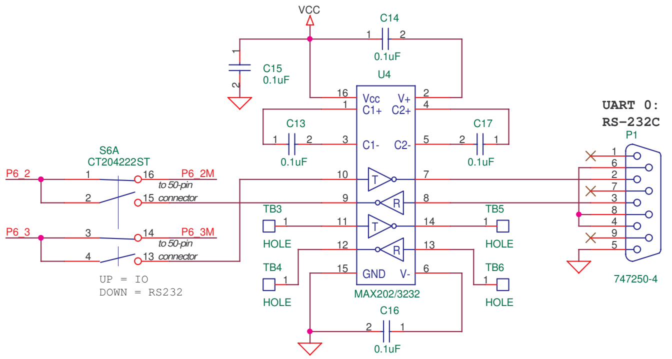
Serial Communications

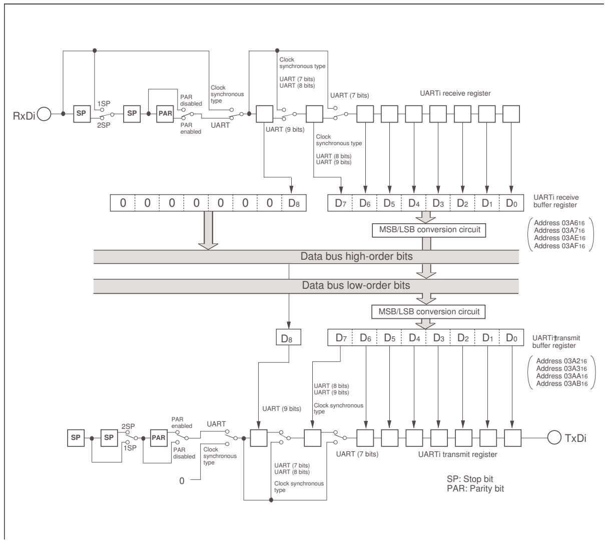
In these notes . . .

General Communications

Serial Communications

- RS232 standard
- UART operation
- Polled Code
- SCI/I^{2}C





Data Communications

There was no standard for networks in the early days and as a result it was difficult for networks to communicate with each other.

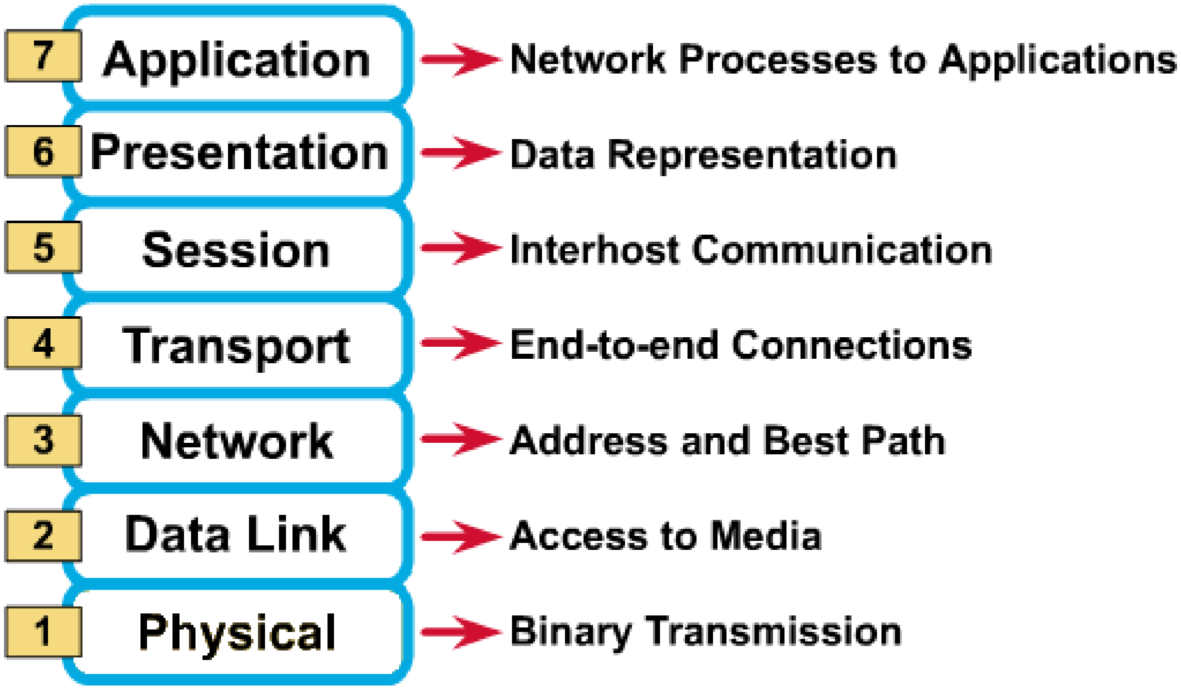
The International Organization for Standardization (ISO) recognized this and in 1984 introduced the Open Systems Interconnection (OSI) reference model.

The OSI reference model organizes network functions into seven numbered layers.

Each layer provides a service to the layer above it in the protocol specification and communicates with the same layer’s software or hardware on other computers.

Layers 5-7 are concerned with services for the applications.

Layers 1-4 are concerned with the flow of data from end to end through the network



7 Application → Network Processes to Applications

6 Presentation → Data Representation

5 Session → Interhost Communication

4 Transport → End-to-end Connections

3 Network → Address and Best Path

2 Data Link → Access to Media

1 Physical → Binary Transmission

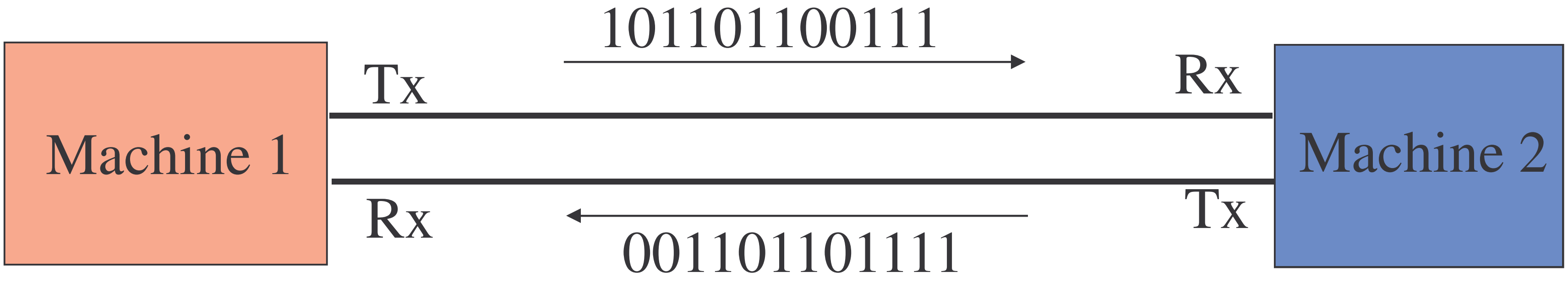
Physical Layer (1) – Serial Communications

The basic premise of serial communications is that one or two wires are used to transmit digital data.

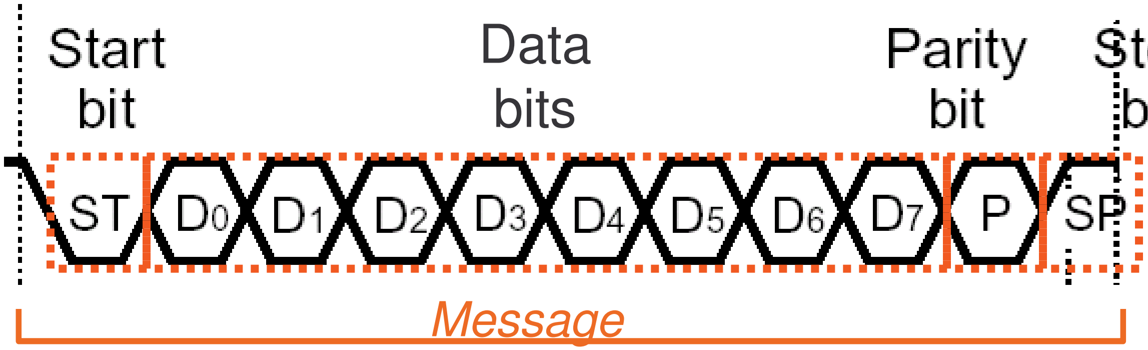
- Of course, ground reference is also needed (extra wire)

Can be one way or two way, usually two way, hence two communications wires.

Often other wires are used for other aspects of the communications (ground, “clear-to-send”, “data terminal ready”, etc).



Serial Communication Basics



Send one bit of the message at a time

Message fields

- Start bit (one bit)
- Data (LSB first or MSB, and size – 7, 8, 9 bits)
- Optional parity bit is used to make total number of ones in data even or odd
- Stop bit (one or two bits)

All devices on network or link must use same communications parameters

- The speed of communication must be the same as well (300, 600, 1200, 2400, 9600, 14400, 19200, etc.)

More sophisticated network protocols have more information in each message

- Medium access control – when multiple nodes are on bus, they must arbitrate for permission to transmit
- Addressing information – for which node is this message intended?
- Larger data payload
- Stronger error detection or error correction information
- Request for immediate response (“in-frame”)

Bit Rate vs. Baud Rate

Bit Rate: how many ***data bits*** are transmitted per second?

Baud Rate: how many ***symbols*** are transmitted per second?

- == How many times does the communication channel change state per second?
- A symbol may be represented by a voltage level, a sine wave's frequency or phase, etc.

These may be different

- Extra symbols (channel changes) may be inserted for framing, error detection, acknowledgment, etc. These ***reduce*** the bit rate
- A single symbol might encode more than one bit. This ***increases*** the bit rate.
  - E.g. multilevel signaling, quadrature amplitude modulation, phase amplitude modulation, etc.

Serial Communication Basics

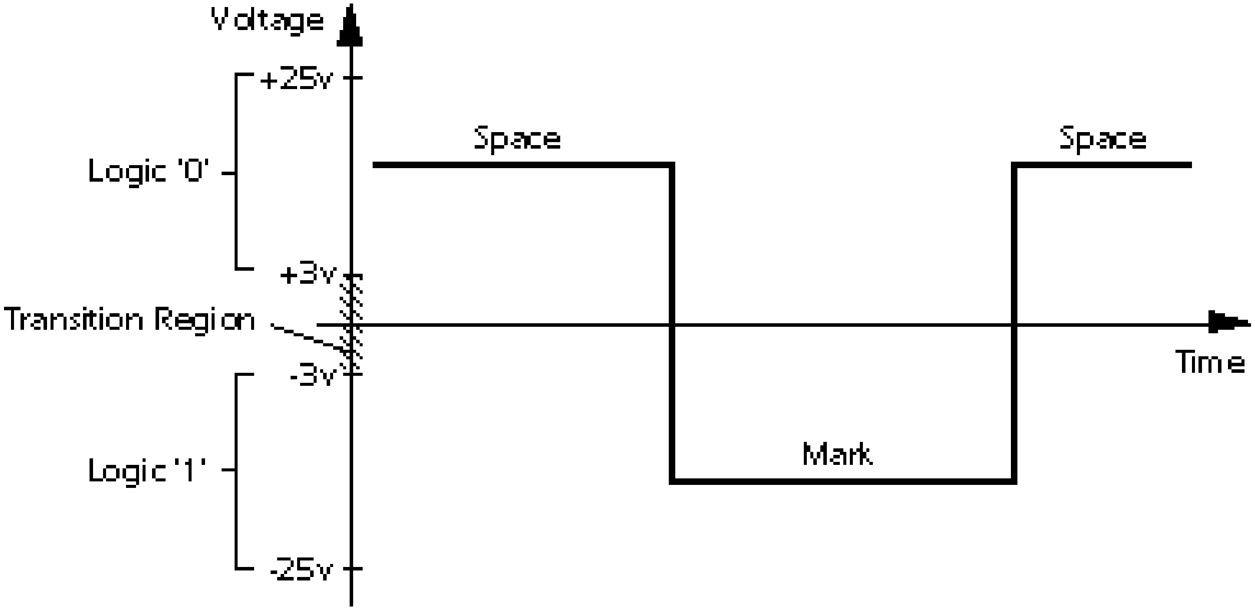
RS232: rules on connector, signals/pins, voltage levels, handshaking, etc.

RS232: Fulfilling All Your Communication Needs, Robert Ashby

Quick Reference for RS485, RS422, RS232 and RS423

Not so quick reference:
The RS232 Standard: A Tutorial with Signal Names and Definitions,
Christopher E. Strangio

Bit vs Baud rates:
http://www.totse.com/en/technology/telecommunications/bits.html



