

CpE 213

Digital Systems Design

Lecture 27
Tuesday 12/2/2003



UNIVERSITY OF MISSOURI-ROLLA
The Name. The Degree. The Difference.

Announcements

- No afternoon office hours today.
- No class or office hours on Thursday Dec. 4.
- Regular office hours next week.
- Project report:
 - One report per group, due by 5 pm on Friday Dec. 5.
 - It should include (see project description):
 - Well-commented code.
 - Anything interesting that you observed while working on the project, in particular any bonus operations that you attempted, even if they didn't succeed.
 - Description of any hardware controls (knobs, etc).
 - You can email it to me, or turn it in to Katie Lortz.
- Don't forget the peer reviews.
 - You are welcome to submit them before the final.
 - Turn them in to me personally, or to Katie Lortz. NO EMAIL.

Outline

- Review
- Serial communication features of the 8051.
- Programming the 8051 for serial I/O expansion.

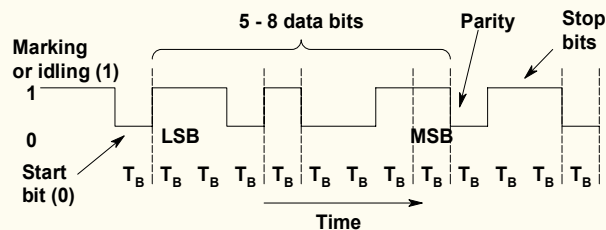
Some slides adapted from Mr. K. T. Ng

Review

Intro to Data Transfer

- Computers transfer data in two ways:
 - Parallel
 - hard disks, printers
 - data is sent one or more bytes at a time
 - 8 or more lines required
 - works well for short distances
 - Serial
 - mouse, PDA sync, keyboard
 - data is sent one bit at a time
 - only 1 line required
 - works well for long distances
 - good for wireless and remote access
- Serial transmission is preferred mostly for its low cost, ease of use and simplicity.

Serial Data Signal



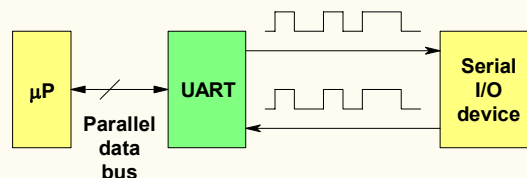
- **Baud rate** is the number of signal changes per second
$$\text{Baud rate} = \frac{1}{T_B} \text{ s}^{-1}$$
- Standard baud rates are: 110 (the old teletype or TTY rate), 300 (early PC modems), 1200, 2400, 9600, 14.4K, 19.2K, 28.8K, 38.4K, and 57.6K

Operations of Asynchronous Transmission

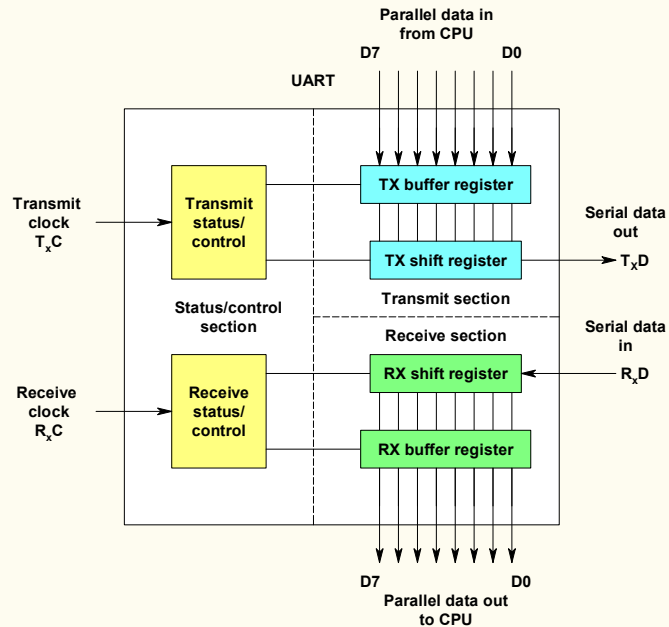
- The line is normally high.
- A START bit (a low bit) is transmitted to synchronize the receiver to the byte.
- The data bits are sent, LSB first.
- An optional PARITY bit is appended. It is used to check to see if any bit in the received byte was corrupted or not.
- Either 1 or 2 STOP bit(s) inform(s) the receiver of the end of the byte.
- This procedure happens for every byte.

Universal Asynchronous Receiver/Transmitter (UART)

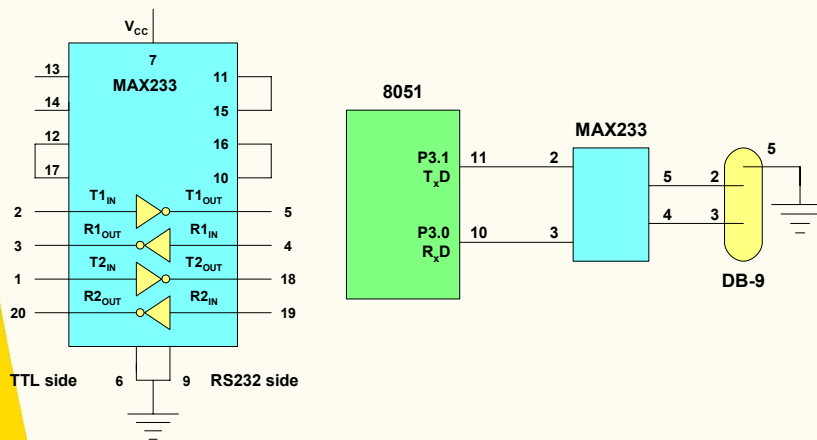
- Asynchronous serial data communication uses a special IC (**UART**) operating in asynchronous mode to serialize and de-serialize data.
- The UART also controls the bit rate, generates the start and stop bits, and provides various control functions.



