USC-3025/USC-3050

USB to SPI Isolated Protocol Adapter

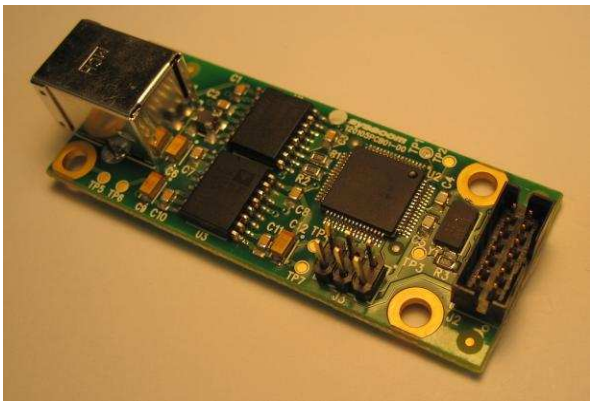
DATASHEET

Rev1.0
May 2012

sysacom
www.sysacom.com

Features

- Bi-directional USB to SPI converter
- 2.5kV and 5kV galvanic isolation available
- USB 2.0 compatible
- Lead-Free / ROHS compliant
- Provide up to 70mA on the 3.3V on the SPI side.



Applications

- Continuous data streaming in bulk transfer
- Data acquisition
- Instrumentation isolation
- Communication
- Medical application

Description

Using the USC-3025 or USC-3050 is an easy way to add a USB host to any device that features an SPI port. It electrically isolates the USB side from the SPI side without any loss of speed. The USC-30xx can be used in various types of applications where a USB host is necessary such as, a data streaming device, a data acquisition device, an instrumentation device or a communication device.

The USC-30xx has a standard SPI port and allows the user to connect with a device in either master or slave mode. It has an interrupt output (INT) to indicate to the SPI master device that the data is ready to send and an output pin (GPX) to indicate the completion of the USB enumeration. The USC-30xx supports the USB 2.0 standard and connects to the host through a type B USB connector. Circular buffers simplify the software interface of the USC-30xx SPI port and USB port.

The USC-30xx is easy to configure with simple protocol commands for the USB port. For more information on the programming of the USC-30xx see the User guide.

The USC-3025 is made with parts that are in accordance with UL recognition 2500 V rms for 1 minute per UL 1577, CSA Component Acceptance Notice #5A IEC 60747-5-2 (VDE 0884, Part 2) VIORM = 560V peak. This device is recommended for lower cost application.

The USC-3050 is made with parts that are in accordance with UL recognition 5000 V rms for 1 minute per UL 1577, CSA Component Acceptance Notice #5A IEC 60747-5-2 (VDE 0884-10):200612, IEC 60601-1: 250 V rms, IEC 60950-1: 400 V VIORM = 846V peak. This device is recommended for medical application.

Contents

Revision History

Revision Number	Revision Date	Description of Changes
01	05/23/2012	Initial release

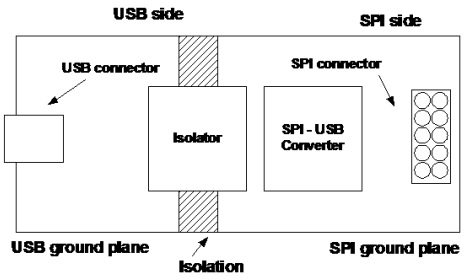
1. Electrical Characteristics

Specifications	USC-3025	USC-3050	Unit
Performance			
USB Compliance	2.0	2.0	
Max. Data Rate ¤			
Data < 835 bytes	1.5	1.5	Mbps
Data > 835 bytes	500	500	Kbps
Environment			
Operating Temp Range	-40 to 85°C	-40 to 85°C	°C
Electrical			
Supply Voltage (from USB)	5.0	5.0	V
Min. Supply Current (from USB)	60	60	mA
Max. Supply Current (from USB)	260	260	mA
Output Voltage SPI Side	3.3	3.3	V
Max. Output Current SPI Side	70	70	mA
Electrical isolation between USB side and SPI side	2.5	5.0	KV
Physical			
Weight	12	12	Grams

Connectors		
Pin	J1[1]	J2[2]
1	VCC	+3.3V
2	D+	MOSI
3	D-	GND
4	GND	SCLK
5		GND
6		SS
7		GND
8		MISO
9		GND
10		INT
11		GND
12		GPX

1- USB Type B connector
2- Molex 87831-1220
Suggest female connector for J2 :87568-1273

2. Principle of operation



The USC-30xx units are built using a Sysacom proprietary USB firmware. This software is used to interface from USB to SPI protocols. On the USC-30xx the galvanic isolation has been place in front of the converter core as an improvement over our previous generations the USC-216. The device is powered from the USB port and generates its own 3.3V for the SPI side. The converter can also be used to power the customer end devices.

Only USB bulk transfer mode is supported by the USC-30xx. The units have two endpoints, one IN and one OUT. This allows for a bi-directional USB communication. Please refer to User Guide for detailed Bulk Mode transfer information.

The converter has two circular buffers that can stock up to 835 bytes before sending through the data to the SPI and USB ports. If the number of bytes to send is less than 835, the converter will stock the data and send it after it has received the last byte. In this case, the SPI port should be set at a baud rate lesser than 1,5Mbps. However, when a packet of data higher than 835 bytes is required the converter can manage both communications simultaneously but the baud rate will need to be decreased to 500 Kbps while in continuous mode.

3. DLL content

This section is focused on helping you to use the DLL for communicating with the USC-30xx.
The DLL will be provided on a CD-ROM with your USC-30xx. For more information please refer to USC-216 user guide.

The DLL file is named USC_216_DLL_REV2_02.dll and shall be copied in the WINDOWS\system32 folder with the lib file.

The functions supported by the DLL are:

```
DWORD InitComm( void );
DWORD CloseComm( void );
DWORD GetReadState( void );
DWORD SendData( BYTE*, DWORD );
WORD GetNbReceivedData( void );
DWORD GetReceivedData( BYTE*, DWORD );
DWORD SendCommand( BYTE*, BYTE, BYTE* );
```

All those functions are defined in the USC_216_DLL.h and you can find this file in the Converter USB to SPI Rev2 folder.

The next section explains how to use all those functions.

