framing Error detected
1 = Break condition occurred
0 = no break condition
0 = no new character in SCIRXBUF
1 = new character in SCIRXBUF
0 = No error flags set
1 = Error flag(s) set
9 - 14

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 after a stop bit has been missed. 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



SCI-A FIFO Transmit Register

SCIFFTX @ 0x00705A

SCI Reset
0 = reset
1 = enable operation

SCI 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
SCIRST	SCIFFENA	TXFIFO RESET	TXFFST4	TXFFST3	TXFFST2	TXFFST1	TXFFST0

7	6	5	4	3	2	1	0
TXFFINT	TXFFINT CLR	TXFFIENA	TXFFIL4	TXFFIL3	TXFFIL2	TXFFIL1	TXFFIL0

TX FIFO Interrupt Flag (read-only)
0 = not occurred
1 = occurred

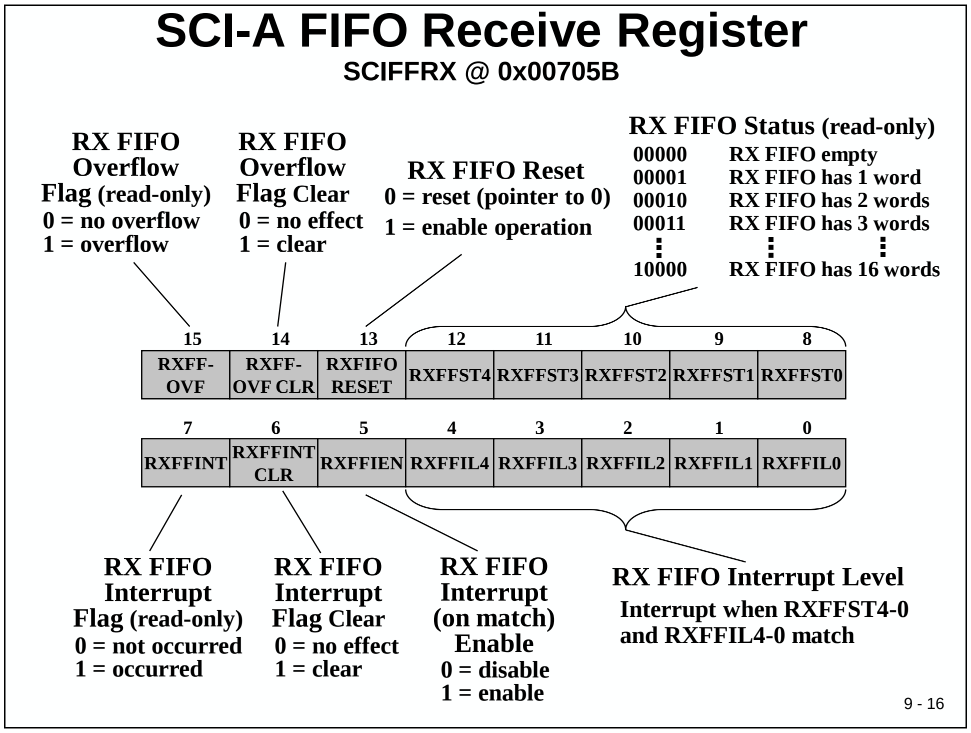
TX FIFO Interrupt Flag Clear
0 = no effect
1 = clear

TX FIFO Interrupt (on match) Enable
0 = disable
1 = enable

TX FIFO Interrupt Level
Interrupt when TXFFST4-0 and TXFFIL4-0 match

9 - 15

The F2833x SCI is equipped with an enhanced buffer mode with 16 levels of FIFO 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 Receive Register