Block Diagram of the UART



Interfacing MAX233 with the 8051



Serial Communication on the 8051 (Long Version of Lecture)

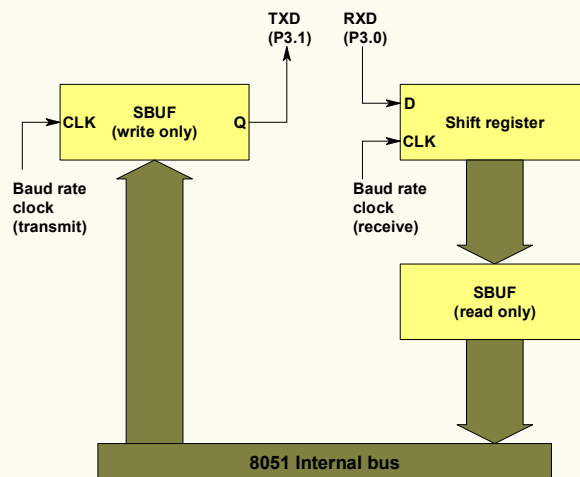
8051's Built-in UART

- One of the 8051's many powerful features is its built-in UART, otherwise known as a serial port.
- The serial port is full-duplex, so it can receive and transmit data simultaneously.
- The port is also buffered in receive mode, so it can receive a second data byte before the first data byte has been read from the register.
- SBUF: SFR to hold data at 99H.
 - This is physically two registers.
 - One is write-only and is used to hold data to be transmitted.
 - The other is read-only and is used to hold received data.

Control Registers

- The 8051 has two SFR registers that control the serial port
 - SCON: Serial Port Control Register at 98H, bit-addressable SFR. It is the serial port control and status register.
 - PCON: Power Control Register at 87H. It is the SFR used to control data rates. Not bit-addressable.

8051 Serial Port Block Diagram



SCON: Serial Control Register

SCON Address = 98H Reset Value = 0000 0000B
 Bit Addressable

Bit:	7	6	5	4	3	2	1	0
	SMO/FE	SM1	SM2	REN	TB8	RB8	T1	R1

Symbol	Function			
FE	Framing Error bit. This bit is set by the receiver when an invalid stop bit is detected. The FE bit is not cleared by valid frames but should be cleared by software. The SMOD0 bit must be set to enable access to the FE bit. SMOD0 is located at PCON6			
SM0	Serial Port Mode Bit 0 SMOD0 must = 0 to access bit SM0. SMOD0 is located at PCON6.			
SM1	Serial Port Mode Bit 1.			
SM0	SM1	Mode	Description	Baud Rate**
0	0	0	shift register	FOSC/12
0	1	1	8-bit UART	variable
1	0	2	9-bit UART	FOSC/64 or FOSC/32
1	1	3	9-bit UART	variable
**FOSC = oscillator frequency				

**FOSC = oscillator frequency

SCON: Serial Control Register (Cont.)

Symbol	Function
SM2	Enables the Automatic Address Recognition feature in Modes 2 or 3. If SM2 = 1 then R1 will not be set unless the received 9th data bit (RB8) is 1, indicating an address, and the received byte is a Given or Broadcast Address. In Mode 1, if SM2 = 1 then R1 will not be activated unless a valid stop bit was received, and the received byte is a Given or Broadcast Address. In Mode 0, SM2 should be 0.
REN	Enables serial reception. Set by software to enable reception. Clear by software to disable reception.
TB8	The 9th data bit that will be transmitted in Modes 2 and 3. Set or clear by software as desired.
RB8	In modes 2 and 3, the 9th data bit that was received. In Mode 1, if SM2 = 0, RB8 is the stop bit that was received. In Mode 0, RB8 is not used.
T1	Transmit interrupt flag. Set by hardware at the end of the 8th bit time in Mode 0, or at the beginning of the stop bit in the other modes, in any serial transmission. Must be cleared by software.
R1	Receive interrupt flag. Set by hardware at the end of the 8th bit time in Mode 0, or halfway through the stop bit time in the other modes, in any serial reception (except see SM2). Must be cleared by software.

PCON: Power Control Register

PCON Address = 87H Reset Value = 00XX 0000B
Not Bit Addressable

Bit:	7	6	5	4	3	2	1	0
	SMOD1	SMOD0	--	N/A	N/A	N/A	N/A	N/A

Symbol	Function
SMOD1	Double Baud rate bit. When set to a 1 and Timer 1 is used to generate baud rates, and the Serial Port is used in modes 1, 2, or 3.
SMOD0	When set, Read/Write accesses to SCON.7 are to the FE bit. When clear, Read/Write accesses to SCON.7 are to the SM0 bit.
--	Not implemented, reserved for future use.
N/A	Not Applicable. These bits are used for power control capability and have nothing to do with the serial port.

