4. SERIAL COMMUNICATION

Topics:

- Serial communication and RS-232c
- ASCII ladder logic functions
- Design case

Objectives:

- To understand serial communications with RS-232
- Be able to use serial communications with a PLC

4.1 INTRODUCTION

Multiple control systems will be used for complex processes. These control systems may be PLCs, but other controllers include robots, data terminals and computers. For these controllers to work together, they must communicate. This chapter will discuss communication techniques between computers, and how these apply to PLCs.

The simplest form of communication is a direct connection between two computers. A network will simultaneously connect a large number of computers on a network. Data can be transmitted one bit at a time in series, this is called serial communication. Data bits can also be sent in parallel. The transmission rate will often be limited to some maximum value, from a few bits per second, to billions of bits per second. The communications often have limited distances, from a few feet to thousands of miles/kilometers.

Data communications have evolved from the 1800's when telegraph machines were used to transmit simple messages using Morse code. This process was automated with teletype machines that allowed a user to type a message at one terminal, and the results would be printed on a remote terminal. Meanwhile, the telephone system began to emerge as a large network for interconnecting users. In the late 1950s Bell Telephone introduced data communication networks, and Texaco began to use remote monitoring and control to automate a polymerization plant. By the 1960s data communications and the phone system were being used together. In the late 1960s and 1970s modern data communications techniques were developed. This included the early version of the Internet, called ARPAnet. Before the 1980s the most common computer configuration was a centralized mainframe computer with remote data terminals, connected with serial data line. In the 1980s the personal computer began to displace the central computer. As a result, high speed networks are now displacing the dedicated serial connections. Serial communications and networks are both very important in modern control applications.

An example of a networked control system is shown in Figure 47. The computer and PLC are connected with an RS-232 (serial data) connection. This connection can only connect two devices. Devicenet is used by the Computer to communicate with various actuators and sensors. Devicenet can support up to 63 actuators and sensors. The PLC inputs and outputs are connected as normal to the process.

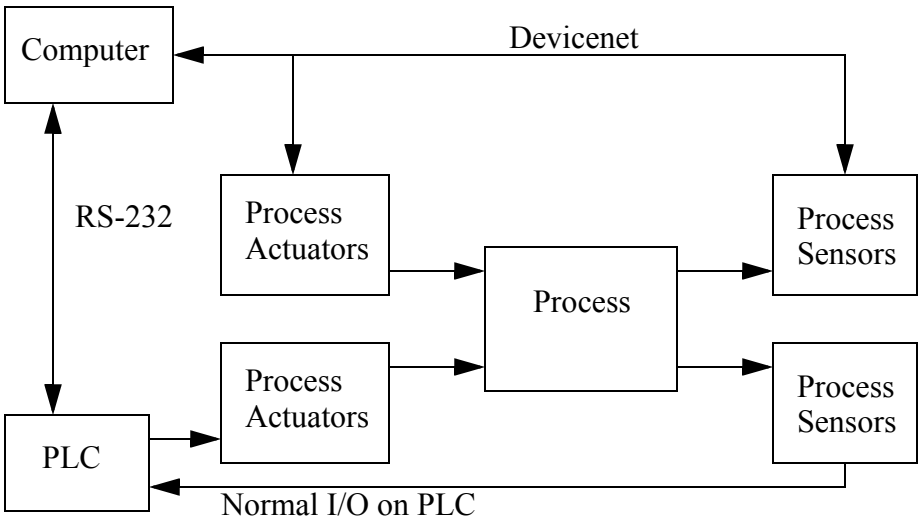


Figure 47 A Communication Example

4.2 SERIAL COMMUNICATIONS

Serial communications send a single bit at a time between computers. This only requires a single communication channel, as opposed to 8 channels to send a byte. With only one channel the costs are lower, but the communication rates are slower. The communication channels are often wire based, but they may also be can be optical and radio. Figure 48 shows some of the standard electrical connections. RS-232c is the most common standard that is based on a voltage change levels. At the sending computer an input will either be true or false. The *line driver* will convert a false value *in* to a *Txd* voltage between +3V to +15V, true will be between -3V to -15V. A cable connects the *Txd* and *com* on the sending computer to the *Rxd* and *com* inputs on the receiving computer. The receiver converts the positive and negative voltages back to logic voltage levels in the receiving computer. The cable length is limited to 50 feet to reduce the effects of electrical noise. When RS-232 is used on the factory floor, care is required to reduce the effects of electrical noise - careful grounding and shielded cables are often used.

