

# Interfacing Slow Peripherals to **D. Modules** via SPI

Doc #1.0

## Application Note

**D. SignT**

Digital Signalprocessing Technology

Norbert Nölker & Adolf Klemenz GbR

Gelderner Straße 36

D - 47647 Kerken

phone +49 (0) 2833 / 570 977

fax +49 (0) 2833 / 33 28

email [info@dsignt.de](mailto:info@dsignt.de)

www <http://www.dsignt.de>

## **D.Module.SPI Revision History**

1.0 July 1999

SPI is a trademark of Motorola Inc.

**1 INTRODUCTION..... 3**

**2 EXAMPLE: INTEL 82527 CAN CONTROLLER..... 4**

**3 SPI INTERFACE ..... 6**

3.1 Interrupt driven Data Transfer ..... 7

## **1 Introduction**

A typical DSP system not only has to acquire data and process it, but also requires secondary devices like temperature sensors, real time clocks (RTCs), digital potentiometers for calibration, or communication channels to receive and pass process control information to and from other processors. The tremendous performance increase of today's DSPs allows to handle typical control tasks by the DSP itself without support by a host processor.

However, most standard communication controllers, RTCs and similar devices are designed to interface to 8 and 16 bit micro controllers. Bus cycles are very slow compared to a DSP: access times of hundreds of nanoseconds are quite common, compared to a typical 30 nsecs bus cycle time of a DSP. Several devices also use multiplexed data and address busses to reduce pin count and provide a glueless interface to common micro controllers like the 8051 and 68HC11 types. DSPs use non-multiplexed busses to increase throughput, thus additional multiplexers and latches are required to connect such peripherals. The 'slow' bus interfaces may cause severe problems if high-speed data acquisition and/or intense use of external memory is a requirement. If for example the DSP has to read a four channel A/D converter at 1 MHz sampling frequency per channel, the external bus of the DSP must not be locked by other devices for more than 200 nsecs, otherwise data may be lost. As an alternative, FIFOs may be used to buffer the A/D data stream at the expense of increased system complexity and costs.

The SPI interface provides an alternative method to connect 'slow' peripherals to a DSP system. Implemented in the D.Module's user-programmable CPLD, the SPI interface is accessed with no wait states in a single 20..30 nsecs. bus cycle. Many communication controllers, RTCs and other secondary peripherals feature a SPI interface. As an example, let's take a look at the Intel 82527 CAN controller:

## **2 Example: Intel 82527 CAN Controller**

The 82527 offers four parallel bus configurations, three of them for multiplexed busses which are difficult to use with DSPs. A fourth mode allows non-multiplexed operation. The minimum bus cycle time in this mode is 90 nsecs, but access time for many registers may be as high as 290 nsecs or even 538 nsecs if a write preceded the read operation. Alternatively, a double-read mechanism may be used: the CAN controller's register is read in a fast cycle, the result is discarded, and valid data is read from a high-speed read register in a second cycle. Besides this, constraints for successive write operations exist: a delay of at least 250 nsecs must be inserted between write accesses. To guarantee the required timings, the 82527 DSACK0# signal may be used to request wait states from the DSP, but this may cause conflicts with high-speed data acquisition requirements.

A SPI interface however will relax this situation: the 82527 accepts a serial clock as high as 4 MHz, resulting in a 500 kByte/sec data transfer rate. This is more than 4 times the maximum usable bit rate of the CAN bus and sufficient to communicate with maximum throughput. The serial protocol used by the 82527 uses one address byte, followed by a control byte, and up to 15 bytes of data. This allows to read or write an entire message object. The DSP can set-up the message and transmit it interrupt controlled as a background task. SPI communication requires the master (the DSP) to write data to the SPI interface even if only data should be read from the slave (the 82527). Reading a complete message object will occupy the DSP external bus for 34 zero-wait-state cycles: 17 \* (1 write + 1 read). The following table compares the external bus occupation of the three possible interfaces.

<b>SPI</b>	<b>parallel standard</b>	<b>parallel with double-read</b>
34 zero wait state cycles (30 nsecs assumed)	17 * 290 nsecs. cycles	30 * 90 nsecs. cycles
1020 nsecs	4350 nsecs.	2700 nsecs.

*Table 2-1 Bus Cycle Time to read complete message object*

The SPI implementation dramatically reduces external bus load. No precautions have to be taken for successive writes as the serial data stream is 'slow' enough to guarantee a sufficient delay. Finally, the SPI interface offers additional advantages: no bus drivers, address decoders or other glue logic are required.

