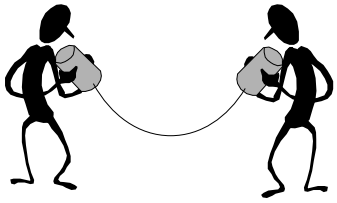
Revision: 13 Oct 2021 (JFF)
Richard W. Wall, University of Idaho, rwall@uidaho.edu

DIGILENT®
www.digilentinc.com
1300 NE Henley Court, Suite 3
Pullman, WA 99163
(509) 334 6306 Voice | (509) 334 6300 Fax

Project 7: Asynchronous Serial Communications



Purpose

The purpose of this project is to learn about asynchronous communications and how to communicate with a microcontroller using a terminal emulation program to implement a point to point serial link between the Cerebot MX7cK and a PC. This project models a system that has both local and remote control.

Minimum Knowledge and Programming Skills

1. Knowledge of C or C++ programming
2. Working knowledge of MPLAB IDE
3. IO pin control
4. ASCII character encoding
5. Use of PC based terminal emulations (Hyperterminal or Putty)

Equipment List

1. Cerebot MX7cK processor board
2. Microchip MPLAB ® X IDE
3. Microchip XC32 Compiler
4. Digilent PmodCLP Parallel Character LCD
5. PC with terminal emulation software (Hyperterminal or Putty)

Software Resources

1. Cerebot MX7cK Board Reference Manual
2. Digilent PmodRS232 Reference Manual
3. MPLAB ® X Integrated Development Environment (IDE)
4. XC32 C/C++ Compiler Users Guide
5. PIC32 Peripheral Libraries for MBLAB C32 Compiler
6. PIC32 Family Hardware Reference Manual Section 21 UART
7. C Programming Reference

Asynchronous Communications Concepts

Asynchronous communications is a serial data protocol that has been in use for many years. Normally eight bits of data are transmitted at a time. There are other less commonly used modes that can send 5, 6, or 7 bits of data. Each byte of data is framed by a start bit and a stop bit. A symbol is defined as a start, data, parity, or stop bit. It is common to define communications speed as bits per second. The bit rate is defined as the inverse of the period of a unit symbol. Although the common standard bit rates are 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, and 115200, communications is possible at any rate provided that the sender and receiver use the same rate. For most asynchronous communications, the term, “baud”, is commonly used interchangeably with the term “bit rate”.

As shown in Figure 1, the idle state of the transmit signal is a logic one or a high voltage level. A start bit is signified by a high to low transition and remains low for one symbol time. The start bit is followed by eight data bits. A logic zero is sent when the signal is a low voltage level for one symbol time. Similarly, a logic one is sent when the signal is a high voltage level for one symbol time. After the transmitting of the data (least significant bit first) there may be an optional parity bit. The parity bit is used for error detection and can be set for even parity or odd parity. For even parity, the parity bit is set high if the number of ones in the preceding eight data bits are odd thus making the total number of 1’s in the data plus parity bit even. Conversely, for odd parity, the parity bit is set high if the number of ones in the 8 data is even. The stop bit(s) is always a logic 1 or high voltage level. If the signal remains high for longer than one symbol period, the stop bit can be thought of as the stop or idle period and may remain in the idle condition arbitrarily long.

