USC-3250

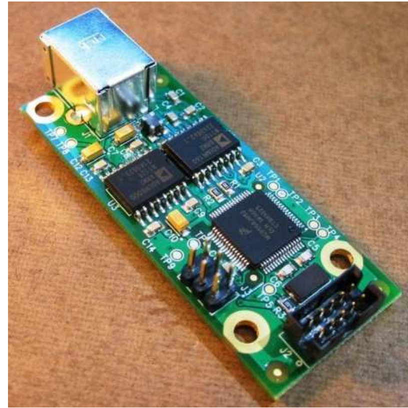
USB to I2C Isolated Protocol Adapter

DATASHEET

Rev1.02
September 2012

sysacom
www.sysacom.com

Features



- Bi-directional USB to I2C converter
- 5kV galvanic isolation
- USB 2.0 compatible
- Lead-Free / ROHS compliant
- Provide up to 70mA on the 3.3V on the I2C side.
- Programmable clock frequencies
- 10-bit address extension
- Baud rate up to 400 kbps

Applications

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

Description

Using the USC-3250 is an easy way to add a USB host to any device that features an Inter-Integrated Circuit (I2C or IIC) port or use an I2C device from an USB port. It electrically isolates the USB side from the I2C side without any loss of speed. The USC-3250 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-3250 has a standard I2C port and allows the user to connect with a device in either master or slave mode. It has an interrupt pin (INT) to indicate to the I2C device that the data is ready to send and an output pin (GPX) to indicate the completion of the USB enumeration. The USC-3250 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-3250 I2C port and USB port.

The USC-3250 is easy to configure with simple protocol commands for the USB port.

The USC-3250 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: 250Vrms, IEC 60950-1: 400V VIORM = 846V peak. This device is recommended for medical application.

Contents

1. ELECTRICAL CHARACTERISTICS **4**

2. PRINCIPLE OF OPERATION **4**

Connectors 4

1- USB TYPE B CONNECTOR **4**

2- MOLEX 87831-1020 .... **4**

3. DLL CONTENT **5**

A. INITCOMM() DETAIL 6
B. CLOSECOMM() DETAIL 6
C. GETREADSTATE() DETAIL 7
D. SENDDATA() DETAIL 8
E. GETNBRECEIVEDDATA() DETAIL 9
F. GETRECEIVEDDATA() DETAIL 9
G. SENDCOMMAND() DETAIL 10
i. Set Mode command 11
ii. Get Mode command 13
iii. Set Clock mode 13
iv. Get Clock mode 15
v. Set Slave Address 15
vi. Get Address 17
vii. Test I2C port 17
viii. Request byte to send 18
ix. Request the version of the firmware 18
H. DATA TRANSMISSION 19
i. USB host to I2C device 19
ii. I2C device to USB host 20
iii. Interface between the API and the driver 22

4. MECHANICAL **23**

5. ORDERING INFORMATION **24**

6. WARRANTY **24**

7. ANNEX A: I2C CLOCK PARAMETERS **25**

Revision History

Revision Number	Revision Date	Description of Changes
01.01	2012/06/25	Initial version

1. Electrical Characteristics

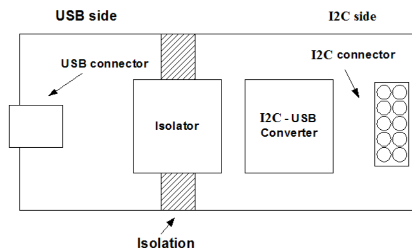
Specifications	USC-3250	Unit
Performance		
USB Compliance	2.0	
Max. Data Rate		
Data < 835 bytes	1.5	Mbps
Data > 835 bytes	500	Kbps
Maximum Bus Capacitance	400	pF
Environment		
Operating Temp Range	-40 to 85°C	°C
Electrical		
Supply Voltage (from USB)	5.0	V
Min. Supply Current (from USB)	60	mA
Max. Supply Current (from USB)	260	mA
Output Voltage I2C Side	3.3	V
Max. Output Current I2C Side	65	mA
Electrical isolation between USB side and I2C side	5.0	KV
Physical		
Weight	12	Grams

Connectors		
Pin	J1[1]	J2[2]
1	VCC	+3.3V
2	D+	+3.3V
3	D-	GND
4	GND	SCL
5		GND
6		SDA
7		GND
8		INT
9		GND
10		GPX

1- USB Type B connector
2- Molex 87831-1020

Suggest female connector for J2 :87568-1073

2. Principle of operation



The USC-3250 units are built using a Sysacom proprietary USB firmware. This software is used to interface from USB to I2C protocols. On the USC-3250 the galvanic isolation has been place between the USB connector and the USB/I2C converter. The device is powered from the USB port and generates its own 3.3V for the I2C side. The converter can also be used to power the customer end devices.

Only USB bulk transfer mode is supported by the USC-3250. 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 I2C 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 I2C 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 500Kbps while in continuous mode.

The I2C is low/medium data rate master/slave communication bus. There are two signals: Serial Clock (SCL) and Serial Data (SDA). The I2C is a true multi-master bus. I2C bus allows the clock stretching by the slave to slow down the bit rate of a transfer. There is also a extended address possibility (10bit).

