Application Note 001:
Programming the Micro-bus Dual RS485 module.

1. Introduction:

The Micro-bus Dual RS485 board uses 2 MAX3100 SPI-bus to UART (Universal Asynchronous Receiver Transmitter) devices. These in tern drive industry standard RS485 line drivers based on the SN75176 device format. The actual device fitted may vary and may be replaced to suit you own requirements. (Driver voltage, unit load, current consumption etc.)

Both MAX3100's are driven from a single 3.6864MHz clock source.

The RTS output is used to change the data direction on the RS485 bus.

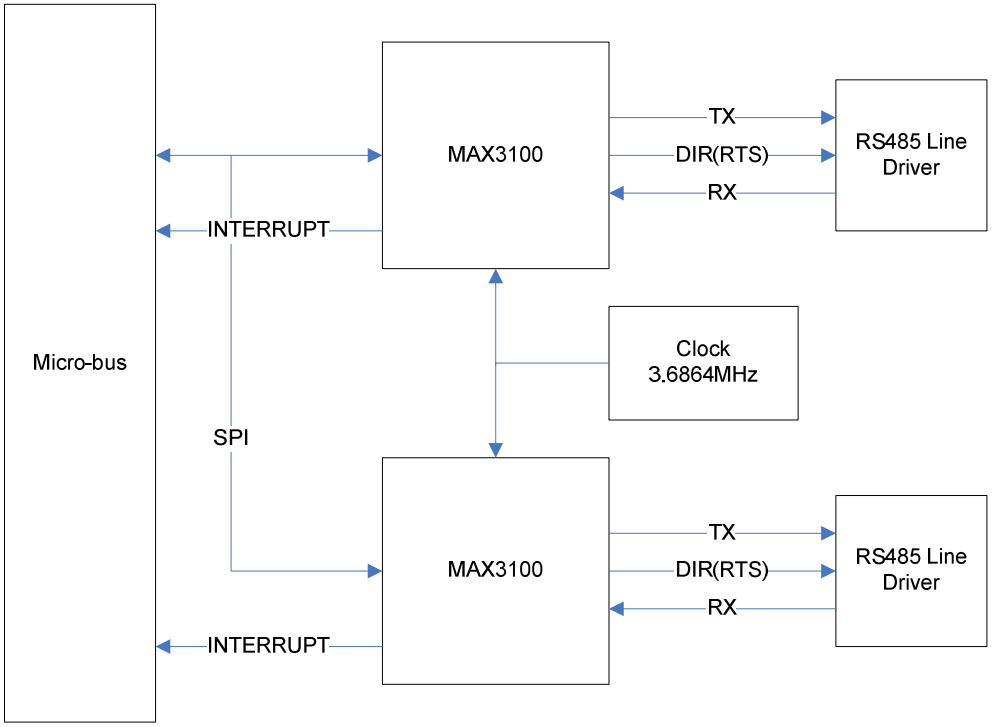

Figure 1 - Dual RS485 Line Driver

The application software configures channel 1 to transmit and channel 2 to receive using interrupts on the reception.

2. Software design and structure

2.1 The SPI bus

The Max3100 datasheet provides all the information needed to complete the design without scouring the web for additional resources. It is not the goal of this application note to repeat this information. Please have this datasheet on hand for reference.

The Micro-bus 8051 CPU module (DN-2006-04) was used as host CPU. This 8051 does not have a dedicated SPI port. I/O pins will be used to emulate the SPI bus interface. The code has been written in 'C'.

The software has been designed following a bottom up approach. The source code files have also been split so that the SPI bus layer can be replaced when SPI bus hardware is available.