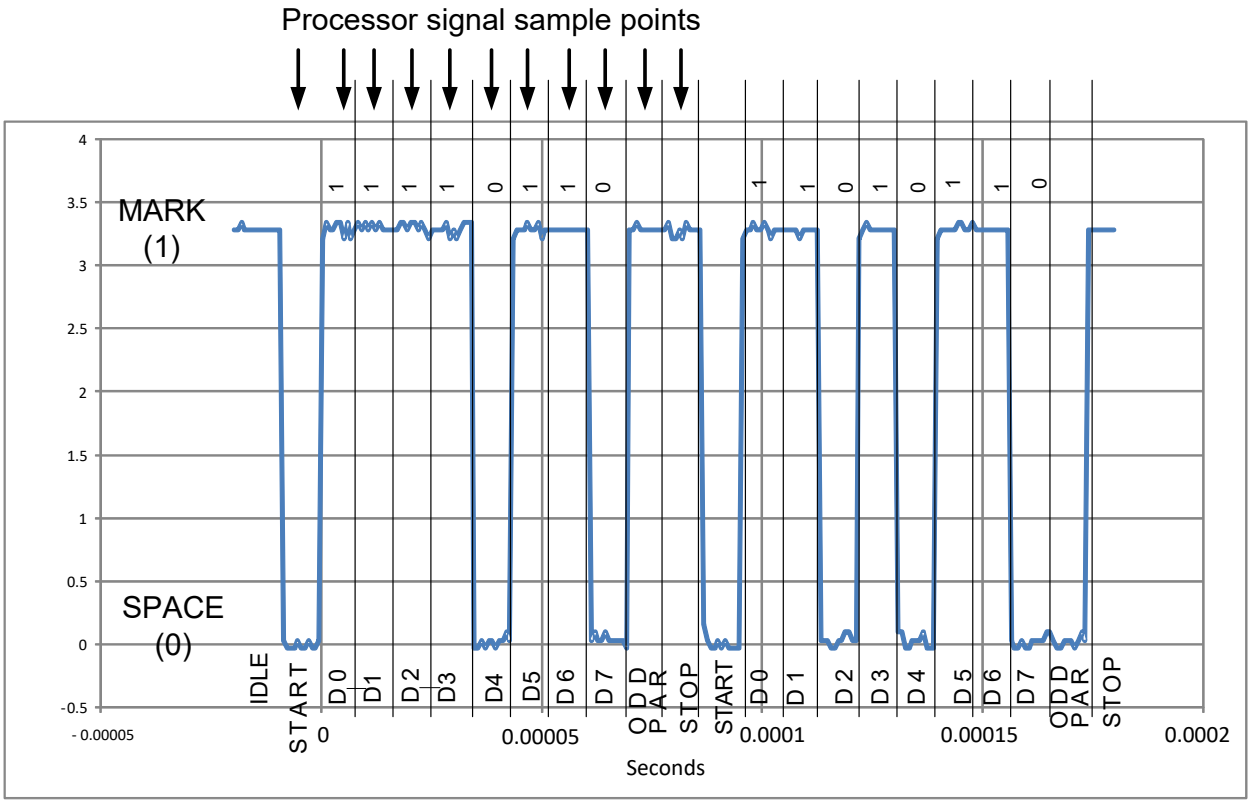


Figure 1. Example of an 115kB asynchronous serial data stream with odd parity

The receiving device unit must use the same bit rate as the sending unit. In the process of receiving serial communications, a processor may generate up to three error flags. A parity error is generated if the parity bit is the incorrect voltage level. The second type of error is is a framing error that is generated if a low voltage level is received in the stop bit position. Thirdly, an overrun error is generated if a software instruction does not read the data byte from the receive buffer before the next byte of data is completely received resulting in the new data byte overwriting the previous data byte.

Each new start bit synchronizes the sampling of the receiving unit. Generally, the receiver recovers the transmission using a clock that is an even multiple of the bit rate. When the receiving unit detects the falling edge of the START bit, it waits one half the symbol period and samples the receive data line. This is illustrated by the processor sample points in Figure 1. If the logic state of the line is zero, then it is recognized as a valid start condition. The receiver then waits full periods to sample the receive data line until all data bits plus any parity bit plus the stop bit have been received. If a stop bit is not received, a framing error is generated.

Just like parallel communications, there are three modes of serial communications: full duplex, half duplex, and simplex. Full duplex describes the operation when a device can simultaneously send and receive data. Half duplex operation allows both sending and receiving but not simultaneously. Simplex operations are where the device can only send or receive data.

The actual data rate is defined as the number of data bits per unit time. Data efficiency is defined as the number of data bits divided by the total number of bits. In this example (RS-232), the efficiency is at best 80% because there are always two extra control bits (start and stop bit) sent for each 8 data bits. If parity is used, the efficiency drops to 73% because 11 bits are needed to communicate 8 bits of data.

UART and RS-232

UART - A Universal Asynchronous Receiver/Transmitter, abbreviated **UART**, is a type of "asynchronous receiver/transmitter". A UART is an electronic device or microprocessor peripheral that translates data between parallel and serial forms. UARTs are usually capable of full duplex communications.

UARTs are typically used in conjunction with asynchronous communication standards such as IrDa, SDI-12, EIA, RS-232, RS-422 or RS-485. RS-232 is a commonly used standard that originated in the early 1960's. The standard defines, the connector pins, the data rates, and voltage levels. The RS-232 standard requires a signal driver that inverts and amplifies the voltages representing logic one and zero. This signal driver IC is on the PmodRS232. As shown in Figure 2, the RS-232 data is bi-polar: +3 TO +12 volts indicate an "ON or 0-state (SPACE) condition" while A -3 to -12 volts indicates an "OFF" 1-state (MARK) condition. (Note the inversion of the logic on the output of the driver IC.) Modern computer equipment ignore the negative level and accepts a zero voltage level as the "OFF" state. For many present day UARTs, the "ON" state may be achieved with lesser positive potential. This means circuits powered by 5 VDC are capable of driving RS-232 circuits directly; however, the overall physical range that the RS-232 signal may be transmitted / received may be dramatically reduced.

Since the RS-232 transmit driver will actively drive the communications line both to high and low voltage levels, it is generally not used in a communications network that consists of many transmitters and receivers sharing a common electrical line. It can, however, be used in a simplex operation where one transmitter communicates with many receivers. Although the PIC32 can host up to six asynchronous serial ports, the Cerebot MX7cK can host up to two – UART1 and UART2.