SCIFFRX @ 0x00705B

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

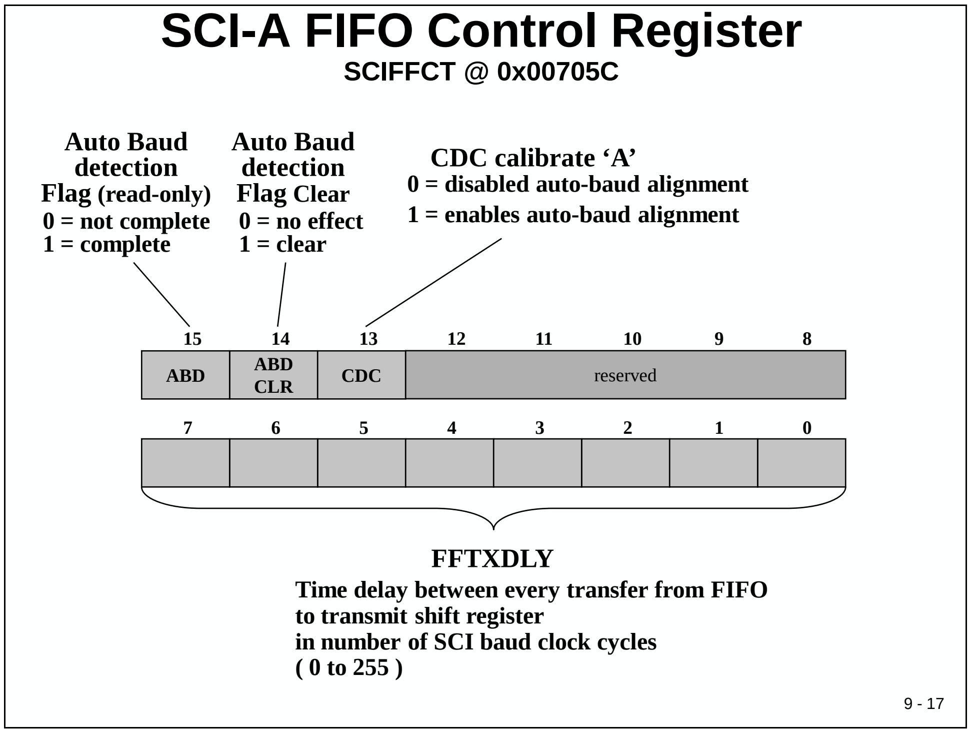
RX FIFO Interrupt Flag (read-only)
0 = not occurred
1 = occurred

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

9 - 16



SCI-A FIFO Control Register

SCIFFCT @ 0x00705C

Auto Baud detection Flag (read-only)
0 = not complete
1 = complete
Auto Baud detection Flag Clear
0 = no effect
1 = clear
CDC calibrate 'A'
0 = disabled auto-baud alignment
1 = enables auto-baud alignment
15 14 13 12 11 10 9 8
ABD
ABD CLR
CDC
reserved
7 6 5 4 3 2 1 0
FFTXDLY
Time delay between every transfer from FIFO to transmit shift register in number of SCI baud clock cycles
(0 to 255)
9 - 17

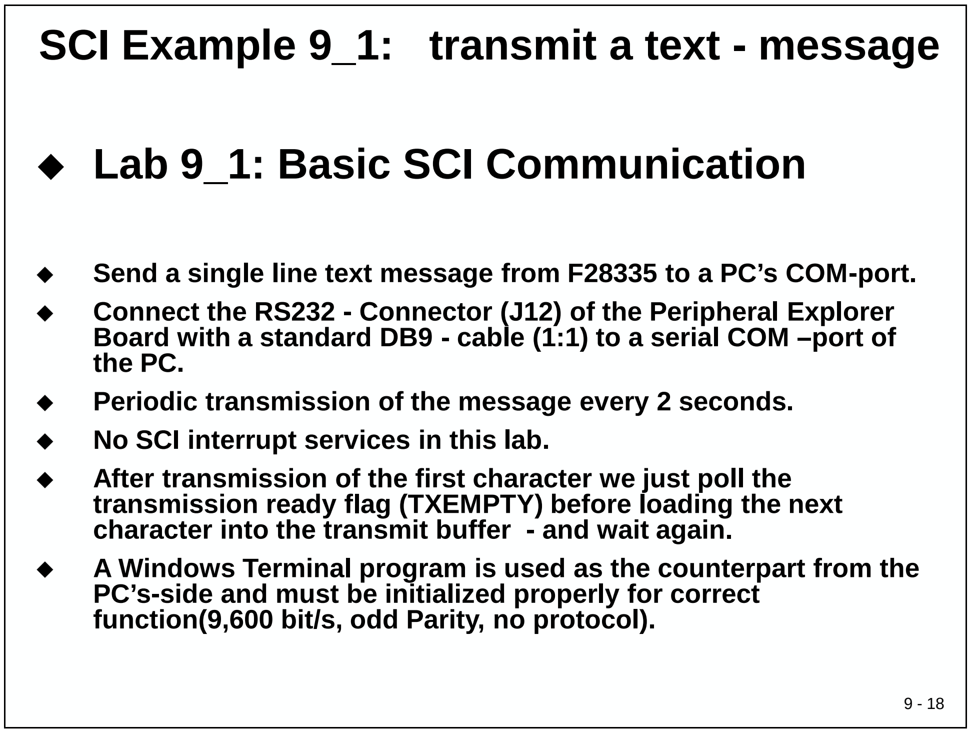
In the enhanced feature set, the SCI module supports auto baud-detect logic in hardware. The following section explains the enabling sequence for auto baud-detect feature. Auto Baud is a feature, which can be used to adjust the data rate of the F2833x to the transmission speed of a host device. If the host sends character 'A' or 'a' the auto baud unit will lock this character and set the internal baud rate registers accordingly.