The DLL has 2 circular buffers to store the packet received from the USC-30xx. The first one is for data and the second is for a command. The length of the data circular buffer is 2432 bytes either the size of two times the maximum number of byte that can be transferred in a millisecond. Those buffers are not accessible directly outside the DLL.

All the commands, values used to communicate with the USC-30xx and values of the error message returned by the DLL functions are defined in the header file.

a. InitComm() detail

Prototype : DWORD InitComm(void);

This function is used to initialize the communication with the USC-30xx. First of all, this function creates the link between the driver and the DLL. With this link the handles (IN and OUT) are initialized to communicate with the driver. When both handles is initialized correctly, the thread to read the USC-30xx in continuous is created and started.

Error Returned	VALUE	Description
ERR_SUCCESS	0x00	The communication with the USC-30xx is done
ERR_INVALID_HANDLE	0x04	The handles were not initialized correctly. The communication is not done.
System error code	xxxx	The error code returned after an error with the creation of the thread. The error system code is defined in the WinError.h file or to this Web address : http://msdn.microsoft.com/en-us/library/ms681381(VS.85).aspx

Table 1: InitComm error Message

Example usage:

```
if( InitComm() == ERR_SUCCESS )
      /* Then the communication is done… */
```

b. CloseComm() detail

Prototype: DWORD CloseComm(void);

This function closes the communication between the application program and the USC-30xx driver. It kills the thread and closes both handles (IN and OUT).

Error Returned	VALUE	Description
ERR_SUCCESS	0x00	The communication with the USC-30xx is correctly closed
ERR_INVALID_FUNCTION	0x05	A problem is arrived during the closure and the communication is not correctly closed.

Table 2: CloseComm error Message

Example usage:

```
if( CloseComm() == ERR_SUCCESS )
      /* The communication is closed, thus the application can be close… */
```

c. GetReadState() detail

Prototype: DWORD GetReadState(void);

This function returns the state of the thread used to read the USC-30xx and the circular buffer used to stored data came from the USC-30xx. This function must be call in a fixed time interval.

Error Returned	VALUE	Description
ERR_SUCCESS	0x00	The thread and the data circular buffer is functional
ERR_CB_OVERLOAD	0x08	The circular buffer used to store the data from USC-30xx is overloaded and some data can be lost.
ERR_THREAD_STOP	0x09	The reading thread is stopped and the communication must be stopped. The thread can be stop if the USC-30xx is disconnected of the computer

Table 3: GetReadState error Message

Example usage:

```
interrupt 10ms_ISR ()
{
	if( GetReadState () == ERR_SUCCESS )
		/* Then the execution may continue */

	if( GetReadState () == ERR_CB_OVERLOAD )
		/* Then stop reading USC-220 and reset data circular buffer */

	if( GetReadState () == ERR_THREAD_STOP )
		/* Then reset the communication with the USC-220 */
}
```

d. SendData() detail

Prototype: DWORD SendData(BYTE*, DWORD);

This function is used to send data to the SPI port through the USC-30xx. The USC-30xx has a circular buffer to store all bytes want to send on the SPI port. The size of this circular buffer is 834 bytes. This function can't send more that 834 bytes at a time but if this function is called several times in rehearsal thus you can send all bytes desired. The SendData function increases the number of data sent each time the function is called until the entire data are send. At this moment the function will return ERR_SUCESS. This method allows returning to the principal application to execute something else during the USC-30xx releases some place in the circular buffer. To send all byte desired this function must be placed in a loop.

Parameter	Description
BYTE*	This parameter points on the start address of the data to send
DWORD	This parameter indicate the number of byte to send to the USC-30xx

Table 4: SendData parameter

Error Returned	VALUE	Description
ERR_SUCCESS	0x00	The transfer is over and no error is occurred
ERR_NOT_COMPLET	0x01	The transfer is not completed and the function must to call again
ERR_WRITE_FAULT	0x02	An error occurred during the transfer.

Table 5: SendData error Message

Example usage:

```
NbDataToSend = 1000;              /* Number of data to send = 1000 */
OutBuffer[NbDataToSend];          /* Buffer to send data */

OutBuffer[0] = WRITE_DATA_CMD;    /* First byte is the command */

for( int i = 1; i < NbDataToSend; i++ )
       OutBuffer[i] = data;       /* Fill the data buffer */

Error = ERR_NOT_COMPLET;          /* Reset the error */
while( Error != ERR_SUCCESS)      /* Loop until all data are sent */
{
       Error = SendData( OutBuffer, NbDataToSend );   /* Send the data */

       if( Error == ERR_WRITE_FAULT )
              /* Stop the transfer ans log the error… */

       if else( Error == ERR_NOT_COMPLET )
              /* Can do something else… */
}
```

e. GetNbReceivedData() detail

Prototype: WORD CLASS_DECLSPEC GetNbReceivedData(void);

This function return the number of byte received from the USC-30xx. Those bytes are stored in the data circular buffer used to receive data from USC-30xx.

f. GetReceivedData() detail

Prototype: DWORD GetReceivedData(BYTE*, DWORD);

This function retrieves one or many value(s) from the circular buffer used to receive data from USC-30xx.

Parameter	Description
BYTE*	This parameter points on the start address of the data to read
DWORD	This parameter indicates the number of byte to read in the circular buffer.

Table 6: GetReceivedData parameter

Error Returned	VALUE	Description
ERR_SUCCESS	0x00	All bytes are put in the BYTE* buffer correctly
ERR_BAD_LENGTH	0x06	The number of byte to read is to much

Table 7: GetReceivedData error Message

Example usage:

```
WORD NbDataReceived;              // Used to read number of byte
BYTE Data[INPORT_DATA_BUFFSIZE]; // Data buffer

NbDataReceived = GetNbReceivedData();

if( NbDataReceived > 0 )
{
       BYTE Data[NbDataReceived]; /* Buffer used to read data from USC-30xx */

GetReceivedData( Data,  NbDataReceived );
/* Then Data[] contains byte came from the USC-30xx SPI port */
}
```

g. SendCommand() detail

Prototype: DWORD SendCommand(BYTE*, BYTE, BYTE*);

This function sends a command to the USC-30xx. This section will explain how this function shall be used. After, all the command that can be sending to the USC-30xx will be explain. Four error messages can be returned from the SendCommand function. If the ERR_SUCCESS (0x00) is returned the command is send, the USC-30xx executed this command and an acknowledge is received from the USC-30xx.