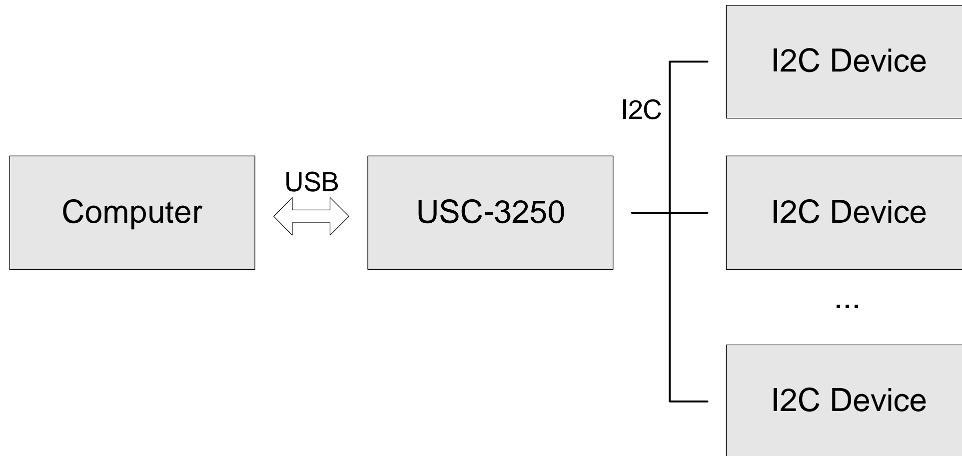


Figure 1: Overall USC-3250 Diagram

3. DLL content

To Define

a. InitComm() detail

Prototype : DWORD InitComm(void);

This function is used to initialize the communication with the USC-3250. 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-3250 in continuous is created and started.

Error Returned	VALUE	Description
ERR_SUCCESS	0x00	The communication with the USC-3250 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-3250 driver. It kills the thread and closes both handles (IN and OUT).

Error Returned	VALUE	Description
ERR_SUCCESS	0x00	The communication with the USC-3250 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-3250 and the circular buffer used to stored data came from the USC-3250. 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-3250 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-3250 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-3250 */
}
```

d. SendData() detail

Prototype: DWORD SendData(BYTE*, DWORD);

This function is used to send data to the I2C port through the USC-3250. The USC-3250 has a circular buffer to store all bytes want to send on the I2C 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-3250 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-3250

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-3250. Those bytes are stored in the data circular buffer used to receive data from USC-3250.

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-3250.

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-3250 */

GetReceivedData( Data,  NbDataReceived );
/* Then Data[] contains byte came from the USC-3250 I2C port */
}
```

g. SendCommand() detail

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

This function sends a command to the USC-3250. This section will explain how this function shall be used. After, all the command that can be sending to the USC-3250 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-3250 executed this command and an acknowledge is received from the USC-3250.

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-3250 after 1sec

Table 9: SendCommand error Message

i. *Set Mode command*

This command is used to set the mode of the I2C port and to handle the INT pin.

- **Mode**: The master mode indicates that the USC-3250 has the control of the I2C clock. For the slave mode, the I2C master device controls the I2C clock, so the USC-3250 has no control of it.
- **Interrupt**: The INT pin can be used in slave or master mode. The INT pin is used by by the I2C slave to indicate to the I2C master device that data is ready to be sent. The Slave INT pin indicates there is ready data to tranfer to I2C Master. The INT pin is not obliged to be used. An acknowledge message will be return by the USC-3250 to confirm the modification of the I2C mode.
- **Slave Address**: This field is used in master mode only. I2C master interacts with this slave address about data transfer.
- **nAck Sensitivity**: TO DEFINE
- **Last nAck:** TO DEFINE

Set I2C mode				
Byte #	Name	Value	Description	Default value
0	Instruction	0x02	Set mode of I2C port	
1	Mode	0x00	Slave mode	0x01
		0x01	Master mode	
2	INT pin	0x00	Disable INT pin	0x00
		0x01	Enable INT pin	
3	Slave Address	0xXX	7 lower bit of Remote Slave address	0x00
4	Slave Address	0xXX	3 upper bit	0x00
5	nAck Sensitive	0x00		0x00
		0x01		
6	Last nAck	0x00		0x00
		0x01		

Table 10: Set I2C Mode

Acknowledge of Set I2C mode			
Byte #0	Byte #1	Byte #2	Byte #3
Instruction(0x02)	Mode	INT pin	Slave Address

Table 11: Return Set I2C Mode

Example usage:

```
BYTE OutBuffer[4];
BYTE InBuffer[4];

/* Set the Set I2C mode command */
OutBuffer[0] = SET_I2C_MODE_CMD;
OutBuffer[1] = MASTER_MODE;
OutBuffer[2] = INT_ENABLE;
OutBuffer[3] = 0x12;       /* Slave Address    */

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

if( Error == ERR_SUCCESS )
{
       /* The I2C mode is changed
          Then the InBuffer must contain:
              InBuffer[0] == SET_I2C_MODE_CMD;
              InBuffer[1] == MASTER_MODE;
              InBuffer[2] == INT_ENABLE;
              InBuffer[3] == 0x12; */
```

Figure 2 shows an example of I2C communication. There are the both I2C signals: the serial clock (SCL) and the serial data (SDA). The I2C communication begins with a START condition (SCL HIGH while SDA makes a HIGH to LOW transition) and finishes with a STOP condition (SCL high while SDA makes a LOW to HIGH transition).

