8. PLC OPERATION

Topics:
- The computer structure of a PLC
- The sanity check, input, output and logic scans
- Status and memory types

Objectives:
- Understand the operation of a PLC.

8.1 INTRODUCTION

For simple programming the relay model of the PLC is sufficient. As more complex functions are used the more complex VonNeuman model of the PLC must be used. A VonNeuman computer processes one instruction at a time. Most computers operate this way, although they appear to be doing many things at once. Consider the computer components shown in Figure 8.1.

![Simplified Personal Computer Architecture](image)

*Figure 8.1* Simplified Personal Computer Architecture

Input is obtained from the keyboard and mouse, output is sent to the screen, and the disk and memory are used for both input and output for storage. (Note: the directions of these arrows are very important to engineers, always pay attention to indicate where information is flowing.) This figure can be redrawn as in Figure 8.2 to clarify the role of
inputs and outputs.

Figure 8.2 An Input-Output Oriented Architecture

In this figure the data enters the left side through the inputs. (Note: most engineering diagrams have inputs on the left and outputs on the right.) It travels through buffering circuits before it enters the CPU. The CPU outputs data through other circuits. Memory and disks are used for storage of data that is not destined for output. If we look at a personal computer as a controller, it is controlling the user by outputting stimuli on the screen, and inputting responses from the mouse and the keyboard.

A PLC is also a computer controlling a process. When fully integrated into an application the analogies become;

inputs - the keyboard is analogous to a proximity switch
input circuits - the serial input chip is like a 24Vdc input card
computer - the 686 CPU is like a PLC CPU unit
output circuits - a graphics card is like a triac output card
outputs - a monitor is like a light
storage - memory in PLCs is similar to memories in personal computers
It is also possible to implement a PLC using a normal Personal Computer, although this is not advisable. In the case of a PLC the inputs and outputs are designed to be more reliable and rugged for harsh production environments.

### 8.2 OPERATION SEQUENCE

All PLCs have four basic stages of operations that are repeated many times per second. Initially when turned on the first time it will check it’s own hardware and software for faults. If there are no problems it will copy all the input and copy their values into memory, this is called the input scan. Using only the memory copy of the inputs the ladder logic program will be solved once, this is called the logic scan. While solving the ladder logic the output values are only changed in temporary memory. When the ladder scan is done the outputs will updated using the temporary values in memory, this is called the output scan. The PLC now restarts the process by starting a self check for faults. This process typically repeats 10 to 100 times per second as is shown in Figure 8.3.

![PLC Scan Cycle](image)

**Figure 8.3**  PLC Scan Cycle

The input and output scans often confuse the beginner, but they are important. The
input scan takes a snapshot of the inputs, and solves the logic. This prevents potential problems that might occur if an input that is used in multiple places in the ladder logic program changed while halfway through a ladder scan. Thus changing the behaviors of half of the ladder logic program. This problem could have severe effects on complex programs that are developed later in the book. One side effect of the input scan is that if a change in input is too short in duration, it might fall between input scans and be missed.

When the PLC is initially turned on the normal outputs will be turned off. This does not affect the values of the inputs.

8.2.1 The Input and Output Scans

When the inputs to the PLC are scanned the physical input values are copied into memory. When the outputs to a PLC are scanned they are copied from memory to the physical outputs. When the ladder logic is scanned it uses the values in memory, not the actual input or output values. The primary reason for doing this is so that if a program uses an input value in multiple places, a change in the input value will not invalidate the logic. Also, if output bits were changed as each bit was changed, instead of all at once at the end of the scan the PLC would operate much slower.

8.2.2 The Logic Scan

Ladder logic programs are modeled after relay logic. In relay logic each element in the ladder will switch as quickly as possible. But in a program elements can only be examined one at a time in a fixed sequence. Consider the ladder logic in Figure 8.4, the ladder logic will be interpreted left-to-right, top-to-bottom. In the figure the ladder logic scan begins at the top rung. At the end of the rung it interprets the top output first, then the output branched below it. On the second rung it solves branches, before moving along the ladder logic rung.
The logic scan sequence become important when solving ladder logic programs which use outputs as inputs, as we will see in Chapter 8. It also becomes important when considering output usage. Consider Figure 8.5, the first line of ladder logic will examine input $A$ and set output $X$ to have the same value. The second line will examine input $B$ and set the output $X$ to have the opposite value. So the value of $X$ was only equal to $A$ until the second line of ladder logic was scanned. Recall that during the logic scan the outputs are only changed in memory, the actual outputs are only updated when the ladder logic scan is complete. Therefore the output scan would update the real outputs based upon the second line of ladder logic, and the first line of ladder logic would be ineffective.

Note: It is a common mistake for beginners to unintentionally repeat the same ladder logic output more than once. This will basically invalidate the first output, in this case the first line will never do anything.
8.3 PLC STATUS

The lack of keyboard, and other input-output devices is very noticeable on a PLC. On the front of the PLC there are normally limited status lights. Common lights indicate;

- power on - this will be on whenever the PLC has power
- program running - this will often indicate if a program is running, or if no program is running
- fault - this will indicate when the PLC has experienced a major hardware or software problem

These lights are normally used for debugging. Limited buttons will also be provided for PLC hardware. The most common will be a run/program switch that will be switched to program when maintenance is being conducted, and back to run when in production. This switch normally requires a key to keep unauthorized personnel from altering the PLC program or stopping execution. A PLC will almost never have an on-off switch or reset button on the front. This needs to be designed into the remainder of the system.

The status of the PLC can be detected by ladder logic also. It is common for programs to check to see if they are being executed for the first time, as shown in Figure 8.6. The 'first scan' input will be true the very first time the ladder logic is scanned, but false on every other scan. In this case the address for 'first scan' in a PLC-5 is 'S2:1/14'. With the logic in the example the first scan will seal on 'light', until 'clear' is turned on. So the light will turn on after the PLC has been turned on, but it will turn off and stay off after 'clear' is turned on. The 'first scan' bit is also referred to at the 'first pass' bit.

![Figure 8.6](image)

8.4 MEMORY TYPES

There are a few basic types of computer memory that are in use today.

RAM (Random Access Memory) - this memory is fast, but it will lose its contents
when power is lost, this is known as volatile memory. Every PLC uses this memory for the central CPU when running the PLC.

ROM (Read Only Memory) - this memory is permanent and cannot be erased. It is often used for storing the operating system for the PLC.

EPROM (Erasable Programmable Read Only Memory) - this is memory that can be programmed to behave like ROM, but it can be erased with ultraviolet light and reprogrammed.

EEPROM (Electronically Erasable Programmable Read Only Memory) - This memory can store programs like ROM. It can be programmed and erased using a voltage, so it is becoming more popular than EPROMs.

All PLCs use RAM for the CPU and ROM to store the basic operating system for the PLC. When the power is on the contents of the RAM will be kept, but the issue is what happens when power to the memory is lost. Originally PLC vendors used RAM with a battery so that the memory contents would not be lost if the power was lost. This method is still in use, but is losing favor. EPROMs have also been a popular choice for programming PLCs. The EPROM is programmed out of the PLC, and then placed in the PLC. When the PLC is turned on the ladder logic program on the EPROM is loaded into the PLC and run. This method can be very reliable, but the erasing and programming technique can be time consuming. EEPROM memories are a permanent part of the PLC, and programs can be stored in them like EPROM. Memory costs continue to drop, and newer types (such as flash memory) are becoming available, and these changes will continue to impact PLCs.

8.5 SOFTWARE BASED PLCS

The dropping cost of personal computers is increasing their use in control, including the replacement of PLCs. Software is installed that allows the personal computer to solve ladder logic, read inputs from sensors and update outputs to actuators. These are important to mention here because they don’t obey the previous timing model. For example, if the computer is running a game it may slow or halt the computer. This issue and others are currently being investigated and good solutions should be expected soon.

8.6 SUMMARY

• A PLC and computer are similar with inputs, outputs, memory, etc.
• The PLC continuously goes through a cycle including a sanity check, input scan, logic scan, and output scan.
• While the logic is being scanned, changes in the inputs are not detected, and the outputs are not updated.
• PLCs use RAM, and sometime EPROMs are used for permanent programs.
8.7 PRACTICE PROBLEMS

1. Does a PLC normally contain RAM, ROM, EPROM and/or batteries.

2. What are the indicator lights on a PLC used for?

3. A PLC can only go through the ladder logic a few times per second. Why?

4. What will happen if the scan time for a PLC is greater than the time for an input pulse? Why?

5. What is the difference between a PLC and a desktop computer?

6. Why do PLCs do a self check every scan?

7. Will the test time for a PLC be long compared to the time required for a simple program.

8. What is wrong with the following ladder logic? What will happen if it is used?

9. What is the address for a memory location that indicates when a PLC has just been turned on?

8.8 PRACTICE PROBLEM SOLUTIONS

1. Every PLC contains RAM and ROM, but they may also contain EPROM or batteries.

2. Diagnostic and maintenance

3. Even if the program was empty the PLC would still need to scan inputs and outputs, and do a self check.

4. The pulse may be missed if it occurs between the input scans

5. Some key differences include inputs, outputs, and uses. A PLC has been designed for the factory floor, so it does not have inputs such as keyboards and mice (although some newer types can). They also do not have outputs such as a screen or sound. Instead they have inputs and outputs for voltages and current. The PLC runs user designed programs for specialized tasks,
whereas on a personal computer it is uncommon for a user to program their system.

6. This helps detect faulty hardware or software. If an error were to occur, and the PLC continued operating, the controller might behave in an unpredictable way and become dangerous to people and equipment. The self check helps detect these types of faults, and shut the system down safely.

7. Yes, the self check is equivalent to about 1ms in many PLCs, but a single program instruction is about 1 micro second.

8. The normal output $Y$ is repeated twice. In this example the value of $Y$ would always match $B$, and the earlier rung with $A$ would have no effect on $Y$.


8.9 ASSIGNMENT PROBLEMS

1. Describe the basic steps of operation for a PLC after it is turned on.

2. Repeating a normal output in ladder logic should not be done normally. Discuss why.

3. Why does removing a battery from some PLCs clear the memory?
9. LATCHES, TIMERS, COUNTERS AND MORE

Topics:

- Latches, timers, counters and MCRs
- Design examples
- Internal memory locations are available, and act like outputs

Objectives:

- Understand latches, timers, counters and MCRs.
- To be able to select simple internal memory bits.

9.1 INTRODUCTION

More complex systems cannot be controlled with combinatorial logic alone. The main reason for this is that we cannot, or choose not to add sensors to detect all conditions. In these cases we can use events to estimate the condition of the system. Typical events used by a PLC include;

- first scan of the PLC - indicating the PLC has just been turned on
- time since an input turned on/off - a delay
- count of events - to wait until set number of events have occurred
- latch on or unlatch - to lock something on or turn it off

The common theme for all of these events is that they are based upon one of two questions "How many?" or "How long?". An example of an event based device is shown in Figure 9.1. The input to the device is a push button. When the push button is pushed the input to the device turns on. If the push button is then released and the device turns off, it is a logical device. If when the push button is released the device stays on, is will be one type of event based device. To reiterate, the device is event based if it can respond to one or more things that have happened before. If the device responds only one way to the immediate set of inputs, it is logical.
9.2 LATCHES

A latch is like a sticky switch - when pushed it will turn on, but stick in place, it must be pulled to release it and turn it off. A latch in ladder logic uses one instruction to latch, and a second instruction to unlatch, as shown in Figure 9.2. The output with an \( L \) inside will turn the output \( D \) on when the input \( A \) becomes true. \( D \) will stay on even if \( A \) turns off. Output \( D \) will turn off if input \( B \) becomes true and the output with a \( U \) inside becomes true (Note: this will seem a little backwards at first). If an output has been latched on, it will keep its value, even if the power has been turned off.
The operation of the ladder logic in Figure 9.2 is illustrated with a timing diagram in Figure 9.3. A timing diagram shows values of inputs and outputs over time. For example the value of input A starts low (false) and becomes high (true) for a short while, and then goes low again. Here when input A turns on both the outputs turn on. There is a slight delay between the change in inputs and the resulting changes in outputs, due to the program scan time. Here the dashed lines represent the output scan, sanity check and input scan (assuming they are very short.) The space between the dashed lines is the ladder logic scan. Consider that when A turns on initially it is not detected until the first dashed line. There is then a delay to the next dashed line while the ladder is scanned, and then the output at the next dashed line. When A eventually turns off, the normal output C turns off, but the latched output D stays on. Input B will unlatch the output D. Input B turns on twice, but the first time it is on is not long enough to be detected by an input scan, so it is ignored. The second time it is on it unlatches output D and output D turns off.