Parameter	Description
BYTE*	This parameter points on the start address of the command to send
BYTE	This parameter is the number of byte of the command
BYTE*	This parameter points on the start address of the Acknowledge message or the returned message.

Table 8: SendCommand parameter

Error Returned	VALUE	Description
ERR_SUCCESS	0x00	The command is sent and the acknowledge or the respond is received correctly
ERR_WRITE_FAULT	0x02	Problem to send the command
ERR_INVALID_DATA	0x03	The acknowledge or the respond received is not good
ERR_NO_ACK_RECEIVED	0x0A	No acknoledge is received from USC-30xx after 1sec

Table 9: SendCommand error Message

i. Set SPI mode command

This command is used to set the mode of the SPI port and to handle the INT pin. The master mode indicates that the USC-30xx has the control of the SPI clock. For the slave mode, the SPI master device controls the SPI clock, so the USC-30xx has no control of it. The USC-30xx can send or received data in 8 bits or 16 bits mode. The INT pin is used only in slave mode to indicate to the SPI master device that data is ready to be sent. The INT pin is not obliged to use to transmit data. An acknowledge message will be return by the USC-30xx to confirm the modification of the SPI mode.

Set SPI mode				
Byte #	Name	Value	Description	Default value
0	Instruction	0x02	Set mode of SPI port	
1	Mode	0x00	Slave mode	Master
		0x01	Master mode	
2	Bit mode	0x00	8 bits mode	8 bits
		0x01	16 bits mode	
3	INT pin	0x00	Disable INT pin	INT pin Disabled
		0x01	Enable INT pin	
4	Seq bit mode	0x00	MSB first	LSB first
		0x01	LSB first	

Table 10: Set SPI mode

Acknowledge of Set SPI mode				
Byte #0	Byte #1	Byte #2	Byte #3	Byte #4
Instruction(0x02)	Mode	Bit mode	INT pin	Seq bit mode

Table 11: Return Get SPI mode

Example usage:

```
BYTE OutBuffer[5];
BYTE InBuffer[5];

/* Set the Set SPI mode command */
OutBuffer[0] = SET_SPI_MODE_CMD;
OutBuffer[1] = MASTER_MODE;
OutBuffer[2] = SIXTEEN_BIT_MODE;
OutBuffer[3] = INT_DISABLE;
OutBuffer[4] = MSB_FIRST_MODE;

/* Send command */
Error = SendCommand( OutBuffer, 5, InBuffer )

if( Error == ERR_SUCCESS )
{
       /* The SPI mode is changed
          Then the InBuffer must contain:
              InBuffer[0] == SET_SPI_MODE_CMD;
              InBuffer[1] == MASTER_MODE;
              InBuffer[2] == SIXTEEN_BIT_MODE;
              InBuffer[3] == INT_DISABLE;
              InBuffer[4] == MSB_FIRST_MODE; */
```

ii. Get SPI mode command

This command is used to get the SPI mode and the INT pin. The answer returned to the host by the USC-30xx is the instruction value of the get command associated with the 3 bytes containing the value of each parameter. The USC-30xx makes accessible the response immediately after it's treatment. The answer is stored in the converter until the API reads it.

Get SPI mode			
Byte #	Name	Value	Description
0	Instruction	0x03	Get mode of SPI port

Table 12: Get SPI mode

Value of Get SPI mode				
Byte #0	Byte #1	Byte #2	Byte #3	Byte #4
Instruction(0x03)	Mode	Bit mode	INT pin	Seq bit mode

Table 13: Return Get SPI mode

Example usage:

```
BYTE OutBuffer[1];
BYTE InBuffer[5];

/* Set the Get SPI mode command */
OutBuffer[0] = GET_SPI_MODE_CMD;

/* Send command */
Error = SendCommand( OutBuffer, 1, InBuffer )

if( Error == ERR_SUCCESS )
{
       /* Then the InBuffer must contain the SPI mode:
              InBuffer[0] == SET_SPI_MODE_CMD;
              InBuffer[1] == MASTER_MODE;
              InBuffer[2] == SIXTEEN_BIT_MODE;
              InBuffer[3] == INT_DISABLE;
              InBuffer[4] == MSB_FIRST_MODE; */
}
```

iii. Set SPI Clock mode

This command is used to set the baud rate, phase and polarity of the SPI clock. The frequency of the SPI serial clock in master mode is set with two fields, the SPI Baud Rate Prescale Divisor field (SPPR) and the SPI Baud Rate Divisor field (SPR). Both fields are on three bits and the baud rate divisor equation is as follow. The minimum SPI BaudRate is 11.718Kbps and the maximum SPI baudrate is 12Mbps.

$$BaudRate = 24Mbps / (Prescale\ Divisor \bullet Divisor)$$

Equation 1: Baudrate equation

For example, having a baud rate of 500Kbps we may have a prescale divisor of 6 (0x05) and a divisor of 8 (0x02).

Field	Description
SPPR[2 :0]	SPI Baud Rate Prescale Divisor: This 3-bit field selects one of eight divisors for the SPI baud rate prescale.
SPR[2:0]	SPI Baud Rate Divisor: This 3-bit field selects one of eight divisors for the SPI baud rate.

Table 14: Baud rate field description

SPPR2: SPPR1: SPPR0	Prescale Divisor	SPR2: SPR1: SPR0	Divisor
0:0:0	1	0:0:0	2
0:0:1	2	0:0:1	4
0:1:0	3	0:1:0	8
0:1:1	4	0:1:1	16
1:0:0	5	1:0:0	32
1:0:1	6	1:0:1	64
1:1:0	7	1:1:0	128
1:1:1	8	1:1:1	256

Table 15: Value of Baud rate

The USC-30xx allows the user to set the clock polarity and the clock phase. The clock polarity selects an inverter or non-inverter bit that is in series with the SPI clock. This controls the active level of the SPI clock. The clock phase chooses between two different clock phase relationships (clock and data). The next figure shows the application of these possibilities.