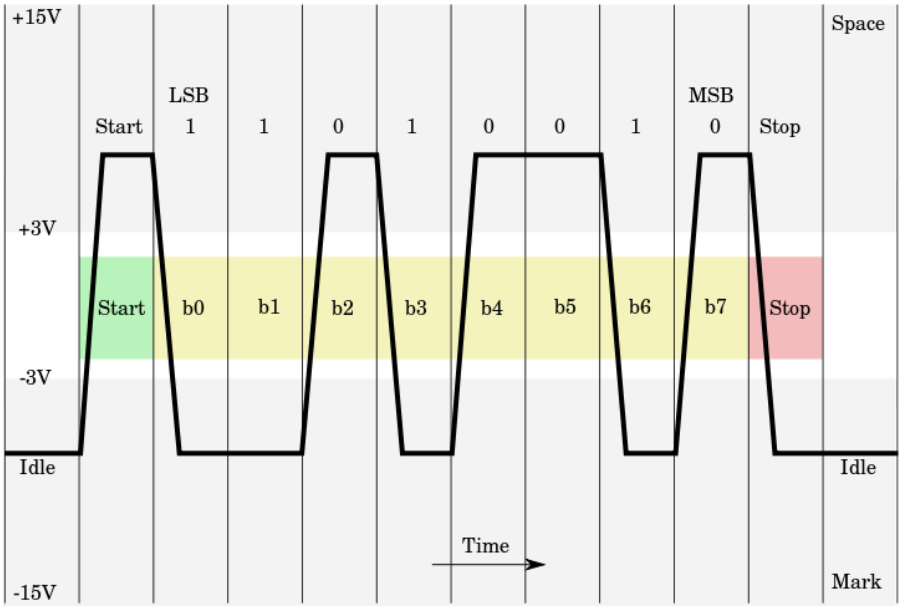


Figure 2. RS-232 signal levels for a asynchronous serial communications

RS-232 Connector Hardware

PCs and other computer systems that have capability to communicate using the RS-232 standard typically use a DB-9 connector to interface with the communications wires. The pin number assignments for a female DB 9 connector are shown in Figure 3. The functionality of the DB 9 connector pins are made on the basis of the connector attaching to Data Terminal Equipment (DTE) or Data Communications Equipment (DCE). The DTE refers to the end device that uses or generates the information. It is the connector on the back of the PC. The DCE refers to equipment that transmits or receives analog or digital signals over the connecting media. It is sometimes called the modem (modulator / demodulator). The connecting media can be wire, fiber optic cable, water (in the case of acoustic modems), or space for cases such as radio frequency (RF) and infrared (IR) modems. Figure 4 shows the pin and functionality for a DTE to DCE connection.

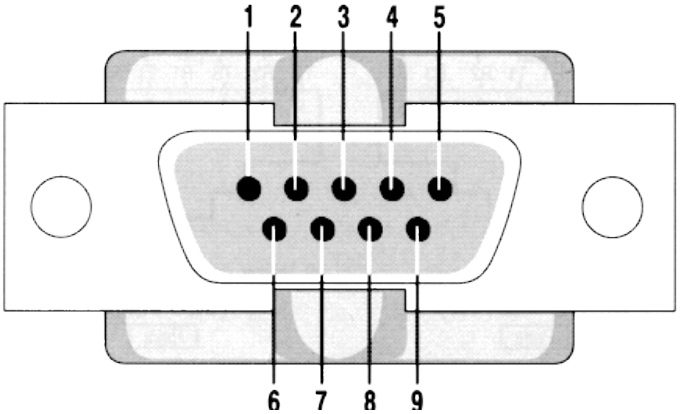


Figure 3. Pin numbering for a female 9 pin Sub D connector

Modem Cable - Straight Cable DB9 to DB9

DTE Device (Computer) **DB9** — **DTE to DCE** Connections — DCE Device (Modem) **DB9**

Pin# DB9	RS-232 Signal Names		Signal Direction	Pin# DB9	RS-232 Signal Names	
#1	Carrier Detector (DCD)	CD	←	#1	Carrier Detector (DCD)	CD
#2	Receive Data (Rx)	RD	←	#2	Receive Data (Rx)	RD
#3	Transmit Data (Tx)	TD	→	#3	Transmit Data (Tx)	TD
#4	Data Terminal Ready	DTR	→	#4	Data Terminal Ready	DTR
#5	Signal Ground/Common (SG)	GND	—	#5	Signal Ground/Common (SG)	GND
#6	Data Set Ready	DSR	←	#6	Data Set Ready	DSR
#7	Request to Send	RTS	→	#7	Request to Send	RTS
#8	Clear to Send	CTS	←	#8	Clear to Send	CTS
#9	Ring Indicator	RI	←	#9	Ring Indicator	RI
Soldered to DB9 Metal - Shield		FGND	—	Soldered to DB9 Metal - Shield		FGND

Figure 4. DB 9 pin designations for DTE to DCE connection

A Null Modem is sometimes referred to as a cross-over cable at allows two devices that have the same DTE or DCE connectors to communicate with each other directly. The connection diagram for a NULL modem shown in Figure 5 illustrates the cross over connections for the Tx and Rx signals as well as the optional handshaking signals.

