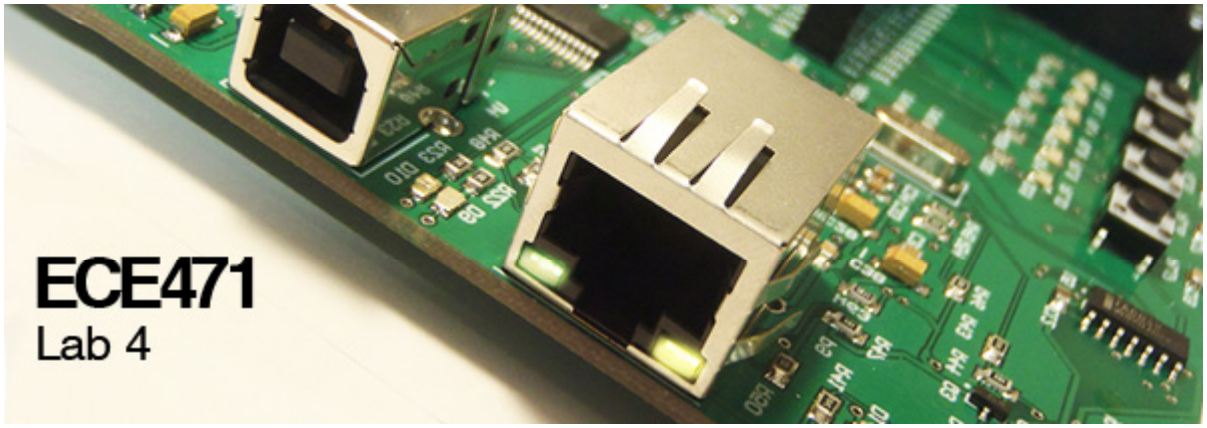


NDSU Embedded Systems II

UART & SPI Mastery

Common Intrachip Protocols

Objective: To obtain familiarity with the following:

- UART
- SPI

By
Nathan Zimmerman

Requirements for lab 4:

Introduction: Intrachip communication protocols are protocols generally used for chip to chip communication internal to a PCB assembly. For example, intrachip would include communication with onboard peripheral and a primary MCU. The most popular protocols used for this type of communication are SPI, I2C, and UART. These protocols are used frequently in industry to communicate with sensors, memory, and other MCUs …ect. External communication protocols such as USB, Ethernet, CAN...ect allows communication between external systems. These forms of external communication will be covered in a latter lab.

Requirements for Lab 4:

1. Implement **Software** based UART via Interrupts in order to receive data.
2. Print received RX data to LCD display
3. Program should be compatible with the FTDI onboard USB->UART interface and putty setup for 9600 baud with no flow control.
4. Replace low level hardware SPI driver with **Software** based SPI driver for LCD display.

Result should look similar to Lab4.axf

Extra Credit:

1. Implement software UART TX and echo characters back to terminal.

Useful Resources:

Putty: (Free HyperTerminal equivalent that works on W7+ OSes)

http://www.putty.org/

Serial Data Wiki Pages

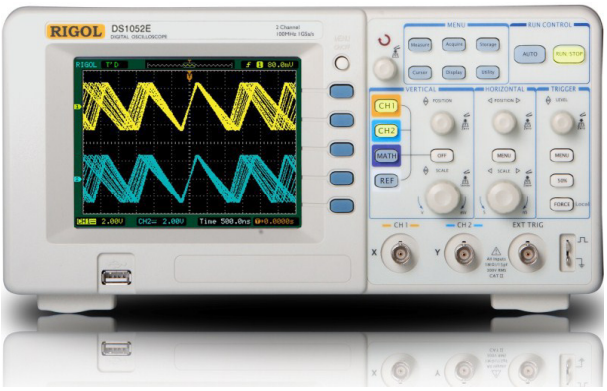
UART:

http://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter

SPI:

http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus

Oscilloscope: Highly recommended you work with oscilloscopes in the lab for this assignment



ASCII

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Source: http://www.asciitable.com/
Info: http://en.wikipedia.org/wiki/Ascii

Serial Data: UART (Universal asynchronous receiver/transmitter)

Specs

Lines: (2+2op)
TXD: Data Transmit
RXD: Data Receive
CTS: Clear to Send (Optional hardware flow control)
RTS: Ready to send (Optional hardware flow control)
Baud rates: 9600, 19200, ….ect