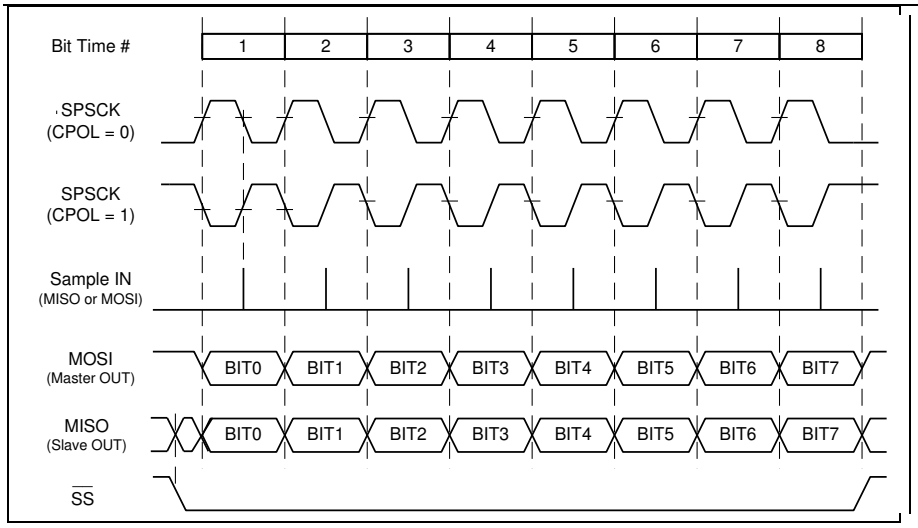


Figure 1: SPI clock format (CPHA = 1)

When CPHA = 1, the slave begins to drive when SS reaches active low. In this case the slave select ($\overline{SS}$) is set to automatic. On the first SPSCK edge, the data is defined and shifts the first data onto the transmission line. The next edge causes the sampling of the bit on the transmission line. The third edge shifts the byte of one bit onto the transmission line and the fourth edge onto the sampling etc.

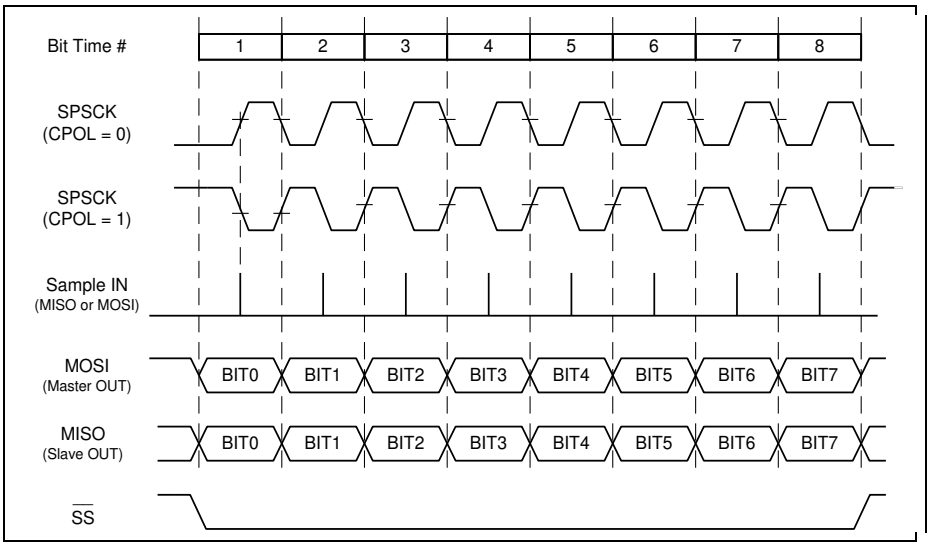


Figure 2: SPI clock format (CPHA = 0)

When CPHA = 0, the slave and the master device begin to drive when SS reaches active low. In this case, the slave select ($\overline{SS}$) is set to automatic. The first SPSCK edge initiates the sampling of the bit on the transmission line. The second edge shifts the byte of one bit onto the transmission line and the third edge onto the sampling etc.

When this command is sent to the USC-30xx, an Acknowledge is responded by the USC-30xx to confirm that the change is really done. This Acknowledge represents the same command send to the USC-30xx. It is gathered in the unit until the API has read it. If the Acknowledge received by the host is not the same

that the API sent, an error has occurred. The SPI baud rate of the USC-30xx cannot be change in slave mode.

Set SPI Clk mode				
Byte #	Name	Value	Description	Default value
0	Instruction	0x04	Set clock of SPI port	
1	SPPR	0xXX	SPI baud rate prescale divisor	0x07
2	SPR	0xXX	SPI baud rate divisor	0x07
3	Clock phase	0x00	Active high SPI clock	0x01
		0x01	Active low SPI clock	
4	Clock polarity	0x00	First edge on SPSCK occurs at the middle of the cycle of a data transfer	0x01
		0x01	First edge on SPSCK occurs at the start of the cycle of a data transfer	

Table 16: Set SPI clock mode

Acknowledge of Set SPI clock mode				
Byte #0	Byte #1	Byte #2	Byte #3	Byte #4
Instruction (0x04)	SPPR	SPR	Clock phase	Clock polarity

Table 17: Acknowledge of Set SPI Clock mode

Example usage:

```
BYTE OutBuffer[5];
BYTE InBuffer[5];

/* Set the Set SPI clock mode command */
OutBuffer[0] = SET_SPI_CLK_CMD;
OutBuffer[1] = SPPR_6;            /* Prescaler Divisor = 6 */
OutBuffer[2] = SPR_32;            /* Rate Divisor = 32 */
OutBuffer[3] = CLK_LOW;           /* Phase Low */
OutBuffer[4] = CLK_START;         /* Polarity at the Start */

/* Send command */
Error = SendCommand( OutBuffer, 1, InBuffer )

if( Error == ERR_SUCCESS )
{
       /* The SPI clock mode is changed
          Then the InBuffer must contain:
              InBuffer[0] == SET_SPI_CLK_CMD;
              InBuffer[1] == SPPR_6;
              InBuffer[2] == SPR_32;
              InBuffer[3] == CLK_LOW;
              InBuffer[4] == CLK_START; */
}
```

iv. Get SPI Clock mode

This command is used to get the SPI clock status. The answer returned to the host by the USC-30xx is the instruction value of the get command associated with the 4 bytes containing the value of each parameter. The USC-30xx makes accessible the response immediately after it's treatment. The answer is stored in the converter until the API reads it.