To use this feature, the following sequence needs to be followed:

1. Enable auto baud-detect mode for the SCI by setting the CDC bit (bit 13) in SCIFFCT and clearing the ABD bit (Bit 15) by writing a 1 to ABDCLR bit (bit 14).
2. Initialize the baud register to be 1 or less than a baud rate limit of 500 Kbps.
3. Allow SCI to receive either character 'A' or 'a' from a host at the desired baud rate. If the first character is either 'A' or 'a', the auto baud- detect hardware will detect the incoming baud rate and set the ABD bit.
4. The auto-detect hardware will update the baud rate register with the equivalent baud value in hex. The logic will also generate an interrupt to the CPU.
5. Respond to the interrupt clear ADB bit by writing a 1 to ABD CLR (bit 14) of SCIFFCT register and disable further auto baud locking by clearing CDC bit by writing a 0.
6. Read the receive buffer for character 'A' or 'a' to empty the buffer and buffer status.
7. If ABD is set while CDC is 1, which indicates auto baud alignment, the SCI transmit FIFO interrupt will occur (TXINT). After the interrupt service CDC bit must be cleared by software.

In the first lab exercises we will not use the auto baud feature. However, if you laboratory time permits, you can add the auto baud unit into your experiments.

Lab 9_1: Basic SCI – Transmission



SCI Example 9_1: transmit a text - message

- **Lab 9_1: Basic SCI Communication**

- Send a single line text message from F28335 to a PC's COM-port.
- Connect the RS232 - Connector (J12) of the Peripheral Explorer Board with a standard DB9 - cable (1:1) to a serial COM –port of the PC.
- Periodic transmission of the message every 2 seconds.
- No SCI interrupt services in this lab.
- After transmission of the first character we just poll the transmission ready flag (TXEMPTY) before loading the next character into the transmit buffer - and wait again.
- A Windows Terminal program is used as the counterpart from the PC's-side and must be initialized properly for correct function(9,600 bit/s, odd Parity, no protocol).

9 - 18

Objective

The objective of this lab is to establish an SCI transmission between the F28338x and a serial port of a PC.

The SCI-A communication channel is used to send data from F28335 to a host, using RS232 voltage levels. The F28335 controlCARD has an onboard RS232-transceiver and the signals Tx and Rx are available at header J12 of the Peripheral Explorer Board. Plug in the serial cable provided to header J12 making sure the red wire aligns with the Rx pin on the Peripheral Explorer Board.

A standard DB9 cable (1:1) with male and female connectors can be used to connect to the host, for example to a COMx – interface of a PC. On the host side you need a terminal program (e.g. Windows XP Hyper Terminal Program or a freeware tool for XP and Vista, such as "Hercules" (www.HW-group.com). The setup for the communication is as follows:

- 9600 bit/second
- 8 characters
- odd parity
- 1 stop bit
- no protocol

The task for the F2833x is to transmit a text message, e.g. "The F28335 – UART is fine!\n\r" periodically. No interrupt services are used for this first and basic test.

