Using the Serial Port with a MATLAB GUI*

Jason Laska
Matthew Berry
Daniel Sachs

This work is produced by OpenStax-CNX and licensed under the
Creative Commons Attribution License 1.0†

Abstract

Explains how to send and receive data with the DSP through the serial port. Example code is shown in both assembly and C (for the DSP) as well as MATLAB for interfacing on the PC. Some example code for creating a MATLAB GUI is also shown.

1 Introduction

The serial port on the back of the DSP box can be used to transmit data between the DSP and the PC during real-time operation. Eight bits of data can be transmitted at one time (signed or unsigned). This can then be used as feedback from the DSP for a variety of applications. The serial port is connected to a data buffer which holds values coming from the DSP until they are read. This allows for sending larger amounts of data (more than 1-byte) at a time from the DSP to the PC. The same is true in the opposite direction.

Our Serial Port Specifications

- Port: com2
- DataRate: 38400
- Parity: None
- StopBits : 1 Stop Bit
- DataBits : 8 Data Bits
- Handshaking : none

These parameters can be used when accessing the serial port through third-party utilities such as HyperTerminal (included with windows).

2 Using the DSP to Access the Serial Port

The serial port data buffer can be written to with either C or assembly code. These functionalities are supplied through the core file, so it must be included with any of this code.

*Version 1.1: Jun 25, 2004 3:47 pm -0500
†http://creativecommons.org/licenses/by/1.0

2.1 Using Assembly to Send/Receive

Accessing the serial port in assembly comes in the form of macros (READSER and WRITSER). These macros allow for multiple bytes of data to be written to the serial port buffer at a time.

- `READSER` : READSER accepts one parameter, an integer number(n). The macro will read n bytes into memory and place them starting at the address `*AR3`. The macro modifies `AR3` and it is left pointing to one byte past the last memory location written. The actual number of data bytes read is put into `AR1`. If `AR1` is zero, there were no available data byte sin the buffer. The calling format is: `READSER n`
- `WRITSER` : Similar to the previous macro, WRITSER takes a single integer parameter n. This macro will add n bytes starting at `*AR3` to the serial port buffer. `*AR3` is left pointing one location after the last memory read. This data is queued in the buffer and will remain there until the PC retrieves the data. The calling format is: `WRITSER n`

WARNING: READSER and WRITSER modify registers AR0, AR1, AR2, AR3, and BK as well as the flag TC.

The core file allows up to 126 characters to be stored in the input and output buffers. No checks to protect against buffer overflows are made, so do not allow the input and output buffers to overflow. (The length of the buffers can be changed by changing `ser_rxlen` and `ser_txlen` values in the core.asm file.) The buffers are 127 characters long; however, the code cannot distinguish between a completely-full and completely-empty buffer. Therefore, only 126 characters can be stored in the buffers.

It is easy to check if the input or output buffers in memory are empty. The input buffer can be checked by comparing the values stored in the memory locations `srx_head` and `srx_tail`; if both memory locations hold the same value, the input buffer is empty. Likewise, the output buffer can be checked by comparing the values stored in memory locations `stx_head` and `stx_tail`. The number of characters in the buffer can be computed by subtracting the head pointer from the tail pointer; add the length of the buffer (normally 127) if the resulting distance is negative.

Example 1

Download the code here ser_echo[1]

```
.copy "v:\54x\dsplib\core.asm"

      .sect ".data"
hold  .word 0

      .sect ".text"
main
      stm #hold,AR3       ; Read to hold location

      READSER 1          ; Read one byte from serial port

      cmpm AR1,#1        ; Did we get a character?
      bc main,NTC        ; if not, branch back to start

      stm #hold,AR3      ; Write from hold location
      WRITSER 1          ; ... one byte

      b main
```

[1] http://cnx.org/content/m12062/latest/ser_echo.asm

On Line 8, we tell `READSER` to receive into the location hold by setting `AR3` to point at it. On Line 9, we call `READSER 1` to read one serial byte into hold; the byte is placed in the low-order bits of the word and the high-order bits are zeroed. If a byte was read, `AR1` will be set to 1. `AR1` is checked in Line 12; Line 13 branches back to the top if no byte was read. Otherwise, we tell reset `AR3` to hold (since `READSER` moved it), then call `WRITSER` to send the word we received on Line 16. On Line 18, we branch back to the start to receive another character.