Timing Diagram

These lines indicate PLC input/output refresh times. At this time all of the outputs are updated, and all of the inputs are read. Notice that some inputs can be ignored if at the wrong time, and there can be a delay between a change in input, and a change in output.

The space between the lines is the scan time for the ladder logic. The spaces may vary if different parts of the ladder diagram are executed each time through the ladder (as with state space code). The space is a function of the speed of the PLC, and the number of Ladder logic elements in the program.
The timing diagram shown in Figure 9.3 has more details than are normal in a timing diagram as shown in Figure 9.4. The brief pulse would not normally be wanted, and would be designed out of a system either by extending the length of the pulse, or decreasing the scan time. An ideal system would run so fast that aliasing would not be possible.

A more elaborate example of latches is shown in Figure 9.5. In this example the addresses are for an Allen-Bradley Micrologix controller. The inputs begin with \( I / \), followed by an input number. The outputs begin with \( O / \), followed by an output number.
A normal output should only appear once in ladder logic, but latch and unlatch instructions may appear multiple times. In Figure 9.5 a normal output $O/2$ is repeated twice. When the program runs it will examine the fourth line and change the value of $O/2$ in memory (remember the output scan does not occur until the ladder scan is done.) The last line is then interpreted and it overwrites the value of $O/2$. Basically, only the last line will change $O/2$.

Latches are not used universally by all PLC vendors, others such as Siemens use
flip-flops. These have a similar behavior to latches, but a different notation as illustrated in Figure 9.6. Here the flip-flop is an output block that is connected to two different logic rungs. The first rung shown has an input $A$ connected to the $S$ setting terminal. When $A$ goes true the output value $Q$ will go true. The second rung has an input $B$ connected to the $R$ resetting terminal. When $B$ goes true the output value $Q$ will be turned off. The output $Q$ will always be the inverse of $\overline{Q}$. Notice that the $S$ and $R$ values are equivalent to the $L$ and $U$ values from earlier examples.

---

**Figure 9.6** Flip-Flops for Latching Values

### 9.3 TIMERS

There are four fundamental types of timers shown in Figure 9.7. An on-delay timer will wait for a set time after a line of ladder logic has been true before turning on, but it will turn off immediately. An off-delay timer will turn on immediately when a line of ladder logic is true, but it will delay before turning off. Consider the example of an old car. If you turn the key in the ignition and the car does not start immediately, that is an on-delay. If you turn the key to stop the engine but the engine doesn’t stop for a few seconds, that is an off delay. An on-delay timer can be used to allow an oven to reach temperature before starting production. An off delay timer can keep cooling fans on for a set time after the
oven has been turned off.

<table>
<thead>
<tr>
<th></th>
<th>on-delay</th>
<th>off-delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>retentive</td>
<td>RTO</td>
<td>RTF</td>
</tr>
<tr>
<td>nonretentive</td>
<td>TON</td>
<td>TOF</td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TON - Timer ON</td>
</tr>
<tr>
<td>TOF - Timer OFF</td>
</tr>
<tr>
<td>RTO - Retentive Timer On</td>
</tr>
<tr>
<td>RTF - Retentive Timer OFF</td>
</tr>
</tbody>
</table>

**Figure 9.7** The Four Basic Timer Types

A retentive timer will sum all of the on or off time for a timer, even if the timer never finished. A nonretentive timer will start timing the delay from zero each time. Typical applications for retentive timers include tracking the time before maintenance is needed. A non retentive timer can be used for a start button to give a short delay before a conveyor begins moving.

An example of an Allen-Bradley TON timer is shown in Figure 9.8. The rung has a single input $A$ and a function block for the $TON$. (Note: This timer block will look different for different PLCs, but it will contain the same information.) The information inside the timer block describes the timing parameters. The first item is the timer number $T4:0$. This is a location in the PLC memory that will store the timer information. The $T4:0$ indicates that it is timer memory, and the $0$ indicates that it is in the first location. The time base is $1.0$ indicating that the timer will work in $1.0$ second intervals. Other time bases are available in fractions and multiples of seconds. The preset is the delay for the timer, in this case it is $4$. To find the delay time multiply the time base by the preset value $4*1.0s = 4.0s$. The accumulator value gives the current value of the timer as $0$. While the timer is running the Accumulated value will increase until it reaches the preset value. Whenever the input $A$ is true the $EN$ output will be true. The $DN$ output will be false until the accumulator has reached the preset value. The $EN$ and $DN$ outputs cannot be changed when programming, but these are important when debugging a ladder logic program. The second line of ladder logic uses the timer $DN$ output to control another output $B$. 
The timing diagram in Figure 9.8 illustrates the operation of the TON timer with a 4 second on-delay. $A$ is the input to the timer, and whenever the timer input is true the $EN$ enabled bit for the timer will also be true. If the accumulator value is equal to the preset value the $DN$ bit will be set. Otherwise, the $TT$ bit will be set and the accumulator value will begin increasing. The first time $A$ is true, it is only true for 3 seconds before turning off, after this the value resets to zero. (Note: in a retentive time the value would remain at 3 seconds.) The second time $A$ is true, it is on more than 4 seconds. After 4 seconds the $TT$ bit turns off, and the $DN$ bit turns on. But, when $A$ is released the accumulator resets to zero, and the $DN$ bit is turned off.

A value can be entered for the accumulator while programming. When the program is downloaded this value will be in the timer for the first scan. If the TON timer is not enabled the value will be set back to zero. Normally zero will be entered for the preset
value.

The timer in Figure 9.9 is identical to that in Figure 9.8, except that it is retentive. The most significant difference is that when the input \( A \) is turned off the accumulator value does not reset to zero. As a result the timer turns on much sooner, and the timer does not turn off after it turns on. A reset instruction will be shown later that will allow the accumulator to be reset to zero.

![Diagram of an Allen Bradley Retentive On-Delay Timer](image)

**Figure 9.9**  An Allen Bradley Retentive On-Delay Timer

An off delay timer is shown in Figure 9.10. This timer has a time base of 0.01s, with a preset value of 350, giving a total delay of 3.5s. As before the \( EN \) enable for the timer matches the input. When the input \( A \) is true the \( DN \) bit is on. Is is also on when the input \( A \) has turned off and the accumulator is counting. The \( DN \) bit only turns off when the input \( A \) has been off long enough so that the accumulator value reaches the preset. This type of timer is not retentive, so when the input \( A \) becomes true, the accumulator resets.
Retentive off-delay (RTF) timers have few applications and are rarely used, therefore many PLC vendors do not include them.

An example program is shown in Figure 9.11. In total there are four timers used in this example, T4:1 to T4:4. The timer instructions are shown with a shorthand notation with the timebase and preset values combined as the delay. All four different types of counters have the input I/1. Output O/I will turn on when the TON counter T4:1 is done. All four of the timers can be reset with input I/2.
A timing diagram for this example is shown in Figure 9.12. As input I/1 is turned on the TON and RTO timers begin to count and reach 4s and turn on. When I/2 becomes true it resets both timers and they start to count for another second before I/1 is turned off. After the input is turned off the TOF and RTF both start to count, but neither reaches the 4s preset. The input I/1 is turned on again and the TON and RTO both start counting. The RTO turns on one second sooner because it had 1s stored from the 7-8s time period. After I/1 turns off again both the off delay timers count down, and reach the 4 second delay, and turn on. These patterns continue across the diagram.
Consider the short ladder logic program in Figure 9.13 for control of a heating oven. The system is started with a Start button that seals in the Auto mode. This can be stopped if the Stop button is pushed. (Remember: Stop buttons are normally closed.) When the Auto goes on initially the TON timer is used to sound the horn for the first 10 seconds to warn that the oven will start, and after that the horn stops and the heating coils start. When the oven is turned off the fan continues to blow for 300s or 5 minutes after.
A program is shown in Figure 9.14 that will flash a light once every second. When the PLC starts, the second timer will be off and the $T4:1/DN$ bit will be off, therefore the normally closed input to the first timer will be on. $T4:0$ will start timing until it reaches 0.5s, when it is done the second timer will start timing, until it reaches 0.5s. At that point $T4:1/DN$ will become true, and the input to the first time will become false. $T4:0$ is then set back to zero, and then $T4:1$ is set back to zero. And, the process starts again from the beginning. In this example the first timer is used to drive the second timer. This type of arrangement is normally called cascading, and can use more than two timers.

Figure 9.13  A Timer Example
There are two basic counter types: count-up and count-down. When the input to a count-up counter goes true the accumulator value will increase by 1 (no matter how long the input is true.) If the accumulator value reaches the preset value the counter $DN$ bit will be set. A count-down counter will decrease the accumulator value until the preset value is reached.

An Allen Bradley count-up (CTU) instruction is shown in Figure 9.15. The instruction requires memory in the PLC to store values and status, in this case is $C5:0$. The $C5:$ indicates that it is counter memory, and the $0$ indicates that it is the first location. The preset value is 4 and the value in the accumulator is 2. If the input $A$ were to go from false to true the value in the accumulator would increase to 3. If $A$ were to go off, then on again the accumulator value would increase to 4, and the $DN$ bit would go on. The count can continue above the preset value. If input $B$ goes true the value in the counter accumulator will become zero.
Count-down counters are very similar to count-up counters. And, they can actually both be used on the same counter memory location. Consider the example in Figure 9.16, the example input I/1 drives the count-up instruction for counter C5:1. Input I/2 drives the count-down instruction for the same counter location. The preset value for a counter is stored in memory location C5:1 so both the count-up and count-down instruction must have the same preset. Input I/3 will reset the counter.
The timing diagram in Figure 9.16 illustrates the operation of the counter. If we assume that the value in the accumulator starts at 0, then the I/1 inputs cause it to count up to 3 where it turns the counter C5:1 on. It is then reset by input I/3 and the accumulator value goes to zero. Input I/1 then pulses again and causes the accumulator value to increase again, until it reaches a maximum of 5. Input I/2 then causes the accumulator value to decrease down below 3, and the counter turns off again. Input I/1 then causes it to increase, but input I/3 resets the accumulator back to zero again, and the pulses continue until 3 is reached near the end.
The program in Figure 9.17 is used to remove 5 out of every 10 parts from a conveyor with a pneumatic cylinder. When the part is detected both counters will increase their values by 1. When the sixth part arrives the first counter will then be done, thereby allowing the pneumatic cylinder to actuate for any part after the fifth. The second counter will continue until the eleventh part is detected and then both of the counters will be reset.

**Figure 9.17** A Counter Example

### 9.5 MASTER CONTROL RELAYS (MCRs)

In an electrical control system a Master Control Relay (MCR) is used to shut down a section of an electrical system, as shown earlier in the electrical wiring chapter. This concept has been implemented in ladder logic also. A section of ladder logic can be put between two lines containing MCR's. When the first MCR coil is active, all of the intermediate ladder logic is executed up to the second line with an MCR coil. When the first MCR coil is inactive, the ladder logic is still examined, but all of the outputs are forced off.

