16. ADVANCED LADDER LOGIC FUNCTIONS

16.1 INTRODUCTION

This chapter covers advanced functions, but this definition is somewhat arbitrary. The array functions in the last chapter could be classified as advanced functions. The functions in this section tend to do things that are not oriented to simple data values. The list functions will allow storage and recovery of bits and words. These functions are useful when implementing buffered and queued systems. The program control functions will do things that don't follow the simple model of ladder logic execution - these functions recognize the program is executed left-to-right top-to-bottom. Finally, the input output functions will be discussed, and how they allow us to work around the normal input and output scans.

16.2 LIST FUNCTIONS

16.2.1 Shift Registers

Shift registers are oriented to single data bits. A shift register can only hold so many bits, so when a new bit is put in, one must be removed. An example of a shift register.
is given in Figure 16.1. The shift register is the word B3:1, and it is 5 bits long. When A becomes true the bits all shift right to the least significant bit. When they shift a new bit is needed, and it is taken from I:000/0. The bit that is shifted out, on the right hand side, is moved to the control word UL (unload) bit R6:2/UL. This function will not complete in a single ladder logic scan, so the control word R6:2 is used. The function is edge triggered, so A would have to turn on 5 more times before the bit just loaded from I:000/0 would emerge to the unload bit. When A has a positive edge the bits in B3:1 will be shifted in memory. In this case it is taking the value of bit B3:1/0 and putting it in the control word bit R6:2/UL. It then shifts the bits once to the right, B3:1/0 = B3:1/1 then B3:1/1 = B3:1/2 then B3:1/2 = B3:1/3 then B3:1/3 = B3:1/4. Then the input bit is put into the most significant bit B3:1/4 = I:000/00. The bits in the shift register can also be shifted to the left with the BSL function.

Figure 16.1
Shift Register Functions

There are other types of shift registers not implemented in PLC-5s. These are shown in Figure 16.2. The primary difference is that the arithmetic shifts will put a zero into the shift register, instead of allowing an arbitrary bit. The rotate functions shift bits around in an endless circle. These functions can also be implemented using the BSR and BSL instructions when needed.
16.2.2 Stacks

Stacks store integer words in a two ended buffer. There are two basic types of stacks; first-on-first-out (FIFO) and last-in-first-out (LIFO). As words are pushed on the stack it gets larger, when words are pulled off it gets smaller. When you retrieve a word from a LIFO stack you get the word that is the entry end of the stack. But, when you get a word from a FIFO stack you get the word from the exit end of the stack (it has also been there the longest). A useful analogy is a pile of work on your desk. As new work arrives you drop it on the top of the stack. If your stack is LIFO, you pick your next job from the top of the pile. If your stack is FIFO, you pick your work from the bottom of the pile.

Stacks are very helpful when dealing with practical situations such as buffers in production lines. If the buffer is only a delay then a FIFO stack will keep the data in order. If product is buffered by piling it up then a LIFO stack works better, as shown in Figure 16.3.

In a FIFO stack the parts pass through an entry gate, but are stopped by the exit gate. In the LIFO stack the parts enter the stack and lower the plate, when more parts are needed the plate is raised. In this arrangement the order of the parts in the stack will be reversed.
The ladder logic functions are FFL to load the stack, and FFU to unload it. The example in Figure 16.4 shows two instructions to load and unload a FIFO stack. The first time this FFL is activated (edge triggered) it will grab the word (16 bits) from the input card I:001 and store them on the stack, at N7:0. The next value would be stored at N7:1, and so on until the stack length is reached at N7:4. When the FFU is activated the word at N7:0 will be moved to the output card O:003. The values on the stack will be shifted up so that the value previously in N7:1 moves to N7:0, N7:2 moves to N7:1, etc. If the stack is full or empty, an a load or unload occurs the error bit will be set R6:0/ER.
Figure 16.4 FIFO Stack Instructions

As values are loaded on the stack, they will be added sequentially N7:0, N7:1, N7:2, N7:3, then N7:4. When values are unloaded, they will be taken from the last loaded position, so if the stack is full, the value of N7:4 will be removed first.