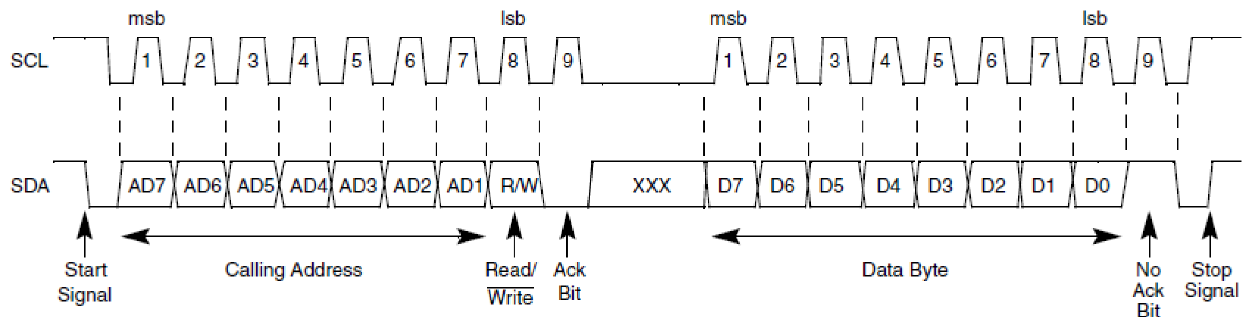


Figure 2: I2C Communication with START and STOP

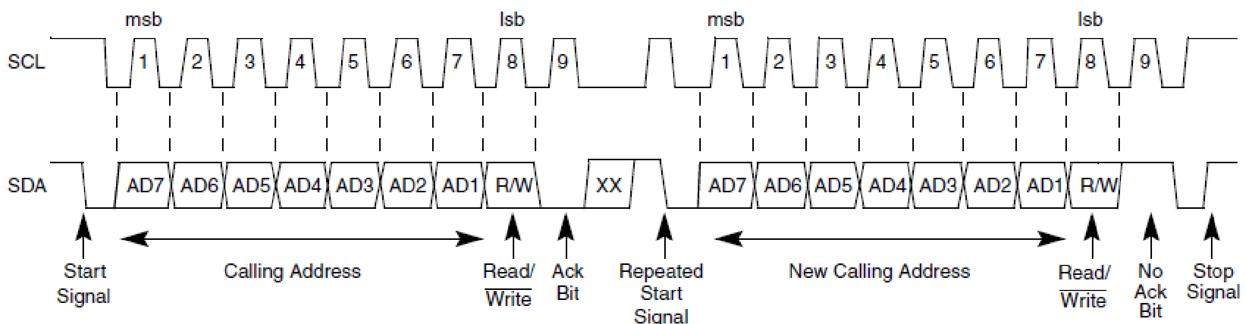


Figure 3: I2C Communication with repeated START

ii. Get Mode command

This command is used to get the I2C mode and the INT pin. The answer returned to the host by the USC-3250 is the instruction value of the get command associated with the 3 bytes containing the value of each parameter. The USC-3250 makes accessible the response immediately after its treatment. The answer is stored in the converter until the API reads it.

Get I2C mode			
Byte #	Name	Value	Description
0	Instruction	0x03	Get mode of I2C port

Table 12: Get I2C mode

Acknowledge of Get I2C mode			
Byte #0	Byte #1	Byte #2	Byte #3
Instruction(0x02)	Mode	INT pin	Slave Address

Table 13: Return Get I2C mode

Example usage:

```
BYTE OutBuffer[1];
BYTE InBuffer[4];

/* Set the Get I2C mode command */
OutBuffer[0] = GET_I2C_MODE_CMD;

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

if( Error == ERR_SUCCESS )
{
	/* Then the InBuffer must contain the I2C mode:
		InBuffer[0] == GET_I2C_MODE_CMD;
		InBuffer[1] == MASTER_MODE;
		InBuffer[2] == INT_ENABLE;
		InBuffer[3] == 0x12; Slave Address */
}
```

iii. Set Clock mode

This command is used to set the baud rate, SDA hold time and SCL start/stop hold times of the I2C clock. The frequency of the I2C serial clock in master mode is set with two fields:

- **Multiplier (MULT)**
- **I2C Clock Rate (ICR)**

Field descriptions are here:

Field	Description
MULT [1:0]	I2C Multiplier Factor. 00 mul = 01 01 mul = 02 10 mul = 04 11 Reserved
ICR [5:0]	I2C Clock Rate: {0x00, 0x3F}

Table 14: Clock rate field description

Here are some examples with different MUTL/ICR values.

MULT	ICR	SCL Freq (kHz)	SDA Hold (µs)	SCL Start (µs)	SCL Stop (µs)
2	0x05	**400.0**	0.750	0.917	1.333
1	0x14	**300.0**	0.708	1.417	1.708
4	0x0B	**150.0**	1.500	2.667	3.500
1	0x1F	**100.0**	1.375	4.917	5.042

Table 15: Exemple of Baud rate and Hold Time

The complete list of MULT/ICR combines is at Section 7.

When this command is sent to the USC-3250, an Acknowledge is responded by the USC-3250 to confirm that the change is really done. This Acknowledge represents the same command send to the USC-3250. 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 I2C baud rate of the USC-3250 cannot be change in slave mode.