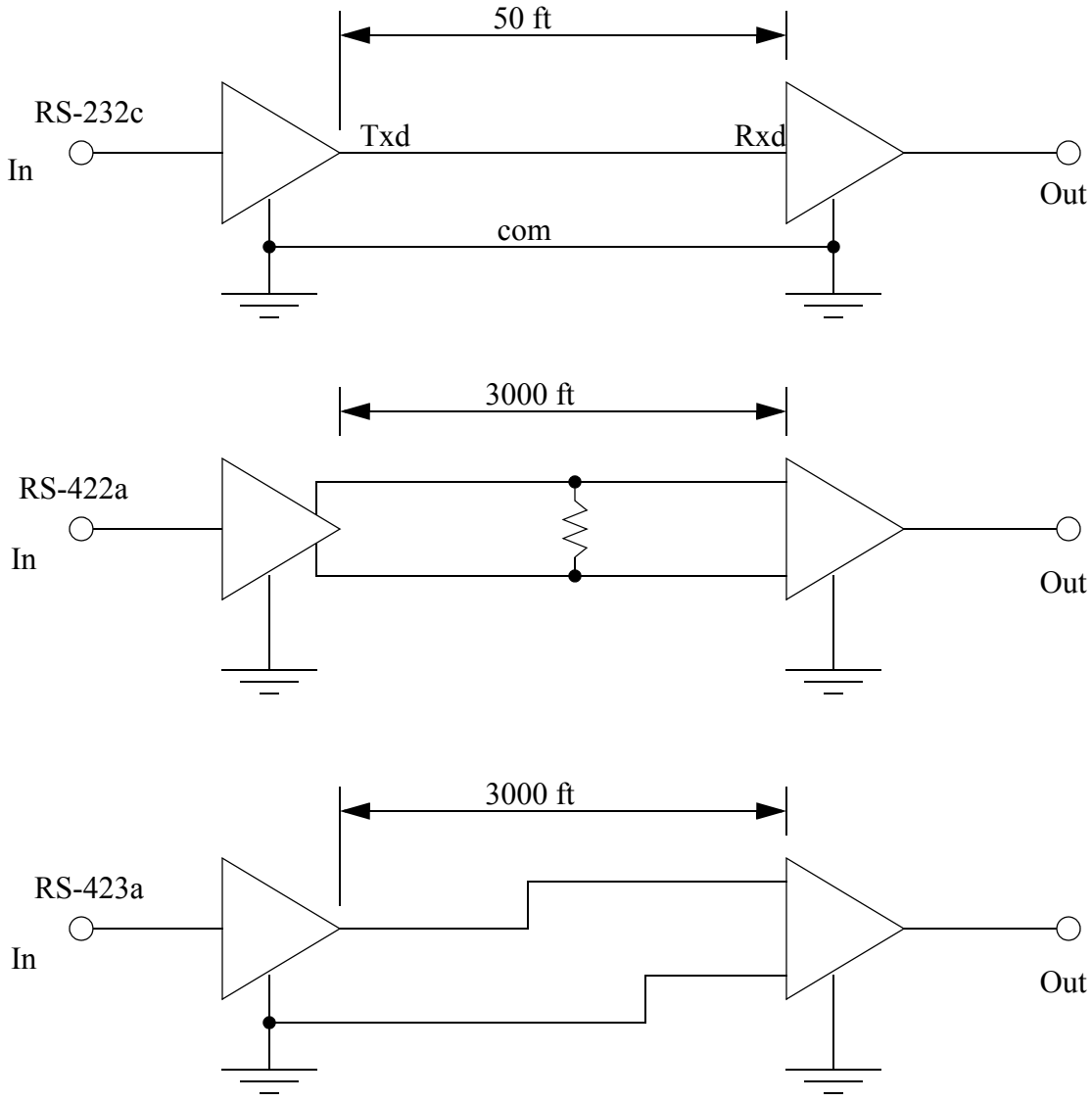


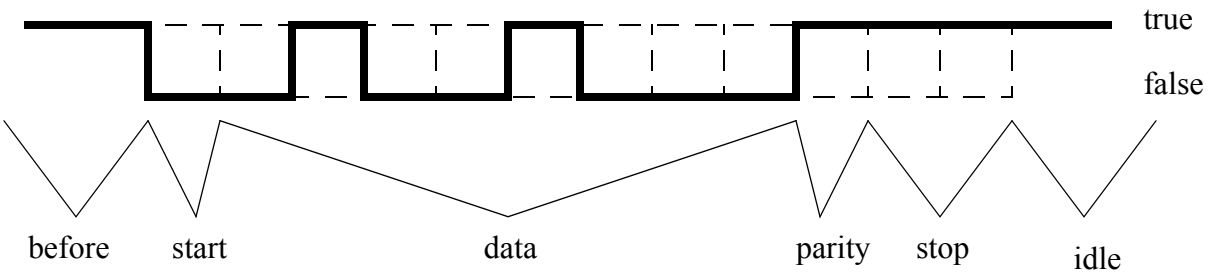
Figure 48 Serial Data Standards

The RS-422a cable uses a 20 mA current loop instead of voltage levels. This makes the systems more immune to electrical noise, so the cable can be up to 3000 feet long. The RS-423a standard uses a differential voltage level across two lines, also making the system more immune to electrical noise, thus allowing longer cables. To provide serial communication in two directions these circuits must be connected in both directions.

To transmit data, the sequence of bits follows a pattern, like that shown in Figure 49. The transmission starts at the left hand side. Each bit will be true or false for a fixed period of time, determined by the transmission speed.

A typical data byte looks like the one below. The voltage/current on the line is made true or false. The width of the bits determines the possible bits per second (bps). The value shown before is used to transmit a single byte. Between bytes, and when the line is idle, the *Txd* is kept true, this helps

the receiver detect when a sender is present. A single start bit is sent by making the *Txd* false. In this example the next eight bits are the transmitted data, a byte with the value 17. The data is followed by a parity bit that can be used to check the byte. In this example there are two data bits set, and even parity is being used, so the parity bit is set. The parity bit is followed by two stop bits to help separate this byte from the next one.



Descriptions:

before - this is a period where no bit is being sent and the line is true.
start - a single bit to help get the systems synchronized.
data - this could be 7 or 8 bits, but is almost always 8 now. The value shown here is a byte with the binary value 00010010 (the least significant bit is sent first).
parity - this lets us check to see if the byte was sent properly. The most common choices here are no parity bit, an even parity bit, or an odd parity bit. In this case there are two bits set in the data byte. If we are using even parity the bit would be true. If we are using odd parity the bit would be false.
stop - the stop bits allow a pause at the end of the data. One or two stop bits can be used.
idle - a period of time where the line is true before the next byte.