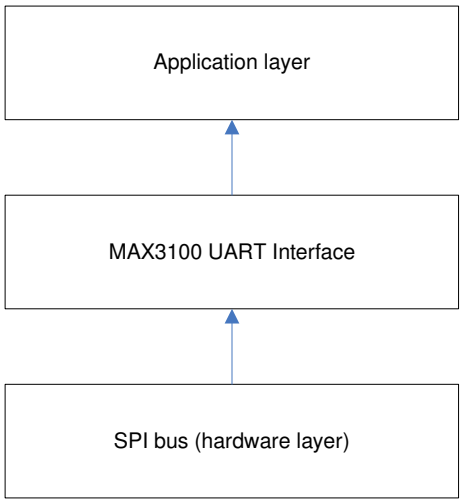

Figure 2 - Software layers

The first goal is to create a function that will read/write from/to the MAX3100. The function in figure 3 will perform both read and write functions on the SPI bus. It has 2 parameters, *device* and *word*. *Device* can be in the range 1 to 8 and selects a physical device on the SPI bus. The Micro-bus SPI bus is limited to 8 physical devices.

Word is an unsigned integer or 16 bits on an 8051. (Check the data size for a 16/32 bit CPU as an unsigned integer will probably be 32-bits. In this case change the data type to an unsigned short)
All reading and writing from/to the MAX3100 is done via 16 bits transfers. *Word* will carry MAX3100 configuration data or the actual data to send depending on the register accessed.

```
unsigned int SPI_Write_Word(unsigned char device, unsigned int word)
{
   data unsigned char i;
   data unsigned int mask;
   data unsigned int temp;

   EX0 = 0;    // disable interrupt

   mask = 0x8000;
   temp = 0;
   SPI_Chip_Select(device,LO);
   for(i=0;i<16;i++)
   {
      if(mask & word) SPI_MOSI = HI;
      else SPI_MOSI = LO;
      if(SPI_MISO == 1) temp |= mask;
      SPI_CLK = HI;
      SPI_CLK = LO;
      mask >>= 1;
   }

   SPI_Chip_Select(device,HI);
   EX0 = 1;    // enable interrupt
   return(temp);
}
```

Figure 3 - SPI Write function

The SPI_Write_Word function will also return an unsigned integer or 16-bits. All MAX3100 accesses will return data or status information.

The function SPI_Chip_Select will select the physical device using the parameter *device*. The code is shown in figure 4.

```
void SPI_Chip_Select(unsigned char device, bit state)
{
   switch(device)
   {
      case SPI_DEVICE_0 : SPI_CS0 = state; break;
      case SPI_DEVICE_1 : SPI_CS1 = state; break;
      case SPI_DEVICE_2 : SPI_CS2 = state; break;
      case SPI_DEVICE_3 : SPI_CS3 = state; break;
      case SPI_DEVICE_4 : SPI_CS4 = state; break;
      case SPI_DEVICE_5 : SPI_CS5 = state; break;
      case SPI_DEVICE_6 : SPI_CS6 = state; break;
      case SPI_DEVICE_7 : SPI_CS7 = state; break;
      default : break;
   }
}
```

Figure 4 - SPI Chip Select

The end result however is that the SPI_CSx port pin will take the *state* as specified. *State* can be either HI (logic one) or LO (logic zero).
Once the SPI bus device has been selected (SPI CSx = logic 0) the data transfer can begin. Data is transferred to and read from the MAX3100 on the rising edge of clock. The loop will repeat 16 times as we are making a 16-bit transfer. *Mask* starts at 10000000/00000000B and after each clock pulse the '1' moves 1 position to the right (`mask >>= 1;`). *Mask* is used in conjunction with *word* to determine the logic level of the MOSI (Master Output / Slave Input) when sending data. It is also used to "build" the input unsigned integer *temp* when examining the MISO (Master Input / Slave Output) port pin.

After the loop completes with SPI_CSx port pin must be returned to its inactive state or logic 1. This completes the read/write function.