Set I2C Clk mode				
Byte #	Name	Value	Description	Default value
0	Instruction	0x04	Set clock of I2C port	
1	MULT	0xXX	Multiplier values see Table 15	0x00
2	ICR	0xXX	Clock rate, see Table 15	0x00

Table 16: Set I2C clock mode

Acknowledge of Set I2C clock mode		
Byte #0	Byte #1	Byte #2
Instruction (0x04)	MULT	ICR

Table 17: Acknowledge of Set I2C Clock mode

Example usage to get I2C clock at 100 kHz:

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

/* Set the Set I2C clock mode command */
OutBuffer[0] = SET_I2C_CLK_CMD;
OutBuffer[1] = MULT_1;           /* Multiplier = 1 */
OutBuffer[2] = ICR_1F;           /* I2C Clock Rate = 0x1F */

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

if( Error == ERR_SUCCESS )
{
       /* The I2C clock mode is changed
          Then the InBuffer must contain:
              InBuffer[0] == SET_I2C_CLK_CMD;
              InBuffer[1] == MULT_1;
              InBuffer[2] == ICR_1F; */
}
```

iv. *Get Clock mode*

This command is used to get the I2C clock status. The answer returned to the host by the USC-3250 is the instruction value of the get command associated with the 2 bytes containing the value of each parameter. The USC-3250 makes accessible the response immediately after it's treatment. The answer is stored in the converter until the API reads it.

Get I2C clock mode			
Byte #	Name	Value	Description
0	Instruction	0x05	Get clock status of I2C port

Table 18: Get I2C clock mode

Acknowledge of Get I2C clock mode		
Byte #0	Byte #1	Byte #2
Instruction (0x04)	MULT	ICR

Table 19: Return Get I2C Clock mode

Example usage:

```
BYTE OutBuffer[1];
BYTE InBuffer[3];

/* Set the Get I2C clock mode command */
OutBuffer[0] = GET_I2C_CLK_CMD;

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

if( Error == ERR_SUCCESS )
{
	/* Then the InBuffer must contain the Clk mode:
		InBuffer[0] == GET_I2C_CLK_CMD;
		InBuffer[1] == MULT_1;
		InBuffer[2] == ICR_1F;	*/
}
```

v. *Set Slave Address*

This command is used to configure the USC-3250 I2C slave address. The address command is used only in slave mode.

- **General call**: This first option deternines if the USC-3250 reacts at a general call (address 0).
- **I2C address type**: I2C address is 7-bit or 10-bit.
- **7bit Address**: it contains 7bit address.
- **10bit Address**: it contains 3 upper bit of 10bit address.

I2C Slave Address				
Byte #	Name	Value	Description	Default value
0	Instruction	0x06	Set Slave Address	
1	General Call	0x00	Disable	0x00
		0x01	Enable	
2	Address type	0x00	7-bit Address	0x00
		0x01	10-bit Address	

3	Address 7bit	0xXX	From bit1 to bit7 are configurable. Bit0 is always 0. Register is [A7 A6 A5 A4 A3 A2 A1 0]	0x00
4	Address 10bit	0xXX	From bit0 to bit2 are configurable.From bit3 to bit 7 are always 0.Register is [0 0 0 0 0 A10 A9 A8].	0x00

Table 20: Set I2C Slave Address

Acknowledge of Set I2C Slave Address	
Byte #0	Byte #1
Instruction (0x06)	Status Slave Select

Table 21: Acknowledge of Set I2C slave select

Example usage:

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

/* Set the Set I2C Slave Address command */
OutBuffer[0] = SET_ADRESS_CMD;
OutBuffer[1] = ADDR_GEN_CALL_DISABLE;
OutBuffer[2] = ADDR_7BIT;
OutBuffer[3] = 0x12;
OutBuffer[4] = 0x00;       /* Don't care value */

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

if( Error == ERR_SUCCESS )
{
       /* The I2C Slave Select pin is changed
          Then the InBuffer must contain:
              InBuffer [0] = SET_ADRESS_CMD;
              InBuffer [1] = ADDR_GEN_CALL_DISABLE;
              InBuffer [2] = ADDR_7BIT;
              InBuffer [3] = 0x12;
              InBuffer [4] = 0x00; */
}
```

Here is the 10bit address pattern:

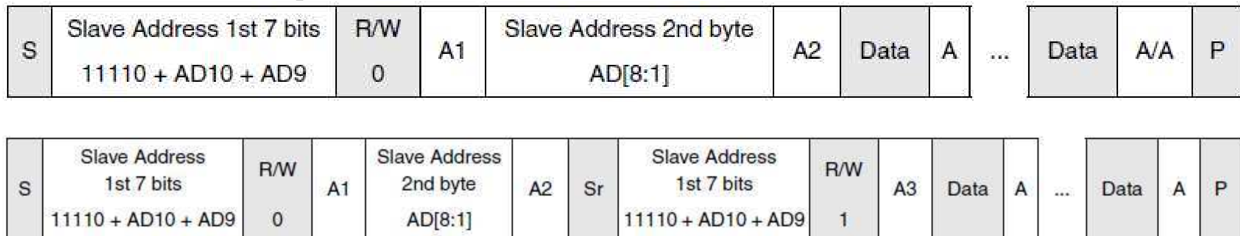


Figure 4: I2C 10bit Address

vi. *Get Address*

This command is used to get address parameters: general call, address type, 7bit address and 10bit address.