Figure 16.5 LIFO Stack Commands

A  B  

LFL
source I:001
LIFO N7:0
Control R6:0
length 5
position 0

LFU
LIFO N7:0
destination O:003
Control R6:0
length 5
position 0

A  B
16.2.3 Sequencers

A mechanical music box is a simple example of a sequencer. As the drum in the music box turns it has small pins that will sound different notes. The song sequence is fixed, and it always follows the same pattern. Traffic light controllers are now controlled with electronics, but previously they used sequencers that were based on a rotating drum with cams that would open and close relay terminals. One of these cams is shown in Figure 16.6. The cam rotates slowly, and the surfaces under the contacts will rise and fall to open and close contacts. For a traffic light controllers the speed of rotation would set the total cycle time for the traffic lights. Each cam will control one light, and by adjusting the circumferential length of rises and drops the on and off times can be adjusted.

![A Single Cam in a Drum Sequencer](image)

A PLC sequencer uses a list of words in memory. It recalls the words one at a time and moves the words to another memory location or to outputs. When the end of the list is reached the sequencer will return to the first word and the process begins again. A sequencer is shown in Figure 16.7. The SQO instruction will retrieve words from bit memory starting at B3:0. The length is 4 so the end of the list will be at B3:0+4 or B3:4 (the total length is actually 5). The sequencer is edge triggered, and each time A becomes true the retrieve a word from the list and move it to O:000. When the sequencer reaches the end of the list the sequencer will return to the second position in the list B3:1. The first item in the list is B3:0, and it will only be sent to the output if the SQO instruction is active on the first scan of the PLC, otherwise the first word sent to the output is B3:1. The mask value is 000Fh, or 000000000000 1111b so only the four least significant bits will be transferred to the output, the other output bits will not be changed. The other instructions allow words to be added or removed from the sequencer list.

As the cam rotates it makes contact with none, one, or two terminals, as determined by the depressions and rises in the rotating cam.
Figure 16.7
The Basic Sequencer Instruction

An example of a sequencer is given in Figure 16.8 for traffic light control. The light patterns are stored in memory (entered manually by the programmer). These are then moved out to the output card as the function is activated. The mask (003F = 0000000000111111) is used so that only the 6 least significant bits are changed.

SQO(start,mask,source,destination,control,length) - sequencer output from table to memory address

SQI(start,mask,source,control,length) - sequencer input from memory address to table

SQL(start,source,control,length) - sequencer load to set up the sequencer parameters

File #B3:0
Mask 000F
Destination O:000
Control R6:0
Length 4
Position 0
Figure 16.9 shows examples of the other sequencer functions. When A goes from false to true, the SQL function will move to the next position in the sequencer list, for example N7:21, and load a value from I:001. If A then remains true the value in N7:21 will be overwritten each scan. When the end of the sequencer list is encountered, the position will reset to 1.

The sequencer input (SQI) function will compare values in the sequence list to the source I:002 while B is true. If the two values match B3/10 will stay on while B remains true. The mask value is 0005h or 0000000000000101b, so only the first and third bits will be compared. This instruction does not automatically change the position, so logic is shown that will increment the position every scan while C is true.

Figure 16.8
A Sequencer For Traffic Light Control

SQO File #B3:0
Mask 003F
Destination O:000
Control R6:0
Length 4
Position 0

0 0 0 0
0 1 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 1 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 1 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0

advance
Sequencer Instruction Examples

These instructions are well suited to processes with a single flow of execution, such as traffic lights.

16.3 PROGRAM CONTROL

16.3.1 Branching and Looping

These functions allow parts of ladder logic programs to be included or excluded from each program scan. These functions are similar to functions in other programming languages such as C, C++, Java, Pascal, etc.

- Entire sections of programs can be bypassed using the JMP instruction in Figure SQI.

```
A
B
C
```

- `ADD` function example:

```
ADD SourceA R6:2.POS SourceB 1 Dest R6:2.POS
```

- `MOV` function example:

```
MOV Source 1 Dest R6:2.POS GT SourceA R6:2.POS SourceB 9
```
If \( A \) is true the program will jump over the next three lines to the line with the `LBL` 01. If \( A \) is false the `JMP` statement will be ignored, and the program scan will continue normally. If \( A \) is false \( X \) will have the same value as \( B \), and \( Y \) can be turned on by \( C \) and off by \( D \). If \( A \) is true then \( X \) and \( Y \) will keep their previous values, unlike the `MCR` statement. Any instructions that follow the `LBL` statement will not be affected by the `JMP` so \( Z \) will always be equal to \( E \). If a jump statement is true the program will run faster.