Figure 49 A Serial Data Byte

Some of the byte settings are optional, such as the number of data bits (7 or 8), the parity bit (none, even or odd) and the number of stop bits (1 or 2). The sending and receiving computers must know what these settings are to properly receive and decode the data. Most computers send the data asynchronously, meaning that the data could be sent at any time, without warning. This makes the bit settings more important.

Another method used to detect data errors is half-duplex and full-duplex transmission. In half-duplex transmission the data is only sent in one direction. But, in full-duplex transmission a copy of any byte received is sent back to the sender to verify that it was sent and received correctly. (Note: if you type and nothing shows up on a screen, or characters show up twice you may have to change the half/full duplex setting.)

The transmission speed is the maximum number of bits that can be sent per second. The units for this is *baud*. The baud rate includes the start, parity and stop bits. For example a 9600 baud transmission of the data in Figure 49 would transfer up to $\frac{9600}{(1 + 8 + 1 + 2)} = 800$ bytes each second. Lower

baud rates are 120, 300, 1.2K, 2.4K and 9.6K. Higher speeds are 19.2K, 28.8K and 33.3K. (Note: When this is set improperly you will get many transmission errors, or *garbage* on your screen.)

Serial lines have become one of the most common methods for transmitting data to instruments: most personal computers have two serial ports. The previous discussion of serial communications techniques also applies to devices such as modems.

4.2.1 RS-232

The RS-232c standard is based on a low/false voltage between +3 to +15V, and an high/true voltage between -3 to -15V (+/-12V is commonly used). Figure 50 shows some of the common connection schemes. In all methods the *txd* and *rxd* lines are crossed so that the sending *txd* outputs are into the listening *rxd* inputs when communicating between computers. When communicating with a communication device (modem), these lines are not crossed. In the *modem* connection the *dsr* and *dtr* lines are used to control the flow of data. In the *computer* the *cts* and *rts* lines are connected. These lines are all used for handshaking, to control the flow of data from sender to receiver. The *null-modem* configuration simplifies the handshaking between computers. The three wire configuration is a crude way to connect to devices, and data can be lost.

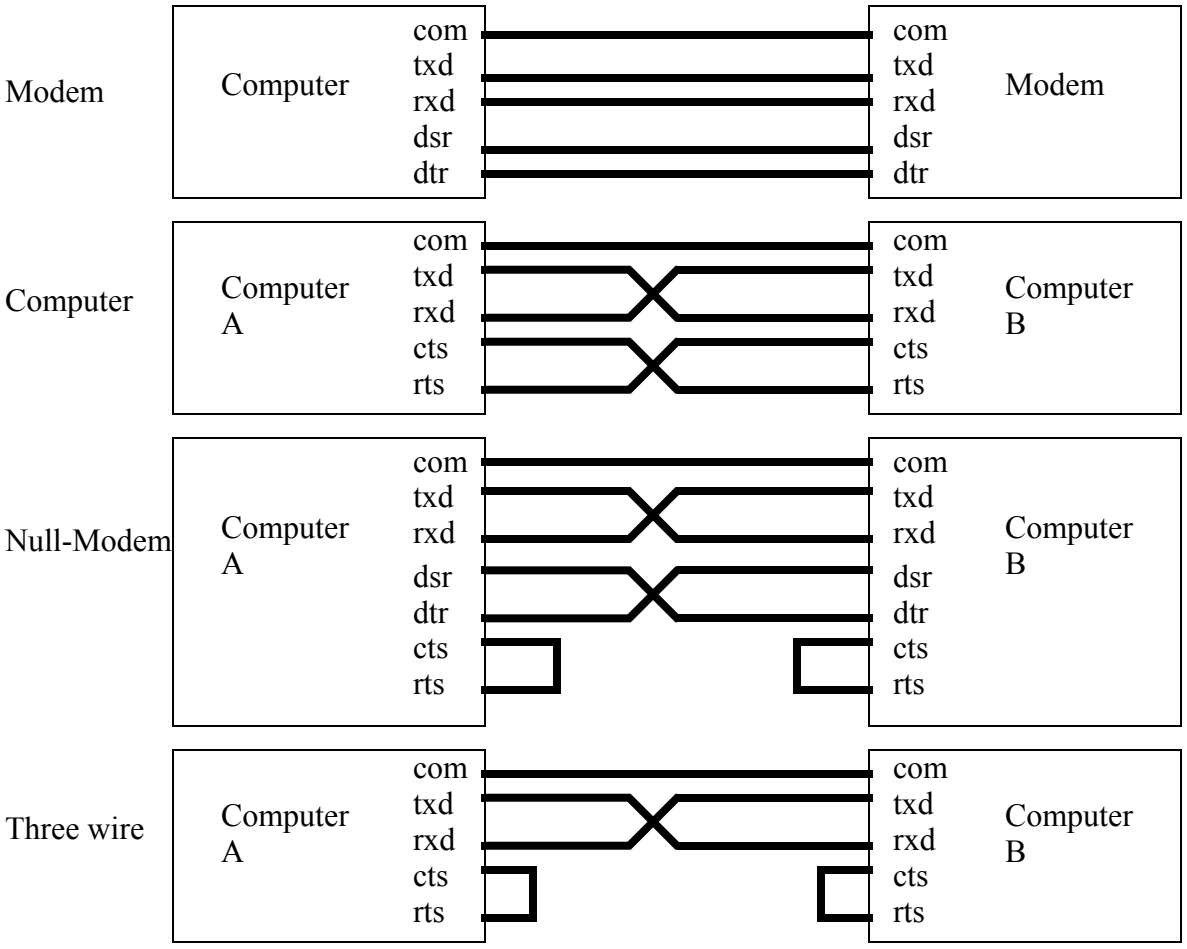
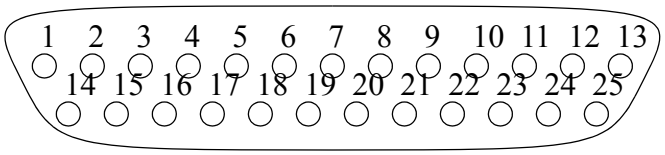