Get I2C slave address			
Byte #	Name	Value	Description
0	Instruction	0x07	Get slave address of I2C port

Table 22: Get I2C slave select

Return Get I2C slave address				
Byte #0	Byte #1	Byte #2	Byte #3	Byte #4
Instruction (0x07)	General Call	Address type	7bit Addr	10bit Addr

Table 23: Return Get I2C slave select

Example usage:

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

/* Set the Get Slave Address command */
OutBuffer[0] = GET_ADDRESS_CMD;

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

if( Error == ERR_SUCCESS )
{
        /* Then the InBuffer must contain the Slave Address mode:
                InBuffer [0] = GET_ADRESS_CMD;
                InBuffer [1] = ADDR_GENERAL_CALL;
                InBuffer [2] = ADDR_7BIT;
                InBuffer [3] = 0x12;
                InBuffer [4] = 0x00; */
}
```

vii. *Test I2C port*

This command is used to test the I2C port by sending the same byte in continuous loop on the I2C port. When this command is sent to the USC-3250, an Acknowledge is responded by the USC-3250 to confirm that the test is started. This Acknowledge represents the same command sent to the USC-3250. The answer is stored in converter until the API reads it.

Test I2C port			
Byte #	Name	Value	Description
0	Instruction	0x08	Get clock of I2C port
1	State	0x00	State of the test (0x00 = Stop or 0x01 = Run)

Table 24: Test I2C port

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

Table 25: Test I2C port

This example usage shows how to stop the sampling:

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

/* Set the test I2C command */
OutBuffer[0] = TEST_I2C_CMD;
OutBuffer[1] = 1;


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

if( Error == ERR_SUCCESS )
{
       /* The sending is started
          Then the InBuffer must contain:
              InBuffer[0] == TEST_I2C_CMD;
              InBuffer[1] == 1; */
}
```

viii. *Request byte to send*

This command is used to ask the USC-3250 how many bytes are empty in the I2C buffer. This buffer is used to store the data sent from host before sending it to the I2C port. It is 834 bytes in length. This request must be done before sending the data to the USC-3250. If an overrun occurs the I2C buffer will reset to prevent a possible error. This command assures that the number of bytes sent won't exceed the length of the I2C buffer.

Request byte to send			
Byte #	Name	Value	Description
0	Instruction	0x09	Request how many byte is empty in I2C buffer

Table 26: 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 27: Return Request byte to send

ix. *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 28: 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 29: 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-3250 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-3250.

i. USB host to I2C device

The USC-3250 may be set in master or slave mode. The method to send data to the USC-3250 in both modes is the same but not for starting the I2C transmission. After configuring the I2C port, we must know how many bytes we can send to the USC-3250 without overrunning the I2C buffer. This information is get 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-3250, 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. The USC-3250 stored the data received in the circular buffer.

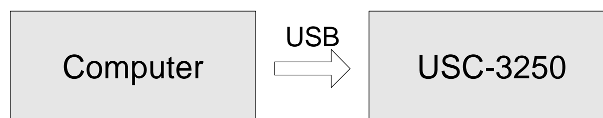


Figure 5: Transfer from USB Host to USC-3250

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 I2C port

Data packet (2nd)			
Byte #	Name	Value	Description
0:63	Data	Byte	Byte to send to I2C 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 I2C port

Table 30: Send data to USC-3250

In master mode, the converter starts the transmission after receiving the last byte of the last packet (all of the data).

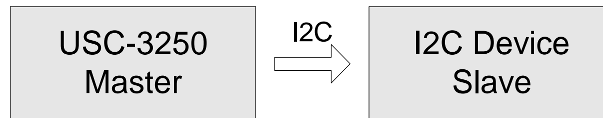


Figure 6: Transfer from USC-3250 (master) to I2C Slave Device.

In slave mode, the INT pin is activated (if enabled by command 0x02) to indicate to the master device there is ready data to send. The master device will enable the clock and the transmission will begin.

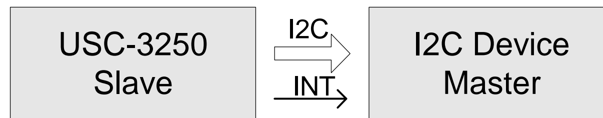


Figure 7: Transfer from USC-3250 (slave) to I2C Master Device.

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

ii. *I2C device to USB host*

The USC-3250 can be set in either master or slave mode. The method used by the USC-3250 to send data to the USB host is the same for both modes (master and slave), but the method to read data on the I2C port is different.

In the slave mode, the I2C master device controls the I2C clock and selects the right slave address. When the I2C master device sends data to USC-3250, this data is stored in a circular buffer and is immediately sent to the USB host.

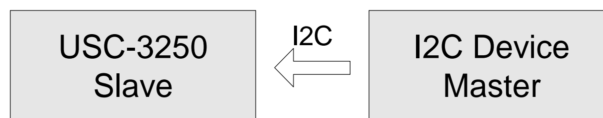


Figure 8: Transfer from I2C Master Device to USC-3250 (slave).

In master mode, the USC-3250 controls the I2C clock and selects the right slave address. Thus, to be able to read the I2C 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 I2C data. To enable the reading mode, the instructions of "send data to USC-3250" must be changed but the transmission protocol should remain the same. When the new byte is received from the I2C port it is automatically sent to the USB host.

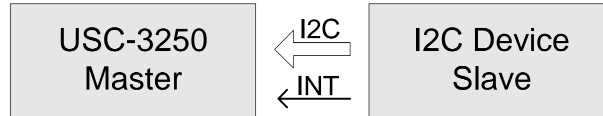


Figure 9: Transfer from I2C Slave Device to USC-3250 (master).

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 I2C port

…

Data packet (Xth)			
Byte #	Name	Value	Description
0:63	Data	Byte	The rest of the byte to send to I2C port

Table 31: Receive and send data from USC-3250

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 packet format when the data is sent from the USC-3250 to the USB host is always the same. All of the packets sent by the USC-3250 to the host contain instructions, that at least one byte has been received by the I2C 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.