Procedure

Open Project "Lab9.pjt"

1. Unzip the provided file "labs_09.zip" and expand the files in C:\DSP2833x_V4\Labs\Lab9. Next, open the project "Lab9_1.pjt"

Modify Source Code

2. Open the file "Lab9_1.c" to edit: double click on "Lab9_1.c" inside the project window.

3. Inside function "Gpio_select()"modify multiplex register GPAMUX2 to use the two SCI-signals "SCIRXDA" and "SCITXDA" for GPIO28 and GPIO29:

```
GpioCtrlRegs.GPAMUX2.bit.GPIO28 = ?;        // SCIRXDA
GpioCtrlRegs.GPAMUX2.bit.GPIO29 = ?;        // SCITXDA
```

4. At the beginning of "main()", define a string variable with the following message:

```
char message[] = {"The F28335 - UART is fine !\n\r"};
```

5. Also at the beginning of "main()", add an integer variable "index". We will use this variable to address the next character of the text message:

```
unsigned int index = 0;
```

6. In "main()", right after the function call of "Gpio_select()", add a function call of "SCIA_Init()". Also add a function prototype at the beginning of your code file "Lab9_1.c".

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

   Inside this function, initialize the following registers:

   - SCICCR:
     - 1 stop bit, no loop back, odd parity, 8 bits per character
   - SCICTL1:
     - Enable TX- and 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. Split this number into a lower 8-bit part and a higher 8-bit part and load the registers.

Finish the main loop

8. Now we can finalize the while(1)-loop of "main()". Recall, we have to add the following actions:

- Load the next character out of the string variable “message[]” into SciaRegs.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 variable “index” to address the next character of the string.
- Add a test if the whole text message has been sent (Hint: Recall that the end of a string variable is always the hidden end of string character ‘\0’). In case your program has reached the end of the message:
  - Reset variable “index” to 0 in preparation of the next transmission sequence.
  - Use CPU Timer 0 and install a wait – loop of 2 seconds (Hint: CPU Timer 0 has been initialized to 50 milliseconds. Each Interrupt Service Routine increments variable “CpuTimer0.InterruptCount”; therefore you can wait until this variable equals 40. While you wait, service the watchdog with 0xAA).
  - At the end of your wait – code, reset variable “CpuTimer0.InterruptCount” to zero in preparation of the next 2 seconds waiting loop.

Build, Load and Run

9. Click the “Rebuild Active Project ” button or perform:

   Project → Rebuild All (Alt +B)

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

10. Load the output file in the debugger session:

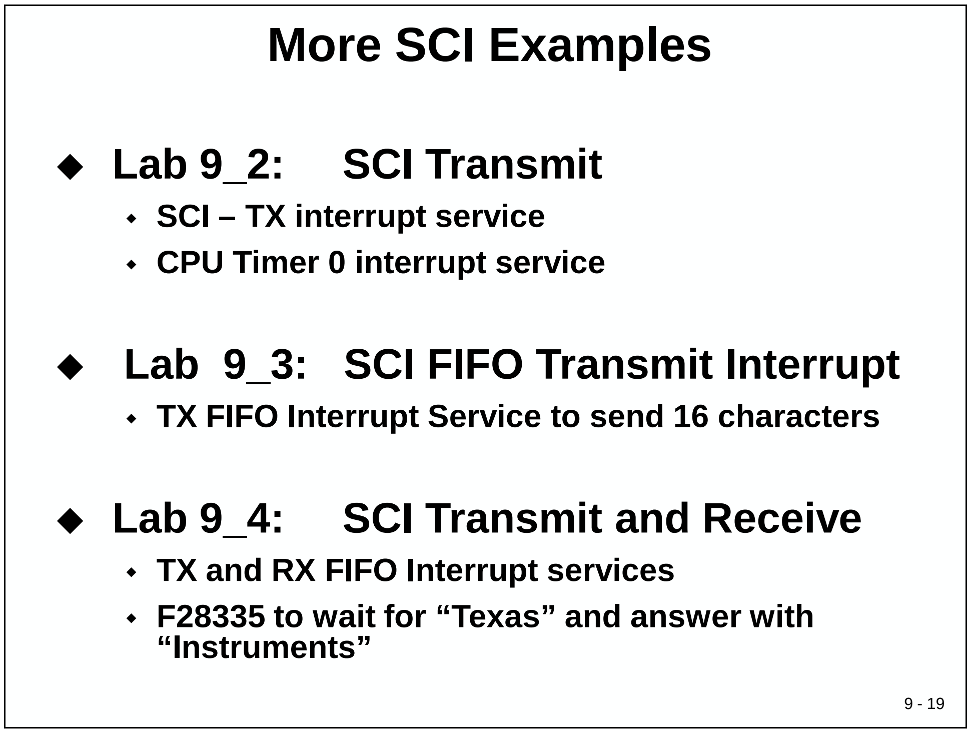
    Target → Debug Active Project

    and switch into the “Debug” perspective.