Modem to Modem Cable - Crossover Cable DB9 to DB9

DCE Device (Modem) DB9 — DCE to DCE Connections — DCE Device (Modem) DB9

Pin# DB9	RS-232 Signal Names	Signal Direction	Pin# DB9	RS-232 Signal Names
#1	Carrier Detector (DCD)	CD	#1	Carrier Detector (DCD)	CD
#2	Receive Data (Rx)	RD	#2	Receive Data (Rx)	RD
#3	Transmit Data (Tx)	TD	#3	Transmit Data (Tx)	TD
#4	Data Terminal Ready	DTR	#4	Data Terminal Ready	DTR
#5	Signal Ground/Common (SG)	GND	#5	Signal Ground/Common (SG)	GND
#6	Data Set Ready	DSR	#6	Data Set Ready	DSR
#7	Request to Send	RTS	#7	Request to Send	RTS
#8	Clear to Send	CTS	#8	Clear to Send	CTS
#9	Ring Indicator	RI	#9	Ring Indicator	RI
Soldered to DB9 Metal - Shield		FGND	Soldered to DB9 Metal - Shield		FGND

Note: Signal directions reversed if devices are DTE to DTE - "Null Modem" cable for DTE devices also connects pins #1 & #6 on each side to simulate Carrier (CD) which is required by some Terminal program software.

Figure 5. DB 9 pin designations for DCE to DCE Null Modem connections

Managing Text Strings

The data that is communicated over a serial communications connection is not constrained to ASCII text. However, using only ASCII text frequently simplifies the development of a communication system. The interface with the LCD used in Project 6 is an example of parallel communication using ASCII text strings. In C, a text string is an array of characters that uses the data value of zero to indicate the end of the string regardless of the array size (provided the string length is less than the array size.) The character data type, "char" is 8-bit that may be signed or unsigned. The data type of signed is assumed if not explicitly declared as unsigned.

There are two methods for handling serial data – a character at a time or a line at a time. The programs provided by the code provided in Appendix C handles both character based and string based drivers for the serial port. There is no constraint for the value of data communicated using character the based functions "*getcU1*" and "*putcU1*". The value of data communicated using string based functions, "*putsU1*" and "*getstrU1*", are constrained to ASCII encoded characters.

Normally, the "*getcU1*" and "*getstrU1*" functions are both blocking functions. This means that when the function "*getcU1*" is called, it will wait indefinitely for a serial character to be received. Since ""*getstrU1*"" calls the function, "*getcU1*" and waits for all characters in the string to be received, it too is a blocking function. The code provided in Appendix C has been written specifically to avoid the blocking problem. Greater detail on the implementation of the non-blocking getcU1 and getstrU1 functions is provided in the code documentation.

Writing an ASCII string to the serial port is handled by the "*putsU1*" function. ASCII control characters, carriage return (CR, 0x0D, '\r') and line feed (LF, 0x0A, '\n') are automatically added to the end of the text string. The prototype for this function is "*int putsU1(const char *s)*". Even

though this prototype implies that a value is returned, in reality the value returned has no meaning. The character array, "str", must be a null terminated text string. Two examples of using the "putsU1" function are shown in Listing 1.

Listing 1. Examples of methods to use the *putsU1* function.

```
putsU1("Hello World");  // Constant text string

char str[32];           // Size of the character array must anticipate the
                        // longest possible string plus 1.
sprintf(str,"Hello World");
putsU1(str);                  // Variable text string set to "Hello World"
```

Both of the methods shown in Listing 1 are equivalent to *printf("Hello World\r\n");*

The "*getstrU1*" function will return an entire string of text. When the string of data is sent over the serial port, the end of the text line is indicated by the carriage return character or when the number of characters received equals the value specified by a parameter to limit the string length. (This string limit value prevents buffer overflows that cause systems to crash.) When a carriage return character is received, it is replaced in the receive string array by a null character (value = 0x00). In the case where the number of characters received reaches the specified limit, the last element in the receiving string array is set to zero (null). The line feed ASCII control character is ignored. The "*getstrU1*" function also processes the BACKSPACE character such that a destructive backspace operation is implemented. (Characters are deleted during backspace operations. This operation is sometimes referred to as destructive backspace that implements the same operation as the backspace key on PC terminal emulator.)

The prototype for the "*getstrU1*" is "*char * getstrU1(char *s, int len);*" The function returns a pointer to the character string "str". Listing 2 shows how to call the *"getstrU1"* function.

Listing 2. Exampe code for using the *"getstrU1" function.*

```
char str[20]; // Buffer size is set to hold the maximum expected
              // number of characters per line plus 1.
while(!getstrU1(str, sizeof(str)));
```

Passing numerical data over text strings can be difficult without a mechanism to parse the data from the fields of data in the string. The Microchip XC32 compiler supports the "*sscanf*" function. To use "s*scanf*", you must include <*stdio.h*> at the top of the program. Additional string functions such as *strcmp(s1,s2)* (string compare) and *strcpy (s1,s2)* (string copy) are available by including <*string.h*>. The reader is referred to the Microchip 32-Bit Language Tools Library guide for additional information on functions to assist processing string data. The code shown in Listing 3 illustrates the use of the "*sscanf*" function. For this illustration, assume that the text "*123 444 dog cat*" is sent to the PIC32MX7 UART.