Consider the example in Figure 9.18. If A is true, then the ladder logic after will be
executed as normal. If \(A\) is false the following ladder logic will be examined, but all of the outputs will be forced off. The second MCR function appears on a line by itself and marks the end of the MCR block. After the second MCR the program execution returns to normal. While \(A\) is true, \(X\) will equal \(B\), and \(Y\) can be turned on by \(C\), and off by \(D\). But, if \(A\) becomes false \(X\) will be forced off, and \(Y\) will be left in its last state. Using MCR blocks to remove sections of programs will not increase the speed of program execution significantly because the logic is still examined.

\[\text{Figure 9.18 MCR Instructions}\]

If the MCR block contained another function, such as a TON timer, turning off the MCR block would force the timer off. As a general rule normal outputs should be outside MCR blocks, unless they must be forced off when the MCR block is off.

Note: If a normal input is used inside an MCR block it will be forced off. If the output is also used in other MCR blocks the last one will be forced off. The MCR is designed to fully stop an entire section of ladder logic, and is best used this way in ladder logic designs.
9.6 INTERNAL RELAYS

Inputs are used to set outputs in simple programs. More complex programs also use internal memory locations that are not inputs or outputs. These are sometimes referred to as 'internal relays' or 'control relays'. Knowledgeable programmers will often refer to these as 'bit memory'. In the Allen Bradley PLCs these addresses begin with 'B3' by default. The first bit in memory is 'B3:0/0', where the first zero represents the first 16 bit word, and the second zero represents the first bit in the word. The sequence of bits is shown in Figure 9.19. The programmer is free to use these memory locations however they see fit.

<table>
<thead>
<tr>
<th>bit number</th>
<th>memory location</th>
<th>bit number</th>
<th>memory location</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B3:0/0</td>
<td>18</td>
<td>B3:1/2</td>
</tr>
<tr>
<td>1</td>
<td>B3:0/1</td>
<td>19</td>
<td>B3:1/3</td>
</tr>
<tr>
<td>2</td>
<td>B3:0/2</td>
<td>20</td>
<td>B3:1/4</td>
</tr>
<tr>
<td>3</td>
<td>B3:0/3</td>
<td>21</td>
<td>B3:1/5</td>
</tr>
<tr>
<td>4</td>
<td>B3:0/4</td>
<td>22</td>
<td>B3:1/6</td>
</tr>
<tr>
<td>5</td>
<td>B3:0/5</td>
<td>23</td>
<td>B3:1/7</td>
</tr>
<tr>
<td>6</td>
<td>B3:0/6</td>
<td>24</td>
<td>B3:1/8</td>
</tr>
<tr>
<td>7</td>
<td>B3:0/7</td>
<td>25</td>
<td>B3:1/9</td>
</tr>
<tr>
<td>8</td>
<td>B3:0/8</td>
<td>26</td>
<td>B3:1/10</td>
</tr>
<tr>
<td>9</td>
<td>B3:0/9</td>
<td>27</td>
<td>B3:1/11</td>
</tr>
<tr>
<td>10</td>
<td>B3:0/10</td>
<td>28</td>
<td>B3:1/12</td>
</tr>
<tr>
<td>11</td>
<td>B3:0/11</td>
<td>29</td>
<td>B3:1/13</td>
</tr>
<tr>
<td>12</td>
<td>B3:0/12</td>
<td>30</td>
<td>B3:1/14</td>
</tr>
<tr>
<td>13</td>
<td>B3:0/13</td>
<td>31</td>
<td>B3:1/15</td>
</tr>
<tr>
<td>14</td>
<td>B3:0/14</td>
<td>32</td>
<td>B3:2/0</td>
</tr>
<tr>
<td>15</td>
<td>B3:0/15</td>
<td>33</td>
<td>B3:2/1</td>
</tr>
<tr>
<td>16</td>
<td>B3:1/0</td>
<td>34</td>
<td>B3:2/2</td>
</tr>
<tr>
<td>17</td>
<td>B3:1/1</td>
<td>etc...</td>
<td>etc...</td>
</tr>
</tbody>
</table>

Figure 9.19 Bit memory

An example of bit memory usage is shown in Figure 9.20. The first ladder logic rung will turn on the internal memory bit 'B3:0/0' when input 'hand_A' is activated, and input 'clear' is off. (Notice that the B3 memory is being used as both an input and output.) The second line of ladder logic similar. In this case when both inputs have been activated, the output 'press on' is active.
9.7 DESIGN CASES

The following design cases are presented to help emphasize the principles presented in this chapter. I suggest that you try to develop the ladder logic before looking at the provided solutions.

9.7.1 Basic Counters And Timers

Problem: Develop the ladder logic that will turn on an output light, 15 seconds after switch A has been turned on.
Figure 9.21  A Simple Timer Example

Problem: Develop the ladder logic that will turn on a light, after switch $A$ has been closed 10 times. Push button $B$ will reset the counters.

Solution:

Figure 9.22  A Simple Counter Example

9.7.2 More Timers And Counters

Problem: Develop a program that will latch on an output $B$ 20 seconds after input $A$ has been turned on. After $A$ is pushed, there will be a 10 second delay until $A$ can have any effect again. After $A$ has been pushed 3 times, $B$ will be turned off.
Solution:

![Diagram of PLC timer setup]

Figure 9.23  A More Complex Timer Counter Example

### 9.7.3 Deadman Switch

Problem: A motor will be controlled by two switches. The Go switch will start the motor and the Stop switch will stop it. If the Stop switch was used to stop the motor, the Go switch must be thrown twice to start the motor. When the motor is active a light should be turned on. The Stop switch will be wired as normally closed.
Problem: A conveyor is run by switching on or off a motor. We are positioning parts on the conveyor with an optical detector. When the optical sensor goes on, we want to wait 1.5 seconds, and then stop the conveyor. After a delay of 2 seconds the conveyor will start again. We need to use a start and stop button - a light should be on when the system is active.

Solution:

Consider:
- what will happen if stop is pushed and the motor is not running?

*Figure 9.24*  A Motor Starter Example

### 9.7.4 Conveyor

Problem: A conveyor is run by switching on or off a motor. We are positioning parts on the conveyor with an optical detector. When the optical sensor goes on, we want to wait 1.5 seconds, and then stop the conveyor. After a delay of 2 seconds the conveyor will start again. We need to use a start and stop button - a light should be on when the system is active.
Solution:

- what is assumed about part arrival and departure?

Figure 9.25  A Conveyor Controller Example

9.7.5 Accept/Reject Sorting

Problem: For the conveyor in the last case we will add a sorting system. Gages have been attached that indicate good or bad. If the part is good, it continues on. If the part is bad, we do not want to delay for 2 seconds, but instead actuate a pneumatic cylinder.
Solution:

Figure 9.26    A Conveyor Sorting Example
9.7.6 Shear Press

Problem: The basic requirements are,

1. A toggle start switch (TS1) and a limit switch on a safety gate (LS1) must both be on before a solenoid (SOL1) can be energized to extend a stamping cylinder to the top of a part.
2. While the stamping solenoid is energized, it must remain energized until a limit switch (LS2) is activated. This second limit switch indicates the end of a stroke. At this point the solenoid should be de-energized, thus retracting the cylinder.
3. When the cylinder is fully retracted a limit switch (LS3) is activated. The cycle may not begin again until this limit switch is active.
4. A cycle counter should also be included to allow counts of parts produced. When this value exceeds 5000 the machine should shut down and a light lit up.
5. A safety check should be included. If the cylinder solenoid has been on for more than 5 seconds, it suggests that the cylinder is jammed or the machine has a fault. If this is the case, the machine should be shut down and a maintenance light turned on.
9.8 SUMMARY

- Latch and unlatch instructions will hold outputs on, even when the power is turned off.
- Timers can delay turning on or off. Retentive timers will keep values, even when inactive. Resets are needed for retentive timers.
- Counters can count up or down.
- When timers and counters reach a preset limit the $DN$ bit is set.
9.28

MCRs can force off a section of ladder logic.

9.9 PRACTICE PROBLEMS

1. What does edge triggered mean? What is the difference between positive and negative edge triggered?

2. Are reset instructions necessary for all timers and counters?

3. What are the numerical limits for typical timers and counters?

4. If a counter goes below the bottom limit which counter bit will turn on?

5. a) Write ladder logic for a motor starter that has a start and stop button that uses latches. b) Write the same ladder logic without latches.

6. Use a timing diagram to explain how an on delay and off delay timer are different.

7. For the retentive off timer below, draw out the status bits.

```
RTF
Timer T4:0
Time Base 0.01
Preset 350
Accum. 0
```

```
A

T4:0/EN
T4:0/DN
T4:0/TT

T4:0.Accum.

0 3 6 10 16 18 20
```
8. Complete the timing diagrams for the two timers below.

RTO

Timer T4:0
Time Base 1.0
Preset 10
Accum. 1

TOF

Timer T4:1
Time Base .01
Preset 50
Accum. 0
9. Given the following timing diagram, draw the done bits for all four fundamental timer types. Assume all start with an accumulated value of zero, and have a preset of 1.5 seconds.

10. Design ladder logic that allows an RTO to behave like a TON.

11. Design ladder logic that uses normal timers and counters to measure times of 50.0 days.

12. Develop the ladder logic that will turn on an output light (O/1), 15 seconds after switch A (I/1) has been turned on.

13. Develop the ladder logic that will turn on a light (O/1), after switch A (I/1) has been closed 10 times. Push button B (I/2) will reset the counters.

14. Develop a program that will latch on an output B (O/1), 20 seconds after input A (I/1) has been turned on. The timer will continue to cycle up to 20 seconds, and reset itself, until input A has been turned off. After the third time the timer has timed to 20 seconds, the output B will be unlatched.

15. A motor will be connected to a PLC and controlled by two switches. The GO switch will start the motor, and the STOP switch will stop it. If the motor is going, and the GO switch is thrown, this will also stop the motor. If the STOP switch was used to stop the motor, the GO switch must be thrown twice to start the motor. When the motor is running, a light should be turned on (a small lamp will be provided).

16. In dangerous processes it is common to use two palm buttons that require a operator to use both hands to start a process (this keeps hands out of presses, etc.). To develop this there are two inputs that must be turned on within 0.25s of each other before a machine cycle may begin.
17. Design a conveyor control system that follows the design guidelines below.
   - The conveyor has an optical sensor $S_1$ that detects boxes entering a workcell
   - There is also an optical sensor $S_2$ that detects boxes leaving the workcell
   - The boxes enter the workcell on a conveyor controlled by output $C_1$
   - The boxes exit the workcell on a conveyor controlled by output $C_2$
   - The controller must keep a running count of boxes using the entry and exit sensors
   - If there are more than five boxes in the workcell the entry conveyor will stop
   - If there are no boxes in the workcell the exit conveyor will be turned off
   - If the entry conveyor has been stopped for more than 30 seconds the count will be reset to zero, assuming that the boxes in the workcell were scrapped.

18. Write a ladder logic program that does what is described below.
   - When button $A$ is pushed, a light will flash for 5 seconds.
   - The flashing light will be on for 0.25 sec and off for 0.75 sec.
   - If button $A$ has been pushed 5 times the light will not flash until the system is reset.
   - The system can be reset by pressing button $B$

19. Write a program that will turn on a flashing light for the first 15 seconds after a PLC is turned on. The light should flash for half a second on and half a second off.

20. A buffer can hold up to 10 parts. Parts enter the buffer on a conveyor controller by output conveyor. As parts arrive they trigger an input sensor enter. When a part is removed from the buffer they trigger the exit sensor. Write a program to stop the conveyor when the buffer is full, and restart it when there are fewer than 10 parts in the buffer. As normal the system should also include a start and stop button.