11. The PC terminal program should display an incoming text message every 2 seconds. If not → Debug!

End of Lab 9_1

Lab 9_2: Interrupt SCI – Transmission



More SCI Examples

- **Lab 9_2: SCI Transmit**
  - **SCI – TX interrupt service**
  - **CPU Timer 0 interrupt service**
- **Lab 9_3: SCI FIFO Transmit Interrupt**
  - **TX FIFO Interrupt Service to send 16 characters**
- **Lab 9_4: SCI Transmit and Receive**
  - **TX and RX FIFO Interrupt services**
  - **F28335 to wait for “Texas” and answer with “Instruments”**

9 - 19

The objective of the next lab exercise is to improve Lab 9_1 by including both the SCI – Transmit interrupt to service an empty transmit buffer. Use your code from Lab9_1 as a starting point.

Procedure

Open Files, Modify Project

1. In project “Lab9” open file “Lab9_1.c” and save it as “Lab9_2.c”

2. Exclude file “Lab9_1.c” from build. Use a right mouse click at file “Lab9_1.c”, and enable “Exclude File(s) from Build”.

Modify Source Code

3. We have to modify the SCI initialization function “SCIA_Init()”. It is not a big change, the only modification is that for this test we have to enable the SCI-Transmit Interrupt:

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

4. The SCI Transmit Interrupt must be also enabled inside the PIE unit and the address of the interrupt service routine must be written into the PIE vector table. We already have some code lines to change such entries for CpuTimer0 (TINT0). Please add the two following lines into your code:

```
PieVectTable.SCITXINTA = &SCIA_TX_isr;
PieCtrlRegs.PIEIER9.bit.INTx2 = 1;
```

5. Also change the setup for register "IER". For this exercise we have to enable lines 1 and 9!

6. If the SCI-TX interrupt is enabled we have to provide an interrupt service routine "SCIA_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 message into SCITXBUF, if index has not already reached the last character of the text message; increment variable "index".
   - If variable "index" points beyond the last character of the message, do NOT load anything into SCITXBUF. Transmission of the message is finished.
   - In every single call of this function acknowledge it's call by resetting the PIEACK-register:

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

7. Because the string variable and the variable "index" are now used both in main and "SCIA_TX_isr" they must be defined as global variables. To make the two variables global, move the definition of the variables from main to the beginning of your code:

   ```
   char message[ ]= {" The F28335 - UART ISR is fine !\n\r"};
   int index =0;
   ```

8. Now it is time change the while(1) – loop of "main()". We do not need the while – wait line to wait for TXEMPTY == 1 from lab 9_1 – because we now will use interrupt services to reload TXBUF as soon as the previous character has been transmitted.

   We can also remove the test code, which we used in lab 9_1 to test for the end of message character ('\0\') – because this will be also done by the interrupt service routine.

   We have to keep the wait construction for a time interval of 2 seconds in the while(1)-loop of "main()".

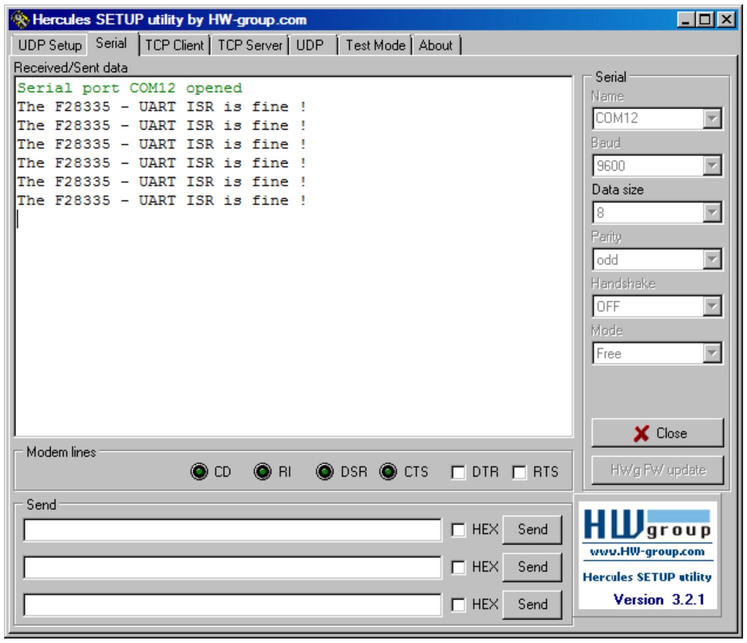
   After the wait – loop reset variables "CpuTimer0.InterruptCount" and "index" both to 0.