8051 Serial Interface Modes

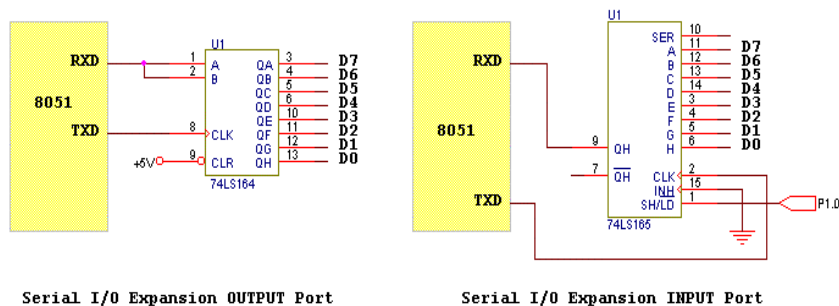
SM0	SM1	Mode	Description	Baud rate
0	0	0	Shift register	fosc/12
0	1	1	8-bit UART	Variable
1	0	2	9-bit UART	fosc/64 or fosc/32
1	1	3	9-bit UART	Variable

- For the above modes, the 8051 uses port 3, bits 1 and 0 as the serial TxD (output) and serial RxD (input) pins, respectively.

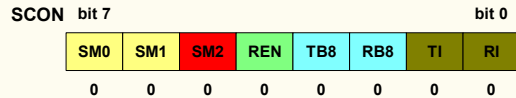
Serial Interface Mode 0 Shift Register

- Synchronous mode (half-duplex)
- Fixed baud rate ($f_{osc}/12$)
 - (Max. 1 MHz for 12MHz osc.)
- 8 data bits (LSB first)
- TxD pin used as shift clock
- RxD pin used for data
- This mode is **not intended for data communication**. It is mainly used for serial I/O expansion.
- Typical external shift register for output is 74LS164, and for input is 74LS165.

Mode 0 Interfacing



Mode 0 Program Example 1



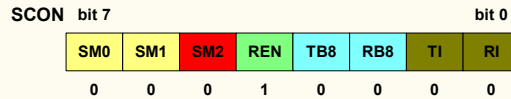
- The following program segment shows how to make use of mode 0 to shift the content of address 20H out to an external shift register (74LS164).

```
START:  MOV    SCON,#00H ;Set mode 0.
        MOV    R0,#20H  ;Set data address.
        MOV    A,@R0    ;Get the parallel output data.
        MOV    SBUF,A    ;Load SBUF for transmission.
WAIT:   JNB    TI,WAIT   ;If TI = 0, wait.
        ;Data byte not yet fully transmitted
        CLR    TI        ;TI must clear by software.
```

Cautions of Shift Register Data Transmission

- Data transmission begins when a byte is written to SBUF by the program, and lasts for eight clock pulses. Data is shifted out LSB first. TI is set after the last data bit has been clocked out by TxD.
- Software must check TI to ensure that the last byte has been transmitted before writing another byte to SBUF.
- The program can easily move bytes to SBUF faster than they can be transmitted.
- Shift register parallel data output is normally enabled after all eight serial bits have been shifted.
- Parallel output data should be latched into an external latch to stabilize the data.

Mode 0 Program Example 2



- The following program fragment shows how to make use of mode 0 to shift in the content of an external shift register (74LS165) and store the result in address 20H.

```
CLR    P1.0      ;Set 74LS165 in load mode, to load data
                ;from its parallel port first.
SETB   P1.0      ;Now set 74LS165 to shift mode.
MOV     SCON,#10H ;Set up mode 0 for data reception.
WAIT:  JNB     RI, WAIT ;Wait for RI to go high.
CLR     RI        ;RI must be cleared by software
MOV     20H, SBUF ;Save the shift reg. to address 20H
```

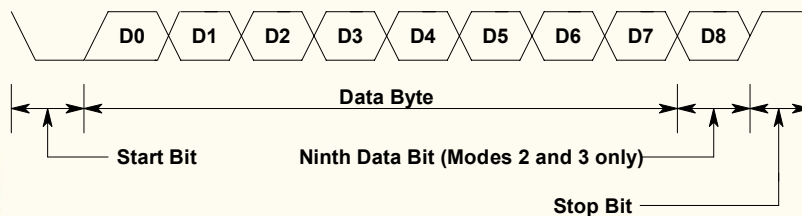
Cautions of Shift Register Data Reception

- Data reception is enabled by setting REN and clearing RI.
- RI is set when the MSB of the received data byte has been shifted into SBUF.
- The program must check to ensure RI is set before reading data from SBUF and then reset RI.
- Having to reset RI to enable further data reception is unique to mode 0. All other modes may receive data irrespective of the state of RI.

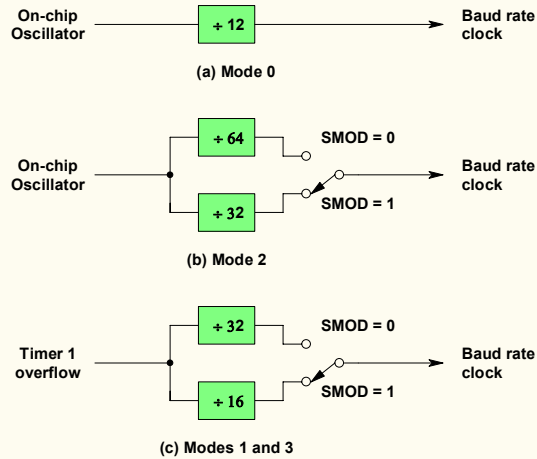
8051 Serial Port Operation Mode 1

- Asynchronous Mode (Full Duplex)
- Variable Baud Rate - can go up to 104.2 KHz (20 MHz osc.).
- 8 data bits (LSB first)
- 1 Start Bit (0); 1 Stop Bit (1)
- The data frame format is compatible with the COM port of IBM/compatible PCs.
 - Makes it the most useful mode.
- On receive, the stop bit goes into RB8 in special function register SCON.