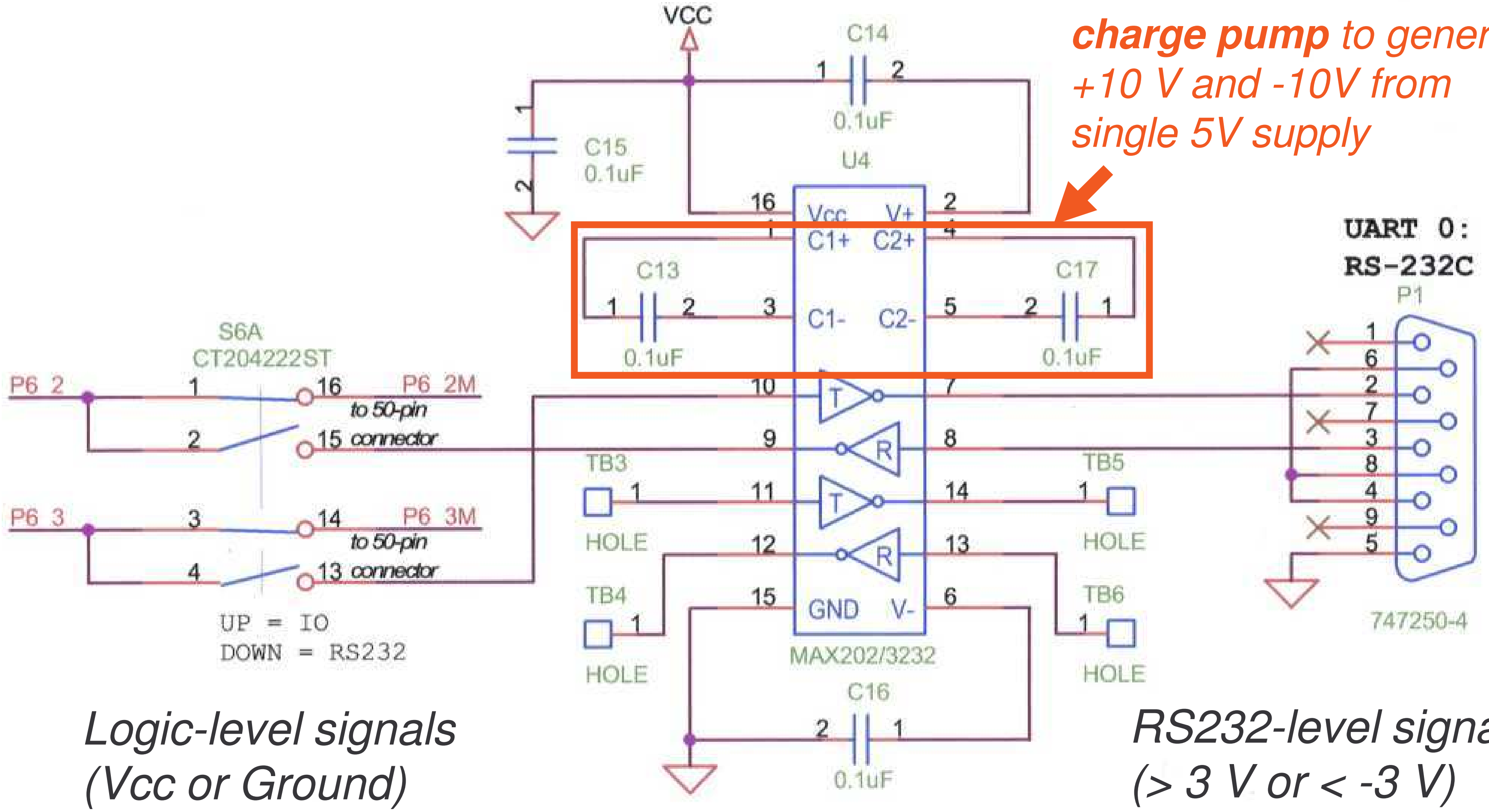
UART Concepts

UART

- Universal – configurable to fit protocol requirements
- Asynchronous – no clock line needed to de-serialize bits
- Receiver/Transmitter

RS232 Communications Circuit

Example RS-232 buffer (level-shifting) circuit



Max202/3232 includes ***charge pump*** *to generate +10 V and -10V from single 5V supply*

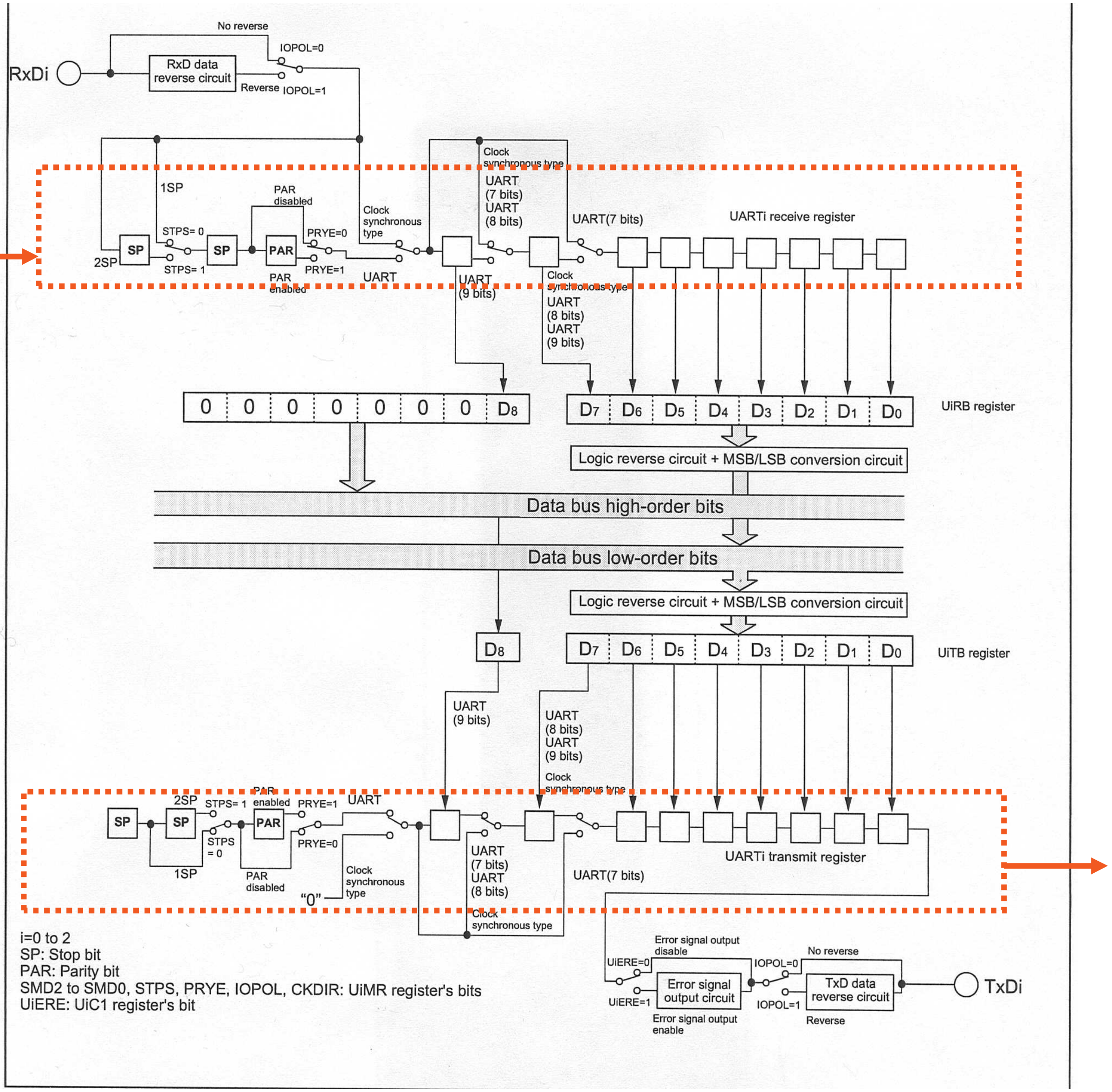
Logic-level signals (Vcc or Ground)

RS232-level signals (> 3 V or < -3 V)

General UART Concepts

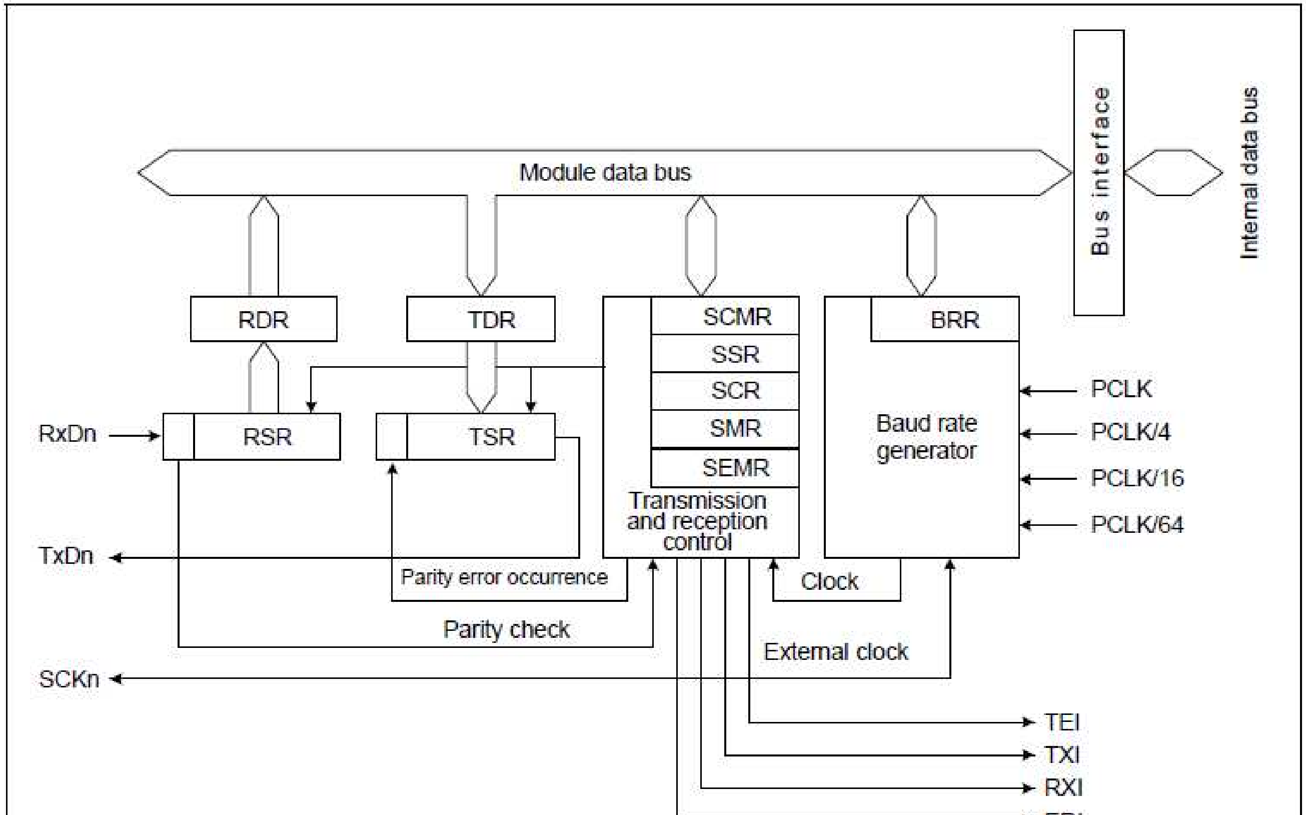
UART subsystems

- Two fancy shift registers
  - Parallel to serial for transmit
  - Serial to parallel for receive
- Programmable clock source
  - Clock must run at 16x desired bit rate
- Error detection
  - Detect bad stop or parity bits
  - Detect receive buffer overwrite
- Interrupt generators
  - Character received
  - Character transmitted, ready to send another



i=0 to 2
SP: Stop bit
PAR: Parity bit
SMD2 to SMD0, STPS, PRYE, IOPOL, CKDIR: UiMR register's bits
UiERE: UiC1 register's bit

Block Diagram of RX62N Serial Comm Interface



[Legend]

RSR:	Receive shift register
RDR:	Receive data register
TSR:	Transmit shift register
TDR:	Transmit data register
SMR:	Serial mode register
SCR:	Serial control register
SSR:	Serial status register
SCMR:	Smart card mode register
BRR:	Bit rate register
SEMR:	Serial extended mode register

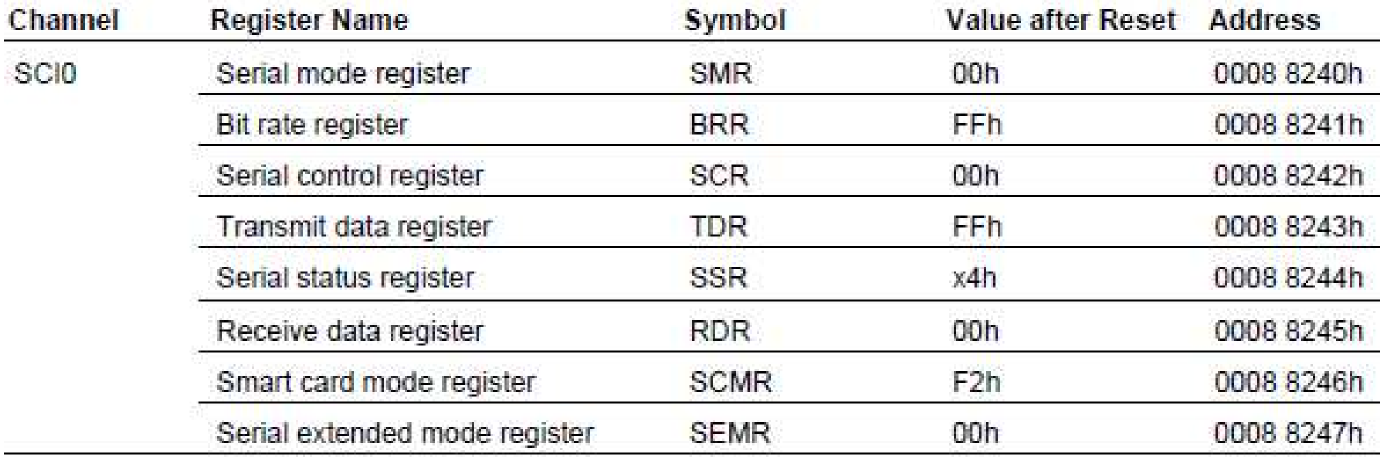
SCI in UART Mode

To communicate from the RX62N chip, you need to set up several registers, including:

- Mode
- Speed
- Parity
- Stop bits
- Configuration

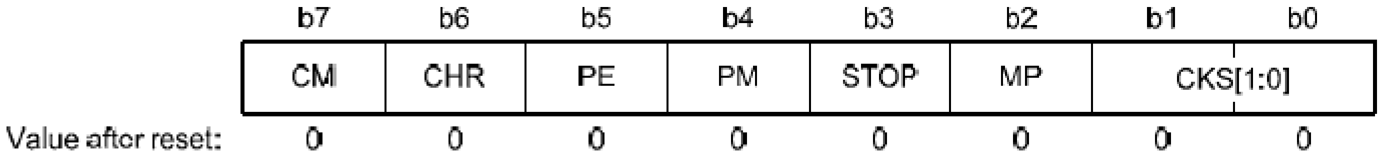
There are two primary “Data Registers”

- SCIx.RDR (Receive Data Register)
- SCIx.TDR (Transmit Data Register)



Channel	Register Name	Symbol	Value after Reset	Address
SCI0	Serial mode register	SMR	00h	0008 8240h
	Bit rate register	BRR	FFh	0008 8241h
	Serial control register	SCR	00h	0008 8242h
	Transmit data register	TDR	FFh	0008 8243h
	Serial status register	SSR	x4h	0008 8244h
	Receive data register	RDR	00h	0008 8245h
	Smart card mode register	SCMR	F2h	0008 8246h
	Serial extended mode register	SEMR	00h	0008 8247h

Serial Mode Register (SMR)



	b7	b6	b5	b4	b3	b2	b1	b0
	CM	CHR	PE	PM	STOP	MP	CKS[1:0]	
Value after reset:	0	0	0	0	0	0	0	0

SCIx.SMR – Operational values of the UART

Each bit is encoded to make a special meaning

CKS: transmission speed (more later)

MP: Multi processor (set to 0)