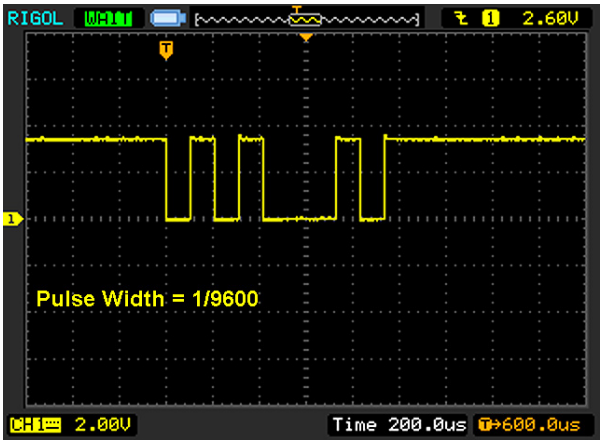
Max Baud*: 115k (if using windows serial port)
Packet Size: 5-8 bits
Packet Overhead: 1 start bit, 1-2 stop bits
Standards: RS-232, RS-422, RS-485 …ect
CLK: asynchronous

UART Packet Example

UART Packet Example: Ascii "E" , Hex 0x45

Idle Line	Start Bit	0 LSB	1	2	3	4	5	6	7 MSB	Stop Bit
1	0	1	0	1	0	0	0	1	0	1

UART Packet Scope Shot:



Info: http://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter

FTDI USB<->UART ASIC

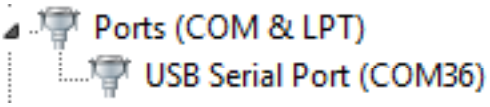
Intro: The FTDI FT232R is a common USB ASIC which implements Its own USB Device Vendor Class which acts as a virtual serial port. This device class is not fully specified by USB 2.0 and consequently a driver is generally required to be installed. Other device classes such as HID are defined and consequently a vendor specific driver install is not required. FTDI has a decent AP note on determining USB classes here: http://www.ftdichip.com/Support/Documents/AppNotes/AN_174_Determining%20USB%20Peripheral%20Device%20Class.pdf

FTDI VCP (Virtual COM Port) driver download here:

http://www.ftdichip.com/Drivers/VCP.htm

You will need to install this driver in order for the FTDI chip on your ECE471 dev board to work properly.

Once installed, the FTDI driver will generate a virtual comport that works almost identically to an actual hardware serial port. Ports can be viewed in windows device manager for example:



On our ECE471 dev board, we use a FTDI breakout board called the MMR232R
Datasheet: http://www.ftdichip.com/Support/Documents/DataSheets/Modules/DS_MM232R.pdf

Reference schematic for connections between the RX and TX Lines

Read and read again the basics of USB: http://en.wikipedia.org/wiki/Universal_Serial_Bus

UART Implementations:

Hardware: A hardware UART embedded into an MCU means that there exists internal circuitry such that UART communication can take place without wasting CLK cycles. Generally a MCU UART module (similar to SPI, I2C…ect) will contain a TX buffer, a RX buffer, setup registers, and some status registers. Once configured properly, the module will implement the communication protocol independently of MCU CLK cycles. CLK cycles will only need to be used to R/W buffers and registers. Using hardware UART (or again SPI, I2C…ect) saves CLK cycles and generally can reduce code size and or complexity. However, in industry, chances are you at one point in time will have to debug a serial interface. Consequently, a software implementation of serial protocol is an excellent way to learn the protocol thoroughly.

Software: A software implementation of a serial protocol is where app code forces the MCU to write and read bits as defined by the protocol. For example, if the serial protocol contains a CLK line, the MCU software must periodically toggle a GPIO in order to generate the signal. Furthermore, upon every CLK edge, the CPU may need to CLK out or read in data on an additional GPIO. Traditionally this is called bit-banging. Although there is already a hardware UART module on the LPC1769, we will be implementing software UART in attempt to better understand the protocol. Hardware UART drivers if desired for latter use can easily be found online.

> **Tips for implementation:** Each UART packet begins with a low start bit. One can program a falling edge GPIO interrupt in order to detect when a packet has started to be transmitted to the LPC1768. The GPIO interrupt can then start a timer which will sample the packet at atleast twice the baud rate of the original protocol. Then by software polling at each timer interrupt, data can be compiled and shifted into a byte based upon what byte number it was received upon.

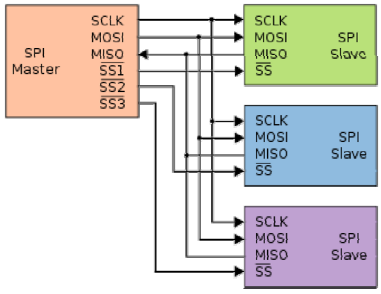
Serial Data: SPI (Serial Peripheral Interface Bus)

SPECS:

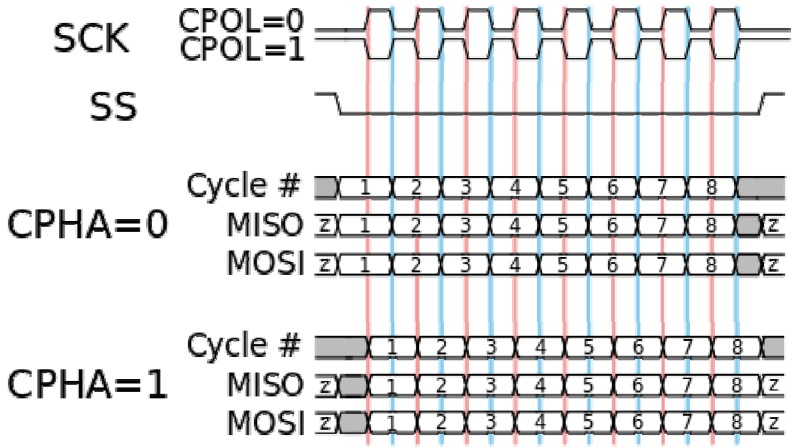
Lines: (3+1op)
MOSI: Master Output Slave Input
MISO: Master Input Slave Output
SCLK: Serial Clock
CS: Chip Select (Sometimes optional if only 1 chip on SPI BUS)

Max Baud*: ~100MHZ (depends on implementation)
Packet Size: user/device defined
Packet Overhead: ~0 (slight overhead due to timing)
Standards: Motorola SPI, Microwire …
CLK: **Synchronous**

Wiring:



Common Packet variation:



Serial Data: SPI (continued)

MAX6675 Example SPI implementation:

Cold-Junction-Compensated K-Thermocouple-to-Digital Converter (0°C to +1024°C)

MAX6675

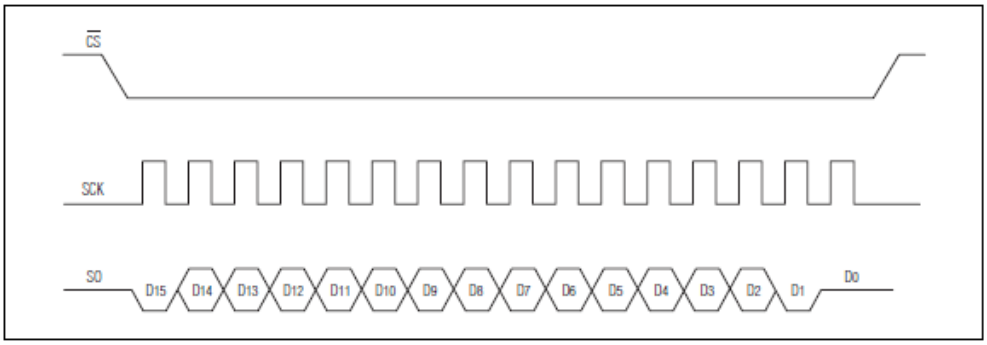


Figure 1a. Serial Interface Protocol

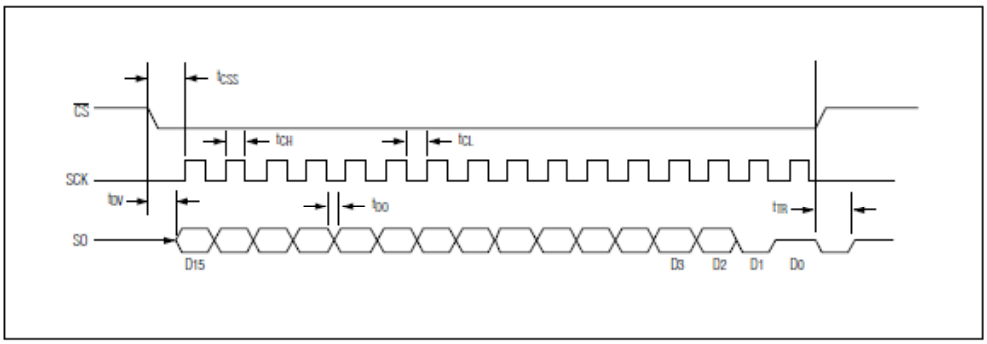


Figure 1b. Serial Interface Timing

BIT	DUMMY SIGN BIT	12-BIT TEMPERATURE READING												THERMOCOUPLE INPUT	DEVICE ID	STATE
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	MSB											LSB		0	Three-state

Lab Report Guidelines:

1. Upload your code files to your assigned GIT repo. You must first create an account on github and email me your account name in order to gain access to your repo.

2. If you are having trouble getting GIT to work initially, email me your code at **NDSU.ECE471@GMAIL.COM.** The code should be attached in the form of .c and .h files or in a complete zip archive . Your email title should be : Lab_Number : Myname. For example: Lab_1 : Nathan Zimmerman. This method of turning in assignments is cumbersome and will not be available long term so attempt to get GIT working ASAP.

   You are allowed to work in groups but you are all expected to turn in your own individual code. You should write the code yourself and it should not be a duplication of someone elses code.

Grading:

100% **: Code runs and by majority meets requirements**

50% **: Code does not meet requirements or does not compile but a reasonable effort was made to complete the lab.**

0% **: Code was not turned in or significantly falls short of meeting requirements.**