   Now the code is prepared for the next repetition of the while(1)-loop.

Build, Load and Run

9. Rebuild the project (Project ➔ Rebuild All), debug the project (Target ➔ Debug Active Project) and switch to the "Debug" – perspective.

10. As we have done in Lab9_1, open a Terminal Program. Use 9600 bit/s, odd parity, 1 stop bit and no protocol (or no handshake) as parameters. Every 2 seconds you should receive the text message from the DSP.



```
Serial port COM12 opened
The F28335 - UART ISR is fine !
The F28335 - UART ISR is fine !
The F28335 - UART ISR is fine !
The F28335 - UART ISR is fine !
The F28335 - UART ISR is fine !
The F28335 - UART ISR is fine !
```

Serial
Name: COM12
Baud: 9600
Data size: 8
Parity: odd
Handshake: OFF
Mode: Free

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.

Optional Exercise

11. Instead of transmitting the text message to the PC your task is now to transmit the current status of the Hexadecimal Encoder input (GPIO12- GPIO15), which is an integer value, to the PC. Recall, to use a PC-Terminal program to display data, you must transmit ASCII-code characters. To convert a long integer into an ASCII-string we can use function a standard C-function “ltoa” (see help menu of CCS).

End of Lab 9_2

Lab 9_3: SCI – FIFO Transmission

The objective of this lab is to improve Lab 9_2 by using the transmit FIFO capabilities of the F2833x. Instead of generating a lot of SCI – transmit interrupts to send the whole text message 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 message!

Use your code from file "Lab9_2.c" as the starting point.

Procedure

Open Files, Modify Project File

1. In project "Lab9" open file "Lab9_2.c" and save it as "Lab9_3.c"

2. Exclude file "Lab9_2.c" from build. Use a right mouse click at file "Lab9_2.c", and enable "Exclude File(s) from Build".

Modify Source Code

3. Open Lab9_3.c to edit: double click on "Lab9_3.c" inside the project window.

   Modify the SCI Initialization in function "SCIA_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)

4. Change the contents of the variable "message[]" from "The F28335 - UART ISR is fine !\n\r" to "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.

5. Search for function "SCIA_TX_isr()" and modify it. Recall that this service will be called when the FIFO interrupt level was hit. Because we have set 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 "SCIA_TX_isr()". Also, do NOT remove the PIEACK- reset instruction at the end of this function!

6. Modify 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 the variable "CpuTimer0.InterruptCount" once every 50ms. There is 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++];
```

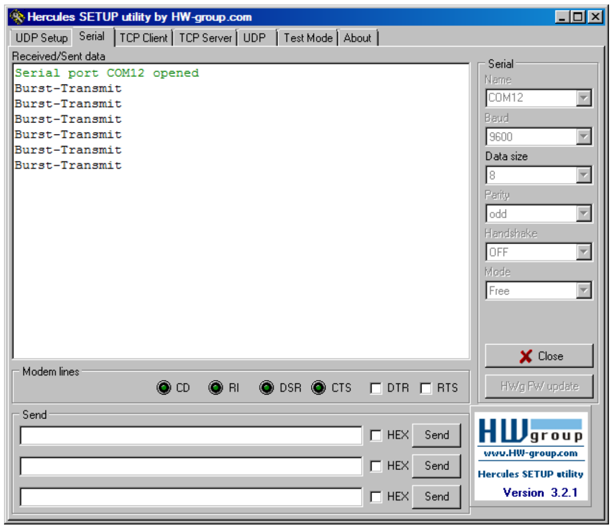
The difference between Lab9_2.c and Lab9_3.c is the initialization of the SCI-unit. In this lab, we have enabled the TX-FIFO interrupt to request a service when the FIFO-level is zero. This will be true immediately after the initialization of the SCI-unit and will cause the first TX-interrupt! The next 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 immediately. To do so, add the following instruction:

```
SciaRegs.SCIFFTX.bit.TXINTCLR = 1;
```

That's it.

Build, Load and Test

7. 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!