Get SPI clock mode			
Byte #	Name	Value	Description
0	Instruction	0x05	Get clock status of SPI port

Table 18: Get SPI clock mode

Return Get SPI clock mode				
Byte #0	Byte #1	Byte #2	Byte #3	Byte #4
Instruction (0x05)	SPPR	SPR	Clock phase	Clock polarity

Table 19: Return Get SPI Clock mode

Example usage:

```
BYTE OutBuffer[1];
BYTE InBuffer[5];

/* Set the Get SPI clock mode command */
OutBuffer[0] = GET_SPI_CLK_CMD;

/* Send command */
Error = SendCommand( OutBuffer, 1, InBuffer )

if( Error == ERR_SUCCESS )
{
       /* Then the InBuffer must contain the Clk mode:
              InBuffer[0] == GET_SPI_CLK_CMD;
              InBuffer[1] == SPPR_6;
              InBuffer[2] == SPR_32;
              InBuffer[3] == CLK_LOW;
              InBuffer[4] == CLK_START; */
}
```

v. Set SPI slave select

This command is used to set the status of the slave select pin of the SPI port. This pin selects a slave device to send it some data and offers three possibilities: Automatic, High and Low. When the slave select is set to Automatic, the USC-30xx handles the SS pin. If the Low status or the High status is selected, this pin is used like an I/O pin and the converter drives the pin Low or High until another SPI slave select command is set. When a slave select command is sent to the USC-30xx, an Acknowledge is responded by the USC-30xx to confirm that the change is really done. This Acknowledge represents the same command sent to the USC-30xx. It is stored in converter until the API has read it. If the Acknowledge received by the host is not the same that the API sent, an error has occurred. The slave select pin status cannot be set in slave mode.

SPI Slave select				
Byte #	Name	Value	Description	Default value
0	Instruction	0x06	Set slave select	
1	Status	0x01	Automatic	0x01
		0x02	Drive high	
		0x03	Drive low	

Table 20: Set SPI slave select

Acknowledge of Set SPI slave select	
Byte #0	Byte #1
Instruction (0x06)	Status slave select

Table 21: Acknowledge of Set SPI slave select

Example usage:

```
BYTE OutBuffer[2];
BYTE InBuffer[2];

/* Set the Set SPI Slave Select command */
OutBuffer[0] = SET_CHIP_SELECT_CMD;
OutBuffer[1] = CS_AUTO;


/* Send command */
Error = SendCommand( OutBuffer, 2, InBuffer )

if( Error == ERR_SUCCESS )
{
       /* The SPI Slave Select pin is changed
          Then the InBuffer must contain:
              InBuffer[0] == SET_CHIP_SELECT_CMD;
              InBuffer[1] == CS_AUTO; */
}
```

vi. Get SPI slave select

This command is used to get the SPI slave select. The message returned to host is the instruction value of the get command associated with the status of the slave select. The USC-30xx makes accessible the response immediately after its treatment. The answer is stored in the converter until the API reads it.

Get SPI slave select			
Byte #	Name	Value	Description
0	Instruction	0x07	Get slave select of SPI port

Table 22: Get SPI slave select

Return Get SPI slave select	
Byte #0	Byte #1
Instruction (0x07)	Status slave select

Table 23: Return Get SPI slave select

Example usage:

```
BYTE OutBuffer[1];
BYTE InBuffer[2];

/* Set the Get Slave Select command */
OutBuffer[0] = GET_CHIP_SELECT_CMD;

/* Send command */
Error = SendCommand( OutBuffer, 1, InBuffer )

if( Error == ERR_SUCCESS )
{
       /* Then the InBuffer must contain the Slave Select mode:
       InBuffer[0] == GET_CHIP_SELECT_CMD;
       InBuffer[1] == CS_AUTO; */
}
```

vii. Start autonomous sampling

This command starts the autonomous sampling of the SPI port with the specific time and number of command wanted. The USC-30xx must be set before to send the command. For example the user must set the sequence bit and the bit mode before to start the autonomous sequence. The command's length is based on the bit mode selected in the USC-30xx. If the bit mode of the USC-30xx is 16 bits the command must have 2 bytes. The acknowledge of the command is the same packet send to the USC-30xx. If during the utilization of this function, the circular buffer of the USC-30xx overrun, the USC-30xx stop the SPI sampling and he will return a toggle value of 0x0000 and 0xFFFF until the stop autonomous command will be received.

Start autonomous (8 bits)			
Byte #	Name	Value	Description
0	Instruction	0x0B	Start autonomous process
1	Time sample	0xXX	Time of the sampling (Between 1(1ms) and 80(80ms))
2	Nb of cmd	0xXX	Number of command at each sampling (1 or 2)
3	Cmd 1	0xXX	Command #1
*4	Cmd 2	0xXX	Command #2 (*if two commands are set)

Table 24: Start autonomous (8 bits)

Acknowledge of Start autonomous (8bit)				
Byte #0	Byte #1	Byte #2	Byte #3	*Byte #4
Instruction (0x0B)	Time Sample	Nb of cmd	Cmd 1	Cmd 2

Table 25: Start autonomous Acknowledge (8 bits)

Start autonomous (16 bits)			
Byte #	Name	Value	Description
0	Instruction	0x0B	Start autonomous process
1	Time sample	0xXX	Time of the sampling (Between 1(1ms) and 80(80ms))
2	Nb of cmd	0xXX	Number of command at each sampling (1 or 2)
3:4	Cmd 1	0xXX	Command #1
*5:6	Cmd 2	0xXX	Command #2 (*if two commands are set)

Table 26: Start autonomous (16 bits)

Acknowledge of Start autonomous (16bit)						
Byte #0	Byte #1	Byte #2	Byte #3	Byte #4	*Byte #5	*Byte #6
Instruction (0x0B)	Time Sample	Nb of cmd	Cmd 1 Byte 1 (LSB)	Cmd 1 Byte 2 (MSB)	Cmd 2 Byte 1 (LSB)	Cmd 2 Byte 2 (MSB)

Table 27: Start autonomous Acknowledge (16 bits)

This example usage shows how to start the sampling of one command:

```
BYTE OutBuffer[5];
BYTE InBuffer[5];

/* Set the start sampling command */
OutBuffer[0] = START_SAMPLING_CMD;
OutBuffer[1] = 0x0A;              /* sampling at 10ms */
OutBuffer[2] = 0x01;              /* Number of command read = 1 */
OutBuffer[3] = 0x00;              /* The command is 0x0400 */
OutBuffer[4] = 0x04;


/* Send command */
Error = SendCommand( OutBuffer, 5, InBuffer )

if( Error == ERR_SUCCESS )
{
       /* The sampling is started
          Then the InBuffer must contain:
              InBuffer[0] == START_SAMPLING;
              InBuffer[1] == 0x0A;
              InBuffer[2] == 0x01;
              InBuffer[3] == 0x00;
              InBuffer[4] == 0x04; */

       /* I started a timer to read the sample at all 50ms. I will read 5 new
          samples (14 bytes including the two first bytes of instruction)*/
       SetTimer( MY_TIMER, 50, NULL );
}
```

viii. Test autonomous sampling

This command tests the autonomous sampling without use the SPI port. A new incremented value is returned to the USB host at each sampling time. The value is incremented by 4 (Value between 0x0000 to 0x0FFC). The acknowledge of the command is the same packet send to the USC-30xx.