![Diagram](image)

**Figure 16.10** A JMP Instruction

Subroutines jump to other programs, as is shown in Figure 16.11. When \( A \) is true the `JSR` function will jump to the subroutine program in file 3. The `JSR` instruction two arguments are passed, \( 7:0 \) and \( 123 \). The subroutine (SBR) function receives these two arguments and puts them in \( 10:0 \) and \( 10:1 \). When \( B \) is true the subroutine will end and return to program file 2 where it was called. The `RET` function can also returns the value \( 10:1 \) to the calling program where it is put in location \( 7:1 \). By passing arguments (instead of having the subroutine use global memory locations) the subroutine can be used for more than one operation. For example, a subroutine could be given an angle in degrees and return a value in radians. A subroutine can be called more than once in a program, but if not called, it will be ignored.
The for-next loop in Figure 16.12 will repeat a section of a ladder logic program 5 times (from 0 to 9 in steps of 2) when \(A\) is true. The loop starts at the \texttt{FOR}\quad and ends at the \texttt{NXT}\quad function. In this example there is an \texttt{ADD}\quad function that will add 1 to the value of \texttt{N7:1}. So when this for-next statement is complete the value of \texttt{N7:1}\quad will be larger. Notice that the label number is the same in the \texttt{FOR}\quad and \texttt{NXT}\quad, this allows them to be matched.

For-next loops can be put inside other for-next loops, this is called nesting. If \(A\) was false the program would skip to the \texttt{NXT}\quad statement. All 5 loops will be completed in a single program scan, so a control word is not required. If \(B\) is true the \texttt{NXT}\quad statement will no longer return the program scan to the \texttt{FOR}\quad instruction, even if the loop is not complete.

Care must be used for this instruction so that the ladder logic does not get caught in an infinite, or long loop - if this happens the PLC will experience a fault and halt.

A JSR (Jump subroutine)

Program File 3

Input par \texttt{N7:0}\quad

Input par 123

Return par \texttt{N7:1}\quad

SBR (subroutine arguments)

Input par \texttt{N10:0}\quad

Input par \texttt{N10:1}\quad

A separate ladder logic program is stored in program file 3. This feature allows users to create their own \texttt{functions}. In this case if \(A\) is true, then the program below will be executed and then when done the ladder scan will continue after the subroutine instruction. The number of data values passed and returned is variable.

If \(B\) is true the subroutine will return and the values listed will be returned to the return par. For this example the value that is in \texttt{N10:1}\quad will eventually end up in \texttt{N7:1}\quad

\texttt{RET}\quad Return par \texttt{N10:1}\quad

Input par \texttt{N10:1}\quad
Ladder logic programs always have an end statement, as shown in Figure 16.13. Most modern software automatically inserts this. PLCs will experience faults if this is not present. The temporary end (TND) statement will skip the remaining portion of a program. If \( C \) is true then the program will end, and the next line with \( D \) and \( Y \) will be ignored. If \( C \) is false then the TND will have no effect and \( Y \) will be equal to \( D \).

FOR label number 0
index N7:0
initial value 0
terminal value 9
step size 2
ADD Source A 1
Source B N7:1
Dest N7:1
NXT label number 0

A Note: if A is true then the loop will repeat 10 times, and the value of N7:1 will be increased by 10. If A is not true, then the ADD function will only be executed once and N7:1 will increase in value by 1.
Figure 16.13

End Statements

The one shot contact in Figure 16.14 can be used to turn on a ladder run for a single scan. When \( A \) has a positive edge the oneshot will turn on the run for a single scan. Bit \( B3:0 \) is used here to track to rung status.