STOP: Stop bits

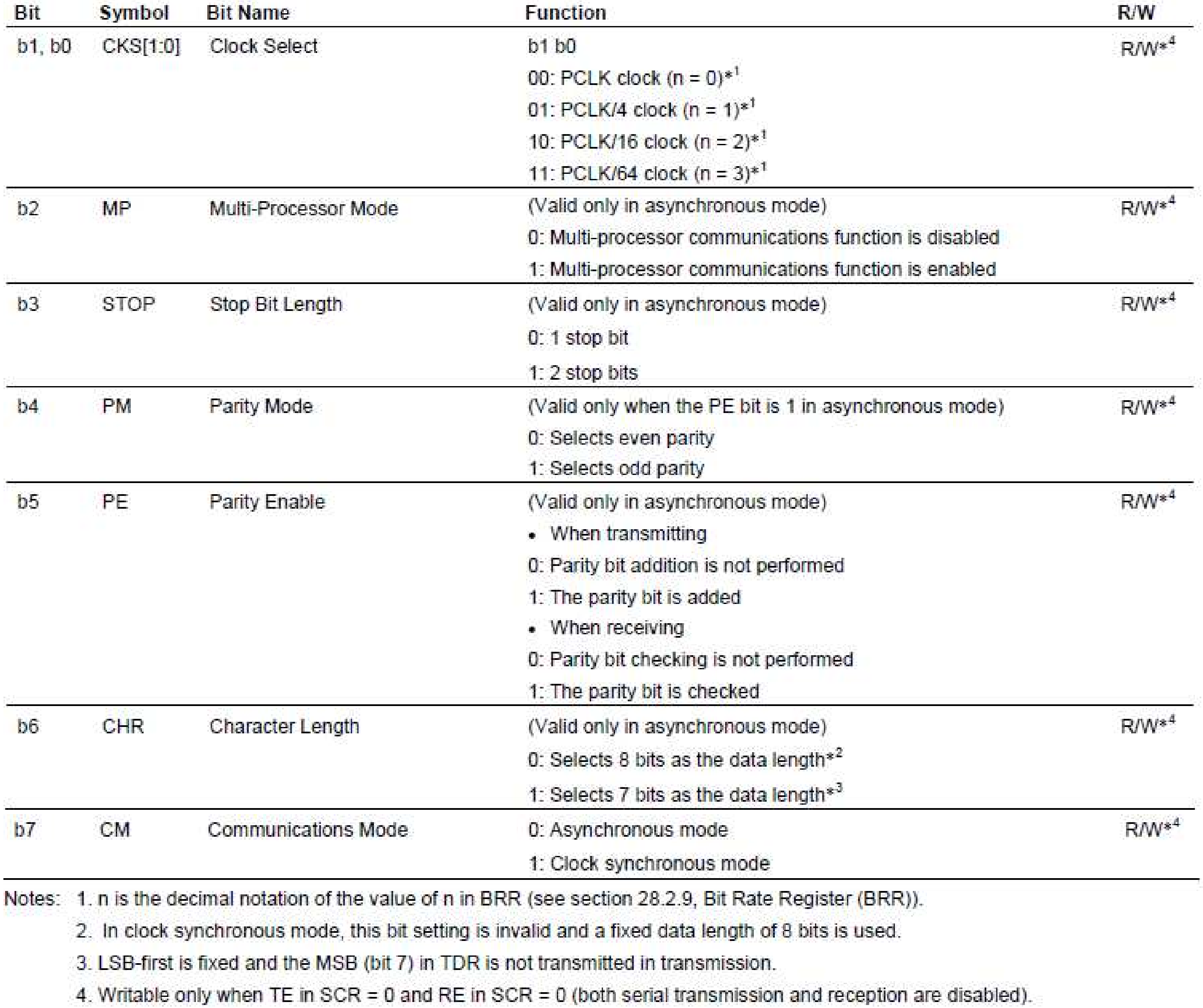
PM: Parity mode

PE: Parity Enable

CHR: Length of data

CM: Communications mode

Reading a Manual (SMR)



Bit	Symbol	Bit Name	Function	R/W
b1, b0	CKS[1:0]	Clock Select	b1 b0 00: PCLK clock (n = 0)*[1] 01: PCLK/4 clock (n = 1)*[1] 10: PCLK/16 clock (n = 2)*[1] 11: PCLK/64 clock (n = 3)*[1]	R/W*[4]
b2	MP	Multi-Processor Mode	(Valid only in asynchronous mode) 0: Multi-processor communications function is disabled 1: Multi-processor communications function is enabled	R/W*[4]
b3	STOP	Stop Bit Length	(Valid only in asynchronous mode) 0: 1 stop bit 1: 2 stop bits	R/W*[4]
b4	PM	Parity Mode	(Valid only when the PE bit is 1 in asynchronous mode) 0: Selects even parity 1: Selects odd parity	R/W*[4]
b5	PE	Parity Enable	(Valid only in asynchronous mode) • When transmitting 0: Parity bit addition is not performed 1: The parity bit is added • When receiving 0: Parity bit checking is not performed 1: The parity bit is checked	R/W*[4]
b6	CHR	Character Length	(Valid only in asynchronous mode) 0: Selects 8 bits as the data length*[2] 1: Selects 7 bits as the data length*[3]	R/W*[4]
b7	CM	Communications Mode	0: Asynchronous mode 1: Clock synchronous mode	R/W*[4]

Notes: 1. n is the decimal notation of the value of n in BRR (see section 28.2.9, Bit Rate Register (BRR)).

2. In clock synchronous mode, this bit setting is invalid and a fixed data length of 8 bits is used.

3. LSB-first is fixed and the MSB (bit 7) in TDR is not transmitted in transmission.

4. Writable only when TE in SCR = 0 and RE in SCR = 0 (both serial transmission and reception are disabled).

Setting up the Serial Control Register

We will use SCI0

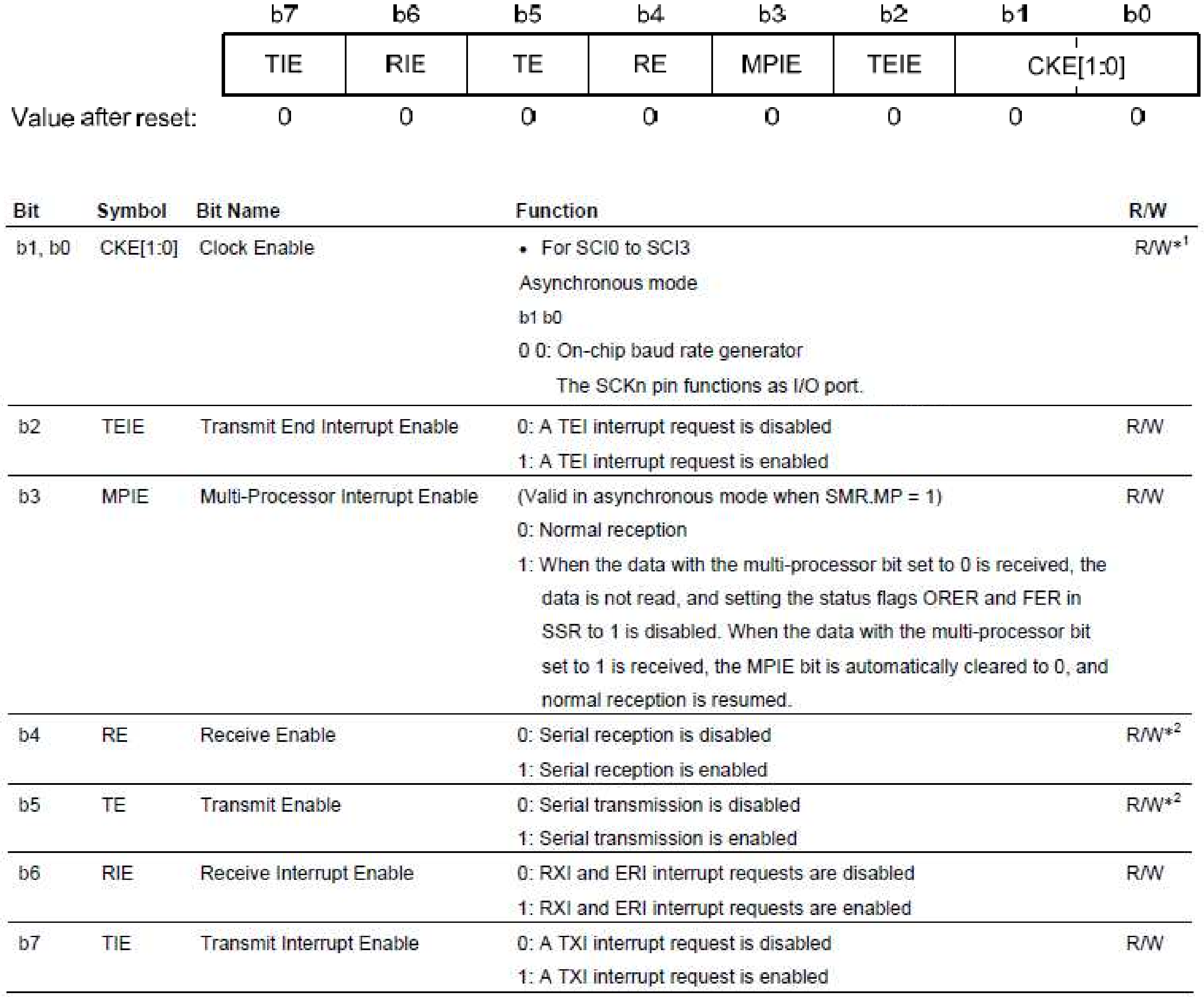
There are several “control” registers you need to set up before you can communicate.

- First, you need to set up the speed of your port.
- Select 8 data bits, no parity, one stop bit (8N1)
- Asynchronous mode

What would the byte be set as?

SCI0.SMR.BYTE =

Serial Control Register

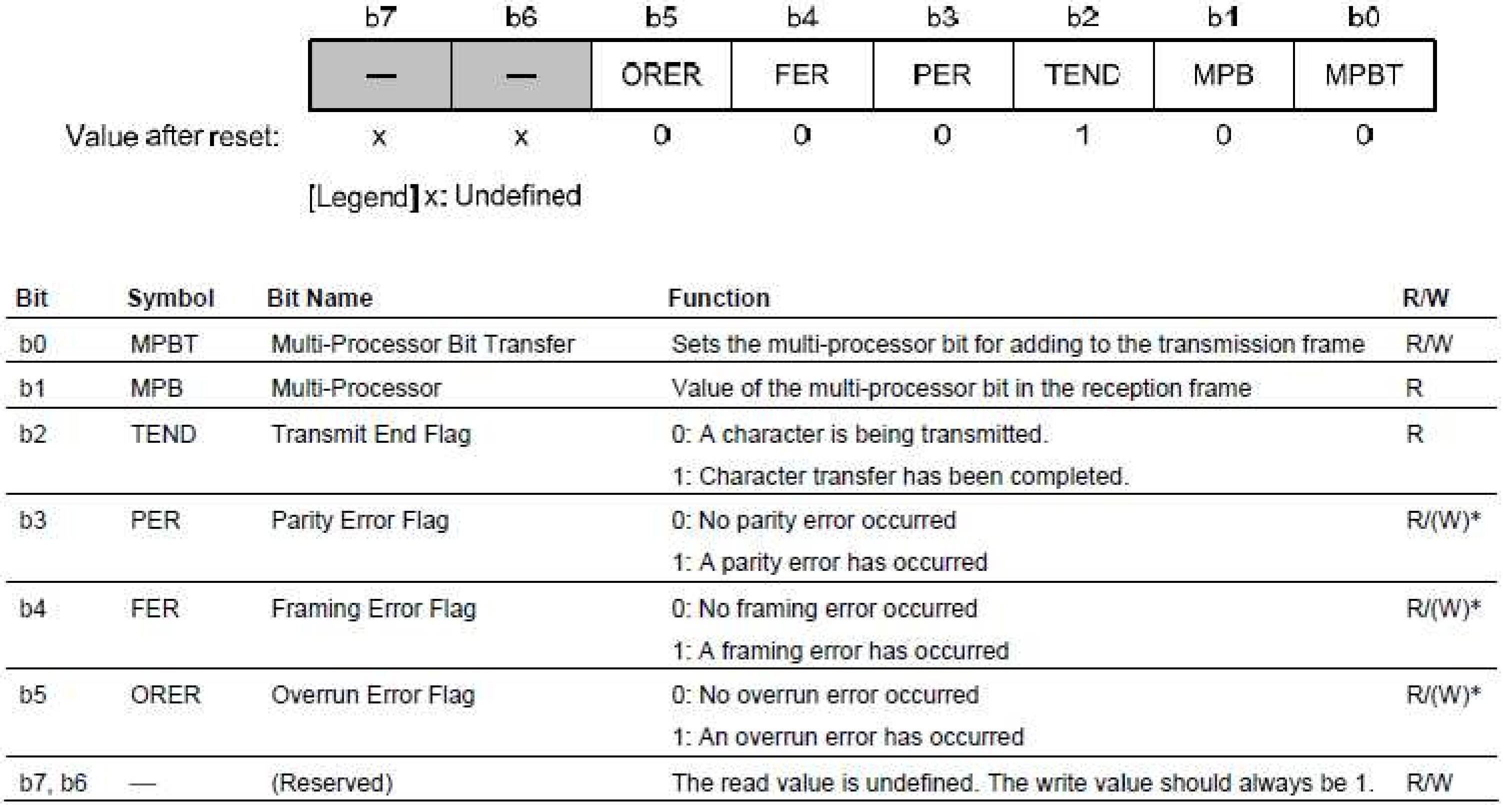


	b7	b6	b5	b4	b3	b2	b1	b0
	TIE	RIE	TE	RE	MPIE	TEIE	CKE[1:0]	
Value after reset:	0	0	0	0	0	0	0	0

Bit	Symbol	Bit Name	Function	R/W
b1, b0	CKE[1:0]	Clock Enable	• For SCI0 to SCI3 Asynchronous mode b1 b0 0 0: On-chip baud rate generator The SCKn pin functions as I/O port.	R/W*1
b2	TEIE	Transmit End Interrupt Enable	0: A TEI interrupt request is disabled 1: A TEI interrupt request is enabled	R/W
b3	MPIE	Multi-Processor Interrupt Enable	(Valid in asynchronous mode when SMR.MP = 1) 0: Normal reception 1: When the data with the multi-processor bit set to 0 is received, the data is not read, and setting the status flags ORER and FER in SSR to 1 is disabled. When the data with the multi-processor bit set to 1 is received, the MPIE bit is automatically cleared to 0, and normal reception is resumed.	R/W
b4	RE	Receive Enable	0: Serial reception is disabled 1: Serial reception is enabled	R/W*2
b5	TE	Transmit Enable	0: Serial transmission is disabled 1: Serial transmission is enabled	R/W*2
b6	RIE	Receive Interrupt Enable	0: RXI and ERI interrupt requests are disabled 1: RXI and ERI interrupt requests are enabled	R/W
b7	TIE	Transmit Interrupt Enable	0: A TXI interrupt request is disabled 1: A TXI interrupt request is enabled	R/W

Serial Control Register – Serial Status Register

Check to see is communications was successful (SCIx.SSR)



	b7	b6	b5	b4	b3	b2	b1	b0
	—	—	ORER	FER	PER	TEND	MPB	MPBT
Value after reset:	x	x	0	0	0	1	0	0

[Legend] x: Undefined

Bit	Symbol	Bit Name	Function	R/W
b0	MPBT	Multi-Processor Bit Transfer	Sets the multi-processor bit for adding to the transmission frame	R/W
b1	MPB	Multi-Processor	Value of the multi-processor bit in the reception frame	R
b2	TEND	Transmit End Flag	0: A character is being transmitted. 1: Character transfer has been completed.	R
b3	PER	Parity Error Flag	0: No parity error occurred 1: A parity error has occurred	R/(W)*
b4	FER	Framing Error Flag	0: No framing error occurred 1: A framing error has occurred	R/(W)*
b5	ORER	Overrun Error Flag	0: No overrun error occurred 1: An overrun error has occurred	R/(W)*
b7, b6	—	(Reserved)	The read value is undefined. The write value should always be 1.	R/W

Identifying Errors

```
char read_sci0_status;
read_sci0_status = SCI0.SSR.BYTE;
```

What does it mean if the value holds 0x04?