Data Frame Format For Modes 1, 2 and 3



Serial Port Clocking Sources



SMOD is the control bit in PCON and can be 0 or 1.
When SMOD = 1, it doubles the baud rate.

Mode 1 Baud Rates

- Timer 1 is used to generate the baud rate for mode 1 by using the overflow flag of the timer to determine the baud frequency.
- When the timer overflows, it signals the processor to send/receive a data bit out/in to the serial port.
- Typically, timer 1 is configured in auto-reload mode (mode 2), and the baud rate is given as:

$$\text{Baud rate} = \frac{2^{\text{SMOD}}}{32} \times \frac{\text{oscillator frequency}}{12 \times [256 - \text{TH1}]}$$

- Very often, the baud rate is given, what we need to determine is the timer reload value.

$$\text{TH1} = 256 - \frac{2^{\text{SMOD}}}{32} \times \frac{\text{oscillator frequency}}{12 \times \text{Baud rate}}$$

Mode 1 Baud Rates (Cont.)

- The oscillator frequency is chosen to help generate both standard and nonstandard baud rates.
- That is why an oddball crystal frequency of 11.0592 MHz (instead of 12 MHz) is used so often for 8051 designs. This is to achieve exact clock rates for most standard baud rates, i.e., 9600, 19200.

Group Exercise

- Assume oscillator frequency = 11.0592 MHz.
Find the timer reload value TH1 for :
 - (a) baud rate = 19200 and SMOD = 1
0FDH
 - (b) baud rate = 1200 and SMOD = 0.
0E8H

Mode 1 Baud Rates (Cont.)

- If timer 1 is not run in timer mode 2, then the baud rate is:

$$\text{Baud rate} = \frac{2^{\text{SMOD}}}{32} \times (\text{Timer 1 overflow rate})$$

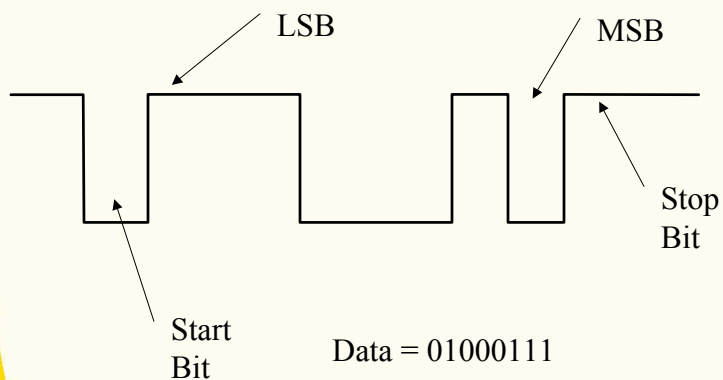
- The baud rate is determined by how frequently timer 1 overflows. The more frequently timer 1 overflows, the higher the baud rate.
- The timer 1 interrupt should be disabled in this application.
- The timer itself can be configured for either “timer” or “counter” operation, in any of its 3 running modes (mode 0, 1 and 2).

Short and Sweet

UART

- UART = Universal Asynchronous Receiver Transmitter
- TX, RX pins are TTL levels. Need level shifter for RS-232 $\pm 12\text{v}$ levels
- Uses timer 1 (or 2) for baud rate generation
- Mode 1 is 'normal' mode. 3 others avail
 - Shift register mode: data plus clock (mode 0)
 - Two nine bit 'multiprocessor' modes (modes 2 and 3)

UART Waveform



UART Timing

- Start bit marks beginning of a character
- Bit sync starts in middle of start bit
- RX and TX agree on timing
- Bits per second not same as Baud!
- Baud is signaling speed, not bits per sec
- Bit (baud) rates are standard:
110, 300, 1.2k, 2.4k, 9.6k, 14.4k, 28.8k,
etc

UART Timing

Reload value for T1 (TH1)

$$TH1 = 256 - (K * f_{clk}) / (12 * 32 * \text{baud})$$

K = SMOD + 1 (SMOD is PCON.7)

Example:

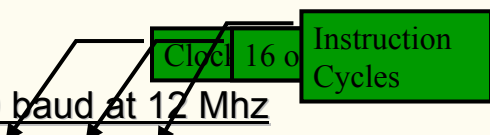
9600 bits per second, $f_{clk} = 11.059 \text{ Mhz}$

$$\begin{aligned} TH1 &= -11.059 \text{E}6 / (384 * 9600) = -3 \\ &= 0xFD \end{aligned}$$

But why 11.059 Mhz???