Figure 16.14

One Shot Instruction

When the end (or End Of File) is encountered the PLC will stop scanning the ladder, and start updating the outputs. This will not be true if it is a subroutine or a step in an SFC.
16.3.2 Fault Detection and Interrupts

The PLC can be set up to run programs automatically using interrupts. This is routinely done for a few reasons;

- to deal with errors that occur (e.g. divide by zero)
- to run a program at a regular timed interval (e.g. SPC calculations)
- to respond when a long instruction is complete (e.g. analog input)
- when a certain input changed (e.g. panic button)

These interrupt driven programs are put in their own program file. The program file number is then put in a status memory location. Some other values are also put into status memory to indicate the interrupt conditions.

A fault condition can stop a PLC. If the PLC is controlling a dangerous process this could lead to significant damage to personnel and equipment. There are two types of faults that occur; terminal (major) and warnings (minor). A minor fault will normally set an error bit, but not stop the PLC. A major failure will normally stop the PLC, but an interrupt can be used to run a program that can reset the fault bit in memory and continue operation (or shut down safely). Not all major faults are recoverable. A complete list of these faults is available in PLC processor manuals.

Figure 16.15 shows two programs. The default program (file 2) will set the interrupt program file to 3 by moving it to S2:29 on the first scan. When A is true a compute function will interpret the expression, using indirect addressing. If B becomes true then the value in N7:0 will become negative. If A becomes true after this then the expression will become N7:-10 +10. The negative value for the address will cause a fault, and program file 3 will be run. In fault program status memory S2:12 is checked the error code 21, which indicates a bad indirect address. If this code is found the index value N7:0 is set back to zero, and S2:11 is cleared. As soon as S2:11 is cleared the fault routine will stop, and the normal program will resume. If S2:11 is not cleared, the PLC will enter a fault state and stop (the fault light on the front of the PLC will turn on).
A Fault Recovery Program

A timed interrupt will run a program at regular intervals. To set a timed interrupt, the program in file number should be put in S2:31. The program will be run every S2:30 times 1 millisecond. In Figure 16.16, program 2 will set up an interrupt that will run program 3 every 5 seconds. Program 3 will add the value of \( I:000 \) to \( N7:10 \). This type of timed interrupt is very useful when controlling processes where a constant time interval is important. The timed interrupts are enabled by setting bit S2:2/1 in PLC-5s.

\[
\begin{align*}
\text{MOV} & \quad \text{Source} 3 \quad \text{Dest} \quad S2:29 \\
\text{CPT} & \quad \text{Dest} \quad N7:1 \\
\text{Expression} & \quad N7: [N7:0] + 10 \\
\end{align*}
\]

\[
\begin{align*}
\text{S2:1/15} & - \text{first scan} \\
\text{A} & - \text{EQU} \\
\text{SourceA} & \quad S2:12 \\
\text{SourceB} & \quad 21 \\
\text{CLR} & \quad \text{Dest.} \quad S2:11 \\
\text{program file 2} & - \text{program file 3} \\
\text{MOV} & \quad \text{Source} 0 \quad \text{Dest} \quad N7:0 \\
\text{MOV} & \quad \text{Source} -10 \quad \text{Dest} \quad N7:0 \\
\text{B} & \quad \\
\end{align*}
\]
Interrupts can also be used to monitor a change in an input. This is useful when waiting for a change that needs a fast response. The relevant values that can be changed are listed below.

- **S:46** - the program file to run when the input bit changes
- **S:47** - the rack and group number (e.g. if in the main rack it is 000)
- **S:48** - mask for the input address (e.g. 0000000000000100 watches 02)
- **S:49** - for positive edge triggered = 1 for negative edge triggered = 0
- **S:50** - the number of counts before the interrupt occurs. 1 = always up to 32767

Figure 16.17 shows an interrupt driven interrupt. Program 2 sets up the interrupt to run program file 3 when input I:002/02 has 10 positive edges. (Note: the value of 0004 in binary is 0000 0000 0000 0100, or input 02.) When the input goes positive 10 times the bit B3/100 will be set.

```
MOV Source 3 Dest S2:31
MOV Source 500 Dest S2:30
S2:1/15 - first scan
ADD SourceA I:000 SourceB N7:10
```
Figure 16.17

An Input Driven Interrupt

When activated, interrupt routines will stop the PLC, and the ladder logic is interpreted immediately. If the PLC is in the middle of a program scan this can cause problems.