Listing 3. Example code for using the string scanf function

```
#include <stdio.h>
#include <string.h>
int main(void)
{
char str[21];      // Buffer size is set to hold the maximum expected
                   // number of characters per line plus the NULL terminator
int x1, x2;
char s1[5], s2[5];

      while(!getstrU1(str, sizeof(str)));
      sscanf(str,"%d %d %s %s", &x1, &x2, s1, s2);
}
```

The format of the string to be parsed with the "*sscanf*" function indicates that two integer values will be followed by two strings. The ASCII space character for this example is the delimiter (field separator). After the "*scanf*" function has executed, x1 will equal 123 and x2 will equal 444. The character arrays s1 and s2 will contain the null terminated strings "dog" and "cat" respectively. These strings s1 and s2 must be of sufficient length to hold the maximum number of characters parsed into these strings to avoid run time errors. Notice the arguments in the variable list are passed by reference. When entering the input string, the space character is the delimiter or separator between parameters. If more than one space character is entered between parameters, the additional space characters are ignored.

UART Programming

The UART peripheral must be initialized to set the communications bit rate, the parity option, and the number of data bits to send and receive. Additionally, the transmit and receive have separate enable bits. The XC32 peripheral library provides the UART open function shown in Listing 4.

Listing 4. UART 1 initialization code

```
OpenUART1( (UART_EN | UART_BRGH_FOUR | UART_NO_PAR_8BIT),
                        (UART_RX_ENABLE | UART_TX_ENABLE) , brg );
```

The UART_EN bit sets the transmit and receive IO pins for their alternate function to support UART communications. The parameter, UART_BRGH_FOUR, specifies that the reconstruction clock rate as four times the serial port bit rate. The reconstruction clock is used to set the sample points in the middle of the symbol as illustrated in Figure 1. Hence the third parameter in the *OpenUART* function, *brg*, is determined by dividing the peripheral bus clock by four times the desired bit rate.

The UART_NO_PAR_8BIT option specifies both the data size and no parity bit for error detection. To use even parity error detection, substitute UART_EVEN_PAR_8BIT or, for odd parity, use UART_ODD_PAR_8BIT into this parameter field.

Appendix C contains the code for UART operations. UART1 is initialized by calling the function, *initialize_uart1(int bit_rate, int parity)* from the users application. The *parity* parameter can be specified as NO_PARITY, EVEN_PARITY, or ODD_PARITY. Other serial communication functions provided in the code shown in Appendix C are for transmitting and receiving single characters and sending and receiving text strings.

The serial receive functions, *getcU1* and *getstrU1* were developed specifically to be non-blocking. They will return a value to the calling function indicating whether there is new data available. These functions return a false value if no new data has been received. When new data has been received, a logical true is returned. The actual character or string is contained in a variable that is passed by reference.

The code provided in Appendix C can be easily converted to use UART 2 by changing all references to UART1 to UART2 and U1 to U2.

Project Tasks

The data that is entered on the PC using the terminal emulation program will be echoed to the LCD and it will be decoded to set the stepper motor parameters direction, mode, and stepper motor speed. The code that reads and decodes the serial data and sets the stepper motor operating parameters will run in the background while all of the preemptive code from Project 5 will run in the foreground. For this project, you will need to replace fixed pre-computed step delay times with a formula that computes the step delay time at run time from a range of RPM values.

After you have correctly merged the Project 5 code with the UART handling code, you will observe that, although the program is waiting for data from the UART, due to the use of preemption, the stepper motor will continue to run and button inputs will continue to be detected.

The specifications require that the LCD be updated in the “while(1)” loop in the main function and also within the decode buttons function called by the change-notice (CN) ISR. Since all LCD operations are non-reentrant, the operations involving the LCD in the main function must be protected from CN interrupts[1]. This is most efficiently accomplished by disabling the CN

[1] Technically, the global variables representing the motor settings should also be protected, but they can be changed much more quickly compared to the LCD.

interrupts using the macro provided by the C32 Peripheral Library, "*mCNIntEnable(FALSE)*" to disable CN interrupts and "*mCNIntEnable(TRUE)*" to re enable CN interrupts.

To prevent getting overwhelmed by the magnitude of the number of files in this project, I suggest that you start with the UART function – sending and receiving text strings and using the "s*scanf*" function to parse the direction, mode, and stepper motor speed data. Temporarily, comment out the portions of the initialization function that enable the Timer 1 and CN interrupts.

Following the development of the UART code, add the LCD operations. Finally, add the Project 5 operations by removing the comments for the section of code that enables the interrupts. Remember to call the initializing functions for the UART and the LCD prior to attempting to use those resources.