```
Serial port COM12 opened
Burst-Transmit
Burst-Transmit
Burst-Transmit
Burst-Transmit
Burst-Transmit
Burst-Transmit
Burst-Transmit
```

Serial
Name: COM12
Baud: 9600
Data size: 8
Parity: odd
Handshake: OFF
Mode: Free

8. Summary: The big improvement of this Lab9_3 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 F2833x SCI-transmit FIFO enhancement compared to a standard UART interface.

END of LAB 9_3

Lab 9_4: SCI – Receive & Transmit

The objective of this final exercise is to add the SCI receiver. Lab9_4 should wait until the message “Texas” has been received and answer transmitting “Instruments” back to the PC.

Use your code from Lab9_3 as a starting point.

Procedure

Open Files, Modify Project File

1. In project “Lab9” open file “Lab9_3.c” and save it as “Lab9_4.c”

2. Exclude file “Lab9_3.c” from build. Use a right mouse click at file “Lab9_3.c”, and enable “Exclude File(s) from Build”.

Modify Source Code

3. Open file “Lab9_4.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 interrupt vector table load 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;
     ```

   - Remove the start instruction for CpuTimer0:

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

   - Change the interrupt enable register to:

     ```
     IER = 0x100;
     ```

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

     ```
     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!

4. Change the contents of the variable “message[ ]” to “ Instruments! \n\r”. Note: Make sure to have 16 characters in this text message; ‘\n’ and ‘\r’ count as single characters!

5. Now we have to introduce a new interrupt service routine for the SCI receiver, called “SCIA_RX_isr()”. Declare its prototype at the beginning of your code:

```
interrupt void SCIA_RX_isr(void);
```

6. Replace the entry for this function inside the PIE vector table. Add this line directly after the entry-instruction for TXAINT:

```
PieVectTable.RXAINT  = &SCIA_RX_isr;
```

7. Enable the PIE interrupt for RXAINT:

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

8. Modify the initialization function for the SCI: “SCIA_Init()”.

   - Inside register “SCICTL2” set bit “RXBKINTENA” to 1 to enable the receiver interrupt.
   - For register “SCIFFTX”, do <u>NOT</u> 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.

9. At the end of your source code add interrupt function “SCIA_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 “SciaRegs.SCIRXBUF.bit.RXDT” you can empty the FIFO into a local variable “buffer[16]”.
   - The C standard string compare 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
```

10. Delete global variable “index”.

11. At the beginning of “Lab9_4.c”, add an include instruction for header file “string.h”:

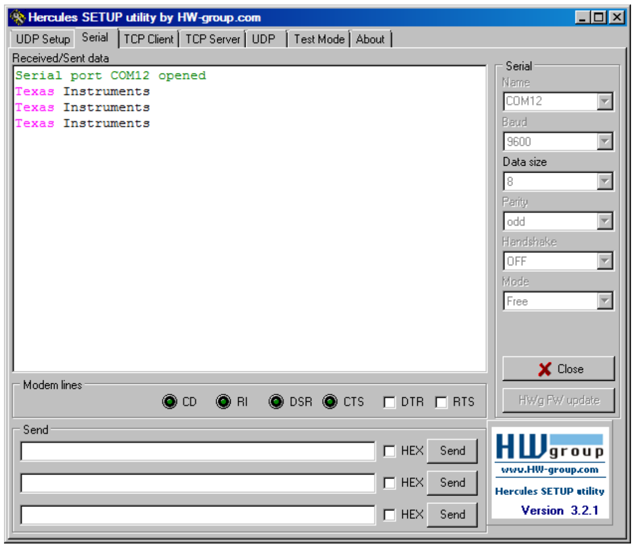
```
#include <string.h>
```

That’s it.

Build, Load and Test

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

13. Start your Terminal program and type in the text “Texas”. The F2833x will respond with the string “ Instruments\n\r”. If not ➔ debug!



```
Serial port COM12 opened
Texas Instruments
Texas Instruments
Texas Instruments
```

Serial
Name: COM12
Baud: 9600
Data size: 8
Parity: odd
Handshake: OFF
Mode: Free

Optional Exercise 9_5

DSC – Junkies only! ➔ Remote Control of the F2833x by a PC!

Try to combine the “binary counter” exercise “Lab6.c” with the SCI-lab “Lab9_4.c”. Let the PC send a string with a numerical value and use this value to control the speed of the “binary counter”!

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).

This page has been left blank intentionally.