To overcome this a program can disable interrupts temporarily using the UID and UIE functions. Figure 16.18 shows an example where the interrupts are disabled for a FAL instruction. Only the ladder logic between the **UID** and **UIE** will be disabled, the first line of ladder logic could be interrupted. This would be important if an interrupt routine could change a value between **N7:0** and **N7:4**. For example, an interrupt could occur while the FAL instruction was at **N7:7 = N7:2 + 5**. The interrupt could change the values of **N7:1** and **N7:4**, and then end. The FAL instruction would then complete the calculations. But, the results would be based on the old value for **N7:1** and the new value for **N7:4**.
16.4 INPUT AND OUTPUT FUNCTIONS

16.4.1 Immediate I/O Instructions

The input scan normally records the inputs before the program scan, and the output scan normally updates the outputs after the program scan, as shown in Figure 16.19. Immediate input and output instructions can be used to update some of the inputs or outputs during the program scan.
Figure 16.19 shows a segment within a program that will update the input word I:001, determine a new value for O:010/01, and update the output word O:010 immediately. The process can be repeated many times during the program scan allowing faster than normal response times.

The normal operation of the PLC is:
- Fast [input scan]
- Slow [ladder logic is checked]
- Fast [outputs updated]

Input values scanned
Outputs are updated in memory only, as the ladder logic is scanned
Output values are updated to match values in memory
Immediate Inputs and Outputs

16.4.2 Block Transfer Functions

Simple input and output cards use a single word. Writing one word to an output card sets all of the outputs. Reading one word from an input card reads all of the inputs. As a result the PLC is designed to send and receive one word to input and from output cards.

Later we will discuss more complex input and output cards (such as analog I/O) that require more than one data word. To communicate multiple words, one word must be sent at a time over multiple scans. To do this we use special functions called Block Transfer Write (BTW) and Block Transfer Read (BTR).

Figure 16.21 shows a BTW function. The module type is defined from a given list, in this case it is an Example Output Card. The next three lines indicate the card location as 00, 3, or 003, the module number should normally be zero (except when using two slot addressing). This instruction is edge triggered, and special control memory BT10:1 is used in this example to track the function progress (Note: regular control memory could have also been used, but the function will behave differently). The instruction will send 10 words from N9:0 to N9:9 to the output card when A becomes true. The enabled bit BT10:1/EN is used to block another start until the instruction is finished. If the instruction enabled bit BT10:1/EN is used to block another start until the instruction is finished. If the instruction

Note: When these instructions are used the normal assumption that all inputs and outputs are updated before and after the program scan is no longer valid.

Figure 16.21

Block Transfer Write (BTW)
is restarted before it is done an error will occur. The length and contents of the memory
N9:0 to N9:9 are specific to the type of input and output card used, and will be discussed
later for specific cards. This instruction is not continuous, meaning that when done it will
stop. If it was continuous then when the previous write was done the next write would
begin.

Figure 16.21
A BTW Function

The BTR function is similar to the BTW function, except that it will read multiple
values back from an input card. This gets values from the card O:000, and places
9 values in memory from N9:4 to N9:13. The function is continuous, so when it is complete, the
process of reading from the card will begin again.

Figure 16.22
A BTR Function

<table>
<thead>
<tr>
<th>Block Transfer Write Module Type Example Output Card</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rack 00</td>
</tr>
<tr>
<td>Group 3</td>
</tr>
<tr>
<td>Module 0</td>
</tr>
<tr>
<td>Control Block BT10:1</td>
</tr>
<tr>
<td>Data File N9:0</td>
</tr>
<tr>
<td>Length 10</td>
</tr>
<tr>
<td>Continuous No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rack: 00</th>
<th>Group: 0</th>
<th>Module: 0</th>
<th>BT Array: BT10:0</th>
<th>Data File: N9:4</th>
<th>Length: 9</th>
<th>Continuous: Yes</th>
</tr>
</thead>
</table>
16.5 DESIGN TECHNIQUES

16.5.1 State Diagrams

The block logic method was introduced in chapter 8 to implement state diagrams using MCR blocks. A better implementation of this method is possible using subroutines in program files. The ladder logic for each state will be put in separate subroutines.

