CML Microcircuits

COMMUNICATION SEMICONDUCTORS

Application Note

C Code For C-BUS

Publication: AN/General/C-Bus/SPICode/1 March 2011

Additional Resources	The Application Note, **C-BUS to SPI Interfacing**, is a guide to the hardware interconnection of these two standard interfaces and is available to download from the Design Support area of the CML website. The Application Note, **C-BUS Microcontroller Interface**, is a guide to the structure of C-BUS transactions and is available to download from the Design Support area of the CML website.

1 Introduction

C-BUS is the communications and control interface almost universally applied across the CML product range. C-BUS is very similar to SPI and can be 'talked to' via any SPI interface using SPI Mode 0. When an SPI interface is not available, it is necessary to use GPIO to talk to C-BUS. This Application Note looks at some simple 'C' core functions to provide an emulation of an SPI function in Mode 0 using 3 GPIO port pins.

2 Contents

3 History

Version	Changes	Date
1.0	First Release	14/03/2011

4 Main Text

The soft SPI port emulation described herein uses the name SPI and standard SPI signal names to refer to the port emulation. C-BUS peripherals are SPI slaves and so the emulation only describes bus mastering.

There are many different ways to write a C function, or set of functions, to achieve a particular purpose. The functions were written as presented because they can be concatenated in any combination and will always result in consistent performance. They are implemented efficiently, compactly and are expected to be free of any compiler dependencies. The functions are written for small host micros. Fast micros may require time delays to achieve the necessary C-BUS timing and this is discussed as each function is presented. These functions may be used, with modifications, as drivers in real-time kernels and multi-tasking operating systems.

5 SPI C Functions

To access the C functions directly go straight to:

7.2 SPI write functions in C
7.4 SPI read functions in C

For setting up simple delays in the C functions to meet C-BUS timing requirements go to:

8 Timing Checks

6 C-BUS Timing

6.1 SPI Mode 0 timing

It is vital to meet the timing constraints given for C-BUS or a design that works in the lab may fail in the field, due, for example to a shift in a crystal supply voltage or a temperature extreme. Not all CML C-BUS peripherals share the same timing so check carefully in the device's Data Sheet.

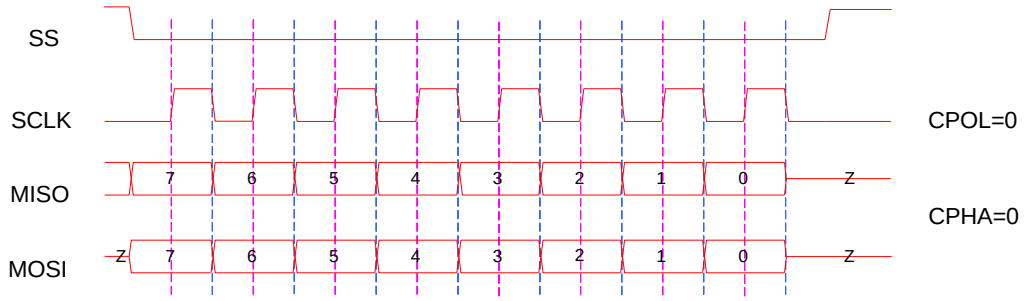

Figure 1. SPI Mode 0 Timing

Figure 1 shows the relative signal timing for SPI Mode 0. This is dissected in the following figures to illustrate the relationship between the clock edges and the required events. C-BUS labelling is added to make the source and destination of the data clearer. The zero level for the traces are indicated by the centre of the trace label.

C-BUS peripherals are always slaves, so the SPI port on the host micro always drives SCLK. SS selects the required peripheral, in this case the CML chip with the C-BUS interface. SS connects to the CSN on this peripheral. The host micro determines which C-BUS port is selected by pulling that C-BUS port's CSN pin low.

Note that the registers on C-BUS peripherals are either read-only or write-only.

6.2 SPI data write to C-BUS

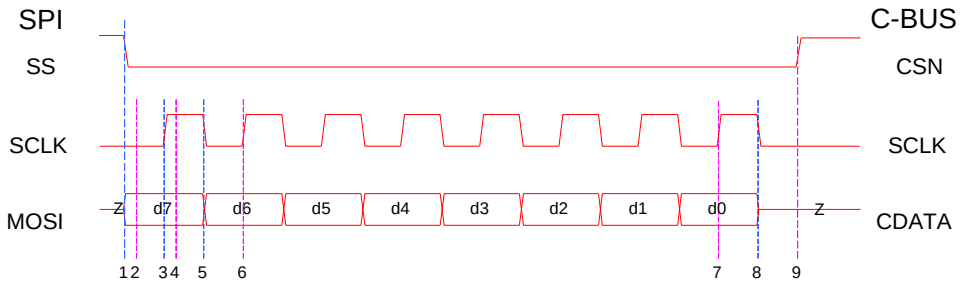

Figure 2. SPI Write to C-BUS

Figure 2 shows the same timing diagram from Figure 1 with the connected C-BUS traces added. SPI SS connects directly to C-BUS CSN and SPI SCLK connects directly to C-BUS SCLK. This is true for the remaining traces.

