Expandable Serial Port Multiplexer

Abstract

The current trend in Personal Computers is to move away from standard RS232 serial ports in favor of USB or other interfaces. This is an acceptable solution for the usual I/O products such as mouse, printer, or modem, but a whole field of products that have embraced RS232 has been left out in the cold. Many test equipment and industrial equipment companies manufacture products that communicate using RS232. In my own work with some of these companies I have faced the challenge of trying to interface with devices that require multiple RS232 ports. For example, my current company produces laboratory instruments which, in their full configurations, can require up to four RS232 ports.

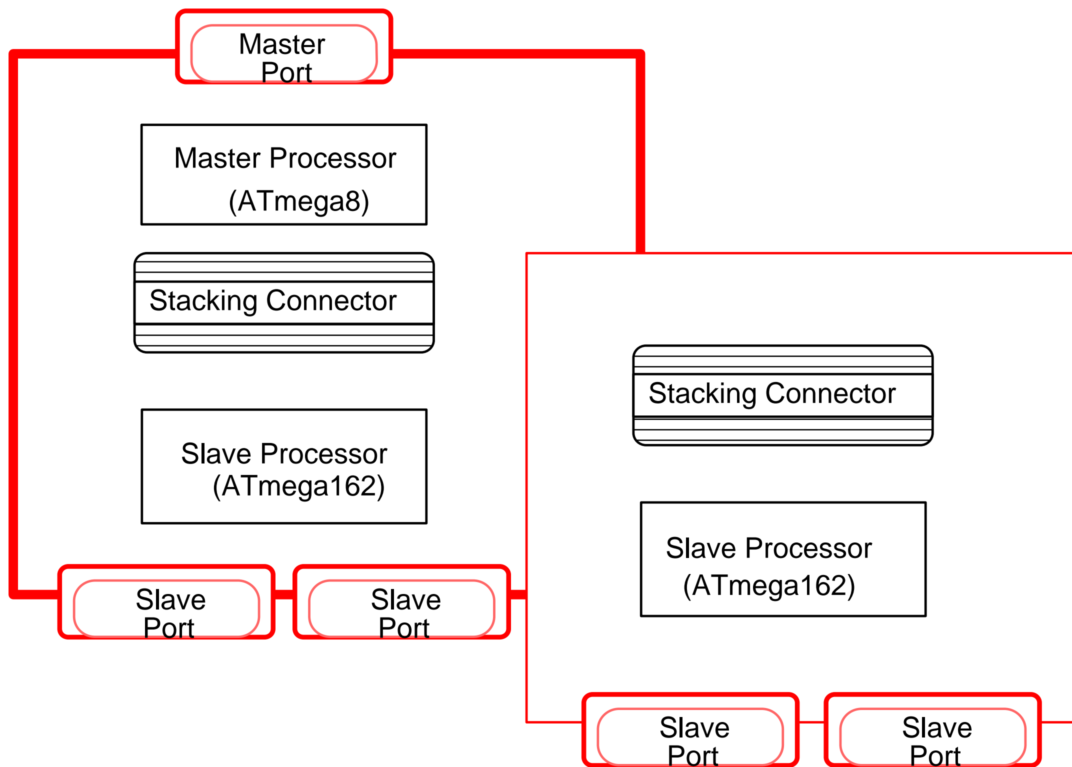
On a desktop PC you can always add more serial ports through add-on boards (some costing $100 or more). On a Laptop there are no internal slots, so you need to use PCMCIA or USB to support multiple RS232 ports. In my experience this can be a hit or miss situation. Missing or out-of-date drivers, driver/resource conflicts, or ports that just don't seem to function as expected, can turn an apparently easy task into a complete nightmare.

This project is a cost-effective solution that provides connection flexibility without the nightmare. The Expandable Serial Port Multiplexer allows a single RS232 serial port to service multiple instruments requiring RS232 ports. Each port is configurable in terms of baud rate, parity, number of bits and number of stop. On the host side the serial streams could be multiplexed through the use of a Windows driver or a special DLL program.

The Multiplexer uses a master processor (ATmega8) connected to a PC through a serial port. Multiple slaves (ATmega162) connect to master, via a parallel bus, to provide two serial ports each. Multiplexed commands sent to the master are routed to the appropriate slave. Likewise slave-received signals are routed back, through the master, to the PC in a multiplexed stream.