Consider the state diagram in Figure 16.23. This state diagram shows three states with four transitions. There is a potential conflict between transitions A and C.

Figure 16.23: A State Diagram

The main program for the state diagram is shown in Figure 16.24. This program is stored in program file 2 so that it is run by default. The first rung in the program resets the states so that the first scan state is on, while the other states are turned off. Each state in the diagram is given a value in bit memory, so ST A=B3/0, STB=B3/1 and STC=B3/2. The following logic will call the subroutine for each state. The logic that uses the current state is placed in the main program. It is also possible to put this logic in the state subroutines.
The Main Program for the State Diagram (Program File 2) includes logic to give state C higher priority, by blocking state A when C is active.

```
S2:1/15 - first scan

L U B3/0 - STA
B3/1 - STB
B3/2 - STC

B3/0 - STA
JSR program 3

B3/1 - STB
JSR program 4

B3/2 - STC
JSR program 5

LO:000/0
LO:000/1
LO:000/2

B3/0 - STA
B3/1 - STB
B3/2 - STC
```
The arrangement of the subroutines in Figure 16.24 and Figure 16.25 could experience race conditions. For example, if STA is active, and both B and C are true at the same time the main program would jump to subroutine 3 where STB would be turned on, then the main program would jump to subroutine 4 where STC would be turned on. For the output logic STB would never have been on. If this problem might occur, the state diagram can be modified to slow down these race conditions. Figure 16.26 shows a technique that blocks race conditions by blocking a transition out of a state until the transition into a state is finished. The solution may not always be appropriate.
Another solution is to force the transition to wait for one scan as shown in Figure 16.27 for state ST A. A wait bit is used to indicate when a delay of at least one scan has occurred since the transition out of the state B became true. The wait bit is set by having the exit transition B true. The B3/10-ST A will turn off the wait B3/1-wait when the transition to state B3/1-STB has occurred. If the wait was not turned off, it would still be on the next time we return to this state.
16.6 DESIGN CASES

16.6.1 If-Then

Problem: Convert the following C/Java program to ladder logic.

Solution:

```c
void main()
{
    int A;
    for (A = 1; A < 10; A++)
    {
        if (A >= 5) then A = add(A);
    }
}

int add(int x)
{
    x = x + 1;
    return x;
}
```

Problem: Design and write ladder logic for a simple traffic light controller that has a single fixed sequence of 16 seconds for both green lights and 4 seconds for both yellow lights. Use either stacks or sequencers.

Solution: The sequencer is the best solution to this problem.
Shift registers move bits through a queue.

Stacks will create a variable length list of words.

Sequencers allow a list of words to be stepped through.

Parts of programs can be skipped with jump and MCR statements, but MCR statements shut off outputs.

Subroutines can be called in other program files, and arguments can be passed.

For-next loops allow parts of the ladder logic to be repeated.

Interrupts allow parts to run automatically at fixed times, or when some event happens.

Immediate inputs and outputs update I/O without waiting for the normal scans.

Block transfer functions allow communication with special I/O cards that need more than one word of data.

**OUTPUTS**

- **O:000/00 NSG - north south green**
- **O:000/01 NSY - north south yellow**
- **O:000/02 NSR - north south red**
- **O:000/03 EWG - east west green**
- **O:000/04 EWY - east west yellow**
- **O:000/05 EWR - east west red**

**TON**

- **T4:0** preset 4.0 sec

**SQO**

- **File #N7:0** mask 003F

**T4:0/DN**

- **Dest. O:000**
- **Control R6:0**
- **Length 10**

**Addr.**

- **N7: 0**
- **N7: 1**
- **N7: 2**

**Contents (in binary)**

- **0000000000100001**
- **0000000000100001**
- **0000000000100001**
PRACTICE PROBLEMS

1. Design and write ladder logic for a simple traffic light controller that has a single fixed sequence of 16 seconds for both green lights and 4 second for both yellow lights. Use shift registers to implement it.