Figure 50 Common RS-232 Connection Schemes

Common connectors for serial communications are shown in Figure 51. These connectors are either male (with pins) or female (with holes), and often use the assigned pins shown. The DB-9 connector is more common now, but the DB-25 connector is still in use. In any connection the *RXD* and *TXD* pins must be used to transmit and receive data. The *COM* must be connected to give a common voltage reference. All of the remaining pins are used for *handshaking*.

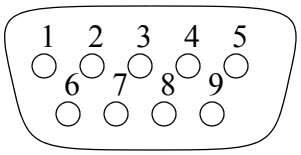
DB-25



Commonly used pins
- 1 - GND (chassis ground)
- 2 - TXD (transmit data)
- 3 - RXD (receive data)
- 4 - RTS (request to send)
- 5 - CTS (clear to send)
- 6 - DSR (data set ready)
- 7 - COM (common)
- 8 - DCD (Data Carrier Detect)
- 20 - DTR (data terminal ready)

Other pins
- 9 - Positive Voltage
- 10 - Negative Voltage
- 11 - not used
- 12 - Secondary Received Line Signal Detector
- 13 - Secondary Clear to Send
- 14 - Secondary Transmitted Data
- 15 - Transmission Signal Element Timing (DCE)
- 16 - Secondary Received Data
- 17 - Receiver Signal Element Timing (DCE)
- 18 - not used
- 19 - Secondary Request to Send
- 21 - Signal Quality Detector
- 22 - Ring Indicator (RI)
- 23 - Data Signal Rate Selector (DTE/DCE)
- 24 - Transmit Signal Element Timing (DTE)
- 25 - Busy

DB-9



- 1 - DCD
- 2 - RXD
- 3 - TXD
- 4 - DTR
- 5 - COM
- 6 - DSR
- 7 - RTS
- 8 - CTS
- 9 - RI

Note: these connectors often have very small numbers printed on them to help you identify the pins.

Figure 51 Typical RS-232 Pin Assignments and Names

The *handshaking* lines are to be used to detect the status of the sender and receiver, and to regulate the flow of data. It would be unusual for most of these pins to be connected in any one application. The most common pins are provided on the DB-9 connector, and are also described below.

TXD/RXD - (transmit data, receive data) - data lines
DCD - (data carrier detect) - this indicates when a remote device is present
RI - (ring indicator) - this is used by modems to indicate when a connection is about to be made.
CTS/RTS - (clear to send, ready to send)
DSR/DTR - (data set ready, data terminal ready) these handshaking lines indicate when the remote machine is ready to receive data.
COM - a common ground to provide a common reference voltage for the TXD and RXD.

When a computer is ready to receive data it will set the *CTS* bit, the remote machine will notice this on the *RTS* pin. The *DSR* pin is similar in that it indicates the modem is ready to transmit data. *XON* and *XOFF* characters are used for a software only flow control scheme.

Many PLC processors have an RS-232 port that is normally used for programming the PLC. Figure 52 shows a PLC connected to a personal computer with a Null-Modem line. It is connected to the *channel 0* serial connector on the PLC processor, and to the *com 1* port on the computer. In this example the *terminal* could be a personal computer running a terminal emulation program. The ladder logic below will send a string to the serial port *channel 0* when *A* goes true. In this case the string is stored is string memory *'example'* and has a length of 4 characters. If the string stored in *example* is "*HALFLIFE"*, the terminal program will display the string "*HALF"*.

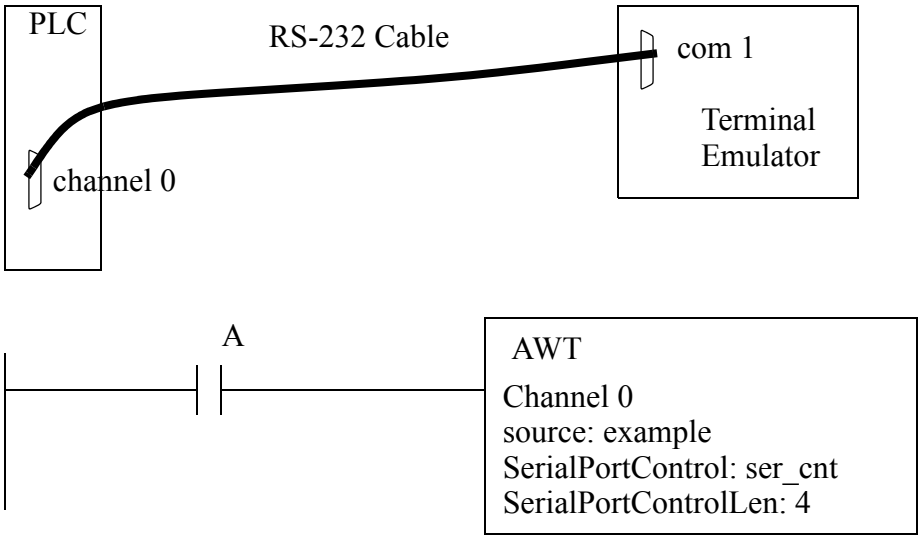


Figure 52 Serial Output Using Ladder Logic

The AWT (Ascii WriTe) function below will write to serial ports on the CPU only.