2.1.1 Alternative Method in Assembly

Many students have found that there are issues with the `READSER` and `WRITSER` macros. Performance of these macros is often "flaky" if they even work at all. Two ECE 320 students I-Ju Liao and Jung-Gyu Lee from the Fall 2002 semester created this alternative method which provides much better assembly support for serial port access. The following is a skeleton for reading data from the serial port onto the DSP:

Example 2
Skeleton of a program for receiving data over the serial port. The function of interest is get_data. In this function, we first recieve one 8 bit value and store it at `value_1`. Then, we receive one 16 bit value and store it at `value_2`.

```
.copy "v:\ece320\54x\dsplib\core.asm"


        .sect ".data"
value_1 .word 0
value_2 .word 0


        .sect ".text"
main:

loop:
        WAITDATA

        call #get_data      ; call function to get serial port data

        stm #BlockLen-1, BRC
        rptb endblock-1

        ;******your code goes here

endblock:
        b loop



get_data:
        pshm AR0            ; we save all registers that are used in
        pshm AR2            ; this function - note that accumulator
        pshm AR3            ; B IS OVERWRITTEN!
        pshm BK
```

```
mvdm #srx_head, AR2 ; srx_head, defined in core.asm, points
                    ; to one element past the last value
                    ; recieved in the serial recieve buffer

stm #ser_rxlen, BK  ; set BK to length of receive buffer

mar *+AR2(-4)%      ; AR2 now points to the most recently
                    ; received block of 24 bits, i.e. one 8
                    ; bit value and one 16 bit value

stm #1, AR0         ; set increment

stm #value_1, AR3   ; get first value
mvdd *AR2+0%, *AR3  ; save at value_1

stm #value_2, AR3   ; get second value
ld *AR2+%, 8, B     ; place first 8 bits in high part of B
or *AR2+%, B        ; combine last 8 bits in low part of B
stl B, *AR3         ; save at value_2

popm BK
popm AR3
popm AR2
popm AR0
ret
```

NOTE: The above program does not describe an alternative means for transmitting data from the DSP board. Some simple sleuthing in the core.asm file starting at stx head should shed some light on the subject.

2.2 Using C to Send/Receive

There are several functions for transmitting and receiving serial data within the C environment:

- `SerialRX()` takes no arguments and returns an integer, which is the next byte waiting in the serial input buffer. If there is no byte waiting, the function returns -1.
- `SerialTX()` takes one argument, an integer to place in the serial output buffer. It returns nothing.
- `SerialRXBufCheck()` takes no arguments and returns the number of bytes waiting in the serial input buffer.
- `SerialRXm()` takes two arguments: the first is the number of bytes to read from the serial input buffer, and the second is a pointer, which is usually the name of an array into which the bytes will be copied. If you attempt to read more bytes than are waiting, the function will only copy those bytes that are waiting and then return. It always returns the number of characters read.
- `SerialTXm()` takes two arguments: the first is the number of characters to write into the serial output buffer, and the second is a pointer, which is usually the name of an array containing the bytes to copy. It returns nothing.

NOTE: The restrictions on buffer length discussed in the assembly section also apply in C: No more than 126 bytes can be stored in the serial transmit buffer or in the serial input buffer, and the core file does not attempt to prevent buffer overflow. Be careful.

Example 3

The following example shows a simple C program that will echo received serial data back through the serial port, much like the assembly example from the previous section.

```
#include "v:/ece320/54x/dspclib/core.h" /* Declarations for core file */

main()
{
    int *Rcvptr, *Xmitptr; /* pointers to Xmit and Rcv Bufs */
    int i;

    while(1)
    {
        WaitAudio(&Rcvptr, &Xmitptr);

        for(i=0; i < BlockLen; i++)
        {
            Xmitptr[6*i] = Rcvptr[4*i];
            Xmitptr[6*i+1] = Rcvptr[4*i];
            Xmitptr[6*i+2] = Rcvptr[4*i];
            Xmitptr[6*i+3] = Rcvptr[4*i+2];
            Xmitptr[6*i+4] = Rcvptr[4*i+2];
            Xmitptr[6*i+5] = Rcvptr[4*i+2];
        }

        i = SerialRX(); /* Check serial port */
        if (i > 0)
            SerialTX(i); /* Echo received byte */

    }
}
```

As you can see, working with the serial port is easier in C than in assembly.