Test autonomous			
Byte #	Name	Value	Description
0	Instruction	0x0C	Test autonomous process
1	Time sample	0xXX	Time of the sampling (Between 1(1ms) and 80(80ms))
2	Nb of cmd	0xXX	Number of command at each sampling (1 or 2)

Table 28: Test autonomous

Acknowledge of Start autonomous (8bit)		
Byte #0	Byte #1	Byte #2
Instruction (0x0C)	Time Sample	Nb of cmd

Table 29: Test autonomous Acknowledge

This example usage shows how to start the uController test sampling:

```
BYTE OutBuffer[3];
BYTE InBuffer[3];

/* Set the Test uController sampling command */
OutBuffer[0] = TEST_SAMPLING_UC_CMD;
OutBuffer[1] = 0x0A;              /* Sampling at 10ms */
OutBuffer[2] = 0x01;              /* Number of axe read = 1 */

/* Send command */
Error = SendCommand( OutBuffer, 3, InBuffer )

if( Error == ERR_SUCCESS )
{
       /* The sampling is started
          Then the InBuffer must contain:
              InBuffer[0] == TEST_SAMPLING_UC;
              InBuffer[1] == 0x0A;
              InBuffer[2] == 0x01; */

       /* I started a timer to read the sample at all 50ms. I will read 5 new
          samples (14 bytes including the two first bytes of instruction)*/
       MyTimer = SetTimer( MY_TIMER, 50, NULL );
}
```

ix. Stop autonomous sampling

This function stops the autonomous sampling. The acknowledge of the command is the same packet send to the USC-30xx.

Stop autonomous			
Byte #	Name	Value	Description
0	Instruction	0x0D	Stop autonomous process

Table 30: Stop autonomous process

Acknowledge of Start autonomous (8bit)
Byte #0
Instruction (0x0D)

Table 31: Stop autonomous process

This example usage shows how to stop the sampling:

```
BYTE OutBuffer[1];
BYTE InBuffer[1];

/* Set the Test Accelerometer sampling command */
OutBuffer[0] = STOP_SAMPLING_CMD;

/* Send command */
Error = SendCommand( OutBuffer, 1, InBuffer )

if( Error == ERR_SUCCESS )
{
/* The sampling is started
          Then the InBuffer must contain:
              InBuffer[0] == STOP_SAMPLING; */

/* I stop the timer for reading the sampling. */
       KillTimer(MyTimer);
}
```

x. Test SPI port

This command is used to test the SPI port by sending the same byte in continuous loop on the PIS port. When this command is sent to the USC-30xx, an Acknowledge is responded by the USC-30xx to confirm that the test is started. This Acknowledge represents the same command sent to the USC-30xx. The answer is stored in converter until the API reads it.

Test SPI port			
Byte #	Name	Value	Description
0	Instruction	0x08	Get clock of SPI port
1	State	0x00	State of the test (0x00 = Stop or 0x01 = Run)

Table 32: Test SPI port

Return request byte to send	
Byte #0	Byte #
Instruction (0x08)	State of the test

Table 33: Test SPI port

This example usage shows how to stop the sampling:

```
BYTE OutBuffer[2];
BYTE InBuffer[1];

/* Set the test SPI command */
OutBuffer[0] = TEST_SPI_CMD;
OutBuffer[1] = 1;


/* Send command */
Error = SendCommand( OutBuffer, 2, InBuffer )

if( Error == ERR_SUCCESS )
{
       /* The sampling is started
          Then the InBuffer must contain:
              InBuffer[0] == TEST_SPI_CMD;
              InBuffer[1] == 1; */
}
```

xi. Request byte to send

This command is used to ask the USC-30xx how many bytes are empty in the SPI buffer. This buffer is used to store the data sent from host before sending it to the SPI port. It is 834 bytes in length. This request must be done before sending the data to the USC-30xx. If an overrun occurs the SPI buffer will reset to prevent a possible error. This command assures that the number of bytes sent won't exceed the length of the SPI buffer.

Request byte to send			
Byte #	Name	Value	Description
0	Instruction	0x09	Request how many byte is empty in SPI buffer

Table 34: Request byte to send

Return request byte to send	
Byte #0	Byte #1 & #2
Instruction (0x09)	Number of byte can send (0x0000 to 0x0342)

Table 35: Return Request byte to send

xii. Request the version of the firmware

This command is used to get the version, the date and the time when the firmware was compiling. The packet returned to host is the instruction of the request command associated with the version on two bytes, the date of compilation on 13 bytes and the time of compilation on 12 bytes. The version is returned in hexadecimal in the form Major : Minor. The date and time is returned in ASCII character.

Request the version of the firmware			
Byte #	Name	Value	Description
0	Instruction	0x0A	Request the version of the firmware

Table 36: Request the version of the firmware

Return request byte to send			
Byte #0	Byte #1 & #2	Byte #3 to #15	Byte #16 to #27
Instruction (0x0A)	Version	Date	Time

Table 37: Return Request the version of the firmware

This example usage shows how to read the sampling:

```
if( nIDEvent == MY_TIMER )
{
	WORD NbDataReceived;			// Used to read number of byte
	BYTE Data[INPORT_DATA_BUFFSIZE];	// Data buffer

	ErrStateThread = GetReadState();
	if ( ErrStateThread == ERR_THREAD_STOP )
	{
		/* The Thread is stop */
		KillTimer(MyTimer); // Stop the timer
		CloseComm();		// Close the comm. between USC-220 and the API
	}
	else if( ErrStateThread == ERR_CB_OVERLOAD )
	{
```

```
                /* The buffer circular of the DLL interface is full */
        }
        else if( ErrStateThread == ERR_SUCCESS )
        {
                /* Read the number of byte received */
                NbDataReceived = GetNbReceivedData();

                if( NbDataReceived > 0)
                {
                        /* Read the sample data */
                        GetReceivedData( Data,  NbDataReceived );

                        /* At this moment, Data[] contains samples data */
                }
        }

}
```

h. Data transmission

The firmware uses circular buffers to handle the data on both sides. The buffers are 834 bytes in length. The converter has complete control of the buffers and no code is necessary to control them. These next parts explain how to send and to receive data with the USC-30xx.