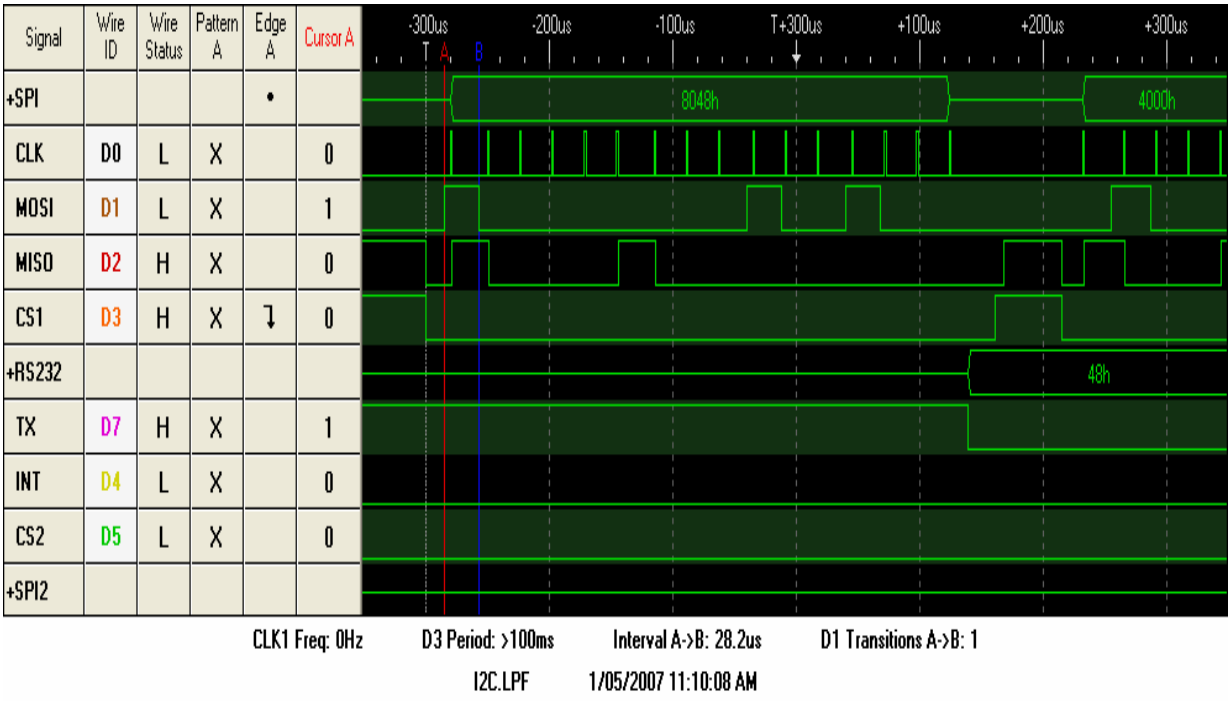

Figure 5 - SPI Write Word Timing (1)

Figure 5 shows the basic timing structure of the SPI bus session. Each clock pulse is 1us in width. (This shows how 'slow' the 8051 is at 11.0592MHz with the clock division of 12.) The transmission data is "HELLO". This plot shows the "H" or 0x48 being sent. You can clearly see that the asynchronous transmission starts soon after the last clock pulse, but just before the chip

select is de-selected. The transition of TX to logic 0 is the start bit of the asynchronous data.