At the start of a data transfer SPI takes SS low. This causes the first data bit, d7 (the most significant bit of the byte) to be transferred to MOSI. Because MOSI is directly connected to CDATA, d7 will appear at this C-BUS pin. If the level of d7 is different to its previous level, then it will take a finite time for d7 to settle. To allow for this, C-BUS will not read d7 into its internal registers until SCLK goes high. So the order of events to start the data transfer is:

1. Ensure SCLK is low. Take SS low. Write d7 to MOSI.
2. Allow d7 to settle.
3. Take SCLK high.
4. Allow d7 to propagate into the C-BUS registers.

In 1, there are three distinct events and these can happen simultaneously. In practice, it may not be possible to do this with a micro's GPIO pins. Provided the C-BUS timing requirements are met, it does not matter what order these three events are processed in. The only timing constraint applying to these events is the time from the end of event (1) to the first SCLK high edge (3), t_{CSE} in C-BUS timing.
Data bits are read into CDATA on the rising edge of SCLK. They must be written to CDATA and allowed to settle before the rising edge of SCLK. For convenience, the low going edge of SCLK can be used to set the point at which the data bit is written. This is the time from (2) until the next high edge of SCLK (3), t_{CDS}, and is the same for every bit written to CDATA. The data must remain stable on CDATA until after the high of SCLK. This period is t_{CDH}.

On the next low edge of SCLK (5), d6 is transferred to MOSI. On the high edge of SCLK that follows (6), d6 is read into CDATA. On each successive cycle of SCLK there is a low edge when a data bit is transferred to MOSI. This is followed by a high edge when the bit is read from MOSI into CDATA. This process continues until the last data bit, the least significant bit d0, is transferred (7). There are two considerations in timing here; the SCLK cycle time, t_{CK} and the SCLK duty cycle. The SCLK duty cycle is very relaxed and both the high time, t_{CH}, and the low time, t_{CL}, can be as fast as 100ns (on the latest CML devices). The sum of t_{CH} and t_{CL} must never be less than t_{CK}. See Section 9, More on SCLK. So, continuing the order of events from the list above:

5. Take SCLK low and write d6 to MOSI.
6. Take SCLK high. d6 is read into CDATA.

To complete the SPI transaction, (5) and (6) are repeated until the last bit is written read into CDATA, (7). SCLK is taken low and CSN is taken high. The only timing constraint here is that the time from (7) to (9) must not be less than t_{CSH}.

So, to complete the event list:

7. Repeat 5 and 6 above until d0 is transferred to MOSI. Take SCLK high. d0 is read into CDATA.
8. Take SCLK low.
9. Take SS high.

Note that SCLK does not have to be low at the end of a the SPI transaction but must be low at the start for the first high edge to be recognised. For convenience, the algorithms presented will leave SCLK high at the end of each transaction, although the SPI Mode 0 timing in Figure 1 shows SCLK low. True SPI Mode 0 emulation is covered at the end of sections 7.2 and 7.4.

6.3 SPI read data from C-BUS

Reading from C-BUS is a little different in that a write to C-BUS always precedes the read. The hex value of the register to be read is first written and then the data from that register can be read. The SS is held low for the duration of both the write and read cycles to make it one C-BUS transaction. After the C-BUS register address has been written, SCLK is cycled eight times to read the data byte from the register.

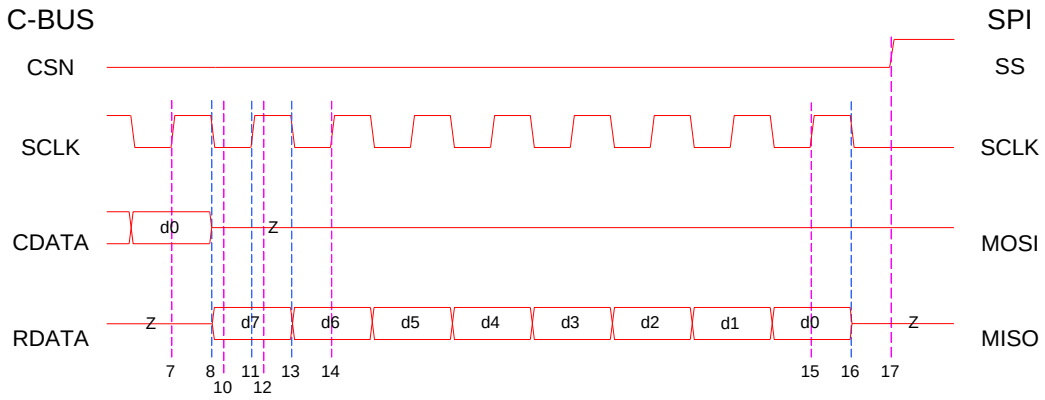

Figure 3. SPI Read of Data from C-BUS

Figure 3 shows the end of the write cycle from Figure 2 when d0, the C-BUS register address, is written to CDATA (7). Note that after this write SCLK continues, CDATA goes high-impedance and RDATA becomes active.