### 3 SPI Interface

The SPI interface is based on a 8 bit shift register. The shift clock (SCK) is provided by the master device. SCK is a gated clock and is only generated during shifting. SCK stays idle between transfers. Transmitting and receiving occurs simultaneously: While the master shifts out its transmit data, data from the slave is shifted in. As a result, the master must always send data in order to generate clocks, even if only data reception is required. The following diagram shows a basic interface and a sample data transfer: Data on SDO is shifted out with the falling SCK edge, data on SDI is sampled on the rising SCK edge. The SCK idle polarity is 'High'.

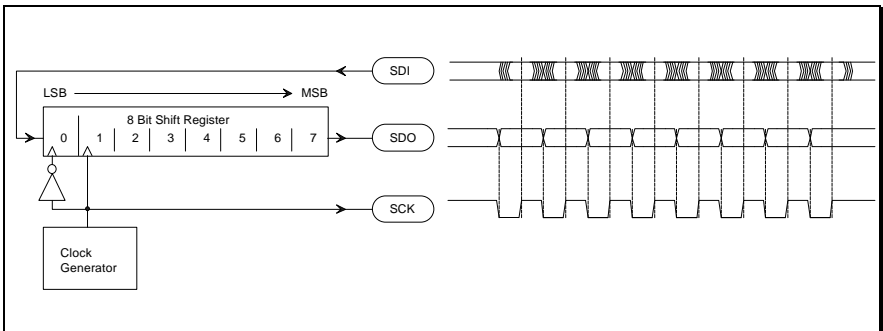


Figure 3-1 Basic SPI Interface

For the entire D.Module family of DSP boards a SPI implementation for the user-programmable CPLD is provided with the support software: as ready-to use JEDEC files, in ABEL source code and with DSP test software. This interface operates in master mode, shifting starts and the serial clock is generated as soon as the DSP writes to the SPI interface. Data transmission is 8 bit, MSB first. A transmit buffer is integrated which stores data written by the DSP until the next transmission is possible. Also included are the shift register, a state machine to control the processes, a pre-scaler to derive SCK from the DSP system clock and/or an additional receive buffer.

The pre-scaler can generate SCK frequencies at 1/2 to 1/16 of the DSP system clock. Alternatively an input clock at twice the desired SCK frequency may be provided externally. In case of the 82527 CAN controller, it's CLOCKOUT signal may be used to provide an 8 MHz clock, resulting in the desired 4 MHz SCK frequency.

The SPI interface generates an interrupt to the DSP after the last bit has been shifted. In uni-directional communication, i.e. if the DSP only writes to the peripheral, DMA can be used too.

### 3.1 Interrupt driven Data Transfer

A sample program to transmit and receive data via SPI is presented below. Please note that some instructions are 'pseudo code' to stay independent of the processor platform. These instructions have to be replaced with processor and compiler specific instructions, e.g. to enable interrupts.

```
/******  
/* Macros and Constants */  
/******  
#define SPI *(char*) CPLD_BASE  
#define BUFFLEN 20  
  
/******  
/* global variables for SPI */  
/******  
bool spi_rdy;          /* flag: transmission complete */  
int spi_cnt;          /* transmission counter */  
int spi_length;       /* transmission length */  
char spi_tx[BUFFLEN]; /* transmit buffer */  
char spi_rx[BUFFLEN]; /* receive buffer */
```

```
/* ***** */
/* SPI interrupt handler */
/* ***** */
void spi_int_func (void)
{
    /* ***** */
    /* read data from SPI, increment counter */
    /* if the entire message buffer is processed */
    /* disable further interrupts and set the */
    /* ready flag */
    /* ***** */
    spi_rx[spi_cnt++] = SPI & 0xFF;
    if (spi_cnt == spi_length)
    {
        spi_cnt = 0;
        disable_interrupt (SPI_INT);
        spi_rdy = true;
    }
    else
    {
        SPI = spi_tx[spi_cnt];
    }
}

/* ***** */
/* Main Program */
/* ***** */
void main (void)
{
    int i;

    /* ***** */
    /* install interrupts, initialise global vars */
    /* ***** */
    install_interrupt (SPI_INT, spi_int_func);
    spi_cnt = 0;
    spi_rdy = false;

    /* ***** */
    /* setup SPI message */
    /* ***** */
    spi_tx = "Hello SPI peripheral";
}
```

```

/*****
/* transmit SPI message          */
/* the first byte is transmitted manually */
/* from now on interrupts are generated */
/*****
spi_length = BUFFLEN;
SPI = spi_tx[0];
enable_interrupt (SPI_INT);

/*****
/* do something useful meanwhile      */
/* then wait until SPI transmission is done */
/* then reset ready flag              */
/*****
while (! spi_rdy) ;
spi_rdy = false;

/*****
/* read the receive buffer */
/*****
...
}

```