What does it mean if the value holds 0x0C?

What does it mean if the value holds 0x24?

What does it mean if the value holds 0x20?

Setting up Speed of the Serial Port

The speed of communications is a combination of

- PCLK
- Bits CKS in the SMR
- The Bit Rate Register (BRR)

Based on formula:

$$N = \frac{PCLK \times 10^6}{64 \times 2^{2n-1} \times B} - 1$$

B=bit rate, N=BRR setting, n=CKS setting

So, if you want to communicate at 38,400 bps, if your PCLK is 50 MHz, set n=0 and N=40

```
SCI0.BRR.BYTE = 40;
```

Example – Change to Slower Clock

What about a slower clock?

$$N = \frac{PCLK \times 10^6}{64 \times 2^{2n-1} \times B} - 1$$

Say, 2400 bps? What do you need to set n and N?

Class Exercise – Set up clock

Set up to 115,200, including writing the code for BRR.

Error rate

The error rate is associated to the settings of n and N, since you will not get the exact value of xx.0.

So, if you want to communicate at 38,400 bps, if your PCLK is 50 MHz, set n=0 and N=40, error is:

$$\text{Error}(\%) = \{ \frac{\text{PCLK} \times 10^6}{B \times 64 \times 2^{2n-1} \times (N+1)} - 1 \} \times 100$$

What is the Maximum Speed??

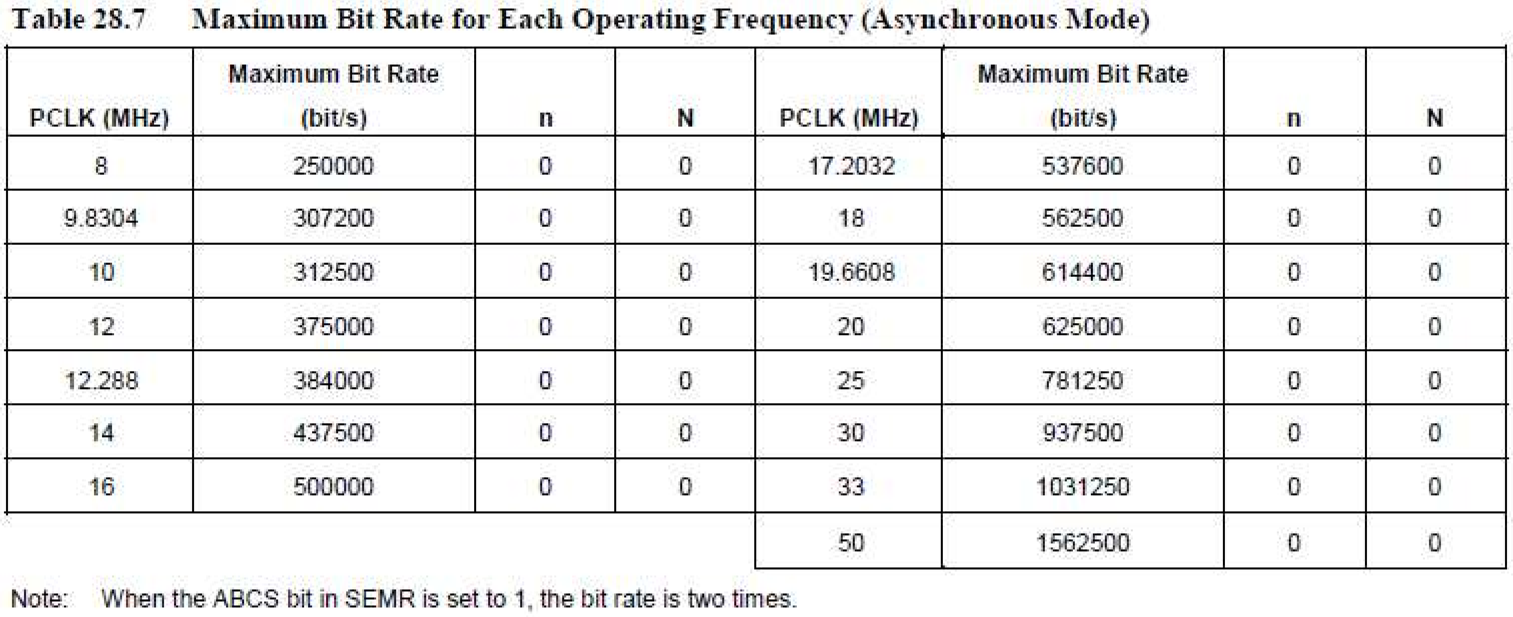


Table 28.7 Maximum Bit Rate for Each Operating Frequency (Asynchronous Mode)

PCLK (MHz)	Maximum Bit Rate (bit/s)	n	N
8	250000	0	0
9.8304	307200	0	0
10	312500	0	0
12	375000	0	0
12.288	384000	0	0
14	437500	0	0
16	500000	0	0
17.2032	537600	0	0
18	562500	0	0
19.6608	614400	0	0
20	625000	0	0
25	781250	0	0
30	937500	0	0
33	1031250	0	0
50	1562500	0	0

Note: When the ABCS bit in SEMR is set to 1, the bit rate is two times.

Example – Set up Communications

Write a function to set up SCI0 to 115,200 bps, 8 data bits, odd parity, 1 stop bit:

Example – Send/receive data

Write a small function to send the string “Sending data” through the SCI0 port. Make sure to wait for the previous char to be transmitted before you send the next char:

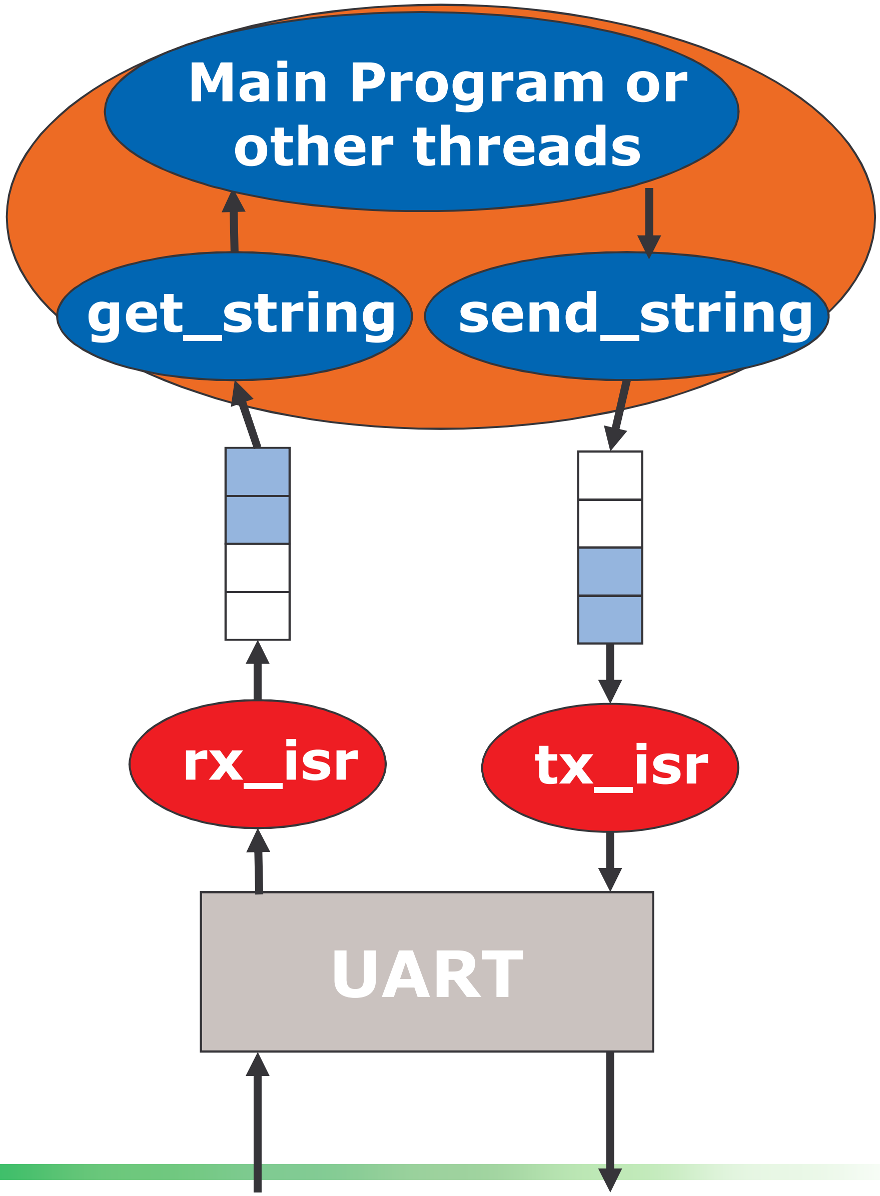
Serial Communications and Interrupts

Now we have three separate threads of control in the program

- main program (and subroutines it calls)
- Transmit ISR – executes when UART is ready to send another character
- Receive ISR – executes when UART receives a character

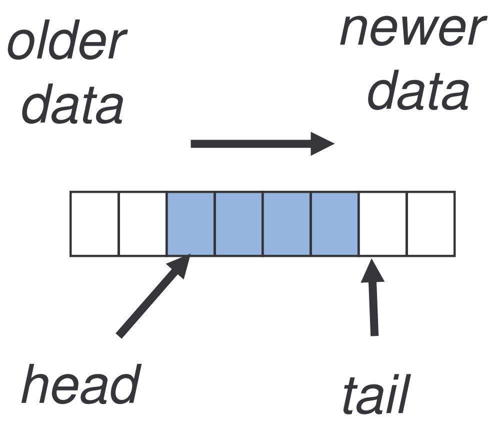
Need a way of buffering information between threads

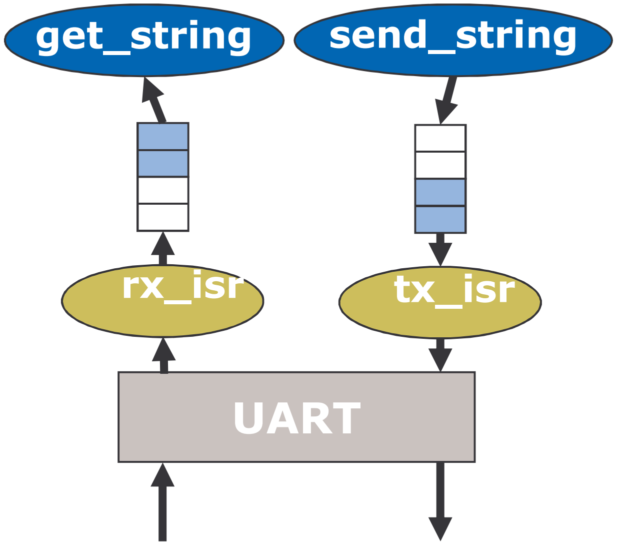
- Solution: circular queue with head and tail pointers
- One for tx, one for rx



Code to Implement Queues

- Enqueue at tail (tail_ptr points to next free entry), dequeue from head (head_ptr points to item to remove)
- #define the queue size to make it easy to change
- One queue per direction
  - tx ISR unloads tx_q
  - rx ISR loads rx_q
- Other threads (e.g. main) load tx_q and unload rx_q
- Need to wrap pointer at end of buffer to make it circular, use % (modulus, remainder) operator
- Queue is empty if size == 0
- Queue is full if size == Q_SIZE





Defining the Queues

```
#define Q_SIZE (32)

typedef struct {
  unsigned char Data[Q_SIZE];
  unsigned int Head; // points to oldest data element
  unsigned int Tail; // points to next free space
  unsigned int Size; // quantity of elements in queue
} Q_T;

Q_T tx_q, rx_q;
```

Initialization and Status Inquiries

```
void Q_Init(Q_T * q) {
  unsigned int i;
  for (i=0; i<Q_SIZE; i++)
    q->Data[i] = 0;  // to simplify our lives when debugging
  q->Head = 0;
  q->Tail = 0;
  q->Size = 0;
}

int Q_Empty(Q_T * q) {
  return q->Size == 0;
}

int Q_Full(Q_T * q) {
  return q->Size == Q_SIZE;
}
```