It should be immediately apparent how similar the data transfer on RDATA is to Figure 2. The only real difference is that data is written from C-BUS on the RDATA pin and it is read into the host micro on MISO. Data is written to and read from the bus on the same edges of SCLK. So the order of events is exactly as shown in Section 6.2.

SPI data write to C-BUS. This is followed by the same set of events but using RDATA and MISO:

1. Ensure SCLK is low. Take SS low. Write d7 to CDATA.
2. Allow d7 to settle.
3. Take SCLK high.
4. Allow d7 to propagate into the C-BUS registers.
5. Take SCLK low and write d6 to MOSI.
6. Take SCLK high. d6 is read into CDATA.
7. Repeat 5 and 6 above until d0 is transferred. Take SCLK high. d0 is read into CDATA.
8. Take SCLK low. CDATA will go high-impedance and d7 is written to RDATA by C-BUS.
9. At the end of the write cycle CSN would go high. Do not do this step because we want CSN to remain low. This signals to C-BUS that the SPI transaction has not yet completed.
10. Allow d7 to settle.
11. Take SCLK high.
12. Allow d7 to propagate into the host micro via MISO.
13. Take SCLK low and write d6 to RDATA.
14. Take SCLK high. d6 is read into MISO.
15. Repeat 5 and 6 above until d0 is transferred. Take SCLK high. d0 is read into MISO.
16. Take SCLK low.
17. Take SS high.

The timing constraints are also identical except for the following:

1. The time between (7) and (11), t_{NXT} in Figure 3, can be indefinite but must not be less than one clock cycle, t_{CK}.
2. SS must be held low throughout both the read and write cycle (from 1 in Figure 2 to 17 in Figure 3).

Similarly to the SPI write transaction, SCLK does not have to be low at the end of an SPI transaction but must be low at the start for the first high edge to be recognised. For convenience, the algorithms presented will leave SCLK high at the end of each transaction, although the SPI Mode 0 timing in Figure 1 shows SCLK low. True SPI Mode 0 emulation is covered at the end of sections 7.2 and 7.4.

6.4 Writing more than 1 byte to a C-BUS register

The only instruction that writes one byte to a C-BUS register is the General Reset command. All other commands require a write to specify the C-BUS register address followed by an 8-bit or a 16-bit write.

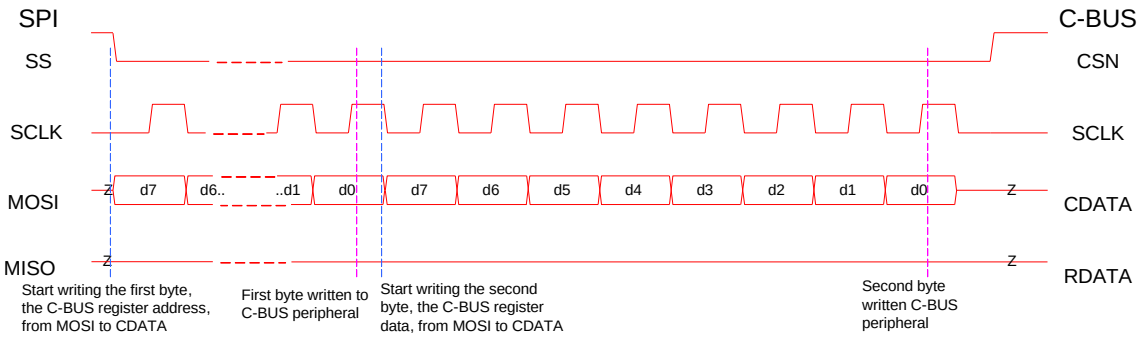

Figure 4. Writing a byte to a C-BUS register

Figure 4 illustrates how two writes from section 6.2 can be concatenated to form an 8-bit write to C-BUS. Note that:

1. SS remains low throughout the write and is de-asserted, taken high, only when both bytes are written.
2. SCLK is continuously cycled.
3. The address byte is written first followed by the data byte in the same bit order.
4. MISO remains hi-impedance.