i. USB host to SPI device

The USC-30xx may be set in master or slave mode. The method to send data to the USC-30xx in both modes is the same but not for starting the SPI transmission. After configuring the SPI port, we must know how many bytes we can send to the USC-30xx without overrunning the SPI buffer. This information is amassed with the "request byte to send" command viewed previously. The number of bytes sent in a packet is 64. If the data contains more than that, the driver automatically separates into packets of 64 bytes. In a data transmission from the USB host to the USC-30xx, the first packet contains the instructions associated with the number of bytes to send and the data. The next few packets contain the rest of the data.

Data packet (1st)			
Byte #	Name	Value	Description
0	Instruction	0x01	The mode is write data
1:2	Nb of byte	0x0XXX	The number of byte to send (0 to 834 bytes)
3:63	Data	Byte	Byte to send to SPI port

Data packet (2nd)			
Byte #	Name	Value	Description
0:63	Data	Byte	Byte to send to SPI port is more that 61 bytes

…

Data packet (Xth)			
Byte #	Name	Value	Description
0:63	Data	Byte	The rest of the byte to send to SPI port

Table 38: Send data to USC-30xx

The USC-30xx stored the data received in the circular buffer. In master mode, the converter starts the transmission after receiving the last byte of the last packet (all of the data). In slave mode, the INT pin must be active to indicate to the master device that the data is ready to send. The master device will enable the clock and the transmission will begin. This pin can be set before or after sending the data.

During the transmission, this same process can be executed to send data while in a continuous state.

ii. SPI device to USB host

The USC-30xx can be set in either master or slave mode. The method used by the USC-30xx to send data to the USB host is the same for both modes (master and slave), but the method to read data on the SPI port is different.

In the slave mode, the SPI master device controls the SPI clock and the slave select line. When the SPI master device sends data to USC-30xx, this data is stored in a circular buffer and is immediately sent to the USB host.

In master mode, the USC-30xx controls the SPI clock and the slave select line. Thus, to be able to read the SPI data from the slave device we must first activate the clock by sending data to it. This will in turn allow us to simultaneously read the SPI data. To enable the reading mode, the instructions of "send data to USC-30xx" must be changed but the transmission protocol should remain the same. When the new byte is received from the SPI port it is automatically sent to the USB host.

Data packet (1st)			
Byte #	Name	Value	Description
0	Instruction	0x11	The mode is read and write data
1:2	Nb of byte	0x0XXX	The number of byte to receive and to send (0 to 834 bytes)
3:63	Data	Byte	Byte to send to SPI port

…

Data packet (Xth)			
Byte #	Name	Value	Description
0:63	Data	Byte	The rest of the byte to send to SPI port

Table 39: Receive and send data from USC-30xx

During the reception phase, the same process can be executed to receive data in a continuous state.

Regardless of the mode, (master or slave) the format of the packet when the data is sent from the USC-30xx to the USB host is always the same. All of the packets sent by the USC-30xx to the host contain instructions, that at least one byte has been received by the SPI port. This is followed by info about the number of bytes in the packet and the bytes read. If the circular buffer contains more than 62 bytes then the data will be divided into multiple packets.

Data received from USC-30xx			
Byte #	Name	Value	Description
0	Instruction	0x0F	Data from SPI port
1	Nb of byte	0xXX	Number of byte in the packet
2:63	Data	Byte	Byte received from SPI port

Table 40: Data received from USC-30xx

iii. Interface between the API and the driver

The USC-30xx works in bulk mode, so the driver must be built to handle this type of communication. The method used to receive data in the bulk mode is polling type; this also allows the USB host to control all transfers. The driver does the USB enumeration automatically.

The link between the driver and the API is a globally unique identifier (GUID). The GUID is also a 128-bit number that uniquely identifies an object. Once the driver has been identified as a unique object by the GUID, the software is able to communicate and becomes fully functional.

The GUID of the driver of Sysacom R&D plus Inc is: `51fe8f30-ef12-11dd-ba2f-0800200c9a66`
This GUID must be used by the API to find the USB device.

The API must initialize one handle for the OUT pipe and one for the IN pipe. As seen previously, the OUT pipe corresponds to the endpoint 2 and the IN pipe to the endpoint 1. This handle will be used to communicate with the USC-30xx.[1]

When the device's handles are done, the API can exchange data with the USC-30xx. The two functions that are used are the WriteFile and ReadFile. Communication is established between the API and the USC-30xx using these two functions. The next figures show a BULK transfer using IN or OUT transaction.

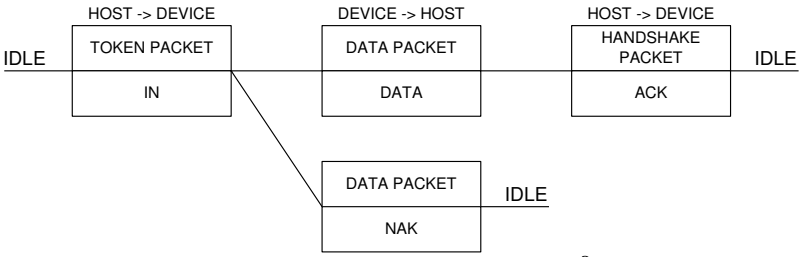


Figure 3: Bulk IN transaction[2]