Project Specifications

The hardware configuration for Project 7 is shown in Figure 6. It requires all of the code and hardware used for Projects 5 and 6 in addition to hardware and software to support the serial communications with the PC. This project will require you to input text data from the serial port that will set the direction, mode, and speed of the stepper motor. This interface will be in addition to all of the controls provided in Project 5. The text from the serial port will be echoed to the LCD. It will be good for you to review the documentation on how the following text manipulating functions are implemented: printf, sprintf, scanf, and strcmp,

The functional specifications are:

1. Communications will use the PC terminal emulation program for a bit rate of 19200, odd parity, 8 data bits and one stop bit.

2. The UART serial text handling code will be contained in the main function "*while(1)*" loop.

3. The text will be entered using the format direction, mode, and speed in rpm with space delimiters. For example the text, "CW FULL 11" will be interpreted as clockwise direction, full step mode, at 11 RPM. You will use the following text to control the stepper motor:
    a. Direction – CW or CCW
    b. Mode – HALF or FULL
    c. RPM – range of 1 to 30 (integer values only)

4. After receiving the line of text from the UART, you will clear the LCD before echoing the received string from the UART to line 1 of the LCD.

5. Next, use the "*sscanf*" function to parse the text string.

a. You will need two string array variables to hold the direction and mode settings. A third integer variable will hold the value for RPM.
b. The direction and mode string variables can be decoded using the string compare function "*strcmp*". An example of using this function would be:

x = strcmp(mode_txt,"FULL");

Only if the string of data in mode_txt is exactly equal to FULL will the value of "x" equal zero. You must include <string.h> to be able to use this function.
c. After decoding the string data, set the global variables that control direction, mode, and step delay (computed from RPM).

6. Create a C source file and call it project7.c. This program will contain the function *main* and process the serial text. Put the following tasks inside the while(1) loop:
   a. Wait for line of text using the "getstr" function
   b. Disable CN interrupts
   c. Clear the LCD
   d. Echo the string to the LCD
   e. Parse the string into the three variables
   f. Set the stepper motor control global variables
   g. Enable CN interrupts
7. Modify the button decode function to report the stepper motor settings to the LCD and to the PC terminal emulation program via the serial port using the same format as in Step 3. The text string should be sent to the UART using the "printf" function.
8. BTN1 and BTN2 will operate as they did in Project 5 in terms of motor settings.

Project Testing

1. Create a function check list and verify that the system performs to the specifications. (Include with your report.) See "Lab report questions" for more details.

Appendix A: Project 7 Parts Configuration

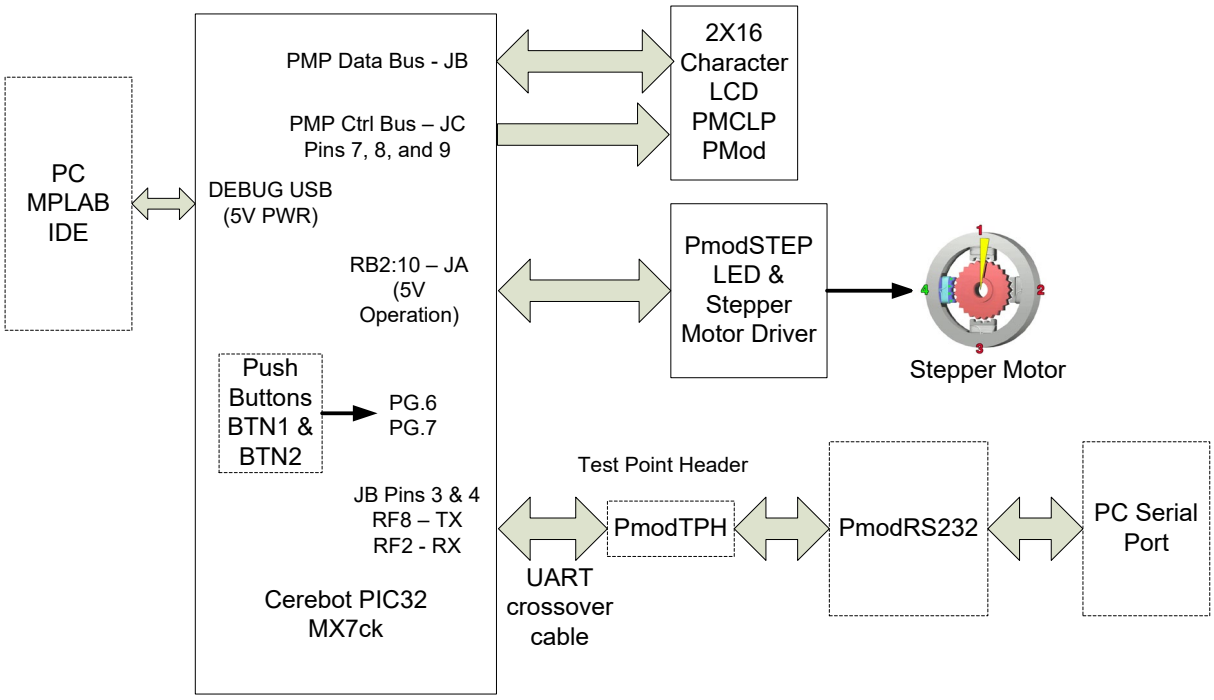


Figure 6. Block diagram of the equipment used in Project 7[2].