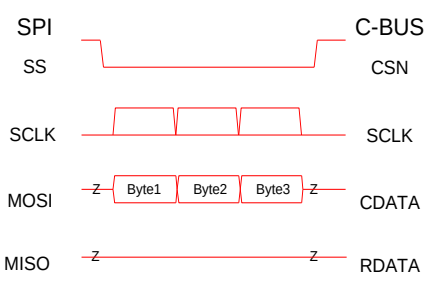

Figure 5. Writing a word to a C-BUS register

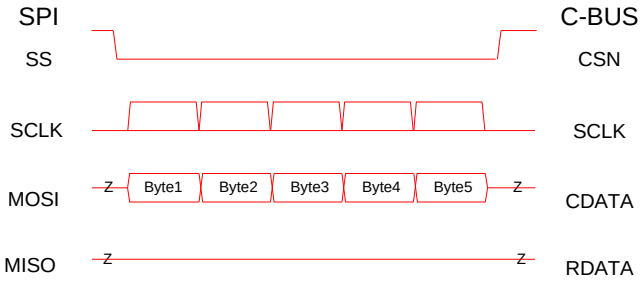

Figure 6. Writing to Streaming C-BUS

Concatenating writes of a data byte to C-BUS can be extended to allow for word writes as shown in Figure 5. Byte1 is the address byte of the C-BUS register and the two following bytes are the word written to that register. SCLK is cycled eight times for each byte written to CDATA.
Streaming C-BUS uses exactly the same principle; a C-BUS write to a register address concatenated with data writes. Figure 6 shows a simplified representation of a streaming write to C-BUS. Byte1 is the address of the C-BUS register and the following bytes constitute the bytes or words written to that register. As long as CSN remains low, more data can be appended. SS takes CSN high to end the transaction. The next write to the Streaming C-BUS register will start with SS going low followed by a write of the Streaming C-BUS register address.

6.5 Reading more than one byte from a C-BUS register

Reading from a C-BUS register requires a single byte write to specify the address of that C-BUS register, followed by sufficient SCLK cycles to clock the data from that register.

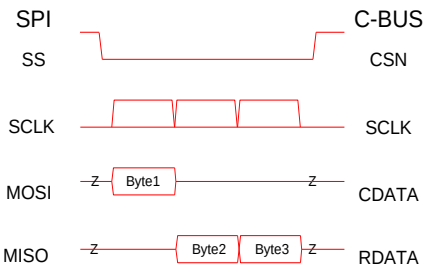

Figure 7. Reading a word from a C-BUS register

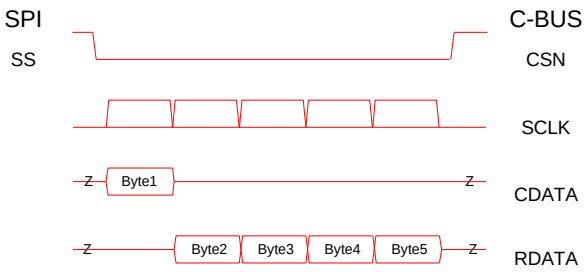

Figure 8. Reading Streaming C-BUS

Figure 7 illustrates a C-BUS read cycle. Byte1 is the address byte of the C-BUS register written to CDATA. Byte2 and Byte3 comprise the word read from that register. Note that the word is read back on RDATA.
Figure 8 illustrates a read on Streaming C-BUS; a C-BUS write to a register address concatenated with data reads. Byte1 is the address of the C-BUS register. Byte2 and the following bytes constitute the bytes or words read from that register. As long as CSN remains low, more data can be read. SS takes CSN high to end the transaction. The next read of the Streaming C-BUS register will start with SS going low followed by a write of the Streaming C-BUS register address.

7 Algorithms for SPI emulation

7.1 Developing the write algorithms

Looking back at Figures 4-6, writing to C-BUS comprises a number of repeated writes of a data byte to CDATA. Each data bit written requires one low to high transition of SCLK to write the data bit into CDATA. This is ideal for a loop construct:

```
loop 8 times
        get data bit n
        write data bit n to MOSI/CDATA
        SCLK=0
        SCLK=1
end loop
```

To manage the SCLK timing, reasonable delays are inserted. For optimum speed the delays can be reduced to allow for instruction, call and return times.

```
loop 8 times
        get data bit n
        write data bit n to MOSI/CDATA
        SCLK=0
        delay 0.5 SCLK cycle
        SCLK=1
        delay 0.5 SCLK cycle
end loop
```

Concatenating this loop for writing two or more bytes from the SPI emulator shows that the SCLK cadence is correct. The data bit is extracted and written to MOSI before taking SCLK low. This gives it the maximum settling time before it is read into CDATA on the rising edge of SCLK.

```
loop 8 times
        get data bit n
        write data bit n to MOSI/CDATA
        SCLK=0
        delay 0.5 SCLK cycle
        SCLK=1
        delay 0.5 SCLK cycle
end loop
loop 8 times
        get data bit n+8
        write data bit n+8 to MOSI/CDATA
        SCLK=0
        delay 0.5 SCLK cycle
        SCLK=1
        delay 0.5 SCLK cycle
end loop
```

The control of CSN is still required so the routine is extended:

```
SS/CSN=0
delay 0.5 SCLK cycle
loop 8 times
        get data bit n+8
        write data bit n+8 to MOSI/CDATA
        SCLK=0
        delay 0.5 SCLK cycle
        SCLK=1
        delay 0.5 SCLK cycle
end loop
delay 0.5 SCLK cycle
SS/CSN=1
```

The delay period from CSN going low to the first high edge of SCLK, t_{CSE}, only occurs at the start of each C-BUS transaction. The delay from the last high edge of SCLK to CSN going high, t_{CSH}, only occurs at the

end of each C-BUS transaction. Half of each delay is handled by the loop, so driving the state of CSN via SS, the other half of these two delays can be handled outside of the core write function. The write function can be called once for the address byte and called again for each data byte to be written. The time delay between each byte of these writes, t_{NXT}, is exactly one SCLK cycle time so there is no need to provide an additional delay between each call of the write function.
So the calling function is now:

SS=0
delay 0.5 SCLK cycle
Call SPI write as many times as required
delay 0.5 SCLK cycle
SS=1

7.2 SPI write functions in C

The core SPI write function can be coded into C as follows. Note that the '0.5 SCLK cycle' delays are not inserted for clarity. Note too, that consecutive C_BUS transactions must be separated by a period of at least t_{CSOFF}.

```
void SPI_wr_byte(unsigned char byte)
{
 unsigned char n;

 for(n=8; n!=0; n--)       //loop 8 times
  {
   if(byte & 0x80)         //test if bit is 1, then copy it to MOSI
     MOSI=1;               //yes
   else
     MOSI=0;               //no
   SCLK=0;                 //take SCLK low
   byte <<= 1;             //get the next bit into position for copying
   SCLK=1;                 //take SCLK high
  }
}
```

To send General Reset to C-BUS:

```
void SPI_wr_gen_reset()
{
 SS=0;                     //take SS low to start the SPI transaction
 SPI_wr_byte(0x01);        //write the C-BUS address for General Reset
 SS=1;                     //take SS high to end the SPI transaction
}
```

This function writes an 8-bit byte <databyte> to C-BUS register <address>:

```
void SPI_wr1(unsigned char address, unsigned char databyte)
{
 SS=0;                     //take SS low to start the SPI transaction
 SPI_wr_byte(address);     //write the C-BUS address byte
 SPI_wr_byte(databyte);    //write the C-BUS data byte
 SS=1;                     //take SS high to end the SPI transaction
}
```

For writing a word to a C-BUS register address.

```
void SPI_wr2(unsigned char address, unsigned int dataword)
{
 unsigned char temp;

 SS=0;                     //take SS low to start the SPI transaction
 SPI_wr_byte(address);     //write the C-BUS address byte
 temp=dataword;            // the LSB of the data word
 dataword >>= 8;           //shift most significant 8 bits down for MSB
 SPI_wr_byte((unsigned char)dataword);         //write MSB
 SPI_wr_byte(temp);        //write LSB
```

```
 SS=1;                      //take SS high to end the SPI transaction
}
```

For writing to Streaming C-BUS from a soft buffer/FIFO:

```
void SPI_wr_stream(unsigned char address, unsigned char *buffer, unsigned int
length)
{
 unsigned int n

 SS=0;                      //take SS low to start the SPI transaction
 SPI_wr_byte(address);      //write the C-BUS address byte
 for (n=0, n<length, n++)         //write <length> bytes from buffer[]
       SPI_wr_byte(*(buffer+n));
 SS=1;                      //take SS high to end the SPI transaction
}
```

For exact emulation of SCLK in SPI Mode 0, add the instruction, SCLK=0 at the end of each of the function calls that generate an SPI transaction. This makes very little difference to the time it takes for any of the SPI transactions to complete. For example:

```
void SPI_wr1(unsigned char address, unsigned char databyte)
{
 SS=0;                      //take SS low to start the SPI transaction
 SPI_wr_byte(address);      //write the C-BUS address byte
 SPI_wr_byte(databyte);     //write the C-BUS data byte
 SS=1;                      //take SS high to end the SPI transaction
 SCLK=0                     //ensure SCLK is low on return
}
```

For information on checking the timing of these routines go to Section 8 Timing Checks.

7.3 Developing the read algorithms

It is possible to create the functions to read the C-BUS using similar methods to the write functions. Figures 7 and 8, show that reads of the C-BUS consists of a byte write to C-BUS followed by one or a number of repeated reads of a data byte from RDATA. Each data bit read requires one low to high transition of SCLK to read the data bit from RDATA. This is used in another loop construct:

```
loop 8 times
        SCLK=0
        delay 0.5 SCLK cycle
        SCLK=1
        copy the next data bit from MISO/RDATA
        store the bit
        delay 0.5 SCLK cycle
end loop
```

The write function is called once for the address byte and the read function called for each data byte to be read from the register. Inserting the core write function shows that the required timing for SCLK is being met and that the data set-up and hold times are maximised throughout

```
loop 8 times
        get data bit n
        write data bit n to MOSI/CDATA
        SCLK=0
        delay 0.5 SCLK cycle
        SCLK=1
        delay 0.5 SCLK cycle
end loop
loop 8 times
        SCLK=0
        delay 0.5 SCLK cycle
        SCLK=1
        copy the next data bit from MISO/RDATA
        store the bit
```

```
        delay 0.5 SCLK cycle
end loop
```

Again, the delay periods t_{CSE} and t_{CSH} occur respectively, only at the beginning and end of each C-BUS transaction so CSN and these two delays are handled outside of the core C-BUS read function. The read function is called for each data byte to be read. The time delay between each byte transfer, t_{NXT}, is exactly one SCLK cycle time so there is no need to provide a delay before calling the write function. Extending the routine for the control of CSN:

```
SS/CSN=0
delay 0.5 SCLK cycle
loop 8 times
        get data bit n
        write data bit n to MOSI/CDATA
        SCLK=0
        delay 0.5 SCLK cycle
        SCLK=1
        delay 0.5 SCLK cycle
end loop
loop 8 times
        SCLK=0
        delay 0.5 SCLK cycle
        SCLK=1
        copy the next data bit from MISO?RDATA
        store the bit
        delay 0.5 SCLK cycle
end loop
delay 0.5 SCLK cycle
SS/CSN=1
```