Example 4

The next example demonstrates how to receive and transmit multiple bytes using `SerialRXm()` and `SerialTXm`.

```
#include "v:/ece320/54x/dspclib/core.h" /* Declarations for core file */

main()
{
    int *Rcvptr, *Xmitptr; /* pointers to Xmit and Rcv Bufs */
    int i;
    int array[10];

    while(1)
    {
        WaitAudio(&Rcvptr,&Xmitptr);

```

```
        for(i=0; i < BlockLen; i++)
        {
            Xmitptr[6*i] = Rcvptr[4*i];
            Xmitptr[6*i+1] = Rcvptr[4*i];
            Xmitptr[6*i+2] = Rcvptr[4*i];
            Xmitptr[6*i+3] = Rcvptr[4*i+2];
            Xmitptr[6*i+4] = Rcvptr[4*i+2];
            Xmitptr[6*i+5] = Rcvptr[4*i+2];
        }

        if ( SerialRXBufCheck() >= 10 )
            SerialRXm(10,array); /* copy serial receive data into array1 */
            SerialTXm(10,array); /* echo all ten bytes */

    }
}
```

3 Using MATLAB to Access the Serial Port (PC)

MATLAB can be used to access the data coming from the serial port. This guide will show the set-up procedures neccesary. Other serial-port readers should also work as long as they are set up with the parameters specified in the introduction.

3.1 Sending Data

Before accessing the serial port, it must be initialized through MATLAB. This is done with this code:

```
% Set serial port mode
!mode com2:38400,n,8,1
```

which sets the port to all of neccesary parameters. The port is still not open for writing, however it is now in the correct mode. To open the port, the fopen command is used, which returns a file descriptor to the port:

```
% open com port for data transfer
fid = fopen('com2:','w');
```

'com2:' is the port on the PC, 'w' means that we are opening the port for writing, and fid is the file descriptor. For our purposes, you can think of the file descriptor as the port buffer itself, and when you write to it, you are writing directly to the buffer. To write to the serial port buffer, the fwrite command is used:

```
fwrite(fid,data,'int8');
```

data is the data to send to the port, fid is the file descriptor of the open port, and 'int8' is the type of data being sent. For a list of different data types, check MATLAB help files with `help serial`. Since the DSP is blind to the different types and we can only use 8 bits at a time, int8 should work.

Before finishing a function, or before executing a read from the serial port, the port MUST BE CLOSED. Failure to close the port, will result in blocking access to other functions and apps on the machine that need to use the port. A reset pulse is sent before closing. The port is closed with the fclose command:

```
% send reset pulse
fwrite(fid,255,'int8');
% close com port connection
fclose(fid);
```

It seems intuitive that to read from the port, it need to be opened with a 'r' or a 'r+' instead of 'w'. According to the MATLAB help files this is true, but in practice it does not work. See the next section for information on how to read from the serial port. Another method of opening the port is using the `fid = serial('com2');` command. This does not seem to work for reading either. See the MATLAB help for more details and methods.

3.2 Receiving Data

Although MATLAB is supposed to support both writing and reading data from the serial port, reading data seems to either produce no result, generate an error, or crash MATLAB. To remedy the situation GetSerialData() has been written. This function will allow you to get vectors of data from the serial port buffer.

3.2.1 Setting Up GetSerialData.cpp

You can download a copy of GetSerialdata.dll[2] and skip this step. If you wish to modify the code for GetSerialData.cpp to handle other serial port protocols (such as handshaking and other features) you can use this to help you re-make the code.