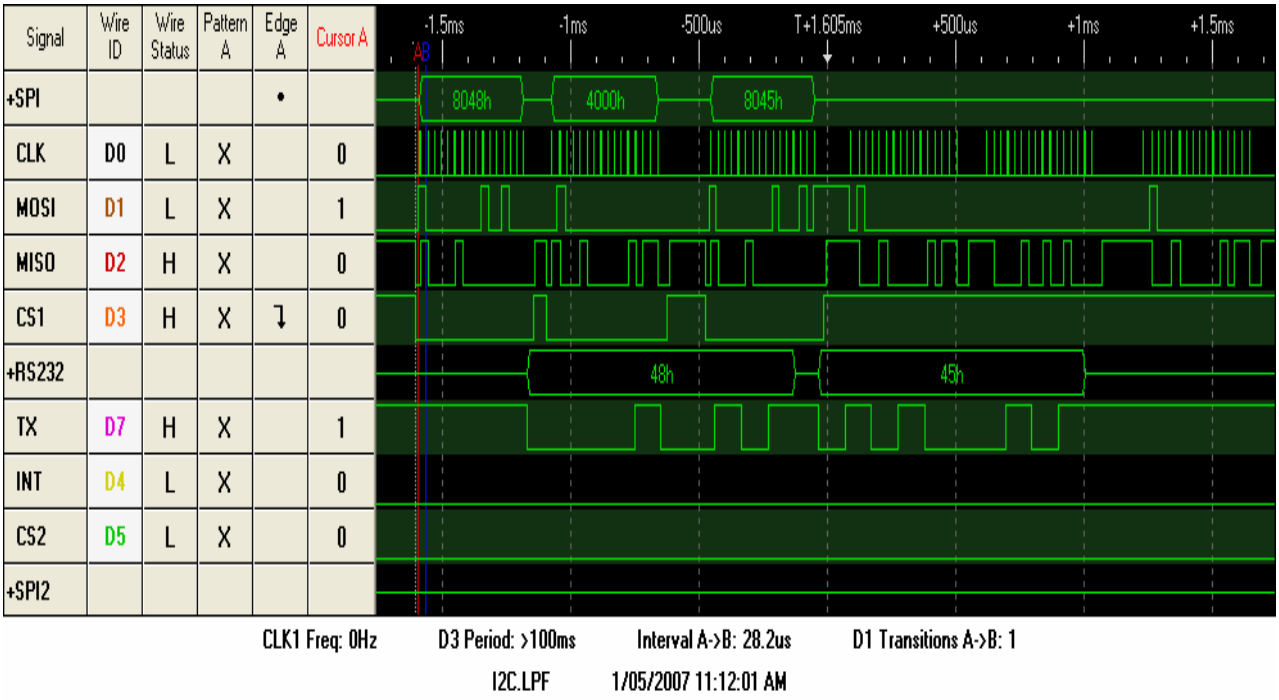

Figure 6 - SPI Write Word Timing (2)

Figure 6 shows the SPI session after "HE" or 0x48/0x45 has been sent. Notice the configuration read 0x4000 after the 0x8048 to test the T bit to see if the TX buffer is empty.

There is one additional bit of code not covered in the SPI_Write_Word function, the de-activation (`EX0 = 0;`) and re-activation (`EX0 = 1;`) of interrupts. In this case INT0 on the 8051. The reason for this will be explained later and depending on the way your code is structured may not be required.

2.2 The MAX3100

Now that we have a function to drive the SPI bus we can concentrate on configuring the MAX3100.

The following functions will write/read the MAX3100 configuration register.

```
void MAX3100_Write_Config(unsigned char device, unsigned int config)
{
   SPI_Write_Word(device,config);
}
```

Figure 7 - MAX3100 Write Configuration

```
unsigned int MAX3100_Read_Config(unsigned char device)
{
   return(SPI_Write_Word(device,MAX3100_READ_CONFIG));
}
```

Figure 8 - MAX3100 Read Configuration

These functions are really just wrapper functions to separate the SPI layer from the MAX3100 UART layer.

Table 1. Write Configuration (D15, D14 = 1, 1)

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIN	1	1	$\overline{FEN}$	SHDNi	$\overline{TM}$	$\overline{RM}$	$\overline{PM}$	$\overline{RAM}$	IR	ST	PE	L	B3	B2	B1	B0
DOUT	R	T	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 9 - MAX3100 Write Configuration Register

Shown above is the MAX3100 configuration register. Each register is prefixed by a 2-bit register identification field. The following #defines make these easy to select.

```
#define MAX3100_READ_CONFIG    0x4000
#define MAX3100_WRITE_CONFIG   0xC000
#define MAX3100_READ_DATA      0x0000
#define MAX3100_WRITE_DATA     0x8000
```

Figure 10 - MAX3100 register definitions

Typical application layer code would configure the MAX3100 as follows. (Note that not all fields have been #defined)

```
MAX3100_Write_Config(MAX3100_RS485_1,MAX3100_WRITE_CONFIG | 0x040B); // 9600,N,8,1
Interrupt + FIFO

MAX3100_Set_RS485_Mode(MAX3100_RS485_1,MAX3100_WRITE_DATA + MAX3100_RS485_TX +
MAX3100_NO_TX);
```

Figure 11 – Application layer configuration code

See the MAX3100 datasheet for comprehensive descriptions of all the configuration bit fields.