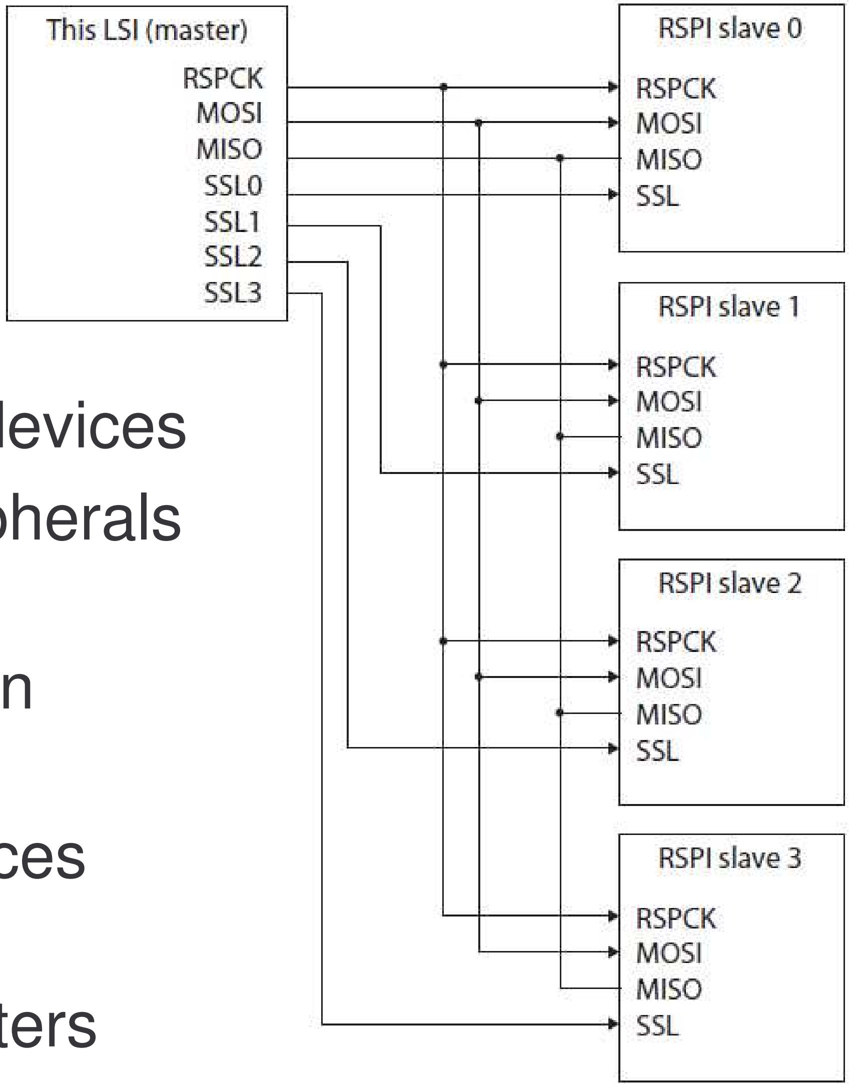
Enqueue and Dequeue

```
// Q_Enqueue - Called by a UART ISR - put a char on the queue
int Q_Enqueue(Q_T * q, unsigned char d) {
  if (!Q_Full(q)) { // What if queue is full?
    q->Data[q->Tail++] = d;
    q->Tail %= Q_SIZE;
    q->Size++;
    return 1; // success
  } else
    return 0; // failure
}

// Q_Dequeue-called by a consumer function-take a char from queue
unsigned char Q_Dequeue(Q_T * q) {
  unsigned char t=0;
  if (!Q_Empty(q)) {    // Must check to see if queue is empty 1st
    t = q->Data[q->Head];
    q->Data[q->Head++] = 0;   // to simplify debugging, clear
    q->Head %= Q_SIZE;
    q->Size--;
  }
  return t;
}
```

Serial Peripheral Interface (SPI)

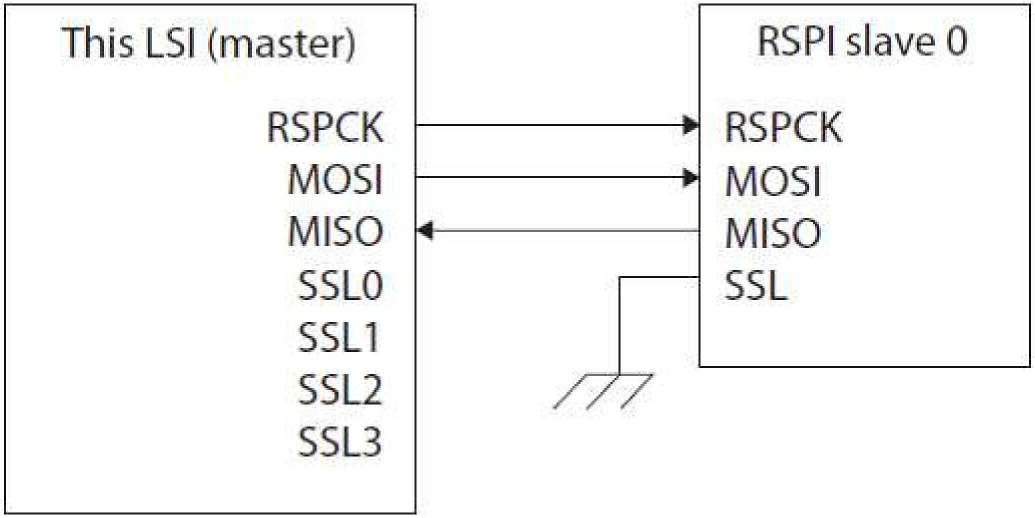
- SPI bus is a de facto standard developed by Motorola
- Can work with as few as three wires, but more needed to access additional devices
- Better method to access peripherals than parallel I/O.
- Common clock means you can transmit at 25.0 Mbps
- Intended for very short distances (i.e. on-board)
- The RX62N has two SPI masters



SPI Details

- •Serial Clock – RSPCK
- •Master Out, Slave in – MOSI (transmission from RX62N)
- •Master In, Slave Out – MISO (transmission from peripheral)
- •Slave Select – SSLx (select one of the peripheral devices)

We will not investigate Multiple Master modes

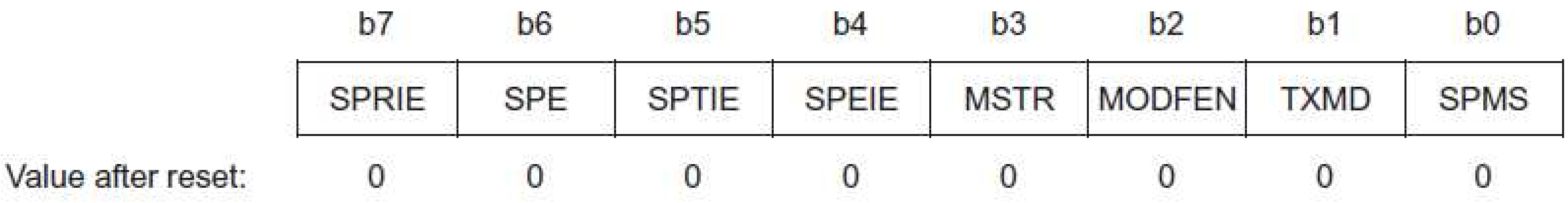


SPI registers

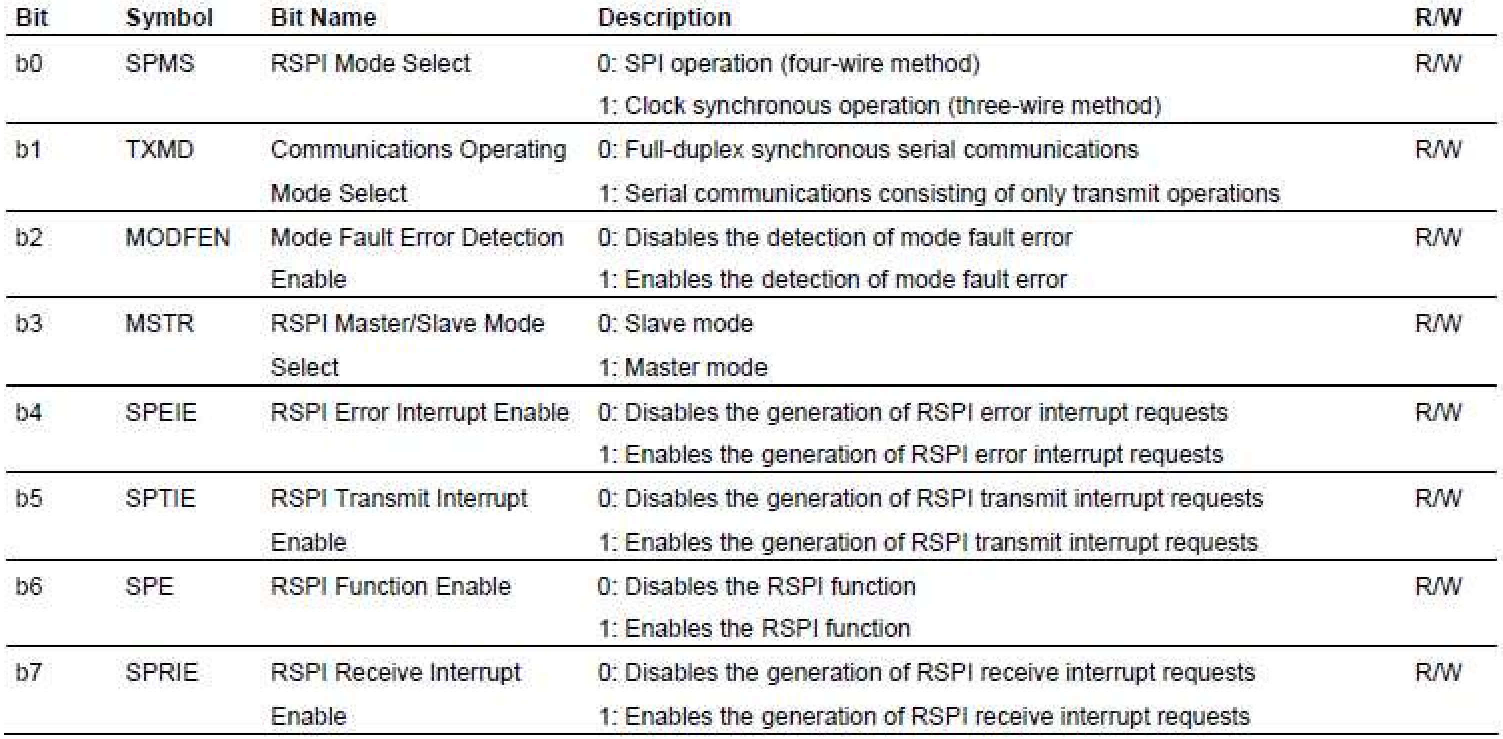
- Serial Peripheral Control Register (SPCR)
- Serial Peripheral Control Register 2 (SPCR2)
- Serial Peripheral Pin Control Register (SPPCR) – set to 0x00
- Slave Select Polarity (SSLP)
- Serial Peripheral Status (SPS)

- Serial Peripheral Data Register (SPDR)

Serial Peripheral Control Register (SPCR)

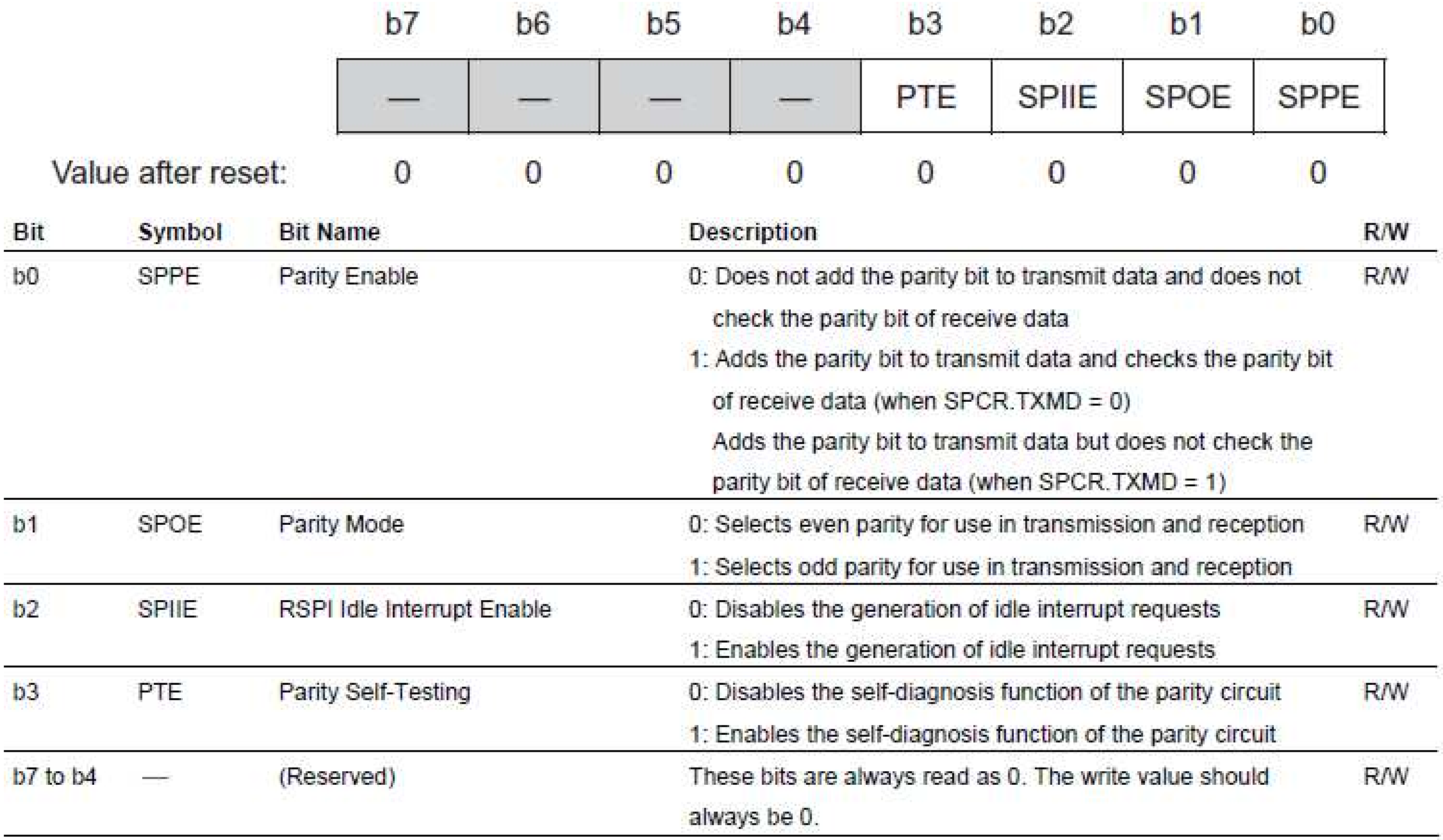


	b7	b6	b5	b4	b3	b2	b1	b0
	SPRIE	SPE	SPTIE	SPEIE	MSTR	MODFEN	TXMD	SPMS
Value after reset:	0	0	0	0	0	0	0	0