21. What is wrong with the following ladder logic? What will happen if it is used?

22. We are using a pneumatic cylinder in a process. The cylinder can become stuck, and we need to detect this. Proximity sensors are added to both endpoints of the cylinder’s travel to indicate when it has reached the end of motion. If the cylinder takes more than 2 seconds to complete a motion this will indicate a problem. When this occurs the machine should be shut down and a light turned on. Develop ladder logic that will cycle the cylinder in and out repeatedly, and watch for failure.
9.10 PRACTICE PROBLEM SOLUTIONS

1. edge triggered means the event when a logic signal goes from false to true (positive edge) or from true to false (negative edge).

2. no, but they are essential for retentive timers, and very important for counters.

3. these are limited by the 16 bit number for a range of -32768 to +32767

4. the un underflow bit. This may result in a fault in some PLCs.

5.

6.
Timer T4:0
Time Base 0.01
Preset 350
Accum. 0

RTF

A

T4:0/EN
T4:0/DN
T4:0/TT
T4:0.Accum.

0 3 6 10 16 18 20
8. PLC Timers - 9.34

RTO

Timer T4:0
Time Base 1.0
Preset 10
Accum. 1

TOF

Timer T4:1
Time Base .01
Preset 50
Accum. 0
9.

input

TON

RTO

TOF

RTF

01234567 sec

10.

A

RTO
Timer T4:0
Base 1.0
Preset 2

RES T4:0

11.

A

T4:0/DN

TON
Timer T4:0
Base 1.0
Preset 3600

C5:0/DN

CTU
Counter C5:0
Preset 1200

Light
12.

```
I/1 ---- B3/0
  `-----
    B3/0
```

```
T4:0/DN ---- O/01
```

```
TON
T4:0
delay 15 sec
```

13.

```
I/2 ---- C5:0
  `-----
    RES
```

```
I/1 ---- C5:0
  `-----
    CTU
    C5:0
    presetR 10
```

```
C5:0/DN ---- O/1
```
14.

- **I/1**
  - **T4:1/DN**
  - **TON**
  - **T4:0**
  - **delay 20 s**

- **T4:0/DN**
  - **TON**
  - **T4:1**
  - **delay 20 s**

- **L**
  - **O/1**

- **CTU**
  - **C5:0**
  - **preset 3**

- **U**
  - **O/1**

- **C5:0/DN**
15.

```
plc timers - 9.38

go stop

motor

CTU
Counter C5:0
Preset 2
Accumulator 1

CTU
Counter C5:1
Preset 3
Accumulator 1

RES
C5:0

RES
C5:1

CTD
Counter C5:0
Preset 2
Accumulator 1

CTD
Counter C5:1
Preset 3
Accumulator 1
```
16.

```
left button

right button

T4:0/TT  T4:1/TT  stop

on
```

```
TON
Timer T4:0
Base 0.01
Preset 25

TON
Timer T4:1
Base 0.01
Preset 25

on
```
18.

A

T4:0/TT

C5:0/DN

T4:0/TT

T4:2/DN

T4:1/DN

T4:1/TT

light

RES

B

counter C5:0 preset 5

timer T4:0 delay 5s

timer T4:1 delay 0.25s

timer T4:2 delay 0.75s
19. The normal output ‘Y’ is repeated twice. In this example the value of ‘Y’ would always match ‘B’, and the earlier rung with ‘A’ would have no effect on ‘Y’.

20.  

21. The normal output ‘Y’ is repeated twice. In this example the value of ‘Y’ would always match ‘B’, and the earlier rung with ‘A’ would have no effect on ‘Y’.
9.11 ASSIGNMENT PROBLEMS

1. Draw the timer and counter done bits for the ladder logic below. Assume that the accumulators
of all the timers and counters are reset to begin with.

2. Write a ladder logic program that will count the number of parts in a buffer. As parts arrive they activate input \( A \). As parts leave they will activate input \( B \). If the number of parts is less than 8 then a conveyor motor, output \( C \), will be turned on.
3. Explain what would happen in the following program when A is on or off.

4. Write a simple program that will use one timer to flash a light. The light should be on for 1.0 seconds and off for 0.5 seconds. Do not include start or stop buttons.

5. We are developing a safety system (using a PLC-5) for a large industrial press. The press is activated by turning on the compressor power relay (R, connected to O:013/05). After R has been on for 30 seconds the press can be activated to move (P connected to O:013/06). The delay is needed for pressure to build up. After the press has been activated (with P) the system must be shut down (R and P off), and then the cycle may begin again. For safety, there is a sensor that detects when a worker is inside the press (S, connected to I:011/02), which must be off before the press can be activated. There is also a button that must be pushed 5 times (B, connected to I:011/01) before the press cycle can begin. If at any time the worker enters the press (and S becomes active) the press will be shut down (P and R turned off). Develop the ladder logic. State all assumptions, and show all work.

6. Write a program that only uses one timer. When an input A is turned on a light will be on for 10 seconds. After that it will be off for two seconds, and then again on for 5 seconds. After that the light will not turn on again until the input A is turned off.

7. A new printing station will add a logo to parts as they travel along an assembly line. When a part arrives a ‘part’ sensor will detect it. After this the ‘clamp’ output is turned on for 10 seconds to hold the part during the operation. For the first 2 seconds the part is being held a ‘spray’ output will be turned on to apply the thermoset ink. For the last 8 seconds a ‘heat’ output will be turned on to cure the ink. After this the part is released and allowed to continue along the line. Write the ladder logic for this process.

8. Write a ladder logic program. that will turn on an output Q five seconds after an input A is turned on. If input B is on the delay will be eight seconds. YOU MAY ONLY USE ONE TIMER.
10. STRUCTURED LOGIC DESIGN

Topics:
- Timing diagrams
- Design examples
- Designing ladder logic with process sequence bits and timing diagrams

Objectives:
- Know examples of applications to industrial problems.
- Know how to design time base control programs.

10.1 INTRODUCTION

Traditionally ladder logic programs have been written by thinking about the process and then beginning to write the program. This always leads to programs that require debugging. And, the final program is always the subject of some doubt. Structured design techniques, such as Boolean algebra, lead to programs that are predictable and reliable. The structured design techniques in this and the following chapters are provided to make ladder logic design routine and predictable for simple sequential systems.

Note: Structured design is very important in engineering, but many engineers will write software without taking the time or effort to design it. This often comes from previous experience with programming where a program was written, and then debugged. This approach is not acceptable for mission critical systems such as industrial controls. The time required for a poorly designed program is 10% on design, 30% on writing, 40% debugging and testing, 10% documentation. The time required for a high quality program design is 30% design, 10% writing software, 10% debugging and testing, 10% documentation. Yes, a well designed program requires less time! Most beginners perceive the writing and debugging as more challenging and productive, and so they will rush through the design stage. If you are spending time debugging ladder logic programs you are doing something wrong. Structured design also allows others to verify and modify your programs.

Axiom: Spend as much time on the design of the program as possible. Resist the temptation to implement an incomplete design.
Most control systems are sequential in nature. Sequential systems are often described with words such as mode and behavior. During normal operation these systems will have multiple steps or states of operation. In each operational state the system will behave differently. Typical states include start-up, shut-down, and normal operation. Consider a set of traffic lights - each light pattern constitutes a state. Lights may be green or yellow in one direction and red in the other. The lights change in a predictable sequence. Sometimes traffic lights are equipped with special features such as cross walk buttons that alter the behavior of the lights to give pedestrians time to cross busy roads.

Sequential systems are complex and difficult to design. In the previous chapter timing charts and process sequence bits were discussed as basic design techniques. But, more complex systems require more mature techniques, such as those shown in Figure 10.1. For simpler controllers we can use limited design techniques such as process sequence bits and flow charts. More complex processes, such as traffic lights, will have many states of operation and controllers can be designed using state diagrams. If the control problem involves multiple states of operation, such as one controller for two independent traffic lights, then Petri net or SFC based designs are preferred.

**Figure 10.1  Sequential Design Techniques**

10.2 PROCESS SEQUENCE BITS

A typical machine will use a sequence of repetitive steps that can be clearly identi-
fied. Ladder logic can be written that follows this sequence. The steps for this design method are:

1. Understand the process.
2. Write the steps of operation in sequence and give each step a number.
3. For each step assign a bit.
4. Write the ladder logic to turn the bits on/off as the process moves through its states.
5. Write the ladder logic to perform machine functions for each step.
6. If the process is repetitive, have the last step go back to the first.

Consider the example of a flag raising controller in Figure 10.2 and Figure 10.3. The problem begins with a written description of the process. This is then turned into a set of numbered steps. Each of the numbered steps is then converted to ladder logic.
Description:
A flag raiser that will go up when an up button is pushed, and down when a
down button is pushed, both push buttons are momentary. There are
limit switches at the top and bottom to stop the flag pole. When turned
on at first the flag should be lowered until it is at the bottom of the pole.

Steps:
1. The flag is moving down the pole waiting for the bottom limit switch.
2. The flag is idle at the bottom of the pole waiting for the up button.
3. The flag moves up, waiting for the top limit switch.
4. The flag is idle at the top of the pole waiting for the down button.

Ladder Logic:

```
first scan

This section of ladder logic forces the flag raiser
to start with only one state on, in this case it
should be the first one, step 1.

step 1

step 1  bottom limit switch

The ladder logic for step 1 turns on the motor to
lower the flag and when the bottom limit
switch is hit it goes to step 2.

step 2  flag up button

The ladder logic for step 2 only waits for the
push button to raise the flag.
```

Figure 10.2 A Process Sequence Bit Design Example
The previous method uses latched bits, but the use of latches is sometimes discouraged. A more common method of implementation, without latches, is shown in Figure 10.4.

The ladder logic for step 3 turns on the motor to raise the flag and when the top limit switch is hit it goes to step 4.

The ladder logic for step 4 only waits for the push button to lower the flag.
10.3 TIMING DIAGRAMS

Timing diagrams can be valuable when designing ladder logic for processes that are only dependant on time. The timing diagram is drawn with clear start and stop times. Ladder logic is constructed with timers that are used to turn outputs on and off at appropri-
ate times. The basic method is;

1. Understand the process.
2. Identify the outputs that are time dependant.
3. Draw a timing diagram for the outputs.
4. Assign a timer for each time when an output turns on or off.
5. Write the ladder logic to examine the timer values and turn outputs on or off.

Consider the handicap door opener design in Figure 10.5 that begins with a verbal description. The verbal description is converted to a timing diagram, with t=0 being when the door open button is pushed. On the timing diagram the critical times are 2s, 10s, 14s. The ladder logic is constructed in a careful order. The first item is the latch to seal-in the open button, but shut off after the last door closes. auto is used to turn on the three timers for the critical times. The logic for opening the doors is then written to use the timers.
Description: A handicap door opener has a button that will open two doors. When the button is pushed (momentarily) the first door will start to open immediately, the second door will start to open 2 seconds later. The first door power will stay open for a total of 10 seconds, and the second door power will stay on for 14 seconds. Use a timing diagram to design the ladder logic.

**Timing Diagram:**

![Timing Diagram](image)

**Ladder Logic:**

```
open button T4:2/DN  auto
  auto  
  auto  

  T4:0/DN  
  Timer T4:0  
  Delay 2s  

  T4:1/DN  
  Timer T4:1  
  Delay 10s  

  T4:2/DN  
  Timer T4:2  
  Delay 14s  

T4:1/TI  
 door 1  

T4:2/TI  
 door 2  

T4:0/DN
```

*Figure 10.5*  Design With a Timing Diagram
10.4 DESIGN CASES

10.5 SUMMARY

- Timing diagrams can show how a system changes over time.
- Process sequence bits can be used to design a process that changes over time.
- Timing diagrams can be used for systems with a time driven performance.

10.6 PRACTICE PROBLEMS

1. Write ladder logic that will give the following timing diagram for $B$ after input $A$ is pushed. After $A$ is pushed any changes in the state of $A$ will be ignored.