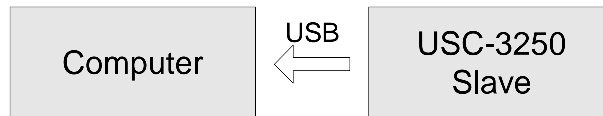


Figure 10: Transfer from USC-3250 to USB Host

Data received from USC-3250			
Byte #	Name	Value	Description
0	Instruction	0x0F	Data from I2C port
1	Nb of byte	0xXX	Number of byte in the packet
2:63	Data	Byte	Byte received from I2C port

Table 32: Data received from USC-3250

iii. *Interface between the API and the driver*

The USC-3250 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:

TO DEFINE

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-3250.[1]

When the device's handles are done, the API can exchange data with the USC-3250. The two functions that are used are the WriteFile and ReadFile. Communication is established between the API and the USC-3250 using these two functions. The next figures show a BULK transfer using IN or OUT transaction.

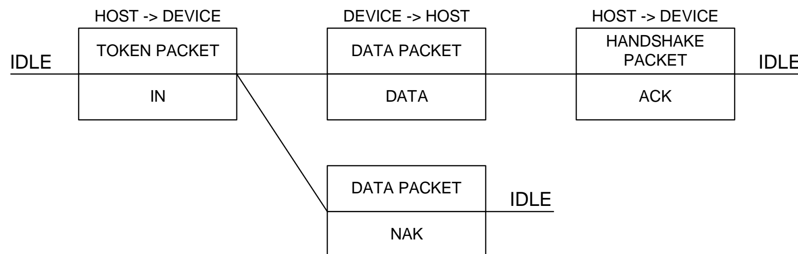


Figure 11: Bulk IN transaction[2]

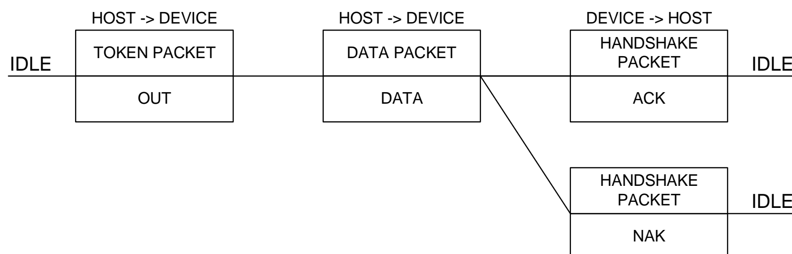


Figure 12: Bulk OUT transaction

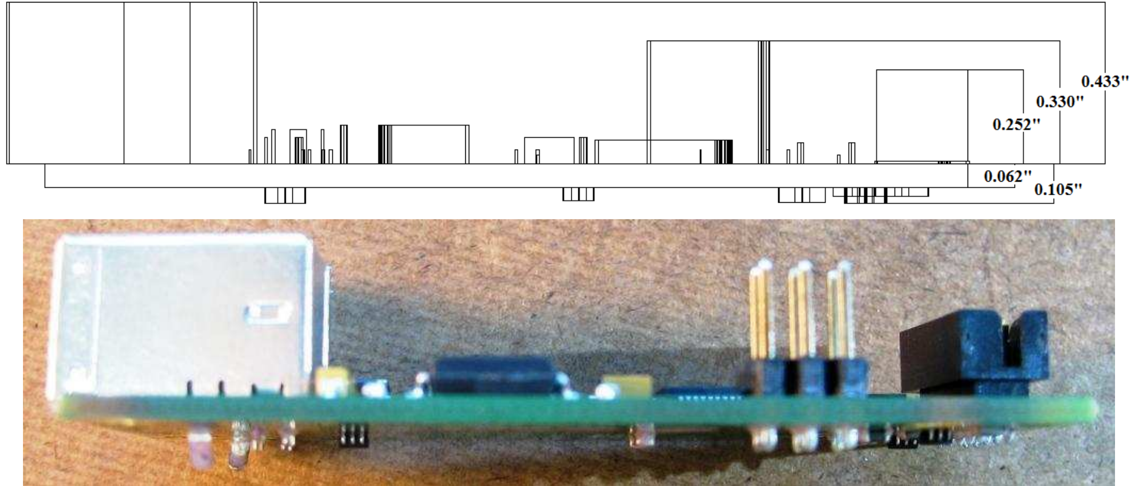
When the USC-3250 has data ready to be sent, the API must begin the communication with a ReadFile function, but the API is not alerted that something is ready to be read. To ease the process a thread can be declared. This thread asks the USC-3250 if it has something to send with the token packet. The USC-3250 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.

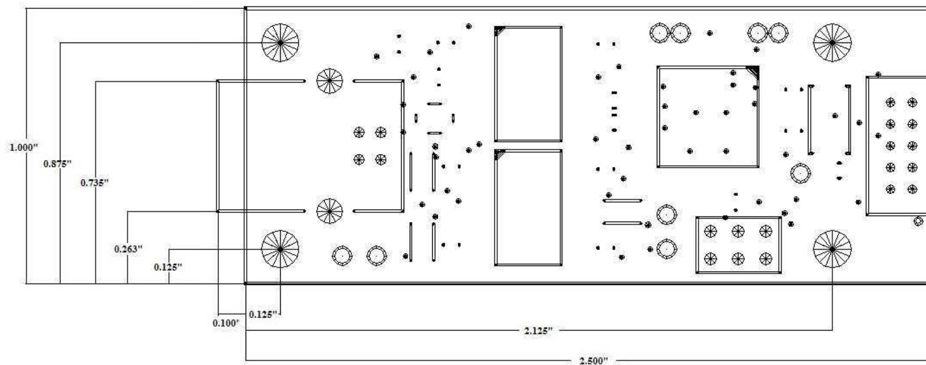
[2] This figure is taken form the "USB COMPLETE SECOND EDITION" from Jan Axelson.