7.4 SPI read functions in C

The core SPI write function can be coded into C as follows. Note that the '0.5 SCLK cycle' delays are not inserted for clarity. Note too, that consecutive C_BUS transactions must be separated by a period of at least t_{CSOFF}.

```
unsigned char SPI_rd_byte(void)
{
 unsigned char byte, unsigned int n;

 for(n=8; n != 0; n--)
  {
   SCLK=0;                  //take SCLK low
   byte <<= 1;              //shift a 0 into b0 ready to build RDATA byte
   SCLK=1;                  //take SCLK high
   if(MISO)                 //append 1 to <byte> if RDATA=1
      byte |= 0x01;
  }
 return(byte);              //return the byte read from C-BUS
}
```

To read a byte from a C-BUS register address

```
unsigned char SPI_rd1(unsigned char address)
{
 unsigned char rbyte;

 SS=0;                      //take SS low to start the SPI transaction
 SPI_wr_byte(address);      //write the C-BUS address byte
 rbyte = SPI_rd_byte();     //read the C-BUS reply byte on MISO
 SS=1;                      //take SS high to end the transaction
 return(rbyte);
}
```

To read a word from a C-BUS register

```
unsigned int SPI_rd2(unsigned char address)
{
 unsigned int rword;

 SS=0;                      //take SS low to start the SPI transaction
 SPI_wr_byte(address);      //write the C-BUS address byte
 rword = (unsigned int) SPI_rd_byte();  //copy 1st read byte to LSB
 rword <<= 8;               //left-shift bits into MSB
 rword |= (unsigned int) SPI_rd_byte();       //copy 2nd read byte to LSB
 SS=1;                      //take SS high to end the transaction
 return(rword);
}
```

To read streaming C-BUS

```
void SPI_rd_stream(unsigned char address, unsigned char *buffer, unsigned int
length)
{
 unsigned int n

 SS=0;                      //take SS low to start the SPI transaction
 SPI_wr_byte(address);      //write the C-BUS address byte
 for (n=0, n<length, n++)   //write <length> bytes from buffer[]
       *(buffer+n)=SPI_rd_byte();
 SS=1;                      //take SS high to end the SPI transaction
}
```

For information on checking the timing of these routines go to Section 8 Timing Checks.

For exact replication of SCLK in SPI Mode 0, add the instruction, SCLK=0 at the end of each of the function calls that generate an SPI transaction. This makes very little difference to the time it takes for any of the SPI transactions to complete. For example:

```
unsigned char SPI_rd1(unsigned char address)
{
 unsigned char rbyte;

 SS=0;                      //take SS low to start the SPI transaction
 SPI_wr_byte(address);      //write the C-BUS address byte
 rbyte = SPI_rd_byte();     //read the C-BUS reply byte on MISO
 SS=1;                      //take SS high to end the transaction
 SCLK=0                     //ensure SCLK is low on return
 return(rbyte);
}
```

8 Timing Checks

The pseudo code routines show where to insert delays to ensure the SPI emulation is compatible with C-BUS. Check the timing in the functions SPI_wr1() and SPI_rd1(). This should be sufficient to cover the other functions with the exception of the streaming routines. Due to the array handling, the timing will be slightly different. Here is an example for applying a timing check of the SPI_rd1 function in the code simulator. This method should not be used for precise timing but can be used to ensure that a reasonable timing overhead is available. These are the timing path to check.

1. At the start of the SPI transaction; SS=0 to first SCLK=1 - t_{CSE}
2. From this SCLK=1 to the first SCLK=0 - t_{CH}
3. From this SCLK=0 to the next SCLK=1 - t_{CL}
4. The sum of 2 and 3 should not be less than t_{CK}
5. From the last SCLK=1 of the first loop to the first SCLK=0 of the following loop - t_{NXT}
6. From this SCLK=0 to the next SCLK=1 - t_{CL}
7. From this SCLK=1 to the next SCLK=0 - t_{CH}
8. The sum of 6 and 7 should not be less than t_{CK}
9. From the last SCLK=1 to SS=1 - t_{CSH}

Here is the function again with the approximate delays included.

```
unsigned char SPI_rd1(unsigned char address)
{
 unsigned char rbyte;

 SS=0;                      //take SS low to start the SPI transaction
 half_cycle_delay()
 SPI_wr_byte(address);      //write the C-BUS address byte
 rbyte = SPI_rd_byte();     //read the C-BUS reply byte on MISO
 SS=1;                      //take SS high to end the transaction
 half_cycle_delay()
 return(rbyte);
}
```

This function is expanded in Figure 9 to show the points between which timing measurements should be made. This method should not be used to set precise timing but can be used where a generous allowance is given for the delays.