- Baud rate = Clk Freq / (16 * 12 * CtrOvf)
- CtrOvf must be a positive integer < 256
- Sample at 16x (or 32x) bit rate
- Timer 1 counts at 1/12 clock frequency
- Example for 9600 Baud:
 $9600 * 16 * 12 * \text{CtrOvf} = \text{ClkFreq}$
 $1843200 * 6 = 11059200 = 11.059 \text{ Mhz}$

More baud rate calculation



CtrOvf for 9600 baud at 12 Mhz
 $\text{BaudRate} = 12\text{E}6 / (16 * 12 * \text{CtrOvf}) \quad \text{smod}=1$
 $\text{CtrOvr} = 12\text{E}6 / (16 * 12 * \text{BaudRate})$
 $1\text{E}6 / (16 * 9600)$
 $1\text{E}6 / 153600 = 6.5 \text{ (must use 6 or 7)}$
 $\text{BaudRate} = 10,417 \text{ for CtrOvf}=6 \text{ (err}=8\%)$
 $= 8,929 \text{ for CtrOvf}=7 \text{ (err}=7\%)$

UART SFR

7		SCON				0	
SM0	SM1	SM2	REN	TB8	RB8	TI	RI

- Mode 00 1Mhz shift register mode.Clock (TX)+Data(RX)
- Mode 01 Normal UART mode
- Mode 10,11 9 bit multiprocessor mode
- SM2 0-normal, 1-RI set when RB8=1 (multiprocessor mode)
- REN 0-disable input, 1-enable input
- TB8, RB8 - 9th data bit in Modes 2,3
- TI, RI - send/receive status bits

Initialize UART

```
TMOD |= 0x20; //Timer 1 mode 2
TH1 = -24;    //1200 baud @ 11.059 Mhz
TR1 = 1;      //start timer
SCON = 0x50; //SM1+REN
```

Output a character

```
char c;
```

```
SBUF= c; //write char to SBUF  
while(!TI); //wait until done
```

Receive a character

```
char c;
```

```
while(!RI); //wait for done  
RI=0;  
c= SBUF;
```

A more complete example

- See family hardware guide PDF for details of UART
- See Puchar.c in Keil/C51/lib
- See Getkey.c in Keil/C51/lib
- Neither one initializes UART so be careful when using printf etc.

The main UART SFRs

- SBUF – 0x99h, not bit addressable
- SCON – 0x98h, bit addressable
- PCON – 0x87h, not bit addressable.

SCON

7							0
SM0	SM1	SM2	REN	TB8	RB8	TI	RI

- SM0 & SM1 – Serial port mode bits
- SM2 – Serial port mod bit 2, Enables multiprocessor communications in mode 2 and 3. R1 is set to 1 if RB8 is 1.
- REN – Receive Enable, must be set to receive characters in SBUF

SCON, cont

7							0
SM0	SM1	SM2	REN	TB8	RB8	TI	RI

- TB8 – Transmit bit 8, is the 9th bit transferred in modes 2 and 3. Set and cleared by software.
- RB8 – Receive bit 8. 9th bit received in modes 2 and 3.
- TI – Transmit interrupt flag. Set at the end of character transmission, cleared by software.
- RI – Receive interrupt flag. Set at the end of character reception, cleared by software.

SBUF

- 8 bit special function register
- Address is 0x99h
- Receives characters into this register.
- Transmits the stored character, LSB first.

PCON

- Bit 7, SMOD: Double-baud rate bit; when set, baud rate is doubled in serial port modes 1, 2, or 3.
- Bit 3 & 2, GF1 & GF0: general purpose flag
- Bit 1, PD: Power down, set to activate, only exit is reset.
- Bit 0, IDL: idle mode, set to activate, only exit is a reset or interrupt.

Serial Modes

- Mode 0 : 8-Bit Shift Register
- Mode 1 : UART with Variable Baud Rate
- Mode 2 : 9-Bit with Fixed Baud Rate
- Mode 3 : 9-Bit with Variable Baud Rate

CpE 213

Example ISM109 - Serial Port Operation

Purpose

This handout is intended to supplement the material in Chapter 5 of ISM on serial port operation for the 8051. A complete hardware interface for an RS232 communication port is shown and programming examples in C are provided.

RS-232 Hardware interface

If we want our 8051 microcontroller system to communicate with another device, there are a number of options available: a) interface the device as a *memory mapped* peripheral using the 8051 in the expanded mode, b) use a parallel bus comprised of a number of parallel port bits (8 is a common number) together with a clock bit and perhaps a direction bit for two-way communication, or c) use one of several bit serial communication protocols. If we want to send data over an appreciable distance (more than a few feet), the last option is usually the only practical approach. There are many bit serial standards in common use, including I²C, SPI, J1850, CAN bus, RS-232, USB, and Ethernet to name a few. The first two are common microcontroller buses, the next two are common automotive network buses, and the last two are commonly used in desktop systems. RS-232 is actually the electrical and physical standard that is commonly used for low speed asynchronous communication used on dumb terminals and telephone modems.

A simple shift register can be used as the basis for a simple serial communication scheme. For example, if `c` is defined as an unsigned char, and `tx` is defined as P1.0, the following C statement can be used to shift each bit out on P1.0:

```
for (i=0;i<8;i++) {tx=c&1; c=c>>1}; (L1)
```

Bit synchronization is a problem in that it is difficult to tell where one bit stops and another one starts. In fact it is even worse; if we put this statement in a loop that sets `c` equal to each character in a string or array of bytes, it is difficult to tell where one character stops and the next one starts. It is a way to start though, and it is simple at least in principle to shift out any number of bits this way and to start at either the LSB like we do here or start at the MSB end. If we modify the statement above just slightly, we can take care of the bit synchronization problem by supplying a clock. In the code below, `sclk` is defined to be P1.1:

```
for (i=0;i<8;i++) {tx=c&1; sclk=1; sclk=0; c=c>>1}; (L2)
```

As simple as it seems, this is the basis for quite a few simple bit serial buses commonly used to expand microcontrollers, such as I²C, and SPI. In order to be complete, the receiving end will need to know the size of the data we are shifting out and which end comes first. That is where standards come in handy.