4.2.2 ASCII Functions

ASCII functions allow programs to manipulate strings in the memory of the PLC. The basic functions are listed in Figure 53.

AWA(channel, string, control, length) - append characters to the output buffer
ABL(channel, control)- reports the number of ASCII characters including line endings
ACB(channel, control) - reports the numbers of ASCII characters in buffer
AHL(channel, mask, mask, control) - does data handshaking
ARD(channel, dest, control, length) - will get characters from the ASCII buffer
ARL(channel, dest, control, length) - will get characters from an ASCII buffer
ASR(string, string) - compares two strings
AWT(channel, string, control, length) - will write characters to an ASCII output
CONCAT(string, string, dest) - concatenate strings
DELETE(string, len, start, dest) - deletes characters from a larger string
DTOS(integer, string) - convert an integer to a string
FIND(string, string, start) - find one string inside another
INSERT(string, string, start, dest) - puts characters inside a string
LOWER(integer, string) - convert a string to lower case
MID(string, start, length, dest) - this will copy a segment of a string out of a larger string
RTOS(integer, string) - convert a real to a string
STOD(string, dest) - convert ASCII string to integer
STOR(string, dest) - convert ASCII string to real
UPPER(string, dest) - convert a string to upper case

Figure 53 PLC ASCII Functions

In the example in Figure 54, the characters "Hi " are placed into string memory *str_in*. The ACB function checks to see how many characters have been received, and are waiting in channel 0. When the number of characters equals 2, the ARD (Ascii ReaD) function will then copy those characters into memory *str_0*, and bit *real_ctl.DN* will be set. This done bit will cause the two characters to be concatenated to the "Hi ", and the result written back to the serial port. So, if I typed in my initial "HJ", I would get the response "HI HJ".

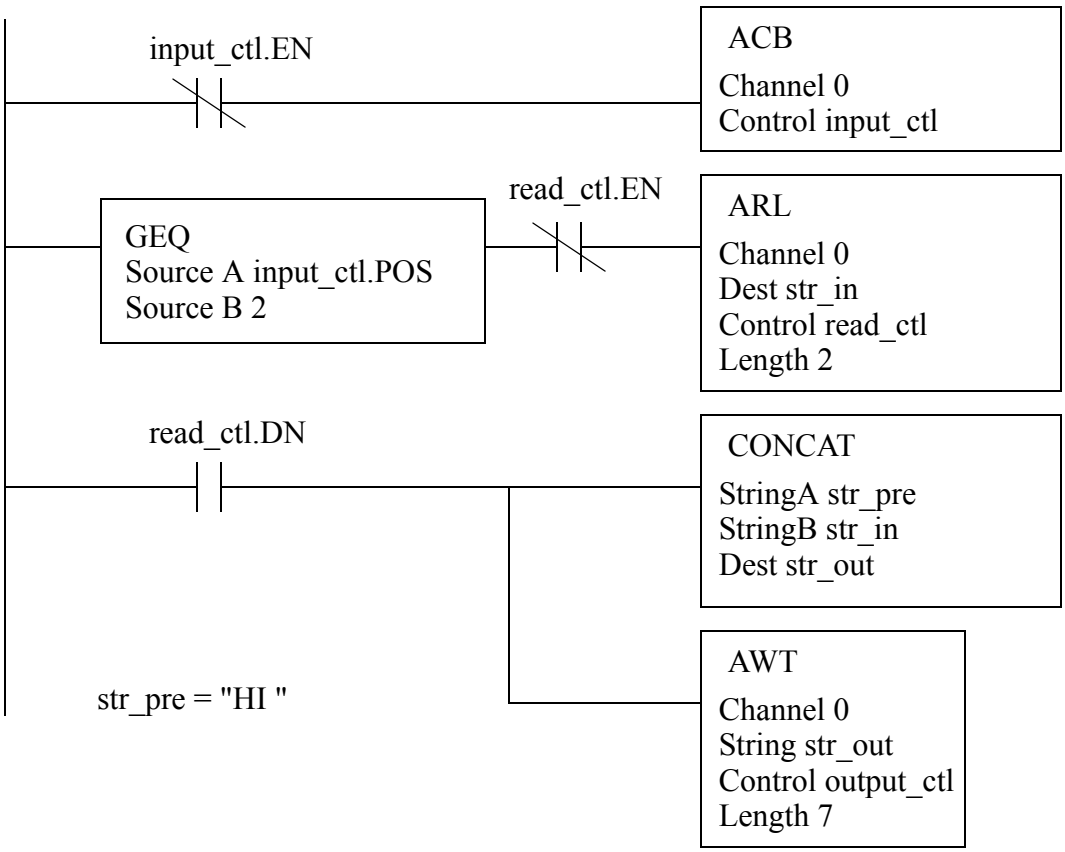


Figure 54 An ASCII String Example

The ASCII functions can also be used to support simple number conversions. The example in Figure 55 will convert the strings in *str_a* and *str_b* to integers, add the numbers, and store the result as a string in *str_c*.