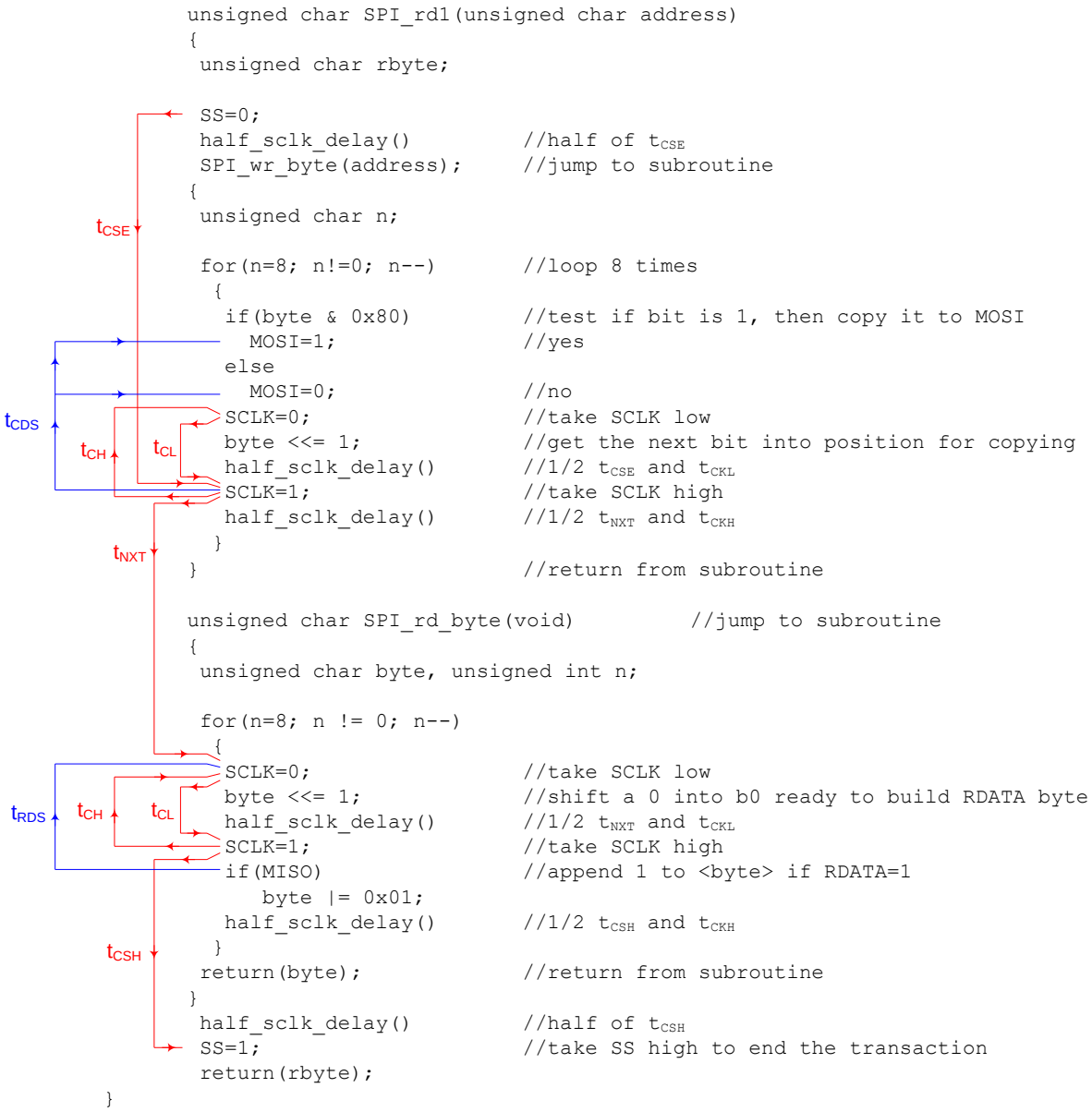

Figure 9. Timing checks in function SPI_rd1()

Ideally, the SPI timing should be checked thoroughly with a protocol analyser to ensure that it meets the requirements of the C-BUS peripheral that is being connected. SPI emulation that runs close to or at the maximum specified capability of C-BUS should be checked across the required temperature range in conjunction with a full-range power supply check. If a logic analyser or scope is available, then use this to trace the C-BUS transactions. Set up the code to run each C-BUS transaction multiple times with dummy data. Set the logic analyser to find violations in setup and hold times. Run the hardware for a period of time at the corners of temperature and supply voltage.
Alternatively, use very generous timings to avoid any possible issues. This will be at the expense of using a faster micro than is actually required.

NOTE that C-BUS data transfer rates have been increasing through newer generation devices. Please check the C-BUS specifications carefully from the latest data sheet published on the CML website.

