Strings in C

A string in the C programming language is simply an array of chars. The length (or end of the string) is determined by the location of the NULL terminator character (\0) in the char array.

char danbuff[50] = {'H','e','l','l','o',' ','W','o','r','l','d','\0'};
or
char danbuff[50] = "Hello World"; This automatically adds the Null terminator
or
char danbuff[] = "Hello World"; This automatically adds the Null terminator and your array is only the size needed for the string (12 chars).

Ascii Character set.
This is a table of the most commonly used characters. Searching the web you can find a complete list of all 256 ascii characters.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
NUL								BS		LF			CR		
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
											ESC				
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	97	98	99	100	101	012	103	104	105	106	107	108	109	110	111
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

ANSI C
int printf(const char *format [, argument]...);
Example:
printf("Robot X=%.2f, Robot Y=%.2f,CamReg= 0x%x.",robx,roby,vCamReg);
In Lab we are going to use a printf function that I wrote for printing to the LCD screen.
int LCDPrintfLine(int line, const char *format[, argument]…); and

Use is identical to printf, but can only print to the 20 characters 1 line of the LCD.
Example:
LCDPrintfLine(1,"X%.2fY%.2fCR0x%x",robx,roby,vCamReg);

To create a character string (Null terminated array of Chars) instead of printing to the LCD use

int sprintf(char *buffer, const char *format [, argument] ...);

Example:
char usb_buffer[125];
sprintf(usb_buffer,"%d,%d,%.3f",count,switchstat,adcvoltvalue);

status of the Interrupt Identification Register. However I have never tested this.

Part 4 : Interfacing Devices to RS-232 Ports

RS-232 Waveforms

So far we have introduced RS-232 Communications in relation to the PC. RS-232 communication is asynchronous. That is a clock signal is not sent with the data. Each word is synchronized using it's start bit, and an internal clock on each side, keeps tabs on the timing.

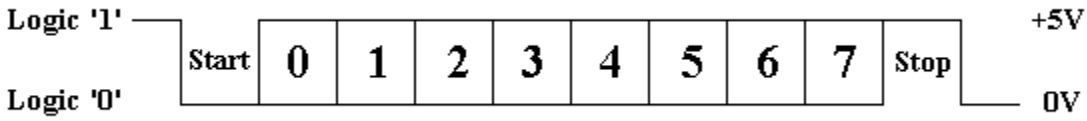


Figure 4 : TTL/CMOS Serial Logic Waveform

The diagram above, shows the expected waveform from the UART when using the common 8N1 format. 8N1 signifies 8 Data bits, No Parity and 1 Stop Bit. The RS-232 line, when idle is in the Mark State (Logic 1). A transmission starts with a start bit which is (Logic 0). Then each bit is sent down the line, one at a time. The LSB (Least Significant Bit) is sent first. A Stop Bit (Logic 1) is then appended to the signal to make up the transmission.

The diagram, shows the next bit after the Stop Bit to be Logic 0. This must mean another word is following, and this is it's Start Bit. If there is no more data coming then the receive line will stay in it's idle state(logic 1). We have encountered something called a "Break" Signal. This is when the data line is held in a Logic 0 state for a time long enough to send an entire word. Therefore if you don't put the line back into an idle state, then the receiving end will interpret this as a break signal.

The data sent using this method, is said to be *framed*. That is the data is *framed* between a Start and Stop Bit. Should the Stop Bit be received as a Logic 0, then a framing error will occur. This is common, when both sides are communicating at different speeds.

The above diagram is only relevant for the signal immediately at the UART. RS-232 logic levels uses +3 to +25 volts to signify a "Space" (Logic 0) and -3 to -25 volts for a "Mark" (logic 1). Any voltage in between these regions (ie between +3 and -3 Volts) is undefined. Therefore this signal is put through a "RS-232 Level Converter". This is the signal present on the RS-232 Port of your computer, shown below.

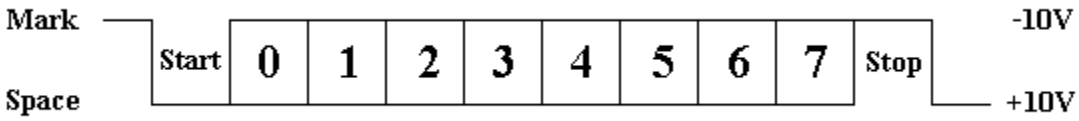


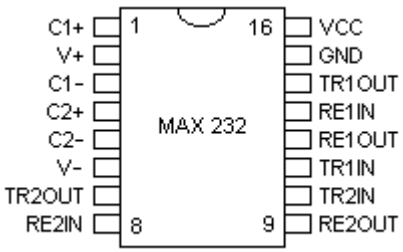
Figure 5 : RS-232 Logic Waveform

The above waveform applies to the Transmit and Receive lines on the RS-232 port. These lines carry serial data, hence the name Serial Port. There are other lines on the RS-232 port which, in essence are *Parallel* lines. These lines (RTS, CTS, DCD, DSR, DTR, RTS and RI) are also at RS-232 Logic Levels.

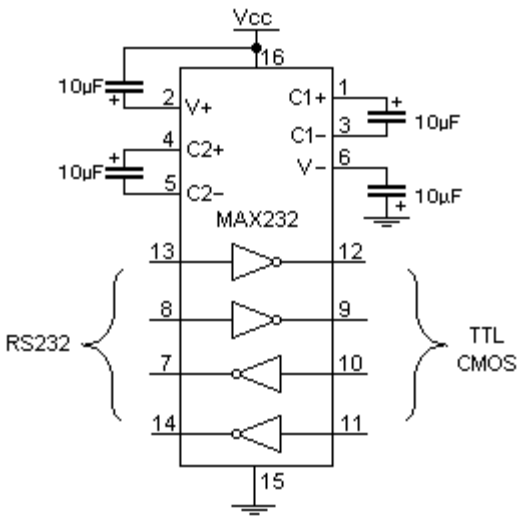
RS-232 Level Converters

Almost all digital devices which we use require either TTL or CMOS logic levels. Therefore the first step to connecting a device to the RS-232 port is to transform the RS-232 levels back into 0 and 5 Volts. As we have already covered, this is done by RS-232 Level Converters.

Two common RS-232 Level Converters are the 1488 RS-232 Driver and the 1489 RS-232 Receiver. Each package contains 4 inverters of the one type, either Drivers or Receivers. The driver requires two supply rails, +7.5 to +15v and -7.5 to -15v. As you could imagine this may pose a problem in many instances where only a single supply of +5V is present. However the advantages of these I.C's are they are cheap.



Above: (Figure 6) Pinouts for the MAX-232,
RS-232 Driver/Receiver.



Right: (Figure 7) Typical MAX-232 Circuit.

Another device is the MAX-232. It includes a Charge Pump, which generates +10V and -10V from a single 5v supply. This I.C. also includes two receivers and two transmitters in the same package. This is handy in many cases when you only want to use the Transmit and Receive data Lines. You don't need to use two chips, one for the receive line and one for the transmit. However all this convenience comes at a price, but compared with the price of designing a new power supply it is very

RS232 Serial Port sending the string 'H','e','y'.

ascii characters 72 (0x48), 101 (0x65) and 121 (0x79).