Appendix B: PmodCLP

[2] We no longer use the PmodRS232. Instead, a USB cable connects the PC to the UART connector on the Cerebot.

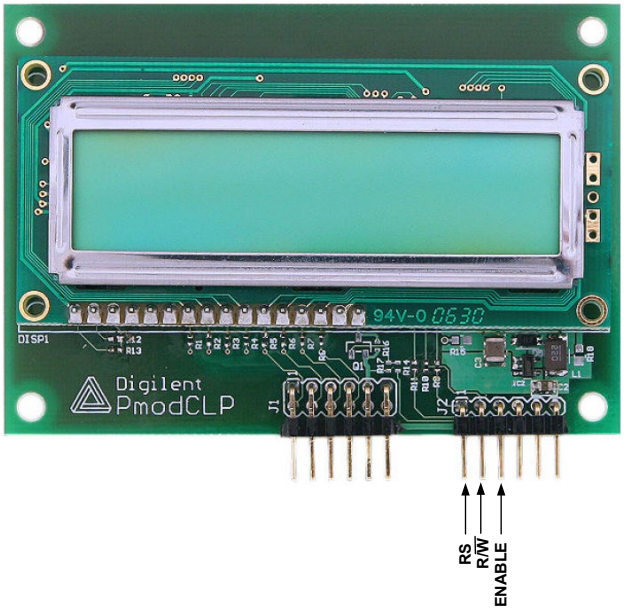


Figure 7. PmodCLP Character LCD pin identification

Appendix C: PIC – UART C Code

Listing 5: comm header file

```
/* File name:      comm.h
 * Author:         Richard Wall
 * Date:           June 12, 2012
 */

#ifndef __COMM_H__
    #define __COMM_H__
    #define BACKSPACE       0x08
    #define NO_PARITY       0
    #define ODD_PARITY      1
    #define EVEN_PARITY     2
#endif

void initialize_uart1(unsigned int bit_rate, int parity);
void _mon_putc(char c); // Called by system to implement "printf" functions
int putcU1( int c);       // Sends single character to UART
int getcU1( char *ch);    // Gets single character from UART
int putsU1( const char *s);     // Send string to UART1
int getstrU1( char *s, unsigned int len ); // Gets string from UART
// End of comm.h
```