9 More on SCLK

The SCLK cycle time, t_{CK} is the time to complete one full cycle of SCLK. This equates to an SCLK frequency of up to 5Mhz for most CML devices or up to 10MHz for the latest devices. Earlier parts may have a lower limit on SCLK so check the timing details. The SCLK duty cycle is the ratio of high to low time. The SCLK duty cycle is very relaxed and both the high time, t_{CH}, and the low time, t_{CL}, can be as fast as 100ns (on the latest CML devices). The sum of t_{CH} and t_{CL} must never be less than t_{Ck}. It is permitted to make SCLK completely asymmetrical, for example a 100ns low time and a 10ms high time.
There are a number of tradeoffs in selecting the speed of SCLK. The host micro will be handling other tasks in between servicing C-BUS and this will dictate how much time is available for C-BUS transfers. It is possible to adjust SCLK to run C-BUS as slowly as possible to maximise power saving but power savings may be greater if C-BUS is burst read at a high rate and the host micro idled between tasks. It is certainly easier to start with a high-speed micro and then tune speed downward, perhaps selecting a slower device in the final design build.

When simulating an SPI port, it is likely that SCLK will not be symmetrical and the symmetry will vary in different parts of the algorithm. This does not matter providing you check three things:

1. Find the fastest SCLK cycle and identify the time that this clock cycle is high and low for.
2. Find the shortest high time for SCLK.
3. Find the shortest low time for SCLK.

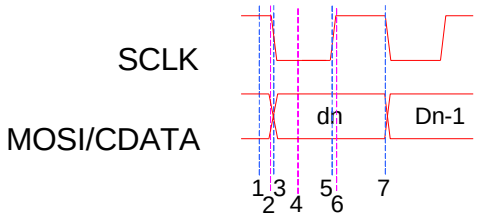

Figure 10. Data Setup and Hold, C-BUS Write

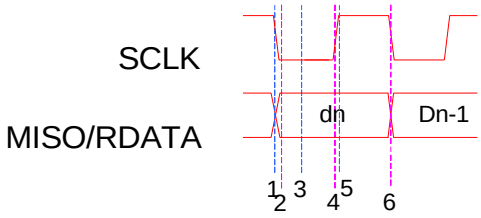

Figure 11. Data Setup and Hold, C-BUS Read

Figure 10 shows the data setup and hold timing when writing to C-BUS.

1. The SPI write function processes the steps to write data bit **n** to MOSI.
2. The data write is completed in firmware and the host sends bit **n** to the GPIO.
3. The SPI write function takes SCLK low.
4. Data bit **n** propagates onto MOSI and settles.
5. The SPI write function takes SCLK high.
6. Data bit **n** read into the C-BUS port is triggered.
7. The SPI write function processes the steps to write data bit **n-1** to MOSI.

Reference (5) in Figure 10 is the point at which data bit **n** must be stable. From (5) to (3) is therefore the available data setup time, t_{CDS}.
Reference (7) in Figure 10 is the point at which the next data bit, **n-1**, will have been processed and is written to the GPIO. Data bit **n** currently on CDATA will be overwritten, so the C-BUS must have completed the read of **n** before this point. From (7) to (5) is therefore the data available hold time, t_{CDH}.

Figure 11 shows the data setup and hold timing when reading from C-BUS.

1. The SPI read function takes SCLK low.
2. When C-BUS detects the low on SCLK, C-BUS writes bit **n** to RDATA.
3. Data bit **n** propagates onto RDATA/MISO and settles.
4. The SPI read function takes SCLK high.
5. The SPI read function reads data bit **n** into the host from the MISO pin.
6. The SPI read function takes SCLK low ready to read the next bit, **n-1**, from C-BUS.

Reference (5) in Figure 11 is the point at which data bit **n** must be stable. From (5) to (3) is therefore the available data setup time, t_{RDS}.
Reference (6) in Figure 11 is the low edge of SCLK that triggers C-BUS to begin to output the next data bit, **n-1**. Data bit **n** currently on RDATA will be overwritten, so the SPI must have completed the read of **n** before this point. From (6) to (5) is therefore the available data hold time, t_{RDH}.

Check each of the four periods to make sure that the data setup and hold times do not exceed the minimum times given in the data sheet.
These checks can be made quite easily by setting the software simulator to run your C-BUS transactions multiple times. Put the minimum setup and hold times in an expression evaluation so if they are exceeded the simulator will signal this. Allow a generous overhead to allow for timing changes in your hardware.

CML Microcircuits (UK) Ltd COMMUNICATION SEMICONDUCTORS	CML Microcircuits (USA) Inc. COMMUNICATION SEMICONDUCTORS	CML Microcircuits (Singapore) Pte Ltd COMMUNICATION SEMICONDUCTORS
Tel: +44 (0)1621 875500 **Fax:** +44 (0)1621 875600 **Sales:** sales@cmlmicro.com **Tech Support:** techsupport@cmlmicro.com	**Tel:** +1 336 744 5050 800 638 5577 **Fax:** +1 336 744 5054 **Sales:** us.sales@cmlmicro.com **Tech Support:** us.techsupport@cmlmicro.com	**Tel:** +65 62 888129 **Fax:** +65 62 888230 **Sales:** sg.sales@cmlmicro.com **Tech Support:** sg.techsupport@cmlmicro.com
- www.cmlmicro.com -		