Bit	Symbol	Bit Name	Description	R/W
b0	SPMS	RSPI Mode Select	0: SPI operation (four-wire method) 1: Clock synchronous operation (three-wire method)	R/W
b1	TXMD	Communications Operating Mode Select	0: Full-duplex synchronous serial communications 1: Serial communications consisting of only transmit operations	R/W
b2	MODFEN	Mode Fault Error Detection Enable	0: Disables the detection of mode fault error 1: Enables the detection of mode fault error	R/W
b3	MSTR	RSPI Master/Slave Mode Select	0: Slave mode 1: Master mode	R/W
b4	SPEIE	RSPI Error Interrupt Enable	0: Disables the generation of RSPI error interrupt requests 1: Enables the generation of RSPI error interrupt requests	R/W
b5	SPTIE	RSPI Transmit Interrupt Enable	0: Disables the generation of RSPI transmit interrupt requests 1: Enables the generation of RSPI transmit interrupt requests	R/W
b6	SPE	RSPI Function Enable	0: Disables the RSPI function 1: Enables the RSPI function	R/W
b7	SPRIE	RSPI Receive Interrupt Enable	0: Disables the generation of RSPI receive interrupt requests 1: Enables the generation of RSPI receive interrupt requests	R/W

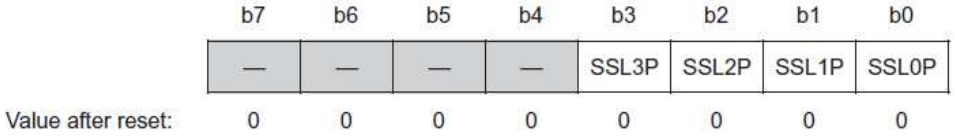
Serial Peripheral Control Register 2 (SPCR2)



	b7	b6	b5	b4	b3	b2	b1	b0
	—	—	—	—	PTE	SPIIE	SPOE	SPPE
Value after reset:	0	0	0	0	0	0	0	0

Bit	Symbol	Bit Name	Description	R/W
b0	SPPE	Parity Enable	0: Does not add the parity bit to transmit data and does not check the parity bit of receive data 1: Adds the parity bit to transmit data and checks the parity bit of receive data (when SPCR.TXMD = 0) Adds the parity bit to transmit data but does not check the parity bit of receive data (when SPCR.TXMD = 1)	R/W
b1	SPOE	Parity Mode	0: Selects even parity for use in transmission and reception 1: Selects odd parity for use in transmission and reception	R/W
b2	SPIIE	RSPI Idle Interrupt Enable	0: Disables the generation of idle interrupt requests 1: Enables the generation of idle interrupt requests	R/W
b3	PTE	Parity Self-Testing	0: Disables the self-diagnosis function of the parity circuit 1: Enables the self-diagnosis function of the parity circuit	R/W
b7 to b4	—	(Reserved)	These bits are always read as 0. The write value should always be 0.	R/W

Slave Select Polarity (SSLP) – set these to 0 (active low)



	b7	b6	b5	b4	b3	b2	b1	b0
	—	—	—	—	SSL3P	SSL2P	SSL1P	SSL0P
Value after reset:	0	0	0	0	0	0	0	0

Serial Peripheral Bit Rate Register (SPBR)

8 Bit value, used with SPCR

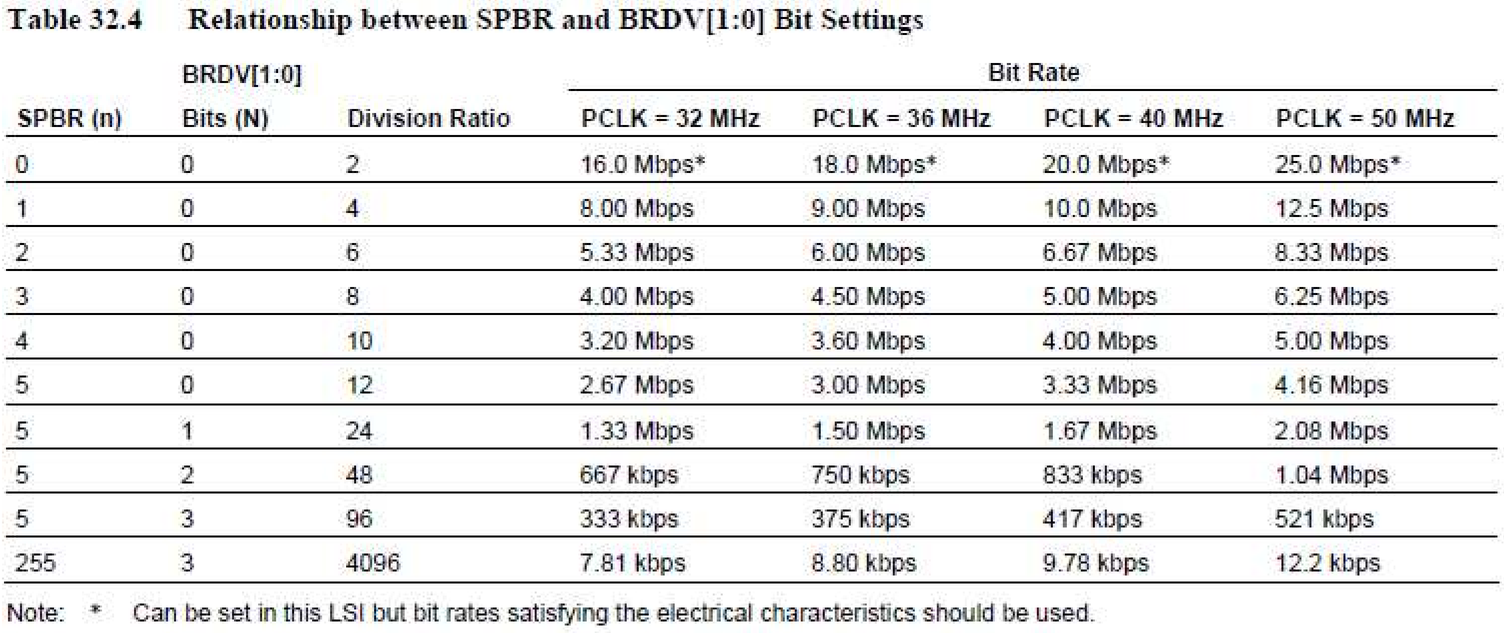
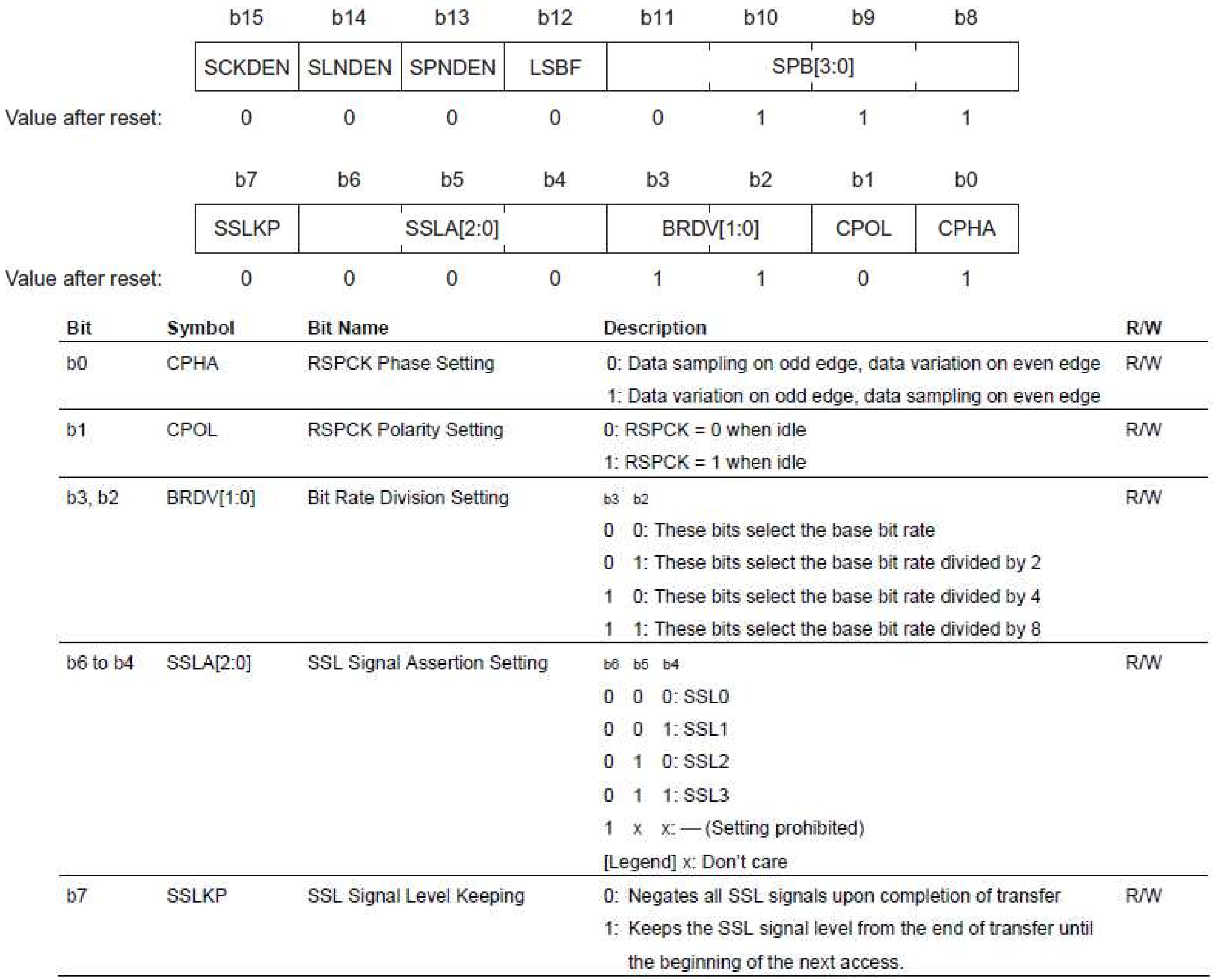


Table 32.4 Relationship between SPBR and BRDV[1:0] Bit Settings

	BRDV[1:0]		Bit Rate			
SPBR (n)	Bits (N)	Division Ratio	PCLK = 32 MHz	PCLK = 36 MHz	PCLK = 40 MHz	PCLK = 50 MHz
0	0	2	16.0 Mbps*	18.0 Mbps*	20.0 Mbps*	25.0 Mbps*
1	0	4	8.00 Mbps	9.00 Mbps	10.0 Mbps	12.5 Mbps
2	0	6	5.33 Mbps	6.00 Mbps	6.67 Mbps	8.33 Mbps
3	0	8	4.00 Mbps	4.50 Mbps	5.00 Mbps	6.25 Mbps
4	0	10	3.20 Mbps	3.60 Mbps	4.00 Mbps	5.00 Mbps
5	0	12	2.67 Mbps	3.00 Mbps	3.33 Mbps	4.16 Mbps
5	1	24	1.33 Mbps	1.50 Mbps	1.67 Mbps	2.08 Mbps
5	2	48	667 kbps	750 kbps	833 kbps	1.04 Mbps
5	3	96	333 kbps	375 kbps	417 kbps	521 kbps
255	3	4096	7.81 kbps	8.80 kbps	9.78 kbps	12.2 kbps

Note: * Can be set in this LSI but bit rates satisfying the electrical characteristics should be used.

Serial Peripheral Command Register (SPCMDx)

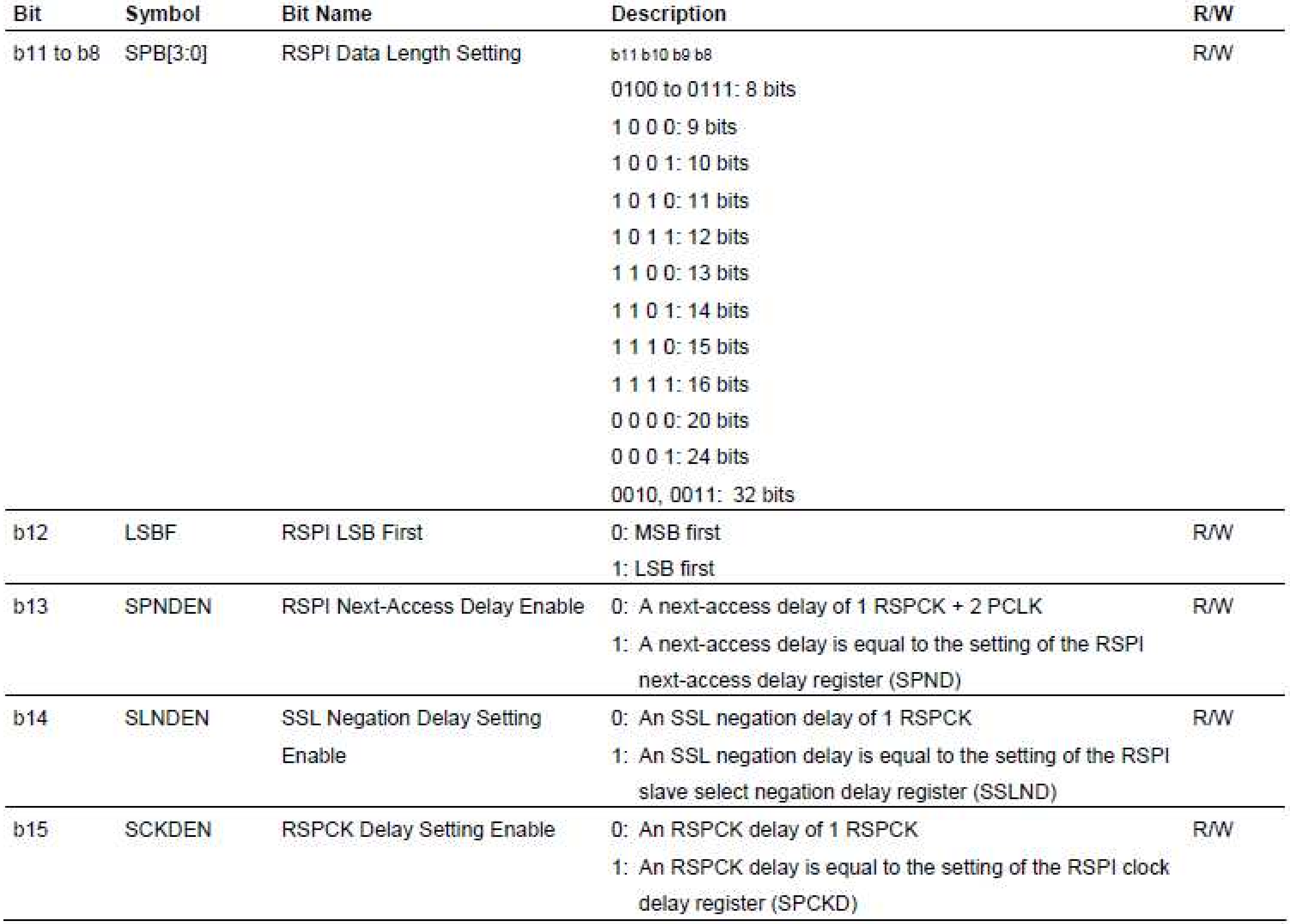


	b15	b14	b13	b12	b11	b10	b9	b8
	SCKDEN	SLNDEN	SPNDEN	LSBF	SPB[3:0]			
Value after reset:	0	0	0	0	0	1	1	1

	b7	b6	b5	b4	b3	b2	b1	b0
	SSLKP	SSLA[2:0]			BRDV[1:0]		CPOL	CPHA
Value after reset:	0	0	0	0	1	1	0	1

