DSP32C Direct Serial Port Communications

This note discusses direct serial port communications between DSP32Cs. With simple pin to pin connections and accompanying software, data can be easily transferred from one DSP32C to another via their high speed serial ports. Both the hardware and software required to set up communications are covered in this note.

From a hardware point of view, the DSP32C serial port interface is comprised of independent input and output sections. Each is equipped with its own clock, load, and data signals, where the load and data are sampled on the rising edge of their respective clock. The input and output ports may be driven by either internally generated clocks and loads or by external user generated signals, and can be run completely asynchronously from each other if external circuitry is being used. Both the input and output sections have buffers which feed into serializing registers, allowing for other DSP execution to continue while a serial transfer is underway. For more specifics on the serial port hardware you should also consult the AT&T DSP32C Information manual, chapter 5.

From a software point of view, the serial input and output ports can be programmed to work either in polled, interrupt, or dma modes. This app note covers using the serial port with a polled program, showing typical code that can be used as a basis for more sophisticated applications. The code listed here can also be found on the app note disk.

HARDWARE CONNECTIONS

For an interface driven by the internal clock and load signals, the connections shown in Fig 1 should be made between two DSP32Cs. For short connections of a foot or less, simple wires between the serial port header pins will be sufficient. For longer runs you may wish to consider adding buffers on each end of the cable along with some termination circuitry. Note that the OEN pin must be grounded for the data out pin DO to be enabled.

These connections assume that the transmitting DSP[0] will generate the output clock and load signals internally. Whether these signals are generated internally or externally is controlled by bit settings in the IOC control register covered later. When driven internally, the clock signal may be selected to have a frequency of either 2.083 or 6.250Mhz. These frequencies were selected by AT&T because they work well with many codecs and other peripheral devices. In fact, the serial port is specified to work at clock frequencies up to 16Mhz, and if you wish, an external clock signal at this frequency may be used. If you do use an external clock, the same signal should be fed to both DSP[0] and DSP[1]. Higher frequency clocks can be derived from small metal can TTL oscillator units.

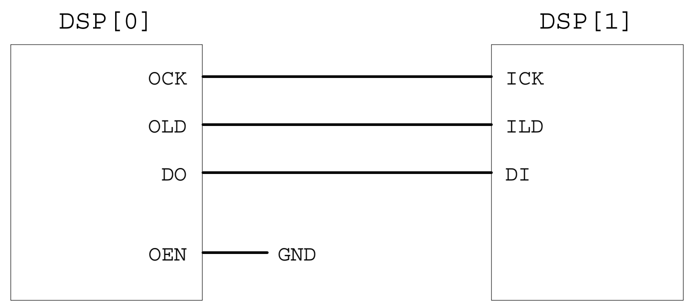

FIG 1

Once the connections are made, you are ready to program the chips to communicate. Note that in addition to the connections diagrammed here, you can also make more complicated topologies involving more DSP32Cs. Some examples would be daisy chaining, rings, and star networks. When using more complicated topologies, we recommend connecting all of the clock pins together and driving them with a single signal to keep all the systems in sync.

IOC REGISTER SETTINGS

The serial port has a number of programmable options to accommodate various operating requirements. These options are controlled by the bit settings in the IOC register. As with the hardware signals, these bit settings are covered in the AT&T DSP32C Information manual. Each of the bits can be related to items on the block diagrams in Figures 5-2 and 5-5 in their manual, and studying these figures can help in understanding the operations.

At the top of programs using the serial port, the IOC control register should be set with an ioc = N instruction. While any legal form of a hex constant N can be specified, a series of defined constants is mnemonic and makes programs easy to maintain. The ones we suggest are listed below and can be bitwise ORed together to form a complete value.

```
constant  /* Serial port IOC bit assignments. */

    SY_EXT     = 0x0 << 0x00,        /* sy external         */
    SY_INT     = 0x1 << 0x00,        /* sy internal         */

    BC_ICK     = 0x0 << 0x01,        /* load/sy from ICK    */
    BC_OCK     = 0x1 << 0x01,        /* load/sy from OCK    */

    SLEN_00    = 0x0 << 0x02,        /* load/sy ratios      */
    SLEN_08    = 0x1 << 0x02,
    SLEN_16    = 0x2 << 0x02,
    SLEN_32    = 0x3 << 0x02,

    ICK_EXT    = 0x0 << 0x04,        /* ICK external        */
    ICK_INT    = 0x1 << 0x04,        /* ICK internal        */
```

```
ILD_EXT     = 0x0 << 0x05,        /* ILD external        */
ILD_INT     = 0x1 << 0x05,        /* ILD internal        */

ILEN_00     = 0x0 << 0x06,        /* input length        */
ILEN_08     = 0x1 << 0x06,
ILEN_16     = 0x2 << 0x06,
ILEN_32     = 0x3 << 0x06,

OCK_EXT     = 0x0 << 0x08,        /* OCK external        */
OCK_INT     = 0x1 << 0x08,        /* OCK internal        */

OLD_EXT     = 0x0 << 0x09,        /* OLD external        */
OLD_INT     = 0x1 << 0x09,        /* OLD internal        */

OLEN_00     = 0x0 << 0x0A,        /* output length       */
OLEN_08     = 0x1 << 0x0A,
OLEN_16     = 0x2 << 0x0A,
OLEN_32     = 0x3 << 0x0A,

SANITY      = 0x1 << 0x0C,        /* set sanity          */

DMA_00      = 0x0 << 0x0D,        /* no dma              */
DMA_01      = 0x1 << 0x0D,        /* in IBF              */
DMA_02      = 0x2 << 0x0D,        /* out OBE             */
DMA_03      = 0x3 << 0x0D,        /* in IBF, out OBE     */
DMA_04      = 0x4 << 0x0D,        /* in/out IBF&OBE      */
DMA_05      = 0x5 << 0x0D,        /* in/out IBF          */
DMA_06      = 0x6 << 0x0D,        /* in/out OBE          */
DMA_07      = 0x7 << 0x0D,        /* in/out IBF|OBE      */

IN_LSB      = 0x0 << 0x10,        /* input lst first     */
IN_MSB      = 0x1 << 0x10,        /* input msb first     */

OUT_LSB     = 0x0 << 0x11,        /* output lsb first    */
OUT_MSB     = 0x1 << 0x11,        /* output msb first    */

CKI_FAST    = 0x0 << 0x12,        /* clock = CKI/8       */
CKI_SLOW    = 0x1 << 0x12,        /* clock = CKI/24      */

O24         = 0x1 << 0x13,        /* out len 24 bits     */
DSZ         = 0x1 << 0x14;        /* dma size            */
```