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 Number	Temperature Range
USC-3250	-40°C ~ 85°C

6. Warranty

Sysacom R&D plus Inc warrants the USC-3250 Isolated USB to I2C 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-3250 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

7. ANNEX A: I2C Clock Parameters

There are two parameters for I2C clock: ICR and MULT. They have an impact on these four specifications:

- SCL Frequency (kHz)
- SDA Hold Value (µs)
- SCL Start/Stop Hold Value (µs)

MULT 1

ICR	SCL Freq	SDA Hold	SCL Start	SCL Stop
0x00	1200.0	0.292	0.250	0.458
0x01	1090.9	0.292	0.292	0.500
0x02	1000.0	0.333	0.333	0.542
0x03	923.1	0.333	0.375	0.583
0x04	857.1	0.375	0.417	0.625
0x05	800.0	0.375	0.458	0.667
0x06	705.9	0.417	0.542	0.750
0x07	600.0	0.417	0.667	0.875
0x08	857.1	0.292	0.417	0.625
0x09	750.0	0.292	0.500	0.708
0x0A	666.7	0.375	0.583	0.792
0x0B	600.0	0.375	0.667	0.875
0x0C	545.5	0.458	0.750	0.958
0x0D	500.0	0.458	0.833	1.042
0x0E	428.6	0.542	1.000	1.208
0x0F	352.9	0.542	1.250	1.458
0x10	500.0	0.375	0.750	1.042
0x11	428.6	0.375	0.917	1.208
0x12	375.0	0.542	1.083	1.375
0x13	333.3	0.542	1.250	1.542
0x14	300.0	0.708	1.417	1.708
0x15	272.7	0.708	1.583	1.875
0x16	230.8	0.875	1.917	2.208
0x17	187.5	0.875	2.417	2.708
0x18	300.0	0.375	1.583	1.708
0x19	250.0	0.375	1.917	2.042
0x1A	214.3	0.708	2.250	2.375
0x1B	187.5	0.708	2.583	2.708
0x1C	166.7	1.042	2.917	3.042
0x1D	150.0	1.042	3.250	3.375
0x1E	125.0	1.375	3.917	4.042
0x1F	100.0	1.375	4.917	5.042

ICR	SCL Freq	SDA Hold	SCL Start	SCL Stop
0x20	150.0	0.708	3.250	3.375
0x21	125.0	0.708	3.917	4.042
0x22	107.1	1.375	4.583	4.708
0x23	93.8	1.375	5.250	5.375
0x24	83.3	2.042	5.917	6.042
0x25	75.0	2.042	6.583	6.708
0x26	62.5	2.708	7.917	8.042
0x27	50.0	2.708	9.917	10.042
0x28	75.0	1.375	6.583	6.708
0x29	62.5	1.375	7.917	8.042
0x2A	53.5	2.708	9.250	9.375
0x2B	46.9	2.708	10.583	10.708
0x2C	41.7	4.042	11.917	12.042
0x2D	37.5	4.042	13.250	13.375
0x2E	31.3	5.375	15.917	16.042
0x2F	25.0	5.375	19.917	20.042
0x30	37.5	2.708	13.250	13.375
0x31	31.3	2.708	15.917	16.042
0x32	26.8	5.375	18.583	18.708
0x33	23.4	5.375	21.250	21.375
0x34	20.8	8.042	23.917	24.042
0x35	18.8	8.042	26.583	26.708
0x36	15.6	10.708	31.917	32.042
0x37	12.5	10.708	39.917	40.042
0x38	18.8	5.375	26.583	26.708
0x39	15.6	5.375	31.917	32.042
0x3A	13.4	10.708	37.250	37.375
0x3B	11.7	10.708	42.583	42.708
0x3C	10.4	16.042	47.917	48.042
0x3D	9.4	16.042	53.250	53.375
0x3E	7.8	21.375	63.917	64.042
0x3F	6.3	21.375	79.917	80.042

MULT 2

ICR	SCL Freq	SDA Hold	SCL Start	SCL Stop
0x00	600.0	0.583	0.500	0.917
0x01	545.5	0.583	0.583	1.000
0x02	500.0	0.667	0.667	1.083
0x03	461.5	0.667	0.750	1.167
0x04	428.6	0.750	0.833	1.250
0x05	400.0	0.750	0.917	1.333
0x06	352.9	0.833	1.083	1.500
0x07	300.0	0.833	1.333	1.750
0x08	428.6	0.583	0.833	1.250
0x09	375.0	0.583	1.000	1.417
0x0A	333.3	0.750	1.167	1.583
0x0B	300.0	0.750	1.333	1.750
0x0C	272.7	0.917	1.500	1.917
0x0D	250.0	0.917	1.667	2.083
0x0E	214.3	1.083	2.000	2.417
0x0F	176.5	1.083	2.500	2.917
0x10	250.0	0.750	1.500	2.083
0x11	214.3	0.750	1.833	2.417
0x12	187.5	1.083	2.167	2.750
0x13	166.7	1.083	2.500	3.083
0x14	150.0	1.417	2.833	3.417
0x15	136.4	1.417	3.167	3.750
0x16	115.4	1.750	3.833	4.417
0x17	93.8	1.750	4.833	5.417
0x18	150.0	0.750	3.167	3.417
0x19	125.0	0.750	3.833	4.083
0x1A	107.1	1.417	4.500	4.750
0x1B	93.8	1.417	5.167	5.417
0x1C	83.3	2.083	5.833	6.083
0x1D	75.0	2.083	6.500	6.750
0x1E	62.5	2.750	7.833	8.083
0x1F	50.0	2.750	9.833	10.083