When reading the MAX3100 configuration register the following bits are important.

Table 2. Read Configuration (D15, D14 = 0, 1)

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIN	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	TEST
DOUT	R	T	$\overline{FEN}$	SHDNo	$\overline{TM}$	$\overline{RM}$	$\overline{PM}$	$\overline{RAM}$	IR	ST	PE	L	B3	B2	B1	B0

Figure 12 - MAX3100 Read Configuration Register

Bit 15 – R and bit 14 – T indicate data reception/transmission status.
If R is set then there is data to read from the FIFO. If the T bit is set then the TX buffer is empty.
All the configuration bits are returned so the actual configuration could be verified. This is useful when debugging.

```
void MAX3100_Write_Data(unsigned char device, unsigned char byte)
{
   data unsigned int w;

   w = (unsigned int)byte | MAX3100_WRITE_DATA | MAX3100_RS485_TX;
   SPI_Write_Word(device,w);
   do
   {
     w = MAX3100_Read_Config(device);
   }while(!(w & 0x4000));    // wait for TX buffer empty
}
```

Figure 13 - MAX3100 Write Data

The MAX3100_Write_Data function has a little more meat. Again the function uses 2 parameters. *Device* selects the physical device as there could be more than 1 MAX3100 on the SPI bus. *Byte* is the character to be sent by the MAX3100.

Byte only fills the lower 8 bits of the SPI_Write_Word function's parameter *word.* We also need the MAX3100 Data Register identification field and we need to set the RTS bit to the appropriate level.

When the RTS bit in the configuration register is logic 0 then the RTS pin is logic 1. The SN75176 (or similar device) uses logic 1 on the DE/RE pins to set the transmit state for the RS485 bus.

After a SPI write we read back the configuration register to confirm that the MAX3100 is ready for the next byte.

From this basic function we can now build the following more useful function.

```
void MAX3100_Print(unsigned char device, char *string)
{
   while(*string)
   {
      MAX3100_Write_Data(device, *string);
      string++;
   }
}
```

Figure 14 - MAX3100 Print Function

Using this function we can now send NULL terminated character strings to another RS485 device.

Receiving data is best done from an Interrupt routine.

```
void MAX3100_Interrupt_Handler(void)
{
   unsigned int w;
   unsigned char x;

   w = MAX3100_Read_Config(MAX3100_RS485_2);
   if((w & 0x8000) == 0x8000)
   {
      x = SPI_UART_Read_Data(MAX3100_RS485_2);
      Add_Byte_buffer(x);
   }
}
```

Figure 15 - MAX3100 Interrupt Routine

This function is called from the 8051 interrupt handler. It does not make use of any parameters. If a read of the MAX3100 configuration register yields R=1 then there is data in the FIFO. The 8051 external interrupt must be configured as a level triggered interrupt and must be active LO. The MAX3100 interrupt will remain logic 0 as long as there is data in the FIFO which means the interrupt will re-trigger as long as the MAX3100 interrupt pin is logic 0.

The interrupt routine could also loop and re-read the configuration register to see if R is still equal to 1 in which case more data can be read.

The character read from the MAX3100 is put into a circular buffer for later retrieval.

There is one flaw in placing this code inside an interrupt routine. If the interrupt is triggered while another SPI session is in progress then both sessions will fail. One solution is to add the interrupt enable/disable code in the SPI_Write_Word function. If a session is active the interrupt routine will be blocked until interrupts are re-enabled.