Thanks to the self-programming abilities of the ATMEL processors, it would be realistic to download routines to the master and/or slave processor to handle tasks like low level protocol translation. This is something that has been on my wish list for sometime now. I have worked with some ancient protocols that communicate in byte-at-a-time mode, requiring the target to acknowledge each byte send, before the next can be transmitted. Communication of this nature might work well with embedded processors on both sides, but hosted on Windows, in a multitasking environment, a 9600 baud line has the throughput equivalent of about 50 baud. With low level translation, the complete message could be sent to the slave port, and it could handle the translation at speeds approaching 4800 baud (it is still necessary to round trip each byte).

In addition to the slave serial port cards, other devices could be built and added to the stack to provide other services of interest to system integrators. Both A/D and D/A functional blocks could be added (based on the same ATMEL parts), to provide analog processing.



Master/Slave Communications

The master and slave processors communicate with each other through a bi-directional, 8-bit parallel bus. The bus extends from master to slave on the main board and through the stacking connector to other slave modules. The bus is made up of the following signals:

Name	Direction	Notes

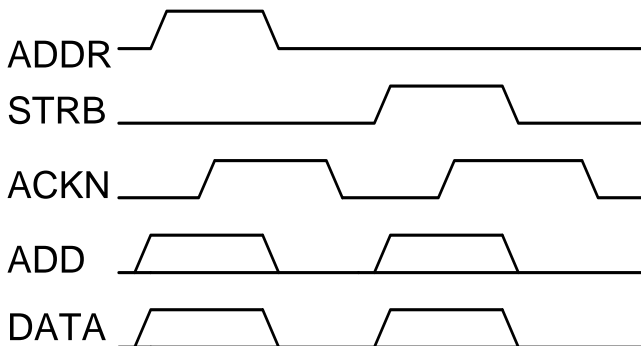
ADDR	Master->Slave	Asserted (high) by Master to Address module.
ACKN	Slave->Master	Raised by slave to Acknowledge ADDR or STRB (tri-stated by slave when not addressed).
STRB	Master->Slave	Asserted by (high) by the Master to continue a conversation started by ADDR.
ADD1-3	Master<->Slave	These lines contain module address when ADDR is high or the high bits of the DATA bus when STRB is high.
DATA1-5	Master<->Slave	These lines contain the command word when ADDR is high or the low bits of the DATA bus when STRB is high.

The getStatus routine demonstrates the complete read cycle between master and slave, by first calling sendCmnd to initiate the conversation, and then getData to read the data and complete the conversation. The sendCmnd routine sets the ADD lines (Addr), DATA lins (Cmnd) and the ADDR bit onto the data bus. The routine then waits until the addressed slave module signals ready by asserting the ACKN signal. Upon receipt of the ACKN, the master clears ADDR (and the ADD lines).

The Slave responds to the ADDR line going high by entering a pin change interrupt routine (the ADDR line is the only bit enabled). The address bits (ADD1-3) are read and the state of the ADDR line is checked. If the ADDR is high and the address matches the slave module address, the ACKN line is raised to acknowledge the command. The command bits are saved and the background task is informed that there is a command pending,