2. Design ladder logic for the timing diagram below. When an input $A$ becomes active the sequence should start.

3. A wrapping process is to be controlled with a PLC. The general sequence of operations is described below. Develop the ladder logic using process sequence bits.
   1. The folder is idle until a part arrives.
   2. When a part arrives it triggers the part sensor and the part is held in place by actuating the hold actuator.
3. The first wrap is done by turning on output paper for 1 second.
4. The paper is then folded by turning on the crease output for 0.5 seconds.
5. An adhesive is applied by turning on output tape for 0.75 seconds.
6. The part is release by turning off output hold.
7. The process pauses until the part sensors goes off, and then the machine returns to idle.

10.7 PRACTICE PROBLEM SOLUTIONS

1.
2.

A

T4:0/EN

stop

TON
T4:0
0.100 s

TON
T4:1
0.300 s

TON
T4:2
0.500 s

TON
T4:3
0.700 s

TON
T4:4
0.900 s

TON
T4:5
1.100 s

TON
T4:6
1.900 s

X

Y

Z

T4:0/TT
T4:2/DN
T4:0/DN
T4:2/DN
T4:4/DN
T4:5/TT

T4:6/DN
T4:1/DN
T4:3/DN
T4:5/DN
3.

(for both solutions)

- step2
- step3
- step4
- step5
- step2
- step3
- step4
- hold
- paper
- crease
- tape
(without latches)

1. **Step 1**
   - Part
   - Stop

2. **Step 2**
   - Part
   - T4:0/DN
   - Stop

3. **Step 3**
   - T4:0/DN
   - T4:1/DN
   - Stop

4. **Step 4**
   - T4:1/DN
   - T4:2/DN
   - Stop

5. **Step 5**
   - T4:2/DN
   - Part
   - Stop
10.8 ASSIGNMENT PROBLEMS

1. Convert the following timing diagram to ladder logic. It should begin when input ‘A’ becomes
true.

2. Use the timing diagram below to design ladder logic. The sequence should start when input X turns on. X may only be on momentarily, but the sequence should continue to execute until it ends at 26 seconds.

3. Use the timing diagram below to design ladder logic. The sequence should start when input X turns on. X may only be on momentarily, but the sequence should execute anyway.

4. Write a program that will execute the following steps. When in steps b) or d), output C will be true. Output X will be true when in step c).
   a) Start in an idle state. If input G becomes true go to b)
   b) Wait until P becomes true before going to step c).
   c) Wait for 3 seconds then go to step d).
   d) Wait for P to become false, and then go to step b).

5. Write a program that will execute the following steps. When in steps b) or d), output C will be true. Output X will be true when in step c).
a) Start in an idle state. If input G becomes true go to b)
b) Wait until P becomes true before going to step c). If input S becomes true then go to step a).
c) Wait for 3 seconds then go to step d).
d) Wait for P to become false, and then go to step b).

6. A PLC is to control an amusement park water ride. The ride will fill a tank of water and splash a tour group. 10 seconds later a water jet will be ejected at another point. Develop ladder logic for the process that follows the steps listed below.
   1. The process starts in ‘idle’.
   2. The ‘cart_detecl opens the ‘filling’ valve.
   3. After a delay of 30 seconds from the start of the filling of the tank the tank ‘outlet’ valve opens. When the tank is ‘full’ the ‘filling’ valve closes.
   4. When the tank is empty the ‘outlet’ valve is closed.
   5. After a 10 second delay, from the tank outlet valve opening, a water ‘jet’ is opened.
   6. After ‘2’ seconds the water ‘jet’ is closed and the process returns to the ‘idle state’.

7. Write a ladder logic program to extend and retract a cylinder after a start button is pushed. There are limit switches at the ends of travel. If the cylinder is extending if more than 5 seconds the machine should shut down and turn on a fault light. If it is retracting for more than 3 seconds it should also shut down and turn on the fault light. It can be reset with a reset button.

8. Design a program with sequence bits for a hydraulic press that will advance when two palm buttons are pushed. Top and bottom limit switches are used to reverse the advance and stop after a retract. At any time the hands removed from the palm button will stop an advance and retract the press. Include start and stop buttons to put the press in and out of an active mode.

9. A machine has been built for filling barrels. Use process sequence bits to design ladder logic for the sequential process as described below.
   1. The process begins in an idle state.
   2. If the ‘fluid_pressure’ and ‘barrel_present’ inputs are on, the system will open a flow valve for 2 seconds with output ‘flow’.
   3. The ‘flow’ valve will then be turned off for 10 seconds.
   4. The ‘flow’ valve will then be turned on until the ‘full’ sensor indicates the barrel is full.
   5. The system will wait until the ‘barrel_present’ sensor goes off before going to the idle state.

10. Design ladder logic for an oven using process sequence bits. (Note: the solution will only be graded if the process sequence bit method is used.) The operations are as listed below.
    1. The oven begins in an IDLE state.
    2. An operator presses a start button and an ALARM output is turned on for 1 minute.
    3. The ALARM output is turned off and the HEAT is turned on for 3 minutes to allow the temperature to rise to the acceptable range.
    4. The CONVEYOR output is turned on.
    5. If the STOP input is activated (turned off) the HEAT will be turned off, but the CONVEYOR output will be kept on for two minutes. After this the oven returns to IDLE.
11. We are developing a safety system (using a PLC-5) for a large industrial press. The press is activated by turning on the compressor power relay (R, connected to O:013/05). After R has been on for 30 seconds the press can be activated to move (P connected to O:013/06). The delay is needed for pressure to build up. After the press has been activated (with P for 1.0 seconds) the system must be shut down (R and P off), and then the cycle may begin again. For safety, there is a sensor that detects when a worker is inside the press (S, connected to I:011/02), which must be off before the press can be activated. There is also a button that must be pushed 5 times (B, connected to I:011/01) before the press cycle can begin. If at any time the worker enters the press (and S becomes active) the press will be shut down (P and R turned off). Develop the process sequence and sequence bits, and then ladder logic for the states. State all assumptions, and show all work.

12. A machine is being designed to wrap boxes of chocolate. The boxes arrive at the machine on a conveyor belt. The list below shows the process steps in sequence.

1. The box arrives and is detected by an optical sensor (P), after this the conveyor is stopped (C) and the box is clamped in place (H).
2. A wrapping mechanism (W) is turned on for 2 seconds.
3. A sticker cylinder (S) is turned on for 1 second to put consumer labelling on the box.
4. The clamp (H) is turned off and the conveyor (C) is turned on.
5. After the box leaves the system returns to an idle state.

Develop ladder logic programs for the system using the following methods. Don’t forget to include regular start and stop inputs.

i) a timing diagram
ii) process sequence bits
11. FLOWCHART BASED DESIGN

Topics:
- Describing process control using flowcharts
- Conversion of flowcharts to ladder logic

Objectives:
- Be able to describe a process with a flowchart.
- Be able to convert a flowchart to ladder logic.

11.1 INTRODUCTION

A flowchart is ideal for a process that has sequential process steps. The steps will be executed in a simple order that may change as the result of some simple decisions. The symbols used for flowcharts are shown in Figure 11.1. These blocks are connected using arrows to indicate the sequence of the steps. The different blocks imply different types of program actions. Programs always need a start block, but PLC programs rarely stop so the stop block is rarely used. Other important blocks include operations and decisions. The other functions may be used but are not necessary for most PLC applications.

![Flowchart Symbols](image-url)

*Figure 11.1 Flowchart Symbols*
A flowchart is shown in Figure 11.2 for a control system for a large water tank. When a start button is pushed the tank will start to fill, and the flow out will be stopped. When full, or the stop button is pushed the outlet will open up, and the flow in will be stopped. In the flowchart the general flow of execution starts at the top. The first operation is to open the outlet valve and close the inlet valve. Next, a single decision block is used to wait for a button to be pushed. When the button is pushed the yes branch is followed and the inlet valve is opened, and the outlet valve is closed. Then the flow chart goes into a loop that uses two decision blocks to wait until the tank is full, or the stop button is pushed. If either case occurs the inlet valve is closed and the outlet valve is opened. The system then goes back to wait for the start button to be pushed again. When the controller is on the program should always be running, so only a start block is needed. Many beginners will neglect to put in checks for stop buttons.
Figure 11.2 A Flowchart for a Tank Filler

The general method for constructing flowcharts is:

1. Understand the process.
2. Determine the major actions, these are drawn as blocks.
3. Determine the sequences of operations, these are drawn with arrows.
4. When the sequence may change use decision blocks for branching.

Once a flowchart has been created ladder logic can be written. There are two basic techniques that can be used, the first presented uses blocks of ladder logic code. The second uses normal ladder logic.

11.2 BLOCK LOGIC

The first step is to name each block in the flowchart, as shown in Figure 11.3. Each of the numbered steps will then be converted to ladder logic
STEP 1: Add labels to each block in the flowchart

![Flowchart Diagram]

Each block in the flowchart will be converted to a block of ladder logic. To do this we will use the MCR (Master Control Relay) instruction (it will be discussed in more detail later.) The instruction is shown in Figure 11.4, and will appear as a matched pair of outputs labelled MCR. If the first MCR line is true then the ladder logic on the following lines will be scanned as normal to the second MCR. If the first line is false the lines to the
next MCR block will all be forced off. If a normal output is used inside an MCR block, it may be forced off. Therefore latches will be used in this method.

Note: We will use MCR instructions to implement some of the state based programs. This allows us to switch off part of the ladder logic. The one significant note to remember is that any normal outputs (not latches and timers) will be FORCED OFF. Unless this is what you want, put the normal outputs outside MCR blocks.

If A is true then the MCR will cause the ladder in between to be executed. If A is false the outputs are forced off.

*Figure 11.4* The MCR Function

The first part of the ladder logic required will reset the logic to an initial condition, as shown in Figure 11.5. The line will only be true for the first scan of the PLC, and at that time it will turn on the flowchart block \( F1 \) which is the *reset all values off* operation. All other operations will be turned off.
STEP 2: Write ladder logic to force the PLC into the first state

The ladder logic for the first state is shown in Figure 11.6. When $F1$ is true the logic between the MCR lines will be scanned, if $F1$ is false the logic will be ignored. This logic turns on the outlet valve and turns off the inlet valve. It then turns off operation $F1$, and turns on the next operation $F2$. 

*Figure 11.5  Initial Reset of States*
STEP 3: Write ladder logic for each function in the flowchart

Figure 11.6  Ladder Logic for the Operation F1

The ladder logic for operation F2 is simple, and when the start button is pushed, it will turn off F2 and turn on F3. The ladder logic for operation F3 opens the inlet valve and moves to operation F4.
The ladder logic for operation $F_4$ turns off $F_4$, and if the tank is full it turns on $F_6$, otherwise $F_5$ is turned on. The ladder logic for operation $F_5$ is very similar.
The ladder logic for operation $F6$ turns the outlet valve on and turns off the inlet valve. It then ends operation $F6$ and returns to operation $F2$. 

*Figure 11.8*  Ladder Logic for Operations $F4$ and $F5*
In general there is a preference for methods that do not use MCR statements or latches. The flowchart used in the previous example can be implemented without these instructions using the following method. The first step to this process is shown in Figure 11.10. As before each of the blocks in the flowchart are labelled, but now the connecting arrows (transitions) in the diagram must also be labelled. These transitions indicate when another function block will be activated.
Figure 11.10  Label the Flowchart Blocks and Arrows

The first section of ladder logic is shown in Figure 11.11. This indicates when the transitions between functions should occur. All of the logic for the transitions should be kept together, and appear before the state logic that follows in Figure 11.12.
The logic shown in Figure 11.12 will keep a function on, or switch to the next function. Consider the first ladder rung for $F_1$, it will be turned on by transition $T_1$ and once function $F_1$ is on it will keep itself on, unless $T_2$ occurs shutting it off. If $T_2$ has occurred the next line of ladder logic will turn on $F_2$. The function logic is followed by output logic that relates output values to the active functions.