2. A PLC is to be used to control a carillon (a bell tower). Each bell corresponds to a musical note and each has a pneumatic actuator that will ring it. The table below defines the tune to be programmed. Write a program that will run the tune once each time a start button is pushed. A stop button will stop the song.

3. Consider a conveyor where parts enter on one end. They will be checked to be in a left or right orientation with a vision system. If neither left nor right is found, the part will be placed in a reject bin. The conveyor layout is shown below.

4. Why are MCR blocks different than JMP statements?

5. What is a suitable reason to use interrupts?

6. When would immediate inputs and outputs be used?

7. Explain the significant differences between shift registers, stacks and sequencers.
8. Design a ladder logic program that will run once every 30 seconds using interrupts. It will check to see if a water tank is full with input I:000/0. If it is full, then a shutdown value (B3/37) will be latched on.

9. At MOdern Manufacturing (MOMs), pancakes are made by multiple machines in three flavors; chocolate, blueberry and plain. When the pancakes are complete they travel along a single belt, in no specific order. They are buffered by putting them on the top of a stack. When they arrive at the stack the input I:000/3 becomes true, and the stack is loaded by making output O:001/1 high for one second. As the pancakes are put on the stack, a color detector is used to determine the pancakes type. A value is put in N7:0 (1=chocolate, 2=blueberry, 3=plain) and bit B3/0 is made true. A pancake can be requested by pushing a button (I:000/0=chocolate, I:000/1=blueberry, I:000/2=plain). Pancakes are then unloaded from the stack, by making O:001/0 high for 1 second, until the desired flavor is removed. Any pancakes removed aren't returned to the stack. Design a ladder logic program to control this stack.

10. a) What are the three fundamental types of interrupts?
   b) What are the advantages of interrupts in control programs?
   c) What potential problems can they create?
   d) Which instructions can prevent this problem?

11. Write a ladder logic program to drive a set of flashing lights. In total there are 10 lights connected to O:000/0 to O:000/11. At any time every one out of three lights should be on. Every second the pattern on the lights should shift towards O:000/11.
16.9 PRACTICE PROBLEM SOLUTIONS

1. TON Timer T4:0

   - Delay 4s
   - Bit address R6:0/UL
   - Length 10

   - File B3: 0
   - Control R6:0

   - File B3: 1
   - Control R6:1
   - Bit address R6:1/UL
   - Length 10

   - File B3: 2
   - Control R6:2
   - Bit address R6:2/UL
   - Length 10

   - File B3: 3
   - Control R6:3
   - Bit address R6:3/UL
   - Length 10

   - File B3: 4
   - Control R6:4
   - Bit address R6:4/UL
   - Length 10

   - File B3: 5
   - Control R6:5
   - Bit address R6:5/UL
   - Length 10

   - B3:0 = 0000 0000 0000 1111 (grn EW)
   - B3:1 = 0000 0000 0001 0000 (yel EW)
   - B3:2 = 0000 0011 1110 0000 (red EW)
   - B3:3 = 0000 0011 1100 0000 (grn NS)
   - B3:4 = 0000 0000 0010 0000 (yel NS)
   - B3:5 = 0000 0000 0001 1111 (red NS)
3. In MCR blocks the outputs will all be forced off. This is not a problem for outputs such as retentive timers and latches, but it will force off normal outputs. JMP statements will skip over logic and not examine it or force it off.

4. Timed interrupts are useful for processes that must happen at regular time intervals. Polled interrupts are useful to monitor inputs that must be checked more frequently than the ladder scan time will permit. Fault interrupts are important for processes where the complete failure of the PLC could be dangerous.