ICR	SCL Freq	SDA Hold	SCL Start	SCL Stop
0x20	75.0	1.417	6.500	6.750
0x21	62.5	1.417	7.833	8.083
0x22	53.6	2.750	9.167	9.417
0x23	46.9	2.750	10.500	10.750
0x24	41.7	4.083	11.833	12.083
0x25	37.5	4.083	13.167	13.417
0x26	31.3	5.417	15.833	16.083
0x27	25.0	5.417	19.833	20.083
0x28	37.5	2.750	13.167	13.417
0x29	31.3	2.750	15.833	16.083
0x2A	26.7	5.417	18.500	18.750
0x2B	23.4	5.417	21.167	21.417
0x2C	20.8	8.083	23.833	24.083
0x2D	18.8	8.083	26.500	26.750
0x2E	15.6	10.750	31.833	32.083
0x2F	12.5	10.750	39.833	40.083
0x30	18.8	5.417	26.500	26.750
0x31	15.6	5.417	31.833	32.083
0x32	13.4	10.750	37.167	37.417
0x33	11.7	10.750	42.500	42.750
0x34	10.4	16.083	47.833	48.083
0x35	9.4	16.083	53.167	53.417
0x36	7.8	21.417	63.833	64.083
0x37	6.3	21.417	79.833	80.083
0x38	9.4	10.750	53.167	53.417
0x39	7.8	10.750	63.833	64.083
0x3A	6.7	21.417	74.500	74.750
0x3B	5.9	21.417	85.167	85.417
0x3C	5.2	32.083	95.833	96.083
0x3D	4.7	32.083	106.500	106.750
0x3E	3.9	42.750	127.833	128.083
0x3F	3.1	42.750	159.833	160.083

MULT 4

ICR	SCL Freq	SDA Hold	SCL Start	SCL Stop
0x00	300.0	1.167	1.000	1.833
0x01	272.7	1.167	1.167	2.000
0x02	250.0	1.333	1.333	2.167
0x03	230.8	1.333	1.500	2.333
0x04	214.3	1.500	1.667	2.500
0x05	200.0	1.500	1.833	2.667
0x06	176.5	1.667	2.167	3.000
0x07	150.0	1.667	2.667	3.500
0x08	214.3	1.167	1.667	2.500
0x09	187.5	1.167	2.000	2.833
0x0A	166.7	1.500	2.333	3.167
0x0B	150.0	1.500	2.667	3.500
0x0C	136.4	1.833	3.000	3.833
0x0D	125.0	1.833	3.333	4.167
0x0E	107.1	2.167	4.000	4.833
0x0F	88.2	2.167	5.000	5.833
0x10	125.0	1.500	3.000	4.167
0x11	107.1	1.500	3.667	4.833
0x12	93.8	2.167	4.333	5.500
0x13	83.3	2.167	5.000	6.167
0x14	75.0	2.833	5.667	6.833
0x15	68.2	2.833	6.333	7.500
0x16	57.7	3.500	7.667	8.833
0x17	46.9	3.500	9.667	10.833
0x18	75.0	1.500	6.333	6.833
0x19	62.5	1.500	7.667	8.167
0x1A	53.6	2.833	9.000	9.500
0x1B	46.9	2.833	10.333	10.833
0x1C	41.7	4.167	11.667	12.167
0x1D	37.5	4.167	13.000	13.500
0x1E	31.3	5.500	15.667	16.167
0x1F	25.0	5.500	19.667	20.167

ICR	SCL Freq	SDA Hold	SCL Start	SCL Stop
0x20	37.5	2.833	13.000	13.500
0x21	31.3	2.833	15.667	16.167
0x22	26.8	5.500	18.333	18.833
0x23	23.4	5.500	21.000	21.500
0x24	20.8	8.167	23.667	24.167
0x25	18.8	8.167	26.333	26.833
0x26	15.6	10.833	31.667	32.167
0x27	12.5	10.833	39.667	40.167
0x28	18.8	5.500	26.333	26.833
0x29	15.6	5.500	31.667	32.167
0x2A	13.4	10.833	37.000	37.500
0x2B	11.7	10.833	42.333	42.833
0x2C	10.4	16.167	47.667	48.167
0x2D	9.4	16.167	53.000	53.500
0x2E	7.8	21.500	63.667	64.167
0x2F	6.3	21.500	79.667	80.167
0x30	9.4	10.833	53.000	53.500
0x31	7.8	10.833	63.667	64.167
0x32	6.7	21.500	74.333	74.833
0x33	5.9	21.500	85.000	85.500
0x34	5.2	32.167	95.667	96.167
0x35	4.7	32.167	106.333	106.833
0x36	3.9	42.833	127.667	128.167
0x37	3.1	42.833	159.667	160.167
0x38	4.7	21.500	106.333	106.833
0x39	3.9	21.500	127.667	128.167
0x3A	3.3	42.833	149.000	149.500
0x3B	2.9	42.833	170.333	170.833
0x3C	2.6	64.167	191.667	192.167
0x3D	2.3	64.167	213.000	213.500
0x3E	2.0	85.500	255.667	256.167
0x3F	1.6	85.500	319.667	320.167