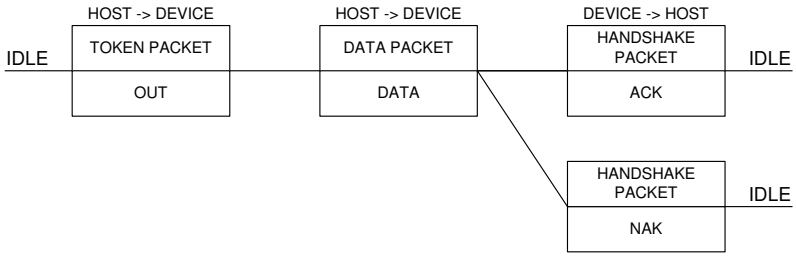


Figure 4: Bulk OUT transaction

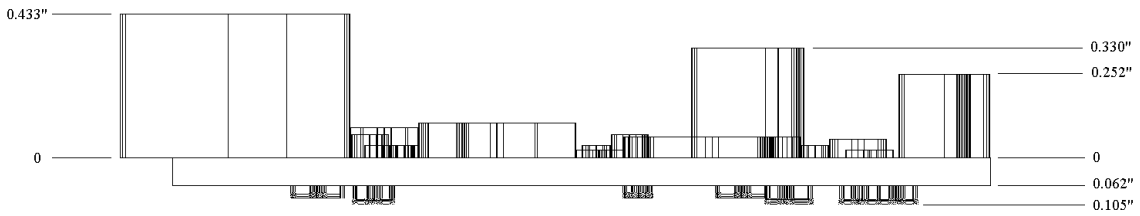
When the USC-30xx has data ready to be sent, the API must begin the communication with a ReadFile function, but the API is not alerted that something is ready to be read. To ease the process a thread can be declared. This thread asks the USC-30xx if it has something to send with the token packet. The USC-30xx responds with the data it has to send or with a no acknowledge (NAK).

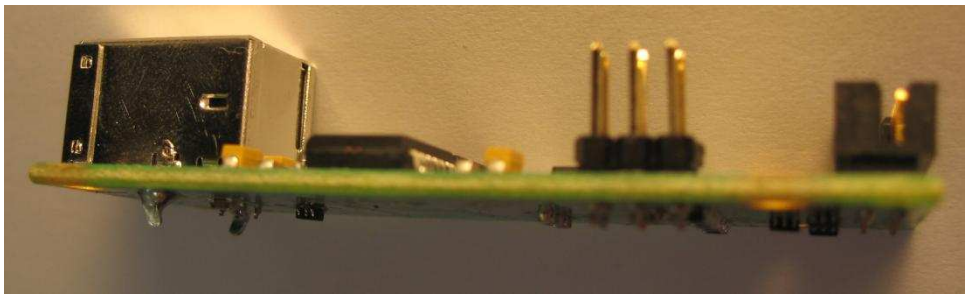
[1] Note: "USB COMPLETE SECOND EDITION" from Jan Axelson describes how to find a device and which function must be use.

[2] This figure is taken form the "USB COMPLETE SECOND EDITION" from Jan Axelson.

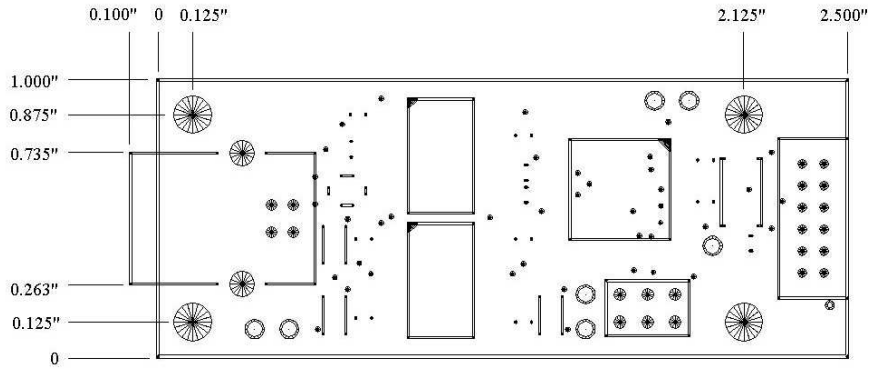
4. Mechanical

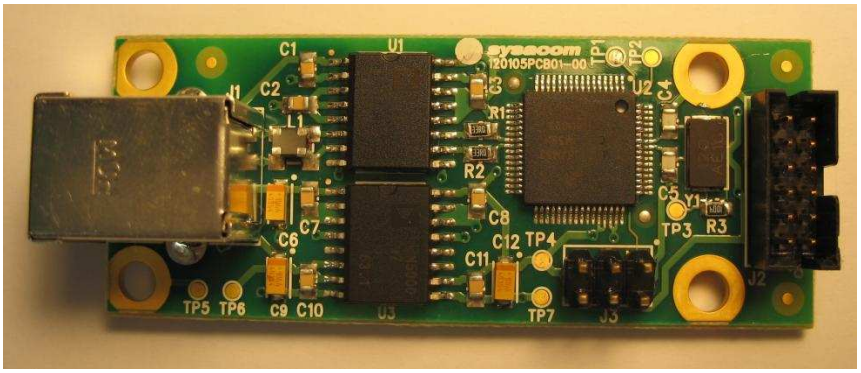
SIDE VIEW





TOP VIEW





* 4 MTH drills of 136 mils and grounded keep out of 250 mils of diameter.

5. Ordering Information

Part	Temperature Range	Isolation
USC-3025	-40°C ~ 85°C	2.5kV
USC-3050	-40°C ~ 85°C	5kV

6. Warranty

Sysacom R&D plus Inc warrants the USC-30xx Isolated USB to SPI converter to be free from malfunctions and defects in both materials and workmanship for one year from the date of purchase.

If the equipment does not function properly during the warranty period due to defects in either materials or workmanship, Sysacom R&D plus Inc will, at its option, either repair or replace the equipment without charge, subject to limitations stated herein. Such repair service will include all labour, as well as any necessary adjustments and / or replacement parts.

LIMITATIONS

The warranty becomes null and void if you fail to pack your USC-30xx in a manner consistent with the original product packaging and damage occurs during shipment.

Sysacom R&D plus Inc makes no other warranties, express, implied, or of merchantability or fitness for a particular purpose for this equipment or software. Repair or replacements without charge are Sysacom R&D plus Inc only obligation under this warranty. Sysacom R&D plus Inc will not be responsible for any special, consequential or incidental damages resulting from the purchase, use, or improper functioning of this equipment regardless of the cause.

Depending on your geographical location, some limitations may not apply.

Sysacom R&D plus inc.
275A Pierre-Le Gardeur Blvd
Repentigny, Quebec
Canada, J5Z 3A7
www.sysacom.com
(450)585-6396