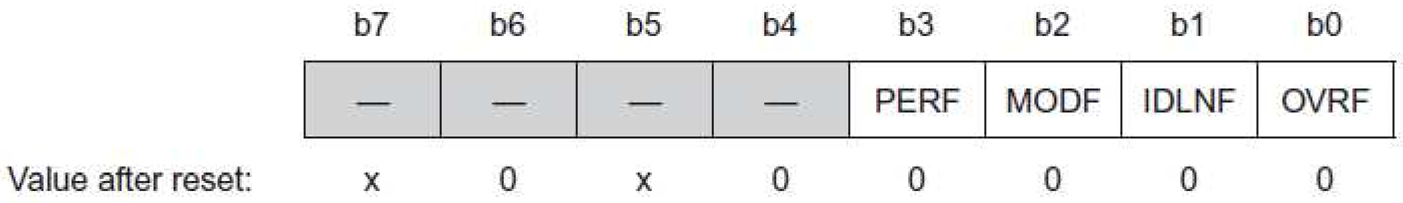
Bit	Symbol	Bit Name	Description	R/W
b0	CPHA	RSPCK Phase Setting	0: Data sampling on odd edge, data variation on even edge 1: Data variation on odd edge, data sampling on even edge	R/W
b1	CPOL	RSPCK Polarity Setting	0: RSPCK = 0 when idle 1: RSPCK = 1 when idle	R/W
b3, b2	BRDV[1:0]	Bit Rate Division Setting	b3 b2 0 0: These bits select the base bit rate 0 1: These bits select the base bit rate divided by 2 1 0: These bits select the base bit rate divided by 4 1 1: These bits select the base bit rate divided by 8	R/W
b6 to b4	SSLA[2:0]	SSL Signal Assertion Setting	b6 b5 b4 0 0 0: SSL0 0 0 1: SSL1 0 1 0: SSL2 0 1 1: SSL3 1 x x: — (Setting prohibited) [Legend] x: Don't care	R/W
b7	SSLKP	SSL Signal Level Keeping	0: Negates all SSL signals upon completion of transfer 1: Keeps the SSL signal level from the end of transfer until the beginning of the next access.	R/W

Serial Peripheral Command Register (SPCMDx)

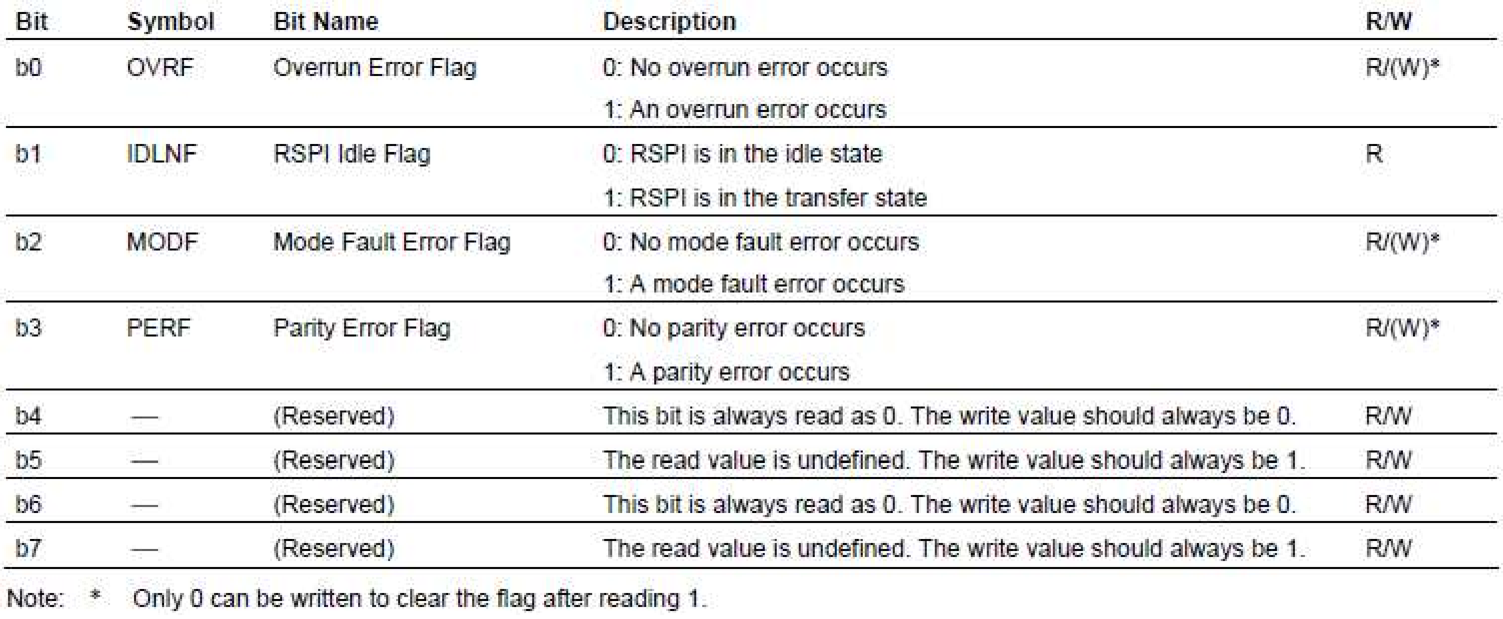


Bit	Symbol	Bit Name	Description	R/W
b11 to b8	SPB[3:0]	RSPI Data Length Setting	b11 b10 b9 b8 0100 to 0111: 8 bits 1 0 0 0: 9 bits 1 0 0 1: 10 bits 1 0 1 0: 11 bits 1 0 1 1: 12 bits 1 1 0 0: 13 bits 1 1 0 1: 14 bits 1 1 1 0: 15 bits 1 1 1 1: 16 bits 0 0 0 0: 20 bits 0 0 0 1: 24 bits 0010, 0011: 32 bits	R/W
b12	LSBF	RSPI LSB First	0: MSB first 1: LSB first	R/W
b13	SPNDEN	RSPI Next-Access Delay Enable	0: A next-access delay of 1 RSPCK + 2 PCLK 1: A next-access delay is equal to the setting of the RSPI next-access delay register (SPND)	R/W
b14	SLNDEN	SSL Negation Delay Setting Enable	0: An SSL negation delay of 1 RSPCK 1: An SSL negation delay is equal to the setting of the RSPI slave select negation delay register (SSLND)	R/W
b15	SCKDEN	RSPCK Delay Setting Enable	0: An RSPCK delay of 1 RSPCK 1: An RSPCK delay is equal to the setting of the RSPI clock delay register (SPCKD)	R/W

Serial Peripheral Status (SPS)



	b7	b6	b5	b4	b3	b2	b1	b0
	—	—	—	—	PERF	MODF	IDLNF	OVRF
Value after reset:	x	0	x	0	0	0	0	0

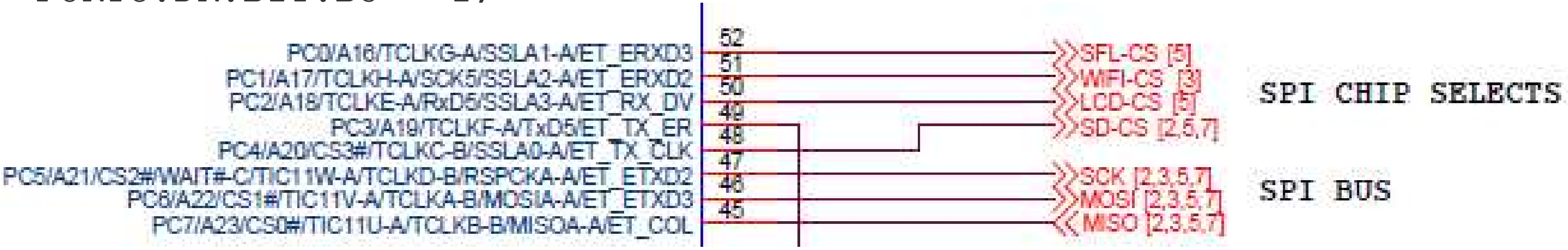


Bit	Symbol	Bit Name	Description	R/W
b0	OVRF	Overrun Error Flag	0: No overrun error occurs 1: An overrun error occurs	R/(W)*
b1	IDLNF	RSPI Idle Flag	0: RSPI is in the idle state 1: RSPI is in the transfer state	R
b2	MODF	Mode Fault Error Flag	0: No mode fault error occurs 1: A mode fault error occurs	R/(W)*
b3	PERF	Parity Error Flag	0: No parity error occurs 1: A parity error occurs	R/(W)*
b4	—	(Reserved)	This bit is always read as 0. The write value should always be 0.	R/W
b5	—	(Reserved)	The read value is undefined. The write value should always be 1.	R/W
b6	—	(Reserved)	This bit is always read as 0. The write value should always be 0.	R/W
b7	—	(Reserved)	The read value is undefined. The write value should always be 1.	R/W

Note: * Only 0 can be written to clear the flag after reading 1.

Code to set up SPI

```
void Init_RSPI(void){
  MSTP(RSPI0) = 0;
  IOPORT.PFGSPI.BIT.RSPIS = 0;
  PORT.PFGSPI.BIT.RSPCKE = 1;
  IOPORT.PFGSPI.BIT.SSL3E = 0;
  IOPORT.PFGSPI.BIT.MOSIE = 1;
  PORTC.DDR.BIT.B4 = 1;
  PORTC.DR.BIT.B4 = 1;
  PORTC.DDR.BIT.B7 = 1;
  PORTC.DR.BIT.B7 = 1;
  PORTC.DDR.BIT.B6 = 1;
  PORTC.DR.BIT.B6 = 1;
  PORTC.DDR.BIT.B5 = 1;
  PORTC.DR.BIT.B5 = 1;
  RSPI0.SPPCR.BYTE = 0x00;
  RSPI0.SPBR.BYTE = 0x00;
  RSPI0.SPDCR.BYTE = 0x00;
  RSPI0.SPCKD.BYTE = 0x00;
  RSPI0.SSLND.BYTE = 0x00;
  RSPI0.SPND.BYTE = 0x00;
  RSPI0.SPCR2.BYTE = 0x00;
  RSPI0.SPCMD0.WORD = 0x0700;
  RSPI0.SPCR.BYTE = 0x6B;
  RSPI0.SSLP.BYTE = 0x08;
  RSPI0.SPSCR.BYTE = 0x00;
}
```


PC0/A16/TCLKG-A/SSLA1-A/ET_ERXD3
PC1/A17/TCLKH-A/SCK5/SSLA2-A/ET_ERXD2
PC2/A18/TCLKE-A/RxD5/SSLA3-A/ET_RX_DV
PC3/A19/TCLKF-A/TxD5/ET_TX_ER
PC4/A20/CS3#/TCLKC-B/SSLA0-A/ET_TX_CLK
PC5/A21/CS2#/WAIT#-C/TIC11W-A/TCLKD-B/RSPCKA-A/ET_ETXD2
PC6/A22/CS1#/TIC11V-A/TCLKA-B/MOSIA-A/ET_ETXD3
PC7/A23/CS0#/TIC11U-A/TCLKB-B/MISOA-A/ET_COL
52
51
50
49
48
47
46
45
SFL-CS [5]
WIFI-CS [3]
LCD-CS [5]
SD-CS [2,5,7]
SCK [2,3,5,7]
MOSI [2,3,5,7]
MISO [2,3,5,7]
SPI CHIP SELECTS
SPI BUS

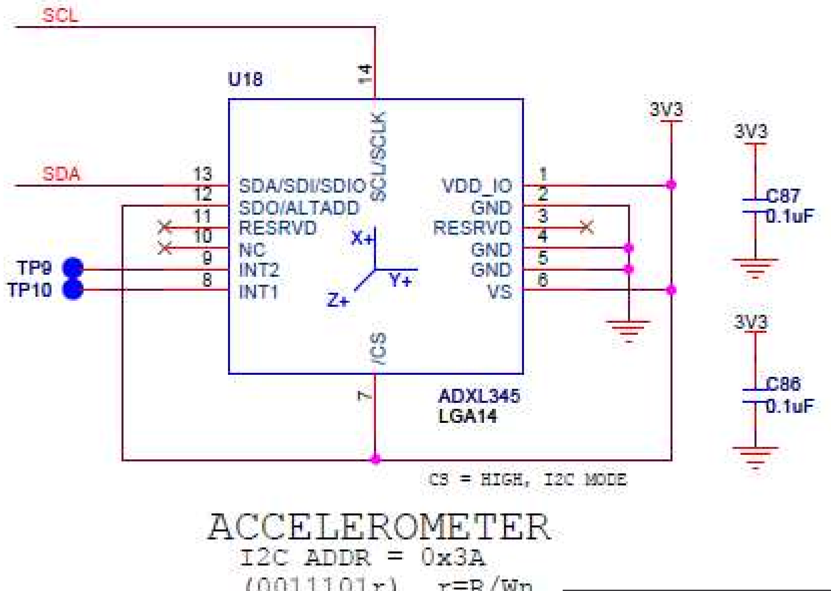
Code to communicate via SPI

```
void RSPI_Transmit_LWord(int16_t sLowWord, int16_t sHighWord){
    PORTC.DR.BIT.B4 = 0;
    while (RSPI0.SPSR.BIT.IDLNF);
    RSPI0.SPDR.WORD.L = sLowWord;
    RSPI0.SPDR.WORD.H = sHighWord;
    while (RSPI0.SPSR.BIT.IDLNF);
    (void)RSPI0.SPDR.WORD.L;
    (void)RSPI0.SPDR.WORD.H;
    PORTC.DR.BIT.B4 = 1 ;     //CS OFF
}
```