For point to point communication over short distances of a few feet, this simple scheme will work fine. More elaborate configurations such as multiple receivers (party line), multiple transmitters (who gets to talk first?), or communication over long distances, more elaborate schemes are needed. This handout focuses on point to point communication using *asynchronous serial* communication with the RS-232 interface standard.

The problem with the *synchronous serial* approach given above is that it requires an extra line for the clock (`sclk`). Self clocking schemes such as representing a '0' by a square pulse with duty cycle much less than 50% and a duty cycle of much greater than 50% for a '1' avoid the extra clock line but aren't commonly used for terminal communication. They are used however for things like TV remote controls.

Asynchronous communication avoids use of a clock to achieve bit synchronization by framing each character with a '0' at the beginning and a '1' at the end. Each character thus starts with a guaranteed '1' to '0' transition that the receiver can count on. If the transmitter and receiver agree on a bit rate, the receiver knows how long each bit lasts, including the initial '0' bit. The receiver simply waits for a '1' to '0' transition, waits for 1/2 bit time, then proceeds to delay 1 bit time, sample the incoming bit, and repeat the last two steps *N* times, where *N* is the number of bits to be received. *N* is usually seven or eight. The following code shows the main idea for an asynchronous transmitter:

```

tx=0; delay_1bit();
for (i=0;i<8;i++){
    tx=c&1;
    c=c>>1;
    delay_1bit();
};
tx=1; delay_1bit();

```

(L3)

This shifts each bit out one at a time as before but now includes a prefix of '0' and a suffix of '1', with bit timing provided by the function 'delay_1bit'. For a data rate of 2400 bits per second, delay_1bit() would need to provide a delay of about 417 μ s.

An asynchronous receiver for the other end of our simple communication scheme could use the following code:

```

while (rx); //wait for rx==0
delay_half_bit(); //wait until midbit
if (rx!=0) error(); //should still be zero here
for (i=0;i<8;i++){
    delay_1bit(); //wait for midbit of next bit
    c=c>>1;
    c |= (rx<<7);
}
delay_1bit();
if (rx!=1) error(); //should be 1 by now

```

(L4)

The first two lines wait until the middle of the first '0' bit. The 3rd line provides some optional error checking in case of spurious '1' to '0' transitions caused by noise. Lines 4 through 7 wait until the middle of subsequent bits and shift each bit in from left to right. Like the transmitter above, the first bit is assumed to be the least significant bit of an eight bit character. After the 8th bit is shifted in, the receiver again delays one bit time which should put it right in the middle of the *stop bit*. This is the final framing bit and should be a '1' if all is well. If not, the last line again provides a little optional error control. Note that the receiver and transmitter can get hopelessly out of synchronization if, for example, the transmitter begins sending a long string of characters and the receiver comes on line in the middle of the string.

If we don't care about standards and merely want to hook two microcontrollers together over a short distances, all we need is a microcontroller with transmitter code at one end, a micro with receiver code at the other, a signal wire to hook the tx and rx ports together and a return path (ground). This one way communication link is called a *simplex* connection. If we include both transmit and receive code in each microcontroller and use the same port for both tx and rx, agree on which micro transmits first, and a protocol for switching directions, we can use our signal wire for two way communication. This is called a *half duplex* connection. Provide a separate tx and rx port at both ends, and include two signal wires and you have a *full duplex* connection.

If you do care about standards and want to communicate with (say) a PC using COM1 or connect your micro to a standard telephone modem, you will need to convert the nominal TTL voltage levels available at the 8051's port pins to EIA/TIA RS-232 standard voltage levels. Some devices use the presence or absence of a 20 ma current flow in the signal wire to indicate a '1' or a '0' or RS-422 (RS-485) which uses a voltage difference across a pair of wires for signalling but these are not commonly used in office equipment.

Maxim (www.maxim-ic.com) supplies a wide range of *transcievers* that can be used to convert between TTL levels and RS-232 or other levels. The awkward thing about RS-232 is that it uses a negative voltage to represent a '1' and a positive voltage to represent a '0'. The specification allows a range of voltages from about ± 5 volts to about ± 15 volts with ± 12 volts being typical. This is a problem since microcontroller supply voltage is typically 5 volts with lower voltages not uncommon in low power applications. Fortunately most modern RS-232 transceivers include an on-chip dc-to-dc converter that derives the bipolar supply from the microcontroller supply voltage. Figure 1 shows a typical 8051 configured for use with a standard RS-232 port. C1 through C4 are part of the Max 221's dc-to-dc converter used to derive the bipolar voltages specified by RS-232. Maxim has a large number of RS-232 driver and receiver chips including some that don't require external capacitors.