5. These can be used to update inputs and outputs more frequently than the normal scan time permits.

6. The main differences are: Shift registers focus on bits, stacks and sequencers on words. Shift registers and sequencers are fixed length, stacks are variable lengths.
TON timer T4:0 delay 1s

S3
S5
B3/0
S2

O:001/0
O:001/1
LFL
source N7:0
LIFO N7:10
Control R6:0
length 10
position 0
LFU
LIFO N7:10
destination N7:1
Control R6:0
length 10
position 0

EQU
SourceA N7:1
SourceB N7:2
B3/2
I:000/0 B3/1
I:000/1
I:000/2
I:000/0

MOV Source 1 Destination N7:2
I:000/1
MOV Source 2 Destination N7:2
I:000/2
MOV Source 3 Destination N7:2
10. a) Timed, polled and fault, b) They remove the need to check for times or scan for memory
changes, and they allow events to occur more often than the ladder logic is scanned. c) A few rungs of ladder logic might count on a value remaining constant, but an interrupt might change the memory, thereby corrupting the logic. d) The UID and UIE

```
TON T4:0 1 s
T4:0/DN

MOV source 1001001001 B
dest. B3:0

FS
BSR
File #B3:0
Control R6:0
Bit R6:0/UL
Length 10
MVM
source B3:0
mask 03FF H
dest O:000

T4:0/DN
```
L U JSR File 3
JSR File 4
JSR File 5
ST 0
ST 1
ST 2
file 2
L U A ST 1 ST 0
file 3
L U C ST 0 ST 1
file 4
L U D ST 1 ST 2
file 5
L U B ST 2 ST 1
C
RET
RET
RET
16.10 ASSIGNMENT PROBLEMS

1. Write a program that will continuously cycle a pattern of 12 lights connected to a PLC output card. The pattern should have one out of every three lights set. The light patterns should appear to move endlessly in one direction.

2. Using 3 different methods write a program that will continuously cycle a pattern of 12 lights connected to a PLC output card. The pattern should have one out of every three lights set. The light patterns should appear to move endlessly in one direction.

3. Look at the manuals for the status memory in your PLC.
   a) Describe how to run program 7 when a divide by zero error occurs.
   b) Write the ladder logic needed to clear a PLC fault.
   c) Describe how to set up a timed interrupt to run program 5 every 2 seconds.

4. Write a program that will run once every 5 seconds and calculate the average of the numbers from F8:0 to F8:19, and store the result in F8:20. It will also determine the median and store it in F8:21.

5. Write a program for SPC (Statistical Process Control) that will run once every 20 minutes using timed interrupts. When the program runs it will calculate the average of the data values in memory locations F8:0 to F8:39 (Note: these values are written into the PLC memory by another PLC using DH+). The program will also find the range of the values by subtracting the maximum from the minimum value. The average will be compared to upper (F8:50) and lower (F8:51) limits. The range will also be compared to upper (F8:52) and lower (F8:53) limits. If the average, or range values are outside the limits, the process will stop, and an 'out of control' light will be turned on. The process will use start and stop buttons, and when running it will set memory bit B3:0/0.

6. Develop a ladder logic program to control a light display outside a theater. The display consists of a row of 8 lights. When a patron walks past an optical sensor the lights will turn on in sequence, moving in the same direction. Initially all lights are off. Once triggered the lights turn on sequentially until all eight lights are on 1.6 seconds latter. After a delay of another 0.4 seconds the lights start to turn off until all are off, again moving in the same direction as the patron. The effect is a moving light pattern that follows the patron as they walk into the theater.

7. Write the ladder logic diagram that would be required to execute the following data manipulation for a preventative maintenance program.
   i) Keep track of the number of times a motor was started with toggle switch #1.
   ii) After 2000 motor starts turn on an indicator light on the operator panel.
   iii) Provide the capability to change the number of motor starts being tracked, prior to triggering of the indicator light. HINT: This capability will only require the change of a value in a compare statement rather than the addition of new lines of logic.
   iv) Keep track of the number of minutes that the motor has run.
   v) After 9000 minutes of operation turn the motor off automatically and also turn...
8. Parts arrive at an oven on a conveyor belt and pass a barcode scanner. When the barcode scanner reads a valid barcode it outputs the numeric code as 32 bits to I:001 and I:002 and sets input I:000/0. The PLC must store this code until the parts pass through the oven. When the parts leave the oven they are detected by a proximity sensor connected to I:000/1. The barcode value read before must be output to O:003 and O:004. Write the ladder logic for the process.

There can be up to ten parts inside the oven at any time.

9. Write the ladder logic for the state diagram below using subroutines for the states.