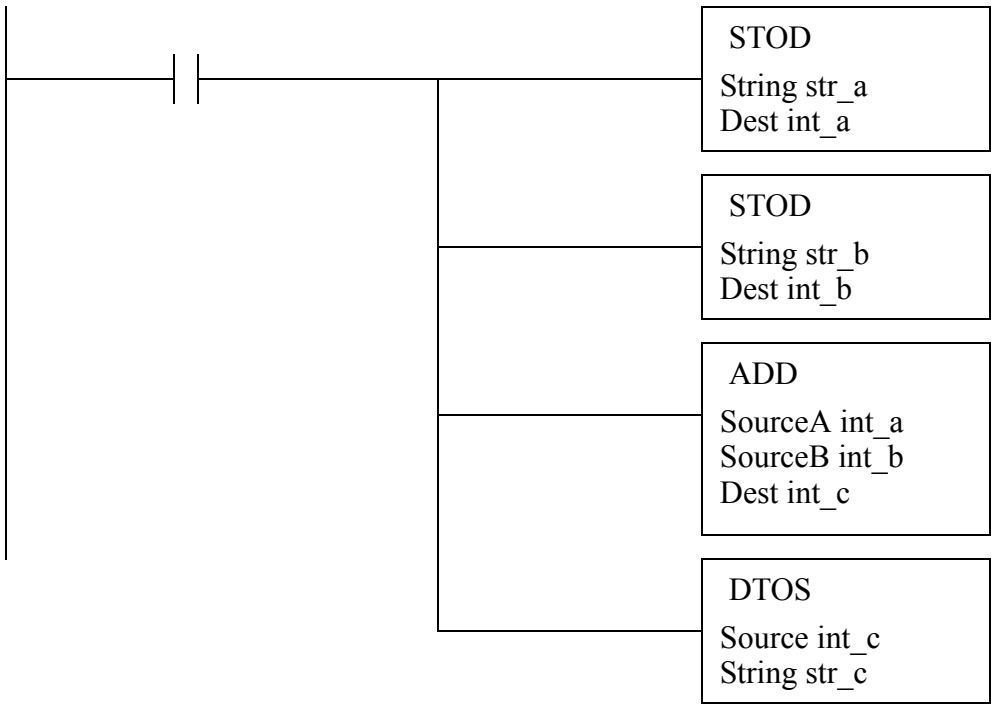


Figure 55 A String to Integer Conversion Example

Many of the remaining string functions are illustrated in Figure 56. When *A* is true the *ABL* and *ACB* functions will check for characters that have arrived on channel 1, but have not been retrieved with an *ARD* function. If the characters "ABC<*CR*>" have arrived (<*CR*> is an ASCII carriage return) the *ACB* would count the three characters, and store the value in *cnt_1.POS*. The *ABL* function would also count the <*CR*> and store a value of four in *cnt_2.POS*. If *B* is true, and the string in *str_a* is "ABCDEFGHIJKL", then "EF" will be stored in *str_b*. The last function will compare the strings in *str_c* and *str_d*, and if they are equal, output *string_match* will be turned on.

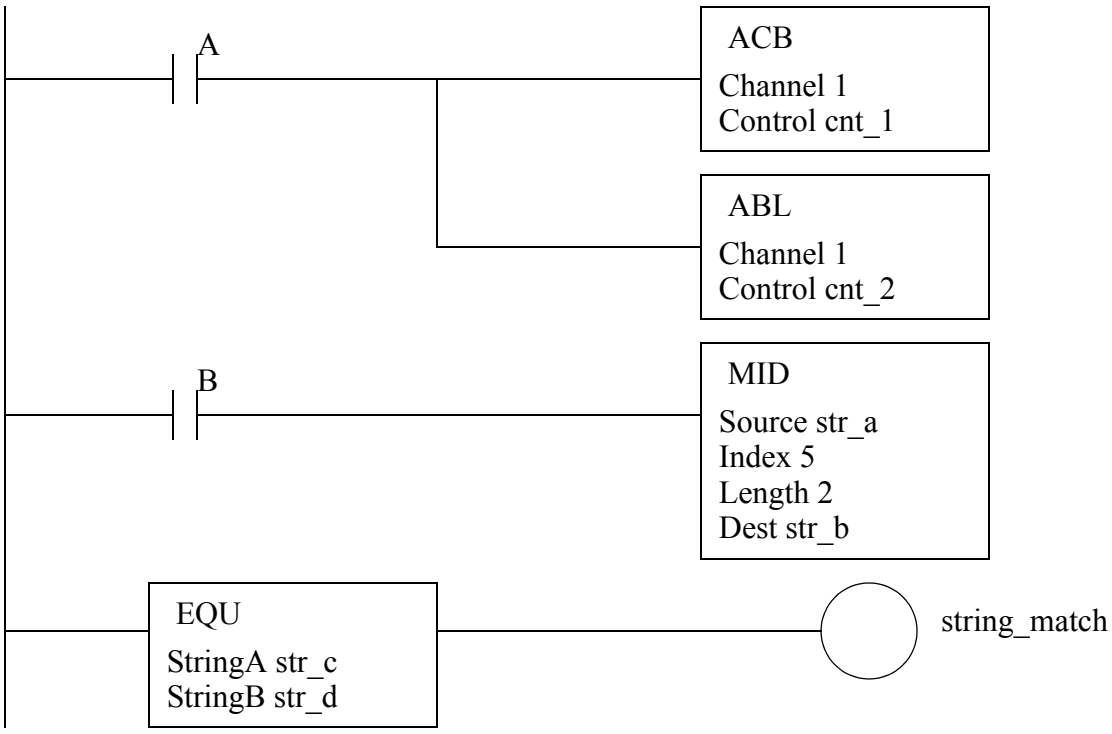


Figure 56 String Manipulation Functions

The *AHL* function can be used to do handshaking with a remote serial device.