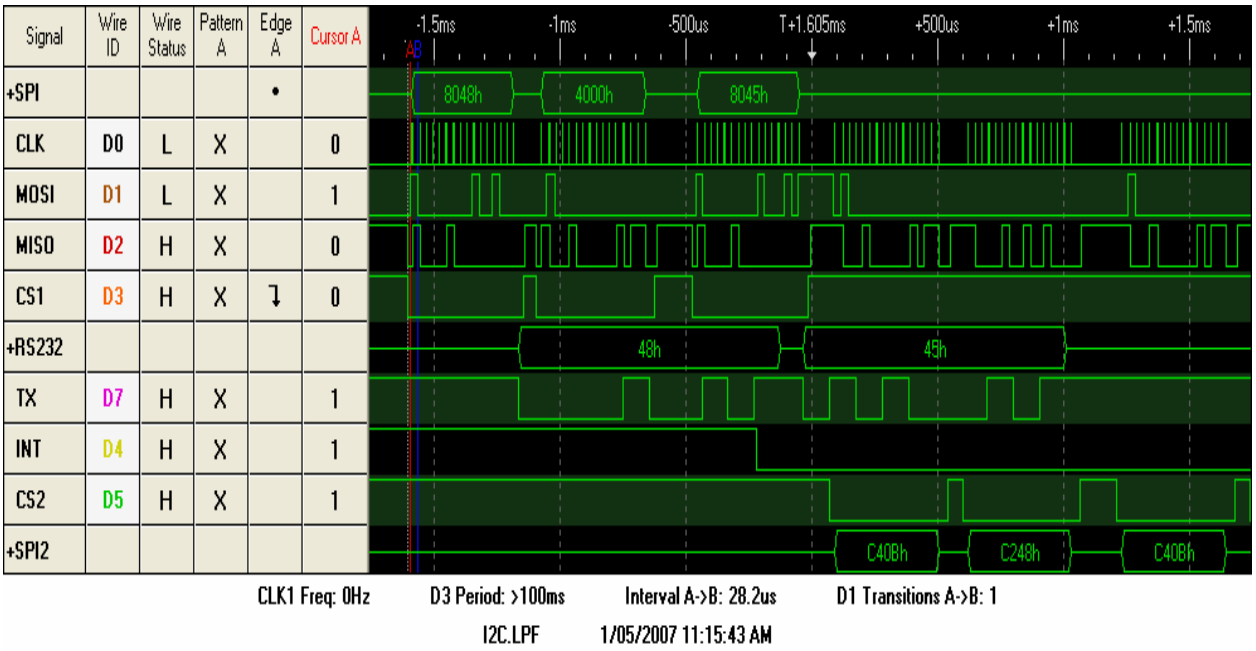

Figure 16 – Complete timing diagram

Figure 16 shows the complete SPI session including the channel 2 interrupt and interrupt SPI access.

Notice that the INT pin remains LO as long as there is data in the MAX3100 FIFO.

Notice also the read of the MAX3100 configuration register (0xC40B on SPI2) to check that the interrupt was in fact caused by this device. If both MAX3100's were sharing the same interrupt then both would need to be read to determine which one needs servicing.
Following the configuration read is 0xC248 where 0x48 or "H", the lower byte, is the asynchronous data received.

Another useful function is to be able to change the state of the RTS pin without actually transmitting data. This is covered in the MAX3100 datasheet.

```
void MAX3100_Set_RS485_Mode(unsigned char device, unsigned int mode)
{
   SPI_Write_Word(device,mode);
}
```

Figure 17 – Controlling the RTS pin

The above code is again just a wrapper of the SPI_Write_Word function.

Once applied it should read something like this.

```
MAX3100_Set_RS485_Mode(MAX3100_RS485_1,MAX3100_WRITE_DATA + MAX3100_RS485_TX +
MAX3100_NO_TX);
```

Figure 18 – Application code for RTS control

The device is selected in the normal way. The RTS bit must be set to the appropriate value. However the TE bit, shown below in the Write Data Register, must be set. This disables transmission resulting in just the RTS pin changing state.

Table 3. Write Data (D15, D14 = 1, 0)

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIN	1	0	0	0	0	$\overline{TE}$	RTS	Pt	D7t	D6t	D5t	D4t	D3t	D2t	D1t	D0t
DOUT	R	T	0	0	0	RA/FE	CTS	Pr	D7r	D6r	D5r	D4r	D3r	D2r	D1r	D0r

Figure 19 – MAX3100 Write Data Register