Files you will need:

- GetSerialData.cpp[3]
- stdafx.h[4]

To compile the code, change to the directory (in MATLAB) with GetSerialData.cpp. Type the command:

```
mex GetSerialData.cpp
```

MATLAB may ask you to set up a compiler. Choose the MATLAB compiler (It is usually one of the first options and has the word MATLAB somewhere in its path). After doing this, repeat the 'mex' command on the code. Note: This code will only work with Windows (only been tested on XP).

Compiling the C code produces a .dll file. The file at this stage is similar to a .m file in that it adds custom functionality to MATLAB. To use the file, place it in the directory where you will use the GetSerialData function.

[2] http://cnx.org/content/m12062/latest/GetSerialData.dll
[3] http://cnx.org/content/m12062/latest/GetSerialData.cpp
[4] http://cnx.org/content/m12062/latest/StdAfx.h

3.2.2 Using GetSerialData with the DSP

GetSerialData should work with both the assembly and C implementations of outputting data to the serial port. Sometimes a DSP will not output any serial port data. Often times this means this feature is broken on the DSP, but occasionally you can get the serial port to output data if you halt your program, press the red button a couple of times, flip the switch twice, and reload your program. To test the port for incoming data, load up the program 'Hyperterm' (StartMenu:Accessories:Communications:Hyperterm). Connect to com2 with data rate 38400 and look for ascii characters. It is suggested that you test for data first with the terminal and not MATLAB because if there is no data coming into MATLAB, it will stall until the function times out.

> NOTE: You do not need to worry about opening or closing the com2 port because the function does it all under the hood. The port must be available for usage (so if MATLAB was writing to the port earlier, it has to be closed).

Once the DSP is running code that outputs data to the serial port, it continuously sends the data. GetSerialData simply 'captures' the output from the buffer and records it to a MATLAB row vector of specified size. The calling format is:

```
y = GetSerialData('port', baud, size);
```

- 'port' is the serial port to which the DSP is connected. For our use it will be 'com2'. The port name must be entered in quotes.
- baud is the speed at which we transfer data. For the DSPs in lab we use 38400.
- size is the length of the row vector you want to acquire.
- y is the output vector.

After calling the function, it will not return until it has receive size bytes from the serial port. If it never receives the bytes, it will eventually time out. Since the serial port only outputs single bytes at a time, the max integer that can be acquired is 255 and the min is 0. If you wish to use signed numbers, a fourth parameter can be entered into the function to specify. To see notes on usage and other baud rates, ports, signed data, etc type:

```
GetSerialData('help');
```

This will bring up a help screen with the full set of options.

> **Example 5**
> This example shows what type of vector would be aquired if the DSP was constantly counting up and outputting these numbers. We are taking in vector of size 6 at some random point in the operation of the DSP:

```
%In the MATLAB terminal:

y = GetSerialData('com2', 38400, 6)

y =

        7 8 9 10 11 12
```

The numbers are counting up as written in the C DSP code. We can also specify signed numbers and if we catch the counting at the right moment we get an output like this:

```
y = getSerialData('com2', 38400, 6, 1)

y =

        125 126 127 0 -1 -2
```

3.2.3 Other Notes

Other functionality can be added to this code. This may include other serial port issues (such as handshaking or parity) or even the formatting of the data coming out of the dsp. For instance, to get numbers larger than bytes in each vector index, you can modify how data is written to the MATLAB vector when it is acquired in the Receive function (in the code). Code for modifying serial port abilities is commented in the main() function where the serial port handle is initialized.

4 Using MATLAB GUI Features

MATLAB has some nice Graphical User Interface (GUI) features which can be used to control the flow of data to and from the serial port. The basic implementation consits of a blank window (figure) which can have different interface elemnts palced on it. These elements can be sliders, buttons, text boxes, etc...

When an element is accessed (for instance, a slider is moved, or a button is pushed), MATLAB will execute a "callback routine" which is a MATLAB function defined by the user. Desgining these interfaces is simple.