*Figure 11.11  The Transition Logic*
Figure 11.12  The Function Logic and Outputs
11.4 SUMMARY

- Flowcharts are suited to processes with a single flow of execution.
- Flowcharts are suited to processes with clear sequences of operation.

11.5 PRACTICE PROBLEMS

1. Convert the following flow chart to ladder logic.

2. Draw a flow chart for cutting the grass, then develop ladder logic for three of the actions/decisions.

3. Design a garage door controller using a flowchart. The behavior of the garage door controller is as follows,
   - there is a single button in the garage, and a single button remote control.
   - when the button is pushed the door will move up or down.
   - if the button is pushed once while moving, the door will stop, a second push will start motion again in the opposite direction.
   - there are top/bottom limit switches to stop the motion of the door.
   - there is a light beam across the bottom of the door. If the beam is cut while the door is closing the door will stop and reverse.
   - there is a garage light that will be on for 5 minutes after the door opens or closes.
11.6 PRACTICE PROBLEM SOLUTIONS

1.

```
11.6 PRACTICE PROBLEM SOLUTIONS

1. start
   - A on?
     - yes
     - no
       - yes
       - no
       - is C on?
         - yes
         - no
         - F4
   - F1
   - MCR
   - F2
   - F3
   - F4
   - MCR
   - F2
   - B
   - F2
   - MCR
   - F3
   - MCR
   - F3
   - MCR
   - F4
   - MCR
   - C
   - MCR
   - F4
   - MCR
   - F1
   - MCR
   - F4
   - MCR
   - F2
   - MCR