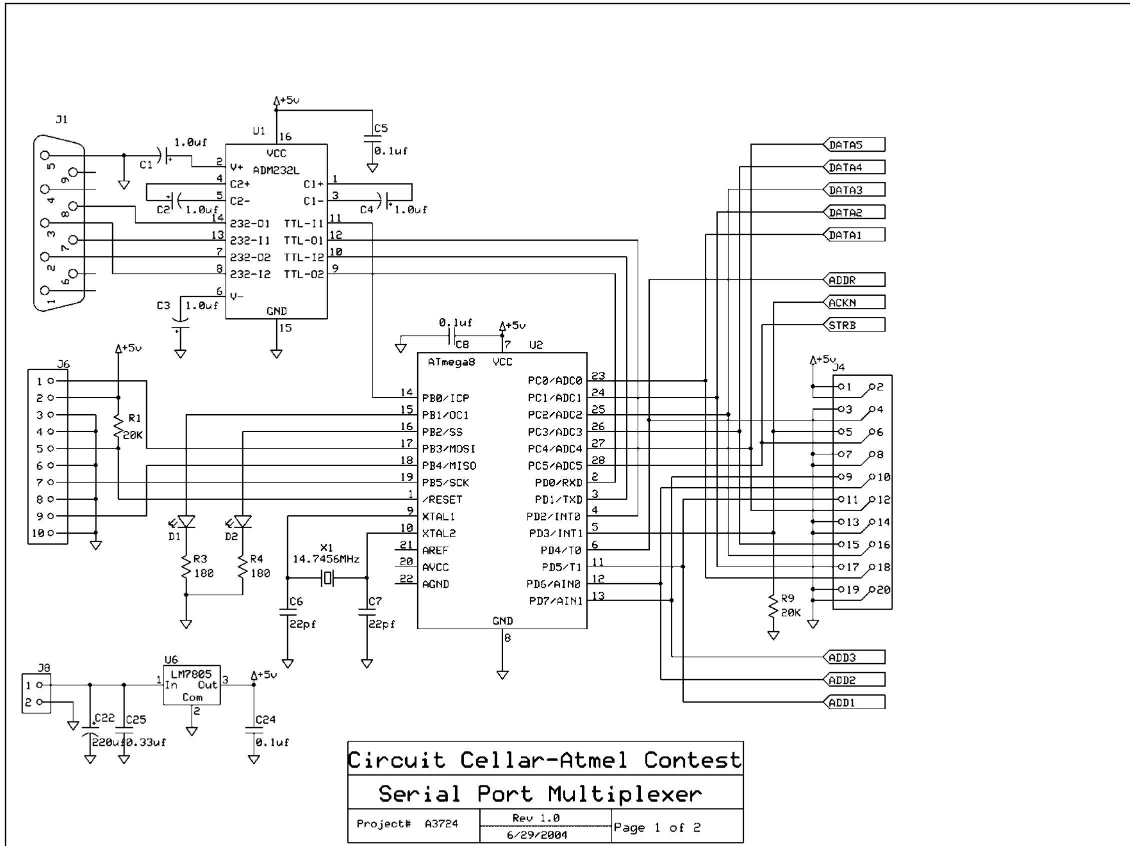
The getData routine is called by the master to transfer data from the addressed slave module. The master begins by tri-stating the DATA bus in anticipation of the data. The master waits for the slave to release the ACKN line (in response to the master releasing the ADDR line). Once the slave releases the ACKN line, the master signals ready for data by raising STRB. The master then waits for the slave to signal that data is ready by once again raising the ACKN line. With this done, the master reads the data on the data bus and exits the wait loop by setting the timeout flag. The loop exits (either normal or timeout) by clearing the bus (releasing STRB) and returns the data read.