4.1 Creating a User Interface with Sliders

Download These Files

- ser_set.m[5] - User Interface
- wrt_slid.m[6] - Callback File

Example 6

```
% ser_set: Initialize serial port and create three sliders

% Set serial port mode
!mode com2:38400,n,8,1

% open a blank figure for the slider
Fig = figure(1);

% open sliders

% first slider
```

[5] http://cnx.org/content/m12062/latest/ser_set.m
[6] http://cnx.org/content/m12062/latest/wrt_slid.m

```
12  sld1 = uicontrol(Fig,'units','normal','pos',[.2,.7,.5,.05],...
13  'style','slider','value',4,'max',254,'min',0,'callback','wrt_slid');
14
15  % second slider
16  sld2 = uicontrol(Fig,'units','normal','pos',[.2,.5,.5,.05],...
17  'style','slider','value',4,'max',254,'min',0,'callback','wrt_slid');
18
19  % third slider
20  sld3 = uicontrol(Fig,'units',normal','pos',[.2,.3,.5,.05],...
21  'style','slider','value',4,'max',254,'min',0,'callback','wrt_slid');
```

Lines 12 through the end create the three sliders for the user interface. Several parameters are used to specify the behavior of each slider. The first parameter, Fig, tells the slider to create itself in the window we created in Line 7. The rest of the parameters are property/value pairs:

- `units`: Normal tells Matlab to use positioning relative to the window boundaries.
- `pos`: Tells Matlab where to place the control.
- `style`: Tells Matlab what type of control to place. slider creates a slider control.
- `value`: Tells Matlab the default value for the control.
- `max`: Tells Matlab the maximum value for the control.
- `min`: Tells Matlab the maximum value for the control.
- `callback`: Tells Matlab what script to call when the control is manipulated. wrt_slid is a Matlab file that writes the values of the controls to the serial port.

Every time a slider is moved, the wrt_slid.m file is called:

Example 7

```
1   % wrt_slid: write values of sliders out to com port
2
3   % open com port for data transfer
4   fid = fopen('com2:','w');
5
6   % send data from each slider
7   v = round(get(sld1,'value'));
8   fwrite(fid,v,'int8');
9
10  v = round(get(sld2,'value'));
11  fwrite(fid,v,'int8');
12
13  v = round(get(sld3,'value'));
14  fwrite(fid,v,'int8');
15
16  % send reset pulse
17  fwrite(fid,255,'int8');
18
19  % close com port connection
20  fclose(fid);
```

Line 7 retrieves the value from the slider using the get function to retrieve the value property. The value is then rounded off to create an integer, and the integer is sent as an 8-bit quantity to the DSP in Line 8. (The

number that is sent at this step will appear when the serial port is read with READSER or a C equivalent in your code.) The other two sliders are sent in the same way. Line 17 sends 0xFF (255) to the DSP, which can be used to indicate that the three previously-transmitted values represent a complete set of data points. This can be used to prevent the DSP and Matlab from losing synchronization if a transmitted character is not received by the DSP.

NOTE: Line 20 closes the serial port. Matlab buffers the data being transmitted, and data is often not sent until the serial port is closed. Make sure you close the port after sending a data block to the DSP.

4.2 Advanced features

The slider example shows some basic features of the gui tools. The file descriptor is generated into the workspace so that it can be used for writing. But other elements, such as text boxes cannot be dealt with as easily. The Parameters from these can be accessed through their returned handles. Some examples:

Example 8

```
%GUI.m
%****Sample GUI, Text and a Button***

%open a blank figure
Fig = figure(1);
set(Fig,'Name','Test GUI');

%Space to enter text
ed2 = uicontrol(Fig,'backgroundcolor','white','units','Normalized','pos',[.1,.6,.4,.05],...
                'string','Default Text','style','edit');

%Button
but1 = uicontrol(Fig,'foregroundcolor','blue','units','Normalized','pos',[.1,.4,.5,.1],...
                'string','Press Me!','style','pushbutton','callback','SampleGUI');
```

A Text box is created with default text in it that says: "Defaul Text". A button is also created, which when pressed, will execute the callback function SampleGUI.m

```
%SampleGUI.m

%Get Text
testText = get(ed2,'string')
```

Now testText holds whatever string was enetered into the text box. The function `get()` is used to retrieve the data from the 'string; parameter in the ed2 handle. MATLAB `help uicontrol` gives the full list of options for interface elements.