4.3 PARALLEL COMMUNICATIONS

Parallel data transmission will transmit multiple bits at the same time over multiple wires. This does allow faster data transmission rates, but the connectors and cables become much larger, more expensive and less flexible. These interfaces still use handshaking to control data flow.

These interfaces are common for computer printer cables and short interface cables, but they are uncommon on PLCs. A list of common interfaces follows.

Centronics printer interface - These are the common printer interface used on most personal computers. It was made popular by the now defunct Centronics printer company.

GPIB/IEEE-488 - (General Purpose Instruments Bus) This bus was developed by Hewlett Packard Inc. for connecting instruments. It is still available as an option on many new instruments.

4.4 DESIGN CASES

4.4.1 PLC Interface To a Robot

Problem: A robot will be loading parts into a box until the box reaches a prescribed weight. A PLC will feed parts into a pickup fixture when it is empty. The PLC will tell the robot when to pick up a part and load it into the box by passing it an ASCII string, "pickup".

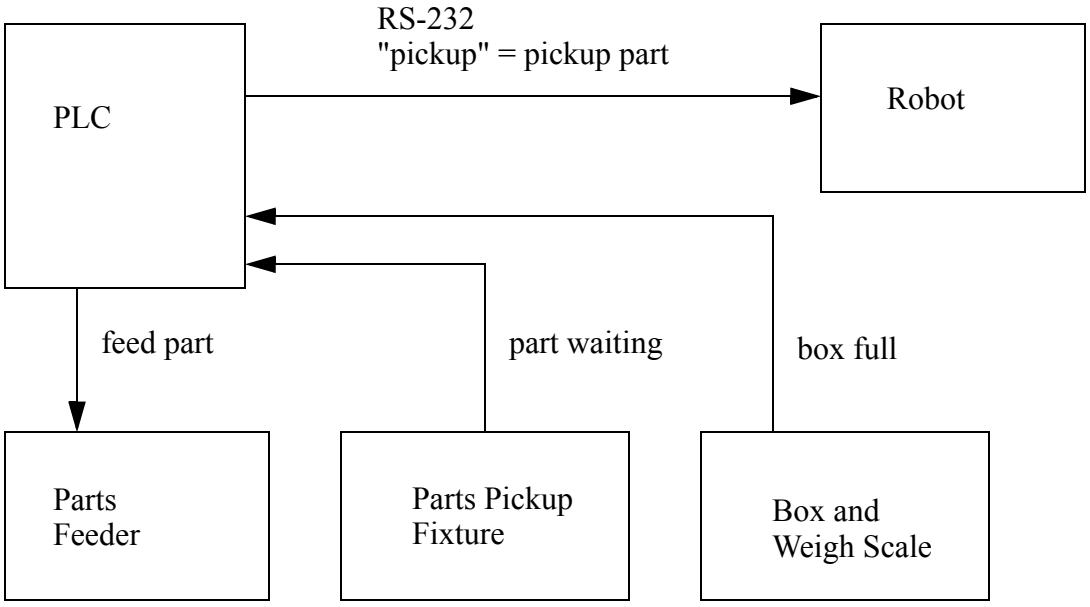


Figure 57 Box Loading System

Solution: The following ladder logic will implement part of the control system for the system in Figure 57.

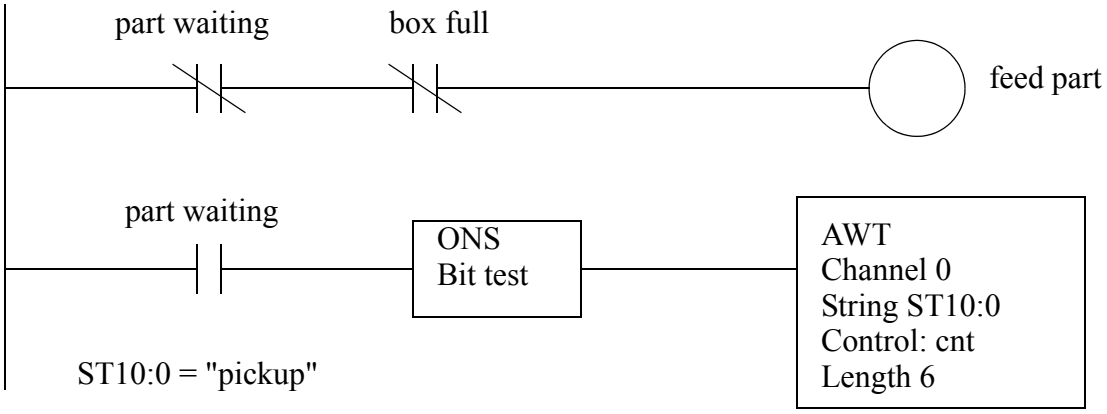


Figure 58 A Box Loading System

4.5 SUMMARY

- Serial communications pass data one bit at a time.
- RS-232 communications use voltage levels for short distances. A variety of communications cables and settings were discussed.
- ASCII functions are available of PLCs making serial communications possible.