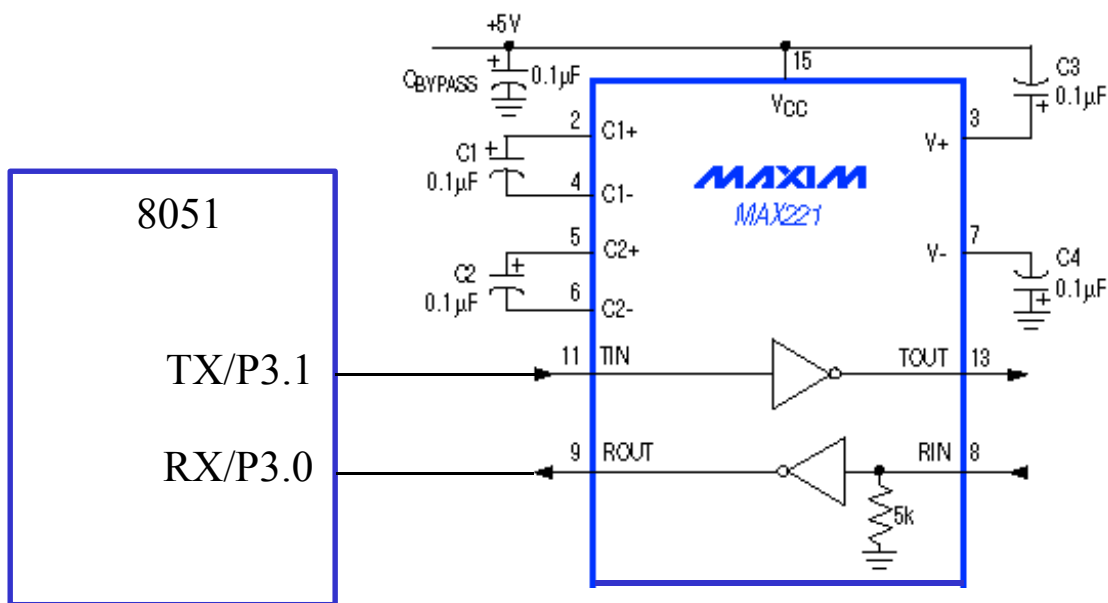


Figure 1 Maxim RS-232 Driver and receiver added to 8051 UART port

Like most digital applications, we can trade off hardware for software and vice versa. The rather slow software shift register examined earlier can be replaced by a hardware shift register that can be clocked at Ghz rates. The software *bit-banger* transmit and receive code can be replaced by a *Universal Asynchronous Receiver Transmitter* or UART (pronounced you-art). Most 8051 variants include at least one UART that uses P3.1 and P3.0 for tx and rx respectively. Timing for the bit delays is derived from one of the 8051's internal timer/counters.

Notice that so far no mention has been made of the term *baud rate*. Although it has become accepted usage, the term baud rate is frequently misused as a synonym for bit rate. The two terms can be quite different. Baud rate is the signalling rate or the rate at which signalling changes are made. Bit rate is simply the number of bits per second. If each change of voltage level represents a bit then the two terms are the same. If, on the other hand, each signal consists of one of four voltage levels, then two bits are sent for every signalling change. 9600 bits per second can be accomplished with only 4800 baud! Of course other parameters can be used besides voltage levels for signalling. Frequency and phase shift are two commonly used parameters for signalling over a communication channel.

Serial port software

If you use the 8051's serial port in an application, you will most likely use it in the 8-bit variable baud rate mode with an RS-232 driver and receiver. This section will show how to initialize the UART with a C program. It will also examine the putchar and getchar routines from the standard C51 library. These are simple non-interrupt driven routines that are called by printf and other standard C library I/O functions. The 8051 family hardware manual (available online as a PDF file) has a section on the standard serial interface that covers the details of using the standard 8051 serial port.

The listing below is a small C program that contains a function called *initser()* to initialize the serial port to mode 1 at 9600 baud. The main program is the familiar 'hello world' program from Kernighan and Ritchie with a call to *getkey()* added.

```

1      /* Use serial port with C51 */
2      #include <reg51.h>
3      #include <stdio.h>
4      #define SMOD 0x80
5
6      /*
7      Initialize the serial port to 9600 baud.
8      TI must be set to 1 so that the transmitter
9      doesn't look busy. This is so putchar will work.
10     */

```

```

11         void initser(void){
12             PCON|= SMOD; //set SMOD bit in PCON
13             TMOD|= 0x20; //T1 in auto-reload mode 2
14             TH0= TH1= -6; //this could also be -3 w/ smod=0
15             TR1= 1;      //turn on timer 1
16             TI= 1;       //set transmitter 'done' flag
17             REN= 1;      //set receive enable bit
18         }
19         void main(void){
20             initser(); //init serial port
21             do printf("Hello World.\n");
22                 while (getkey() != 'q');
23             while(1);
24         }

```

Line 3 includes the stdio.h header file, which includes prototypes for printf and getkey. Getkey is a low level routine that tests RI and when set, returns the character in SBUF. Printf is the usual C library formatted print function. It calls a low level function called putchar that tests TI and writes a character to SBUF when TI==1. Each of these are examined below.