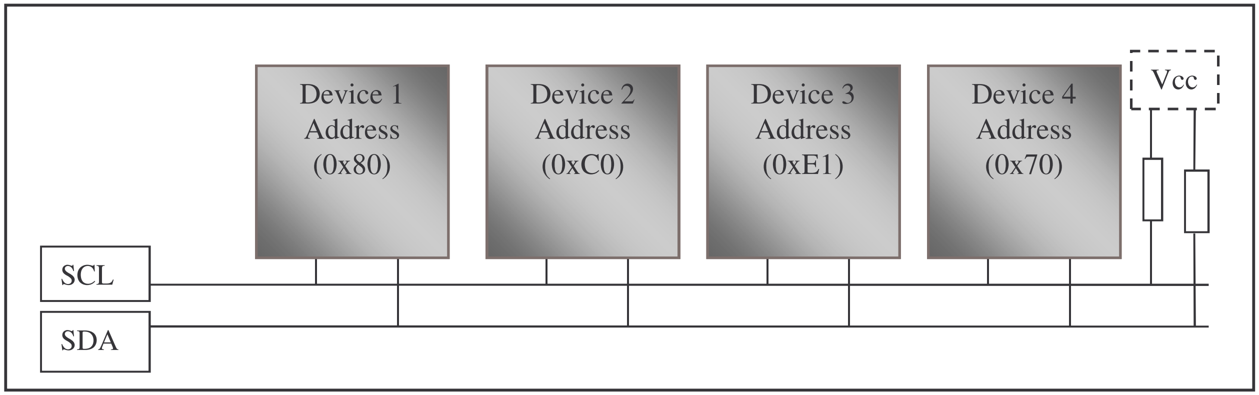
I2C

Inter-Integrated Circuit Bus

- A two line bus for communicating data at high speeds
- Multiple devices on the same bus with only one master controlling the bus
- Needs pull up resistors and is kept at a digital high level when idle



ACCELEROMETER
I2C ADDR = 0x3A
(0011101r), r=R/Wn



I2C: Properties

	Renesas Inter Integrated Chip Bus
Modes	I^{2}C mode or SM bus mode
Max Transfer Speed	Up to 1Mbps (most devices won't support speeds beyond 400kbps)
Max. number of devices/slaves connected per bus	128 (0 to $2^{7\text{ bits}}$-1)
Number of wires required for communication (not including ground)	2
Max. Length of wires[6]	10 meters @ 100kbps
Number of bits per unit transfer (excluding start and stop)	8
Slave Selection method	Through addressing

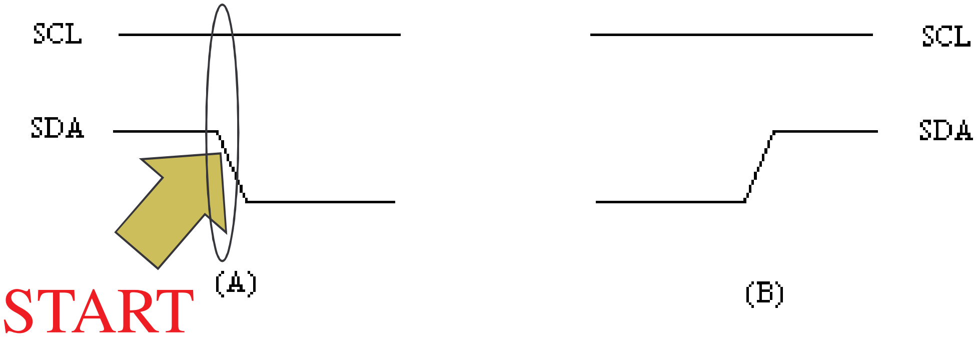
I2C: Working

Two wires:

- SCL (Serial Clock): Synchronizing data transfer on the data line
- SDA (Serial Data): Responsible for transferring data between devices
- Together they can toggle in a controlled fashion to indicated certain important conditions that determine the status of the bus and intentions of the devices on the bus.

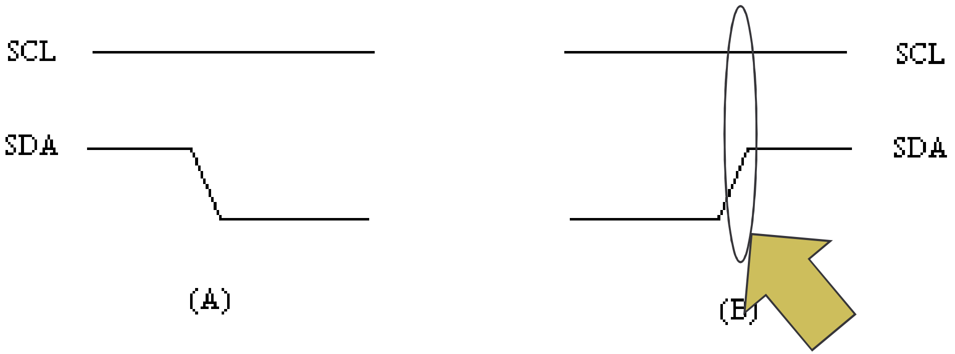
I2C: Working (Contd …) START CONDITION

- Before any form of data transfer takes place, a device wanting to transfer data must take control of the bus (Needs to monitor the bus).
- If the bus is held high, then it is free. A device may issue a START condition and take control of the bus.
- If a START condition is issued, no other device will transmit data on the bus (predetermined behavior for all devices).



I2C: Working (Contd …) STOP CONDITION

- When device is ready to give up control of the bus, it issues a STOP condition
- STOP condition is one in which the SDA line gets pulled high while the SCL line is high.
- Other conditions: RESTART (combination of a START and STOP signal)

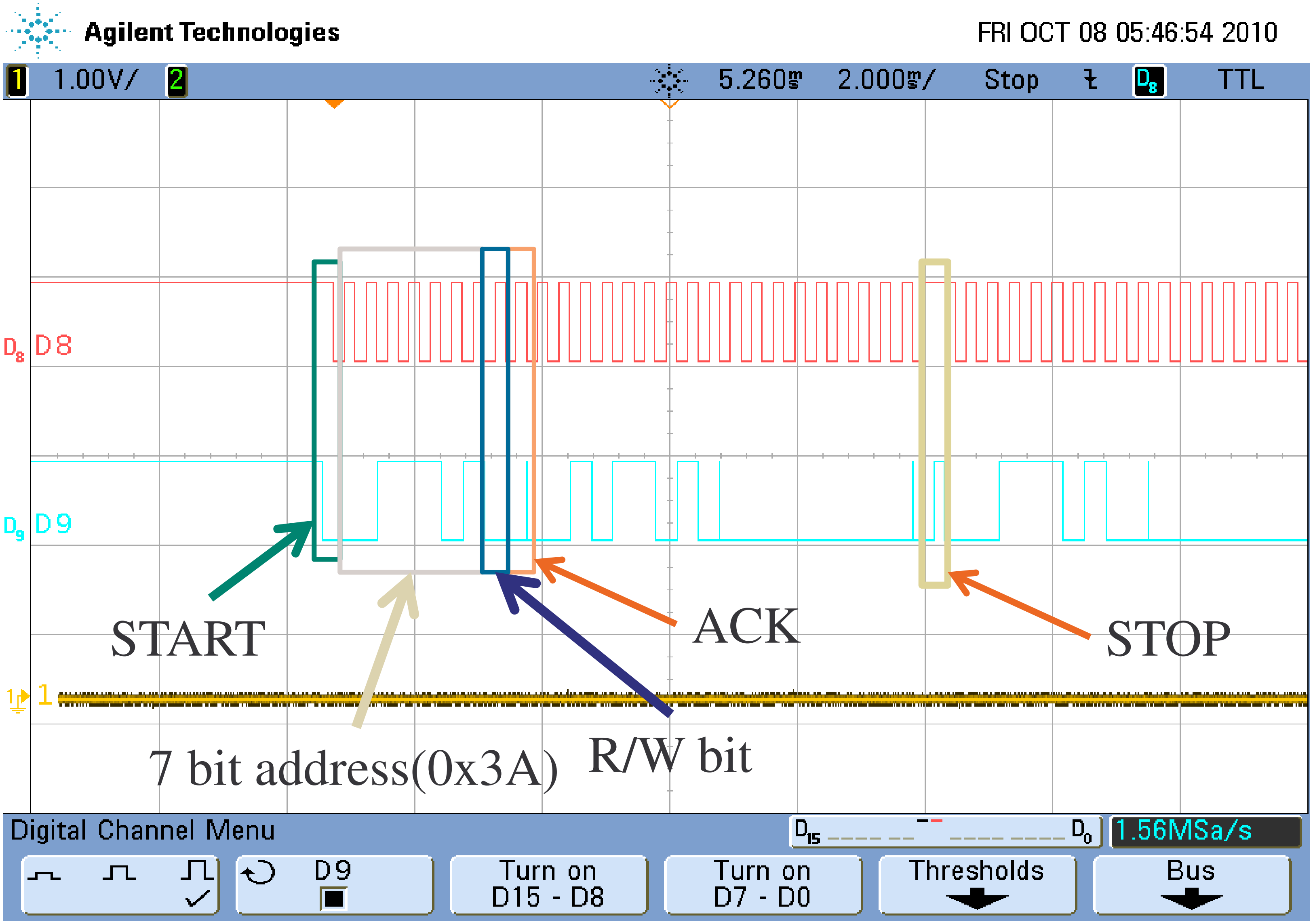


STOP

I2C: Working (Contd …) After START

- Address the slave device with one byte of data which consists of a 7 bit address + 1 bit (R/W)
- If this bit is low, it indicates that the master wants to write to the slave device; if high, the master device wishes to read from the slave. This determines whether the next transactions are going to be read from or written to the addressed slave devices.
- A ninth bit (clock) is transmitted with each byte of data transmitted (ACK(Logic 0)/NACK(logic 1) bit). The slave device must provide an ACK within the 9th cycle to acknowledge receipt of data

For Real???? Let’s have a look ..



I2C: Code: Simple example (START and STOP)

```
void RiicIni(unsigned char in_SelfAddr){
    SYSTEM.MSTPCRB.BIT.MSTPB21 = 0;
    RIIC0.ICCR1.BIT.ICE = 0;
    RIIC0.ICCR1.BIT.IICRST = 1;
    RIIC0.ICCR1.BIT.IICRST = 0;
    RIIC0.SARU0.BIT.FS = 0;
    RIIC0.SARL0.BYTE = in_SelfAddr;
    RIIC0.ICMR1.BIT.CKS = 7;
    RIIC0.ICBRH.BIT.BRH = 28;
    RIIC0.ICBRL.BIT.BRL = 29;
    RIIC0.ICMR3.BIT.ACKWP = 1;
    RIIC0.ICIER.BIT.RIE = 1;
    RIIC0.ICIER.BIT.TIE = 1;
    RIIC0.ICIER.BIT.TEIE = 0;
    RIIC0.ICIER.BIT.NAKIE = 1;
    RIIC0.ICIER.BIT.SPIE = 1;
    RIIC0.ICIER.BIT.STIE = 0;
    RIIC0.ICIER.BIT.ALIE = 0;
    RIIC0.ICIER.BIT.TMOIE = 0;
    PORT1.ICR.BIT.B3 = 1;
    PORT1.ICR.BIT.B2 = 1;
    RIIC0.ICCR1.BIT.ICE = 1;
}
```

```
void RiicUnIni(void){
    SYSTEM.MSTPCRB.BIT.MSTPB21 = 1;
}

unsigned char RiicSendStart(void){
    if(RIIC0.ICCR1.BIT.ICE){
     while(RIIC0.ICCR2.BIT.BBSY);
     RIIC0.ICCR2.BIT.ST=1;

    while(!(RIIC0.ICCR2.BIT.BBSY&&RIIC0.ICS
    R2.BIT.START));
    RIIC0.ICSR2.BIT.START=0;
    return 1;
    }
    else return 0;
}

unsigned char RiicSendStop(void){
    if(RIIC0.ICCR1.BIT.ICE){
          while(RIIC0.ICCR2.BIT.BBSY){
                    RIIC0.ICCR2.BIT.SP=1;
          }
    return 1;
    }
else return 0;
}
```

I2C Code: Reading and writing ….

```
unsigned char RiicReadByte(unsigned char
    slave_addr, unsigned char
    slave_register_num){

    RiicWriteByte(slave_addr,slave_register
    _num);
    RiicSendStop();
    RIIC0.ICSR2.BIT.STOP=0;
    RiicSendStart();
    while(!RIIC0.ICSR2.BIT.TDRE);
    RIIC0.ICDRT=slave_addr|(0x01);
    while(!RIIC0.ICSR2.BIT.RDRF);
    if(RIIC0.ICSR2.BIT.NACKF==0){
      RIIC0.ICMR3.BIT.WAIT=1;
      RIIC0.ICMR3.BIT.ACKBT=1;
      read_byte=RIIC0.ICDRR;
      while(!RIIC0.ICSR2.BIT.RDRF);
      RIIC0.ICSR2.BIT.STOP=0;
      RIIC0.ICCR2.BIT.SP=1;
      read_byte=RIIC0.ICDRR;
      while(!RIIC0.ICSR2.BIT.STOP);
      return read_byte;
      }
    else return 0xFF;
}
```

```
unsigned char RiicWriteByte(unsigned char
    slave_addr, unsigned char data_byte){

    RIIC0.ICDRT=slave_addr&(0xFE);
    while(!RIIC0.ICSR2.BIT.TDRE);
    RIIC0.ICDRT=data_byte;
    while(!RIIC0.ICSR2.BIT.TEND){
      if(RIIC0.ICSR2.BIT.NACKF){
      RIIC0.ICSR2.BIT.NACKF=0;
      return 0;
      }
    }
    while(!RIIC0.ICSR2.BIT.TDRE);
    while(!RIIC0.ICSR2.BIT.TEND) {
      if(RIIC0.ICSR2.BIT.NACKF){
      RIIC0.ICSR2.BIT.NACKF=0;
      return 0;
      }
    }
return 1;
}
```

I2C Code: the Glorious main()

```
void main(void){
    RiicIni(0x10);
    RiicSendStart();
    RiicWriteByte2(0x3A,0x2D,0x00);
    RiicSendStop();
    RiicSendStart();
    i=RiicReadByte(0x3A,0x00);
    RiicSendStop();
    RiicUnIni();
    while(1);
}
```