4.6 PRACTICE PROBLEMS

1. Describe what the bits would be when an *A* (ASCII 65) is transmitted in an RS-232 interface with 8 data bits, even parity and 1 stop bit.

2. Divide the string in 'str_a' by the string in 'str_b' and store the results in 'str_c'. Check for a divide by zero error.

   str_a "100"
   str_b "10"
   str_c

3. How long would it take to transmit an ASCII file over a serial line with 8 data bits, no parity, 1 stop bit? What if the data were 7 bits long?

4. Write a number guessing program that will allow a user to enter a number on a terminal that transmits it to a PLC where it is compared to a value in *target*. If the guess is above "Hi" will be returned. If below "Lo" will be returned. When it matches "ON" will be returned.

5. Write a structured text program that reads inputs from 'channel 0'. An input string of 'CLEAR' will clear a storage array. Up to 100 real values with the format 'XXX.XX' will arrive on 'channel 0' and are to be stored in the array. If the string 'AVG' is received, the average of the array contents will be calculated and written out 'Channel 0'.

4.7 ASSIGNMENT PROBLEMS

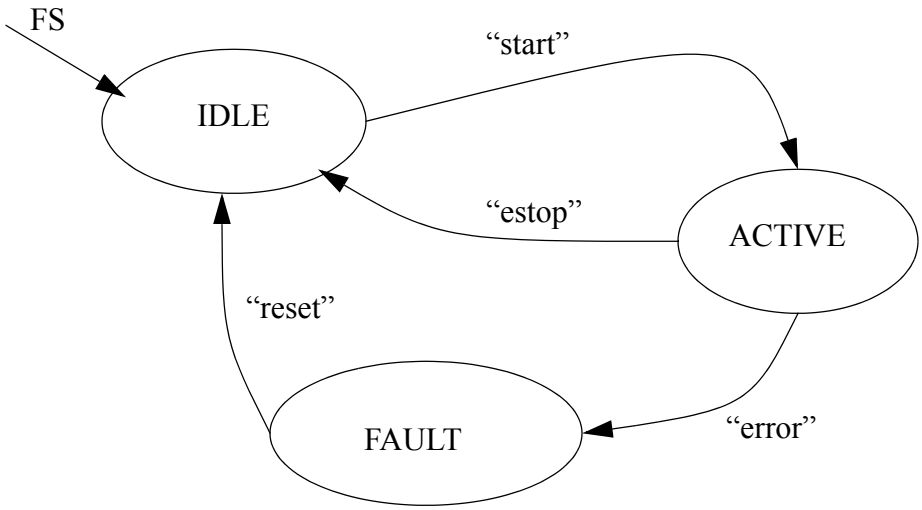
1. Describe an application of ASCII communications.

2. Write a ladder logic program to output an ASCII warning message on channel 1 when the value in 'temp' is less than 10, or greater than 20. The message should be "out of temp range".

3. Write a program that will send an ASCII message every minute. The message should begin with the word 'count', followed by a number. The number will be 1 on the first scan of the PLC, and increment once each minute.

4. A PLC will be controlled by ASCII commands received through the RS-232C communications port. The

commands will cause the PLC to move between the states shown in the state diagram. Implement the ladder logic.



5. A program is to be written to control a robot through an RS-232c interface. The robot has already been programmed to respond to two ASCII strings. When the robot receives the string ‘start’ it will move a part from a feeder to a screw machine. When the robot receives an ‘idle’ command it will become inactive (safe). The PLC has ‘start’ and ‘end’ inputs to control the process. The PLC also has two other inputs that indicate when the parts feeder has parts available (‘part present’) and when the screw machine is done (‘machine idle’). The ‘start’ button will start a cycle where the robot repeatedly loads parts into the screw machine whenever the ‘machine idle’ input is true. If the ‘part present’ sensor is off (i.e., no parts), or the ‘end’ input is off (a stop requested), the screw machine will be allowed to finish, but then the process will stop and the robot will be sent the idle command. Use a structured design method (e.g., state diagrams) to develop a complete ladder logic program to perform the task.

6. A PLC is connected to a scale that measures weights and then sends an ASCII string. The string format is ‘XXXX.XX’. So a weight of 29.9 grams would result in a string of ‘0029.90’. The PLC is to read the string and then check to see if the weight is between 18.23 and 18.95 grams. If it is not then an error output light should be set until a reset button is pushed.

7. Write a program that will convert a numerical value stored in the ’REAL’ value *float* and write it out the RS-232 output with 3 decimal places.

8. A system for testing hydraulic resevoirs is to be designed and built using a PLC. Part of the test will be conducted using a computer based Data AQuisition (DAQ) system for high speed analog inputs. When the test begins a command of ’S’ is sent to the DAQ system, and an output ’pump’ will be turned on. The test is started with a ’start’ input and stopped with a ’stop’ input. The test will be shut down and an error light turned on if the flow sensor does not turn on within 0.1s, or if the pressure input rises above 4V. When the test is done the DAQ system will send a ’D’ to the PLC. The PLC will retrieve the data by sending and ’R’ to the DAQ system. The data is returned in the format ’xxxx.x<cr><lf>’. The last data line will be ’END’. The array of data should be analyzed and the results stored in the real variables ’maximum’, ’average’, ’standard_deviation’, and ’median’. These variables will displayed on an HMI. Write a structured text program for the control system.