```

```
start

F1
A on

F2
is B on?
yes

F3
A off

no

F4
is C on?
yes

```

```
2.

Start

Get mower and gas can

F2
Is gas can empty?

yes → F3 → get gas

no → F4

Fill mower

F5
Pull cord

F6
Is Mower on?

no

yes → F7

Push Mower

F8
Is all lawn cut?

no

yes → F9

Stop mower

F10
Put gas and mower away
F3 MCR
  gas can full
  fill gas tank
  F4
  F3
F4 MCR
F5 MCR
  T4:0/DN
  pour gas
  T4:0/DN
  F5
  F4
F5 MCR
  cord pulled
  pull cord
  cord pulled
  F6
  F5
F6 MCR
  mower on
  F7
  F5
  F6
  MCR
ETC........................
3.

start

is remote or button pushed?

no

yes

ST2 turn on door close

is remote or button or bottom limit pushed?

no

yes

ST4 is light beam on?

yes

no

ST5 turn off door close

ST6 is remote or button pushed?

yes

ST7 turn on door open

is remote or button or top limit pushed?

no

yes

ST8

ST9 turn off door open
first scan

- ST1
- ST2
- ST3
- ST4
- ST5
- ST6
- ST7
- ST8
- ST9
- door open
- door close

- TOF
  - T4:0
  - preset 300s

- garage light
11.7 ASSIGNMENT PROBLEMS

1. Develop ladder logic for the flowchart below.

2. Use a flow chart to design a parking gate controller.

- the gate will be raised by one output and lowered by another. If the gate gets stuck an over current detector will make a PLC input true. If this is the case the gate should reverse and the light should be turned on indefinitely.
- if a valid keycard is entered a PLC input will be true. The gate is to rise and stay open for 10 seconds.
- when a car is over the car detector a PLC input will go true. The gate is to open while this detector is active. If it is active for more that 30 seconds the light should also turn on until the gate closes.
3. A welding station is controlled by a PLC. On the outside is a safety cage that must be closed while the cell is active. A belt moves the parts into the welding station and back out. An inductive proximity sensor detects when a part is in place for welding, and the belt is stopped. To weld, an actuator is turned on for 3 seconds. As normal the cell has start and stop push buttons.
   a) Draw a flow chart
   b) Implement the chart in ladder logic

4. Convert the following flowchart to ladder logic.

5. A machine is being designed to wrap boxes of chocolate. The boxes arrive at the machine on a conveyor belt. The list below shows the process steps in sequence.
   1. The box arrives and is detected by an optical sensor (P), after this the conveyor is stopped (C) and the box is clamped in place (H).
   2. A wrapping mechanism (W) is turned on for 2 seconds.
   3. A sticker cylinder (S) is turned on for 1 second to put consumer labelling on the
The clamp (H) is turned off and the conveyor (C) is turned on. After the box leaves the system returns to an idle state.

Develop ladder logic for the system using a flowchart. Don’t forget to include regular start and stop inputs.
12. STATE BASED DESIGN

Topics:
- Describing process control using state diagrams
- Conversion of state diagrams to ladder logic
- MCR blocks

Objectives:
- Be able to construct state diagrams for a process.
- Be able to convert a state diagram to ladder logic directly.
- Be able to convert state diagrams to ladder logic using equations.

12.1 INTRODUCTION

A system state is a mode of operation. Consider a bank machine that will go through very carefully selected states. The general sequence of states might be idle, scan card, get secret number, select transaction type, ask for amount of cash, count cash, deliver cash/return card, then idle.

A State based system can be described with system states, and the transitions between those states. A state diagram is shown in Figure 12.1. The diagram has two states, State 1 and State 2. If the system is in state 1 and A happens the system will then go into state 2, otherwise it will remain in State 1. Likewise if the system is in state 2, and B happens the system will return to state 1. As shown in the figure this state diagram could be used for an automatic light controller. When the power is turned on the system will go into the lights off state. If motion is detected or an on push button is pushed the system will go to the lights on state. If the system is in the lights on state and 1 hour has passed, or an off pushbutton is pushed then the system will go to the lights off state. The else statements are omitted on the second diagram, but they are implied.
The most essential part of creating state diagrams is identifying states. Some key questions to ask are,

1. Consider the system,
   - What does the system do normally?
   - Does the system behavior change?
   - Can something change how the system behaves?
   - Is there a sequence to actions?

2. List modes of operation where the system is doing one identifiable activity that will start and stop. Keep in mind that some activities may just be to wait.

Consider the design of a coffee vending machine. The first step requires the identification of vending machine states as shown in Figure 12.2. The main state is the idle state. There is an inserting coins state where the total can be displayed. When enough coins have been inserted the user may select their drink of choice. After this the make coffee state will
be active while coffee is being brewed. If an error is detected the service needed state will be activated.

**STATES**

- **idle** - the machine has no coins and is doing nothing
- **inserting coins** - coins have been entered and the total is displayed
- **user choose** - enough money has been entered and the user is making coffee selection
- **make coffee** - the selected type is being made
- **service needed** - the machine is out of coffee, cups, or another error has occurred

**Notes:**
1. These states can be subjective, and different designers might pick others.
2. The states are highly specific to the machine.
3. The previous/next states are not part of the states.
4. There is a clean difference between states.

*Figure 12.2*  Definition of Vending Machine States

The states are then drawn in a state diagram as shown in Figure 12.3. Transitions are added as needed between the states. Here we can see that when powered up the machine will start in an idle state. The transitions here are based on the inputs and sensors in the vending machine. The state diagram is quite subjective, and complex diagrams will differ from design to design. These diagrams also expose the controller behavior. Consider that if the machine needs maintenance, and it is unplugged and plugged back in, the service needed statement would not be reentered until the next customer paid for but did not receive their coffee. In a commercial design we would want to fix this oversight.
12.1.1 State Diagram Example

Consider the traffic lights in Figure 12.4. The normal sequences for traffic lights are a green light in one direction for a long period of time, typically 10 or more seconds. This is followed by a brief yellow light, typically 4 seconds. This is then followed by a similar light pattern in the other direction. It is understood that a green or yellow light in one direction implies a red light in the other direction. Pedestrian buttons are provided so that when pedestrians are present a cross walk light can be turned on and the duration of the green light increased.

Figure 12.3 State Diagram for a Coffee Machine
The first step for developing a controller is to define the inputs and outputs of the system as shown in Figure 12.5. First we will describe the system variables. These will vary as the system moves from state to state. Please note that some of these together can define a state (alone they are not the states). The inputs are used when defining the transitions. The outputs can be used to define the system state.

We have eight items that are ON or OFF

A simple diagram can be drawn to show sequences for the lights

Note that each state will lead to a different set of outputs. The inputs are often part, or all of the transitions.
Previously state diagrams were used to define the system, it is possible to use a state table as shown in Figure 12.6. Here the light sequences are listed in order. Each state is given a name to ease interpretation, but the corresponding output pattern is also given. The system state is defined as the bit pattern of the 6 lights. Note that there are only 4 patterns, but 6 binary bits could give as many as 64.

**Step 1: Define the System States and put them (roughly) in sequence**

<table>
<thead>
<tr>
<th>State Description</th>
<th>#</th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
<th>L4</th>
<th>L5</th>
<th>L6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Green East/West</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Yellow East/West</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Green North/South</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Yellow North/South</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

A binary number
0 = light off
1 = light on

Here the four states determine how the 6 outputs are switched on/off.

Figure 12.6 System State Table for Traffic Lights

Transitions can be added to the state table to clarify the operation, as shown in Figure 12.7. Here the transition from Green E/W to Yellow E/W is S1. What this means is that a cross walk button must be pushed to end the green light. This is not normal, normally the lights would use a delay. The transition from Yellow E/W to Green N/S is caused by a 4 second delay (this is normal.) The next transition is also abnormal, requiring that the cross walk button be pushed to end the Green N/S state. The last state has a 4 second delay before returning to the first state in the table. In this state table the sequence will always be the same, but the times will vary for the green lights.
Step 2: Define State Transition Triggers, and add them to the list of states

<table>
<thead>
<tr>
<th>Description</th>
<th>#</th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
<th>L4</th>
<th>L5</th>
<th>L6</th>
<th>transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Green East/West</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>S1</td>
</tr>
<tr>
<td>Yellow East/West</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>delay 4 sec</td>
</tr>
<tr>
<td>Green North/South</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>delay 4 sec</td>
</tr>
<tr>
<td>Yellow North/South</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 12.7  State Table with Transitions*

A state diagram for the system is shown in Figure 12.8. This diagram is equivalent to the state table in Figure 12.7, but it can be valuable for doing visual inspection.

*Figure 12.8  A Traffic Light State Diagram*

### 12.1.2 Conversion to Ladder Logic

#### 12.1.2.1 - Block Logic Conversion
State diagrams can be converted directly to ladder logic using block logic. This technique will produce larger programs, but it is a simple method to understand, and easy to debug. The previous traffic light example is to be implemented in ladder logic. The inputs and outputs are defined in Figure 12.9, assuming it will be implemented on an Allen Bradley Micrologix. *first scan* is the address of the first scan in the PLC. The locations B3/1 to B3/4 are internal memory locations that will be used to track which states are on. The behave like outputs, but are not available for connection outside the PLC. The input and output values are determined by the PLC layout.

<table>
<thead>
<tr>
<th>STATES</th>
<th>OUTPUTS</th>
<th>INPUTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>B3/1 - state 1 - green E/W</td>
<td>O/1 - L1</td>
<td>I/1 - S1</td>
</tr>
<tr>
<td>B3/2 - state 2 - yellow E/W</td>
<td>O/2 - L2</td>
<td>I/2 - S2</td>
</tr>
<tr>
<td>B3/3 - state 3 - green N/S</td>
<td>O/3 - L3</td>
<td>S2:1/14 - first scan</td>
</tr>
<tr>
<td>B3/4 - state 4 - yellow N/S</td>
<td>O/4 - L4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>O/5 - L5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>O/6 - L6</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 12.9* Inputs and Outputs for Traffic Light Controller

The initial ladder logic block shown in Figure 12.10 will initialize the states of the PLC, so that only state 1 is on. The first scan indicator *first scan* will execute the MCR block when the PLC is first turned on, and the latches will turn on the value for state 1 *B3/1* and turn off the others.
RESET THE STATES

Figure 12.10  Ladder Logic to Initialize Traffic Light Controller

Note: We will use MCR instructions to implement some of the state based programs. This allows us to switch off part of the ladder logic. The one significant note to remember is that any normal outputs (not latches and timers) will be FORCED OFF. Unless this is what you want, put the normal outputs outside MCR blocks.

The next section of ladder logic only deals with outputs. For example the output O/1 is the N/S red light, which will be on for states 1 and 2, or B3/1 and B3/2 respectively. Putting normal outputs outside the MCR blocks is important. If they were inside the
blocks they could only be on when the MCR block was active, otherwise they would be forced off. Note: Many beginners will make the careless mistake of repeating outputs in this section of the program.

**TURN ON LIGHTS AS REQUIRED**

![Diagram](image)

*Figure 12.11  General Output Control Logic*

The first state is implemented in Figure 12.10. If state 1 is active this will be active. The transition is S1 or I/1 which will end state 1 B3/1 and start state 2 B3/2.
The second state is more complex because it involves a time delay, as shown in Figure 12.13. When the state is active the RTO timer will be timing. When the timer is done state 2 will be unlatched, and state 3 will be latched on. The timer is retentive, so it must also be reset when the state is done, so that it will start at zero the next time the state starts.
Figure 12.13  Ladder Logic for Second State

The third and fourth states are shown in Figure 12.14 and Figure 12.15. Their layout is very similar to that of the first two states.
THIRD STATE WAIT FOR TRANSITIONS

B3/3
\[\text{MCR}\]

I/2

B3/3
\[\text{U}\]

I/2

B3/4
\[\text{L}\]

MCR

Figure 12.14  Ladder Logic for State Three

FOURTH STATE WAIT FOR TRANSITIONS

B3/4
\[\text{MCR}\]

T4:2/DN
\[\text{RTO}\]
delay 4s

T4:2/DN

B3/4
\[\text{U}\]

T4:2/DN

B3/1
\[\text{L}\]

T4:2/DN

T4:2
\[\text{RST}\]

MCR

Figure 12.15  Ladder Logic for State Four
The previous example only had one path through the state tables, so there was never a choice between states. The state diagram in Figure 12.16 could potentially have problems if two transitions occur simultaneously. For example if state STB is active and A and C occur simultaneously, the system could go to either STA or STC (or both in a poorly written program.) To resolve this problem we should choose one of the two transitions as having a higher priority, meaning that it should be chosen over the other transition. This decision will normally be clear, but if not an arbitrary decision is still needed.

*Figure 12.16* A State Diagram with Priority Problems

The state diagram in Figure 12.16 is implemented with ladder logic in Figure 12.17 and Figure 12.18. The implementation is the same as described before, but for state STB additional ladder logic is added to disable transition A if transition C is active, therefore giving priority to C.
Note: if A and C are true at the same time then C will have priority. PRIORITIZATION is important when simultaneous branches are possible.
The Block Logic technique described does not require any special knowledge and the programs can be written directly from the state diagram. The final programs can be easily modified, and finding problems is easier. But, these programs are much larger and less efficient.

### 12.1.2.2 - State Equations

State diagrams can be converted to Boolean equations and then to Ladder Logic. The first technique that will be described is state equations. These equations contain three main parts, as shown below in Figure 12.19. To describe them simply - a state will be on if it is already on, or if it has been turned on by a transition from another state, but it will be turned off if there was a transition to another state. An equation is required for each state in the state diagram.
The state equation method can be applied to the traffic light example in Figure 12.8. The first step in the process is to define variable names (or PLC memory locations) to keep track of which states are on or off. Next, the state diagram is examined, one state at a time. The first equation if for ST1, or state 1 - green NS. The start of the equation can be read as ST1 will be on if it is on, or if ST4 is on, and it has been on for 4s, or if it is the first scan of the PLC. The end of the equation can be read as ST1 will be turned off if it is on, but S1 has been pushed and S2 is off. As discussed before, the first half of the equation will turn the state on, but the second half will turn it off. The first scan is also used to turn on ST1 when the PLC starts. It is put outside the terms to force ST1 on, even if the exit conditions are true.

Informally,

State X = (State X + just arrived from another state) and has not left for another state

Formally,

$STATE_i = \left( STATE_i + \sum_{j=1}^{n} (T_{j,i} \cdot STATE_j) \right) \cdot \prod_{k=1}^{m} (T_{i,k} \cdot STATE_i)$

where, $STATE_i = \text{A variable that will reflect if state i is on}$
$n = \text{the number of transitions to state i}$
$m = \text{the number of transitions out of state i}$
$T_{j,i} = \text{The logical condition of a transition from state j to i}$
$T_{i,k} = \text{The logical condition of a transition out of state i to k}$

Figure 12.19  State Equations
Defined state variables:

\( ST1 = \) state 1 - green NS
\( ST2 = \) state 2 - yellow NS
\( ST3 = \) state 3 - green EW
\( ST4 = \) state 4 - yellow EW

The state entrance and exit condition equations:

\[
ST1 = (ST1 + ST4 \cdot TON_2(ST4, 4s)) \cdot \overline{ST1} \cdot S1 \cdot S2 + FS
\]
\[
ST2 = (ST2 + ST1 \cdot S1 \cdot \overline{S2}) \cdot \overline{ST2} \cdot TON_1(ST2, 4s)
\]
\[
ST3 = (ST3 + ST2 \cdot TON_1(ST2, 4s)) \cdot \overline{ST3} \cdot \overline{S1} \cdot S2
\]
\[
ST4 = (ST4 + ST3 \cdot \overline{S1} \cdot S2) \cdot \overline{ST4} \cdot TON_2(ST4, 4s)
\]