Listing 6: comm C file

```
/* File name:      comm.c
```

```
 * Author:        Richard Wall
 * Date:          June 12, 2013
 *
 * This code also provides access to the "printf" function
 * on UART Serial Port 1
 *
 * Cerebot MX7cK requires UART crossover cable
 * Connector Pin    Pmod Pin      Function
 *    2                4           MX7 Rx
 *    3                3           MX7 Tx
 *    5                5           Gnd
 *    6                6           Vcc
*/

#include <plib.h>
#include <stdio.h>            // required for printf
#include "comm.h"
#include "CerebotMX7ck.h"     // has info regarding the PB clock

/* initialize_comm  FUNCTION DESCRIPTION ***********************************
SYNTAX:           void initialize_comm(unsigned int bit_rate, int parity);
KEYWORDS:         UART, initialization, parity
DESCRIPTION:      Initializes UART1 comm port for specified bit rate using
                  the assigned parity
PARAMETER 1:      bit_rate: the communications bit rate
PARAMETER 2:      parity: NO_PARITY, ODD_PARITY, or EVEN_PARITY
RETURN VALUE:     None
NOTES:            9 bit mode MARK or SPACE parity is not supported
END DESCRIPTION *********************************************************/
void initialize_uart1(unsigned int bit_rate, int parity)
{
unsigned int brg;

    brg = (unsigned short)(( (float)FPB / ( (float)4 * (float) bit_rate))-
            (float)0.5);      // provides rounding for greatest accuracy
    switch(parity)
    {
        case NO_PARITY:
            OpenUART1( (UART_EN | UART_BRGH_FOUR | UART_NO_PAR_8BIT),
                       (UART_RX_ENABLE | UART_TX_ENABLE) , brg );
            break;
        case ODD_PARITY:
            OpenUART1( (UART_EN | UART_BRGH_FOUR | UART_ODD_PAR_8BIT),
                       (UART_RX_ENABLE | UART_TX_ENABLE) , BRG );
            break;
        case EVEN_PARITY:
            OpenUART1( (UART_EN | UART_BRGH_FOUR | UART_EVEN_PAR_8BIT),
                       (UART_RX_ENABLE | UART_TX_ENABLE) , BRG );
```

```
            break;
    }
    printf("\n\rCerebot MX7ck Serial Port 1 ready\n\r");
}

/* _mon_putc FUNCTION DESCRIPTION ****************************************

SYNTAX:           void _mon_putc( char c);
KEYWORDS:         printf, console, monitor
DESCRIPTION:      Sets up serial port to function as console for printf.
                  This function is used only by system.
PARAMETER 1:      c: The character to send to monitor
RETURN VALUE:     None
NOTES:            This function will block until space is available
                  in the transmit buffer
END DESCRIPTION *******************************************************/
void _mon_putc(char c)
{
    while(BusyUART1());
    WriteUART1((unsigned int) c);
} //  end of _mon_putc

/* putcU1 FUNCTION DESCRIPTION ***************************************
SYNTAX:           int putcU1( int c);
KEYWORDS:         UART, character
DESCRIPTION:      Waits while UART1 is busy (buffer full) and then sends a
                  single byte to UART1.
PARAMETER:        c: character to send
RETURN VALUE:     character sent
NOTES:            This function will block until space is available in the
                  transmit buffer. The data type for the parameter, "c" is
                  int but only the least significant 8 bits of the parameter
                  are sent out the serial port.
END DESCRIPTION *******************************************************/
int putcU1( int c)
{
    while(BusyUART1());
    WriteUART1((unsigned int) c);
    return c;
} //  end of putcU1

/* getcU1 FUNCTION DESCRIPTION ***************************************
SYNTAX:           int getcU1( char *ch_ptr);
KEYWORDS:         character, get
DESCRIPTION:      Checks for a new character to arrive to the UART1 serial
                  port.
PARAMETER 1:      ch_ptr: pointer to character
RETURN VALUE:     TRUE = new character received
```

```
                    FALSE = No new character
NOTES:              This function does not block for new character not ready
END DESCRIPTION *****************************************************/
int getcU1( char *ch_ptr)
{
    if( !DataRdyUART1())      // wait for new char to arrive
        return FALSE;         // Return new data not available flag
    else
    {
        *ch_ptr = ReadUART1();// read the char from receive buffer
        return TRUE;          // Return new data available flag
    }
}// end of getcU1

/* putsU1 FUNCTION DESCRIPTION ***********************************************
SYNTAX:             int putsU1( const char *s);
KEYWORDS:           UART, string
DESCRIPTION:        Sends a NULL terminates text string to UART1 with
                    CR and LF appended
PARAMETER 1:        pointer to text string
RETURN VALUE:       Logical TRUE
NOTES:              This function will block until space is available
                    in the transmit buffer
END DESCRIPTION *****************************************************/
int putsU1( const char *s)
{
    putsUART1(s);             // from peripheral library
    putcUART1( '\r');
    putcUART1( '\n');
    return 1;
} // putsU1

/* getstrU1 FUNCTION DESCRIPTION *********************************************
SYNTAX:             int getstrU1( char *s, unsigned int len );
KEYWORDS:           string, get, UART
DESCRIPTION:        This function assembles a line of text until the number of
                    characters assembled exceed the buffer length or an ASCII
                    CR control character is received.  This function echos each
                    received character back to the UART. It also implements a
                    destructive backspace. ASCII LF control characters are
                    filtered out.  The returned string has the CR character
                    removed and a NULL character appended to terminate the text
                    string.
PARAMETER 1:        s: pointer to string
PARAMETER 2:        len: maximum string length
RETURN VALUE:       TRUE = EOL signals the line of text is complete and ready
                    to process.
                    FALSE = waiting for end of line
```

```
NOTES:            It is presumed that the buffer pointer or the buffer length
                  does not change after the initial call asking to
                  receive a new line of text.  This function does not block
                  for no character received. A timeout can be added to this
                  function to free resource. There is no way to restart
                  the function after the first call until a EOL has been
                  received.  Hence this function has denial of service
                  security risks.
END DESCRIPTION **********************************************************/
int getstrU1( char *s, unsigned int len )
{
     static int eol = TRUE;  // End of input string flag
     static unsigned int buf_len;
     static char *p1;        // copy #1 of the the buffer pointer
     static char *p2;        // copy #2 of the  the buffer pointer
     char ch;                // Received new character

    if(eol)             // Initial call to function - new line
    {                   // Make two copies of pointer - one for
        p1 = s;         // receiving characters and one for marking
        p2 = s;         // the starting address of the string.  The
        eol = FALSE;    // second copy is needed for backspacing.
        buf_len = len - 1;  // Save maximum number of received chars
    }

    if(!(getcU1(&ch)))  // Check for character received
    {
        return FALSE;   // Bail out if not
    }
    else
    {
        *p1 = ch;       // Save new character in string buffer
        putcU1( *p1);   // echo character
        switch(ch)      // Test for control characters
       {
            case BACKSPACE:
                if ( p1>p2)
                {
                    putcU1(' ');     // overwrite the last character
                    putcU1(BACKSPACE);
                    buf_len++;
                    p1--;            // back off the pointer
                }
                break;
            case '\r':              // end of line, end loop
                putcU1('\n');      // add line feed
                eol = TRUE;         // Mark end of line
                break;
```

```
            case '\n':              // line feed, ignore it
                break;
            default:
                p1++;               // increment buffer pointer
                buf_len--;          // decrement length counter
        }
    }
    if( buf_len == 0 || eol)        // Check for buffer full or end of line
    {
       *p1 = '\0';                  // add null terminate the string
       eol = TRUE
       return TRUE;                 // Set EOL flag
    }
    return FALSE;                   // Not EOL
} // end of getstrU1

// End of comm.c
```