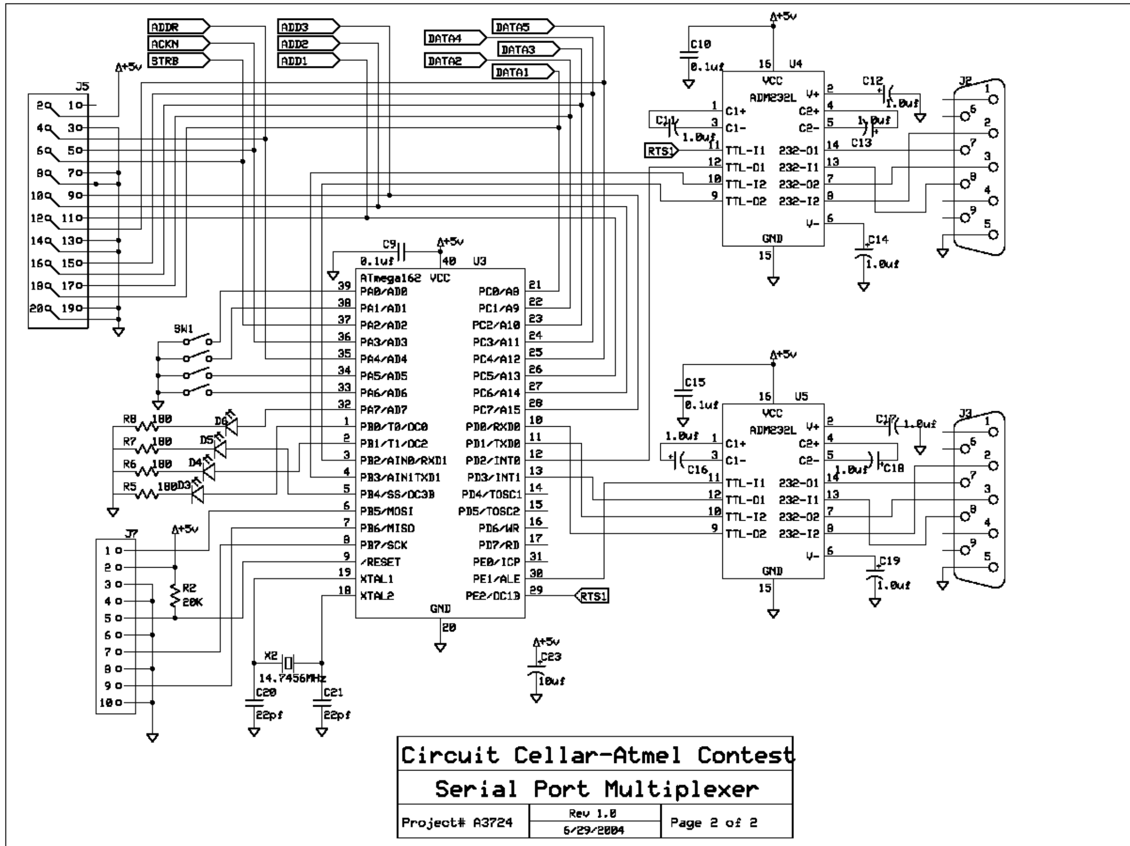
All data transfers are performed in the same manner. Multi-byte transfers follow the same format, with the first get or send after the sendCmnd being used to indicate the number of bytes to follow in the sequence.



```
// MASTER

unsigned char getStatus(unsigned char Addr,
unsigned char Command)
{
    unsigned char input;

    input = 0x00;
    if(sendCmnd(Addr, Command))
    {
        input = getData();
    }
    return( input );
}
```

```
// SLAVE

// Pin change 0-7 interrupt service routine
interrupt [PCINT0] void pin_change_isr0(void)
{
    unsigned char input;

    input = PINC;
    if( PINA.4 && ((input & 0xE0) == Address))
    {
        // mask pin change interrupt
        PCMSK0=0x00;
        PORTA.3 = 1;        // set ACKN bit
        DDRA.3 = 1;         // enable ACKN
        command_pending = 1;
        // isolate command
        command = input & 0x1F;
    }
}

void process_command(void)
{
    // start command processing
    // reset timeout timer
    timeout = 0;
    TCNT0 = 0x01;

    while( PINA.4 && timeout == 0 )
        ;
    // Clear ACKN
    PORTA.3 = 0;
    //clear pin change interrupt flag (ADDR went low by now)
    GIFR=0x08;
    if( timeout == 0 )
    {
        switch(command)
        {
            case READ_STATUS:
                WriteByte(Status);
                break;
        }
    }
```


Circuit Cellar-Atmel Contest
Serial Port Multiplexer
Project# A3724
Rev 1.0
6/29/2004
Page 1 of 2
J1
U1
ADM232L
C1
C2
C3
C4
C5
1.0uf
0.1uf
+5v
U2
ATmega8
C8
X1
14.7456MHz
C6
C7
22pf
J6
R1
20K
D1
D2
R3
R4
180
J8
U6
LM7805
In
Out
Com
C22
C24
C25
J4
R9
20K
DATA5
DATA4
DATA3
DATA2
DATA1
ADDR
ACKN
STRB
ADD3
ADD2
ADD1


ADDR
ACKN
STRB
ADD3
ADD2
ADD1
DATA4
DATA2
DATA5
DATA3
DATA1
J5
SW1
U3
ATmega162
U4
ADM232L
U5
ADM232L
J2
J3
J7
RTS1
14.7456MHz
Circuit Cellar-Atmel Contest
Serial Port Multiplexer
Project# A3724
Rev 1.0
6/29/2004
Page 2 of 2