HALT, RESET, AND THE SERIAL PORT

After a processor reset, all of the IOC bits are set to zero. This ensures that the serial port is not driving any of its output pins and no contention problems can occur until the DSP32C is programmed into a known state.

It is important however to also understand the relationship between the processor halt and reset states to effectively use the serial port. When a halt command is issued from DSPMON or DSPTOOLS, the processor is halted *but is not reset*. Halting the processor leaves the IOC register in its current state. *Only issuing a go command* from DSPMON or DSPTOOLS forces the processor execute its reset sequence and reset the IOC bits.

DSPMON has also been set up so that quitting leaves the DSP32C in its current state and does not even halt the processor. Starting DSPMON halts the processor, *but does not do a reset*. *Only issuing a go command or powering up will force the processor execute its reset sequence*.

This means that if you leave DSPMON with the IOC register programmed to generate internal clock and load signals, the serial output shift register OSR will continue transmitting the same value over and over again. This behavior is fine, but be aware when debugging your programs that even if you have quit and returned to DOS it is likely that the serial port is still operating.

While in principle you might want to turn the serial port on and off, it has been designed to be used in a continuous mode with start and stop signals transmitted in software. This is the approach taken in the example program below, and can be very flexible.

POLLED PROGRAMMED SERIAL PORT COMMUNICATIONS

This program gives a simple example of programming serial communications with dedicated polled loops. It has been designed to run on a DSP_MUL multiple cpu board with DSP[0] transmitting, and DSP[1] receiving, although it should be easy to modify for other setups. The code for the programs can be found on the supplied disk, and the batch file ser_xxx.bat will step you through running the example. The program ser_out.dsp handles serial output on DSP[0], while the program ser_inp.dsp handles serial input on DSP[1].

The sequence of events in ser_xxx.bat is as follows. First, ser_out.dso is downloaded to DSP[0] and should be started by hand while in DSPMON with the go command. Running this program leaves the serial port output shift register OSR cleared and DSP[0] transmitting zeros. After quitting DSPMON with the quit command, ser_inp.dso is downloaded and should be started by hand on DSP[1]. This program waits in an infinite polling loop waiting to receive a nonzero value signaling the start of an incoming transmission. Quitting DSPMON returns to ser_out.dso on DSP[0], and running it a second time executes the transmission sequence while DSP[1] is listening. The transmission is comprised of a word count, followed by data words, and a terminating zero to leave the transmitting OSR cleared. Quitting DSPMON on DSP[0] returns to the receiving DSP[1] so you can make sure the data was received.

In both the output and input programs, the transfer rate is maximized by programming the IOC registers for 32 bit communications. The code fragment for ser_out.dsp is:

```
/* Code fragment to run the SOP serial output port on DSP[0]. */

main {
        register p = r1;

        /* Start the SOP driving OCK and OLD internally. */
        ioc = ( OCK_INT | OLD_INT | OLEN_32 | BC_OCK );

        /* Transmit an array of data. */
        p = output_array;

        while ( p < end_of_output_array ) {
                if ( obe ) { obuf = *p++.l; }

                }

        /* When finished, wait in an infinite loop. */
        finished: goto finished;
}

long output_array[] = {

                0x0000000A,   /* word count .... */

                0x11111111,   /* array to transmit ... */
                0x22222222,
                0x33333334,
                0x44444444,
                0x55555555,
                0x66666666,
                0x77777777,
                0x88888888,
                0x99999999,
                0xAAAAAAAA,

                0x00000000,   /* null terminator ... */

                }, end_of_output_array;
```

Note that the word count, data, and null terminator are saved in a single output array, which is all transmitted in one loop. The ser_inp.dsp program will be waiting for these values, and its code fragment is:

```
/* Code fragment to run the SIP serial input port on DSP[1]. */

main {
        register

                nwords  = r1,
                p       = r2;


        /* Program the SIP to be driven by external ICK and ILD */
        ioc = ( ICK_EXT | ILD_EXT | ILEN_32 );


        /* Wait in an infinite loop to read the SIP. */
wait:
        nwords = ibuf.l;

            if ( nwords > 0 ) {   /* receive the transmission ... */

                p = input_array;

                while ( nwords > 0 ) {

                        if ( ibf ) { *p++ = ibuf.l; nwords--; }
```

```
                                }
                        }

        goto wait;  /* continue waiting for a transmission */
}


<0x100> long

        input_array[0x200] = { 0 };
```

After the transmission is complete you should be able to see the received words at location 0x100 on DSP[1]. Both processors are left in a state so that another transmission could be repeated from DSP[0] if desired.

There are many possible extensions to the code fragments listed above. For example, in addition to sending a leading word count, you may also want to send a target address for where data is to be put, or perhaps a cpu id if several DSPs are connected together. The possibilities can be tailored to your application needs.