Initser starts on line 11. Unlike 'normal' desktop C environments that depend on an operating system for I/O resources, programs generated by C51 must be 'stand alone'. That means the programmer (you) must make arrangements for sending and receiving characters to and from standard input and output (stdin and stdout). The default putchar and getkey routines that do the low level I/O use the 8051's built in serial port but make no assumptions regarding bit rates or modes. The purpose of initser is to put the serial port into mode 1 with a bit rate of 9600 bits per second. Line 12 sets the SMOD bit in PCON. Since PCON isn't bit addressable, line 12 uses an OR assignment operator to set the SMOD bit and leave the others alone. Rather than use a naked constant, the SMOD symbol is defined as 0x80 in line 4. Line 13 performs a similar service for TMOD and sets timer 1 to the eight bit auto reload mode 2. The reload value of -6 is placed into TH1 in line 14 in order to produce a bit rate of 9600. Timer 1 is started in line 15. Lines 16 and 17 complete the initialization by setting the TI bit and enabling the receiver. REN must be set in order for the RI status bit to be set when an incoming character is received. Technically the TI bit is a status bit for the transmitter and should not be set by software. If interrupts were being used, this would not be necessary here. The default putchar is written to test TI first and then if TI==1 the next character is written to SBUF to start the transmitter. This way the processor can do useful work while the character is being shifted out by the transmitter. If TI was tested after writing to SBUF to wait until transmission is complete before returning, initser would not need to set TI but there would be no overlap of output and processing. Main simply calls initser, outputs 'hello world' and waits for a key to be pressed. If you run this program in dscope, you will need to open the serial1 window, make it the active window, and type a character on the keyboard in order for 'hello world' to be written a second time. Main falls through to the while(1) loop if 'q' is typed.

This module uses 39 bytes of code. The executable, which includes the stdio routines from the C library, takes about 1120 bytes. As you can see, the stdio routines are quite large. In most cases, it would probably be worth while to write more specialized I/O routines that are tailored to the application.

Putchar

```

/*****
/* This file is part of the C51 Compiler package */
/* Copyright KEIL ELEKTRONIK GmbH 1990 */
/*****
/*
/* PUTCHAR.C: This routine is the general character output of C51. */
/*
/* To translate this file use C51 with the following invocation: */
/*
/* C51 PUTCHAR.C <memory model> */
/*
/* To link the modified PUTCHAR.OBJ file to your application use the */
/* following L51 invocation: */
/*

```

```

/*      L51 <your object file list>, PUTCHAR.OBJ <controls>          */
/*                                                                    */
/*****                                                                    */

#include <reg51.h>

#define XON  0x11
#define XOFF 0x13

char putchar (char c)  {

    if (c == '\n')  {
        if (RI)  {
            if (SBUF == XOFF)  {
                do  {
                    RI = 0;
                    while (!RI);
                }
                while (SBUF != XON);
                RI = 0;
            }
        }
        while (!TI);
        TI = 0;
        SBUF = 0x0d;                      /* output CR */
    }
    if (RI)  {
        if (SBUF == XOFF)  {
            do  {
                RI = 0;
                while (!RI);
            }
            while (SBUF != XON);
            RI = 0;
        }
    }
    while (!TI);
    TI = 0;
    return (SBUF = c);
}

```

The listing above is taken from the C51 library directory. Source code for several of the library functions that might potentially need to be modified are included there. The last three lines do the actual work. Notice that if TI wasn't set by `initser`, `putchar` would hang in the `while (!TI)` loop. The rest of `putchar` implements *flow control* by testing to see if an XOFF character was received. If so, then transmission halts until an XON character is received. XOFF and XON are Ctrl-S and Ctrl-Q respectively. This kind of flow control is mainly useful when a human is part of the communication system. If `putchar` is being used to communicate with another (unattended) computer it is less useful and more powerful flow control schemes are recommended. The other function performed by `putchar` is to translate a newline character (`'\n'`) to a newline, carriage return sequence. This comes from the old typewriter days where you not only had to return the carriage to the left hand margin, you also needed to advance the paper in order to begin a new line. Newline is also known as line feed.

Getkey

```

/*****
/*  This file is part of the C51 Compiler package          */
/*  Copyright KEIL ELEKTRONIK GmbH 1993                  */
/*****
/*
/*  GETKEY.C:  This routine is the general character input of C51.
/*
/*  To translate this file use C51 with the following invocation:
/*

```

```

/*                                                    */
/*      C51 GETKEY.C  <memory model>                  */
/*                                                    */
/*  To link the modified GETKEY.OBJ file to your application use the */
/*  following L51 invocation:                               */
/*                                                    */
/*      L51 <your object file list>, GETKEY.OBJ <controls>      */
/*                                                    */
/*****/

#include <reg51.h>

char _getkey ()  {
    char c;

    while (!RI);
    c = SBUF;
    RI = 0;
    return (c);
}

```

Getkey is much simpler than putchar and simply returns the contents of SBUF when RI is set. Getkey isn't normally called from a user program. Getchar is the library function normally called from a user C program.

Summary

This example has shown:

- The basis for simple bit serial input/output in C
- Extra hardware required to interface to an RS-232 device
- How to set up the serial port in C
- How the low level serial input/output routines in C work.

Questions

Answer the following questions.

1. The default I/O routines are fine if the 8051 variant you are using has a serial port. Not all do. For example, the 87C752, which is a low cost, low pin count variant, has no serial port. Briefly describe what would need to be done in order to make the 'hello world' program given here work with an 87C752.
2. Rewrite putchar to write the 7 bits of an ascii character to P1 bits 0 through 6 and generate a short active high pulse on P1.7 to act as a clock pulse for whatever is attached to P1.
3. If line 3 was deleted from the 'hello world' program, what would need to be added in order to make the program work? (Hint: examine the file stdio.h)
4. Sketch a timing diagram of P1.0 for listing L1 (the shift register code) if a 12 Mhz clock is used and the character 'c' contains an ASCII 'Z' code.
5. What happens if getkey is replaced with a call to getchar? (Hint: try it and examine the code in the debugger's disassembly window.