Note: Timers are represented in these equations in the form \( TON_i(A, \text{delay}) \). \( TON \) indicates that it is an on-delay timer, \( A \) is the input to the timer, and \( \text{delay} \) is the timer delay value. The subscript \( i \) is used to differentiate timers.

*Figure 12.20  State Equations for the Traffic Light Example*

The equations in Figure 12.20 cannot be implemented in ladder logic because of the NOT over the last terms. The equations are simplified in Figure 12.21 so that all NOT operators are only over a single variable.
Now, simplify these for implementation in ladder logic.

\[
ST1 = (ST1 + ST4 \cdot TON_2(ST4, 4)) \cdot (\overline{ST1} + \overline{S1} + S2) + FS
\]

\[
ST2 = (ST2 + ST1 \cdot S1 \cdot \overline{S2}) \cdot (\overline{ST2} + \overline{TON_1(ST2, 4)})
\]

\[
ST3 = (ST3 + ST2 \cdot TON_1(ST2, 4)) \cdot (\overline{ST3} + S1 + \overline{S2})
\]

\[
ST4 = (ST4 + ST3 \cdot \overline{S1} \cdot S2) \cdot (\overline{ST4} + \overline{TON_2(ST4, 4)})
\]

Figure 12.21  Simplified Boolean Equations

These equations are then converted to the ladder logic shown in Figure 12.22 and Figure 12.23. At the top of the program the two timers are defined. (Note: it is tempting to combine the timers, but it is better to keep them separate.) Next, the Boolean state equations are implemented in ladder logic. After this we use the states to turn specific lights on.
Figure 12.22 Ladder Logic for the State Equations
OUTPUT LOGIC FOR THE LIGHTS

This method will provide the most compact code of all techniques, but there are potential problems. Consider the example in Figure 12.23. If push button $S1$ has been pushed the line for ST1 should turn off, and the line for ST2 should turn on. But, the line for ST2 depends upon the value for ST1 that has just been turned off. This will cause a problem if the value of ST1 goes off immediately after the line of ladder logic has been scanned. In effect the PLC will get lost and none of the states will be on. This problem arises because the equations are normally calculated in parallel, and then all values are updated simultaneously. To overcome this problem the ladder logic could be modified to the form shown in Figure 12.24. Here some temporary variables are used to hold the new state values. After all the equations are solved the states are updated to their new values.

Figure 12.23  Ladder Logic for the State Equations
Figure 12.24  Delayed State Updating

When multiple transitions out of a state exist we must take care to add priorities.
Each of the alternate transitions out of a state should be given a priority, from highest to lowest. The state equations can then be written to suppress transitions of lower priority when one or more occur simultaneously. The state diagram in Figure 12.25 has two transitions $A$ and $C$ that could occur simultaneously. The equations have been written to give $A$ a higher priority. When $A$ occurs, it will block $C$ in the equation for $STC$. These equations have been converted to ladder logic in Figure 12.26.

$$STA = (STA + STB \cdot A) \cdot \overline{STA} \cdot \overline{B}$$
$$STB = (STB + STA \cdot B + STC \cdot D) \cdot \overline{STB} \cdot A \cdot \overline{STB} \cdot \overline{C} + FS$$
$$STC = (STC + STB \cdot C \cdot \overline{A}) \cdot \overline{STC} \cdot \overline{D}$$

*Figure 12.25 State Equations with Prioritization*
Figure 12.26  Ladder Logic with Prioritization

12.1.2.3 - State-Transition Equations
A state diagram may be converted to equations by writing an equation for each state and each transition. A sample set of equations is seen in Figure 12.27 for the traffic light example of Figure 12.8. Each state and transition needs to be assigned a unique variable name. (Note: It is a good idea to note these on the diagram) These are then used to write the equations for the diagram. The transition equations are written by looking at the each state, and then determining which transitions will end that state. For example, if ST1 is true, and crosswalk button $S_1$ is pushed, and $S_2$ is not, then transition $T_1$ will be true. The state equations are similar to the state equations in the previous State Equation method, except they now only refer to the transitions. Recall, the basic form of these equations is that the state will be on if it is already on, or it has been turned on by a transition. The state will be turned off if an exiting transition occurs. In this example the first scan was given it’s own transition, but it could have also been put into the equation for $T_4$.

defined state and transition variables:

- $ST_1$ = state 1 - green NS
- $ST_2$ = state 2 - yellow NS
- $ST_3$ = state 3 - green EW
- $ST_4$ = state 4 - yellow EW

state and transition equations:

- $T_1 = ST_1 \cdot S_1 \cdot \overline{S_2}$
- $T_2 = ST_2 \cdot TON_1(ST_2, 4)$
- $T_3 = ST_3 \cdot \overline{S_1} \cdot S_2$
- $T_4 = ST_4 \cdot TON_2(ST_4, 4)$
- $T_5 = FS$
- $ST_1 = (ST_1 + T_4 + T_5) \cdot \overline{T_1}$
- $ST_2 = (ST_2 + T_1) \cdot \overline{T_2}$
- $ST_3 = (ST_3 + T_2) \cdot \overline{T_3}$
- $ST_4 = (ST_4 + T_3) \cdot \overline{T_4}$

$Figure\ 12.27\ \ State\text{-}Transition\ Equations$

These equations can be converted directly to the ladder logic in Figure 12.28, Figure 12.29 and Figure 12.30. It is very important that the transition equations all occur before the state equations. By updating the transition equations first and then updating the state equations the problem of state variable values changing is negated - recall this problem was discussed in the State Equations section.
Figure 12.28  Ladder Logic for the State-Transition Equations
Figure 12.29  Ladder Logic for the State-Transition Equations
Figure 12.30  Ladder Logic for the State-Transition Equations

The problem of prioritization also occurs with the State-Transition equations. Equations were written for the State Diagram in Figure 12.31. The problem will occur if transitions $A$ and $C$ occur simultaneously. In the example transition $T_2$ is given a higher priority, and if it is true, then the transition $T_3$ will be suppressed when calculating $STC$. In this example the transitions have been considered in the state update equations, but they can also be used in the transition equations.
12.2 SUMMARY

- State diagrams are suited to processes with a single flow of execution.
- State diagrams are suited to problems that has clearly defines modes of execution.
- Controller diagrams can be converted to ladder logic using MCR blocks
- State diagrams can also be converted to ladder logic using equations
- The sequence of operations is important when converting state diagrams to ladder logic.

12.3 PRACTICE PROBLEMS

1. Draw a state diagram for a microwave oven.
2. Convert the following state diagram to equations.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>P</td>
</tr>
<tr>
<td>B</td>
<td>Q</td>
</tr>
<tr>
<td>C</td>
<td>R</td>
</tr>
<tr>
<td>D</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>state</th>
<th>P</th>
<th>Q</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>S1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>S2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

3. Implement the following state diagram with equations.
4. Given the following state diagram, use equations to implement ladder logic.

![State Diagram](image)

5. Convert the following state diagram to logic using equations.

![State Diagram](image)

6. You have been asked to program a PLC-5 that is controlling a handicapped access door opener. The client has provided the electrical wiring diagram below to show how the PLC inputs and outputs have been wired. Button A is located inside and button B is located outside. When either button is pushed the motor will be turned on to open the door. The motor is to be kept on for a total of 15 seconds to allow the person to enter. After the motor is turned off the door will fall closed. In the event that somebody gets caught in the door the thermal relay will go off, and the motor should be turned off. After 20,000 cycles the door should stop working and the light
should go on to indicate that maintenance is required.
a) Develop a state diagram for the control of the door.
b) Convert the state diagram to ladder logic. (list the input and the output addresses first)
c) Convert the state diagram to Boolean equations.

7. Design a garage door controller using a) block logic, and b) state-transition equations. The behavior of the garage door controller is as follows,
   - there is a single button in the garage, and a single button remote control.
   - when the button is pushed the door will move up or down.
   - if the button is pushed once while moving, the door will stop, a second push will start motion again in the opposite direction.
   - there are top/bottom limit switches to stop the motion of the door.
   - there is a light beam across the bottom of the door. If the beam is cut while the door is closing the door will stop and reverse.
   - there is a garage light that will be on for 5 minutes after the door opens or closes.
12.4 PRACTICE PROBLEM SOLUTIONS

1. 

2. 

\[ T_1 = FS \]
\[ T_2 = S_1(B\overline{A}) \]
\[ T_3 = S_2(E(C + D + F)) \]
\[ T_4 = S_1(\overline{F} + \overline{E}) \]
\[ T_5 = S_0(A(C + \overline{D})) \]

\[ P = S_1 + S_2 \]
\[ Q = S_0 + S_2 \]
\[ R = S_0 + S_1 \]
3.

\[
T1 = ST1 \cdot A \\
T2 = ST2 \cdot B \\
T3 = ST1 \cdot C \\
T4 = ST3 \cdot D \\
T5 = ST1 \cdot E \\
T6 = ST4 \cdot F
\]

\[
ST1 = (ST1 + T2 + T4 + T6) \cdot \overline{T1} \cdot \overline{T3} \cdot \overline{T5} \\
ST2 = (ST2 + T1 \cdot \overline{T3} \cdot \overline{T5}) \cdot \overline{T2} \\
ST3 = (ST3 + T3 \cdot \overline{T5}) \cdot \overline{T4} \\
ST4 = (ST4 + T5 + FS) \cdot \overline{T6}
\]
4.

FS = first scan
T1 = ST2 \cdot A
T2 = ST1 \cdot B
T3 = ST3 \cdot (C \cdot B)
T4 = ST2 \cdot (C + B)

ST1 = (ST1 + T1) \cdot \overline{T2} + FS
ST2 = (ST2 + T2 + T3) \cdot \overline{T1} \cdot \overline{T4}
ST3 = (ST3 + T4 \cdot \overline{T1}) \cdot \overline{T3}
5.

\[
TA = ST2 \cdot A \\
TB = ST1 \cdot B \\
TC = ST3 \cdot C \\
TD = ST1 \cdot D \cdot \overline{B} \\
TE = ST2 \cdot E \cdot \overline{A} \\
TF = ST3 \cdot F \cdot \overline{C}
\]

\[
ST1 = (ST1 + TA + TC) \cdot \overline{TB} \cdot \overline{TD}
\]

\[
ST2 = (ST2 + TB + TF) \cdot \overline{TA} \cdot \overline{TE}
\]

\[
ST3 = (ST3 + TD + TE) \cdot \overline{TC} \cdot \overline{TF}
\]
6.

a) 

button A + button B

- door idle
- motor on door opening
- thermal relay + 15 sec delay
- counter > 20,000
- service mode
- reset button - assumed

b) 

Legend
button A  I:001/01
button B  I:001/02
motor      O:000/03
thermal relay I:001/03
reset button I:001/04 - assumed
state 1    B3:0/0
state 2    B3:0/1
state 3    B3:0/2
lamp       O:000/07
c) \[ S_0 = (S_0 + S_1(\text{delay}(15) + \text{thermal}))S_0(\text{buttonA} + \text{buttonB}) \]

\[ S_1 = (S_1 + S_0(\text{buttonA} + \text{buttonB}))S_1(\text{delay}(15) + \text{thermal})S_3(\text{counter}) \]

\[ S_3 = (S_3 + S_2(\text{counter}))S_3(\text{reset}) \]

\text{motor} = S_1

\text{light} = S_3
7.

a) block logic method

- **Door closed** (state 3) → remote OR button
- **Remote OR button or bottom limit**
- **Light sensor**
- **Door opening** (state 4) → remote OR button OR top limit
- **Door opening** (state 4) → remote OR button
- **Door opened** (state 1) → remote OR button OR bottom limit
- **Remote OR button or top limit**
- **Door closing** (state 2) → remote OR button
- **Light sensor**
- **Door closed** (state 3)
plc states - 12.43

first scan

state 1

state 2

state 3

state 4

close door

open door

state 2

state 4

TOF
T4:0
preset 300s

garage light

state 1

MCR

remote

button

state 1

state 2

MCR
b) state-transition equations

\[
\begin{align*}
T1 &= \text{state 1 to state 2} \\
T2 &= \text{state 2 to state 3} \\
T3 &= \text{state 2 to state 4} \\
T4 &= \text{state 3 to state 4} \\
T5 &= \text{state 4 to state 1} \\
\end{align*}
\]

using the previous state diagram.

\[
\begin{align*}
ST1 &= (ST1 + T5) \cdot \overline{T1} \\
ST2 &= (ST2 + T1) \cdot \overline{T2} \cdot \overline{T3} \\
ST3 &= (ST3 + T2) \cdot \overline{T4} \\
ST4 &= (ST4 + T3 + T4) \cdot \overline{T5} \\
FS &= \text{first scan} \\
\end{align*}
\]

\[
\begin{align*}
T1 &= ST1 \cdot (\text{remote} + \text{button}) \\
T2 &= ST2 \cdot (\text{remote} + \text{button} + \text{bottomlimit}) \\
T3 &= ST2 \cdot (\text{remote} + \text{button}) \\
T4 &= ST3 \cdot (\text{lightbeam}) \\
T5 &= ST4 \cdot (\text{remote} + \text{button} + \text{toplimit}) + FS \\
\end{align*}
\]
plc states - 12.48

T1 ST1
T5

T2 T3 ST2
T1

T4 ST3
T2

T5 ST4
T3
T4

ST2 close do

ST4 open do

TOF
T4:0 preset 300s

T4:0/DN garage light
12.5 ASSIGNMENT PROBLEMS

1. Describe the difference between the block logic, delayed update, and transition equation methods for converting state diagrams to ladder logic.

2. Write the ladder logic for the state diagram below using the block logic method.

3. Convert the following state diagram to ladder logic using the block logic method. Give the stop button higher priority.
4. Convert the following state diagram to ladder logic using the delayed update method.

5. Use equations to develop ladder logic for the state diagram below using the delayed update method. Be sure to deal with the priority problems.
6. Implement the State-Transition equations in the figure below with ladder logic.

- \( T_1 = FS \)
- \( T_2 = STB \cdot A \)
- \( T_3 = STB \cdot C \)
- \( T_4 = STC \cdot D \)
- \( T_5 = STA \cdot B \)

7. Write ladder logic to implement the state diagram below using state transition equations.

8. Convert the following state diagram to ladder logic using a) an equation based method, b) a
method that is not based on equations.

9. The state diagram below is for a simple elevator controller. a) Develop a ladder logic program that implements it with Boolean equations. b) Develop the ladder logic using the block logic technique. c) Develop the ladder logic using the delayed update method.

10. Write ladder logic for the state diagram below a) using an equation based method. b) without
11. For the state diagram for the traffic light example, add a 15 second green light timer and speed up signal for an emergency vehicle. A strobe light mounted on fire trucks will cause the lights to change so that the truck doesn’t need to stop. Modify the state diagram to include this option. Implement the new state diagram with ladder logic.

12. Design a program with a state diagram for a hydraulic press that will advance when two palm buttons are pushed. Top and bottom limit switches are used to reverse the advance and stop after a retract. At any time the hands removed from the palm button will stop an advance and retract the press. Include start and stop buttons to put the press in and out of an active mode.

13. In dangerous processes it is common to use two palm buttons that require a operator to use both hands to start a process (this keeps hands out of presses, etc.). To develop this there are two inputs (P1 and P2) that must both be turned on within 0.25s of each other before a machine cycle may begin.

Develop ladder logic with a state diagram to control a process that has a start (START) and stop (STOP) button for the power. After the power is on the palm buttons (P1 and P2) may be used as described above to start a cycle. The cycle will consist of turning on an output (MOVE) for 2 seconds. After the press has been cycled 1000 times the press power should turn off and an output (LIGHT) should go on.
14. Use a state diagram to design a parking gate controller.

- the gate will be raised by one output and lowered by another. If the gate gets stuck an over current detector will make a PLC input true. If this is the case the gate should reverse and the light should be turned on indefinitely.
- if a valid keycard is entered a PLC input will be true. The gate is to rise and stay open for 10 seconds.
- when a car is over the car detector a PLC input will go true. The gate is to open while this detector is active. If it is active for more than 30 seconds the light should also turn on until the gate closes.

15. This morning you received a call from Mr. Ian M. Daasprate at the Old Fashioned Widget Company. In the past when they built a new machine they would used punched paper cards for control, but their supplier of punched paper readers went out of business in 1972 and they have decided to try using PLCs this time. He explains that the machine will dip wooden parts in varnish for 2 seconds, and then apply heat for 5 minutes to dry the coat, after this they are manually removed from the machine, and a new part is put in. They are also considering a premium line of parts that would call for a dip time of 30 seconds, and a drying time of 10 minutes. He then refers you to the project manager, Ann Nooyed.

You call Ann and she explains how the machine should operate. There should be start and stop buttons. The start button will be pressed when the new part has been loaded, and is ready to be coated. A light should be mounted to indicate when the machine is in operation. The part is mounted on a wheel that is rotated by a motor. To dip the part, the motor is turned on until a switch is closed. To remove the part from the dipping bath the motor is turned on until a second switch is closed. If the motor to rotate the wheel is on for more that 10 seconds before hitting a switch, the machine should be turned off, and a fault light turned on. The fault condition will be cleared by manually setting the machine back to its initial state, and hitting the start button twice. If the part has been dipped and dried properly, then a done light should be lit. To select a premium product you will use an input switch that needs to be pushed before the start button is pushed. She closes by saying she will be going on vacation and you need to have it done before she returns.

You hang up the phone and, after a bit of thought, decide to use the following outputs and inputs,
16. Design ladder logic with a state diagram for the following process description.

a) A toggle start switch (TS1) and a limit switch on a safety gate (LS1) must both
be on before a solenoid (SOL1) can be energized to extend a stamping cylinder
to the top of a part. Should a part detect sensor (PS1) also be considered?
Explain your answer.

b) While the stamping solenoid is energized, it must remain energized until a limit
switch (LS2) is activated. This second limit switch indicates the end of a stroke.
At this point the solenoid should be de-energized, thus retracting the cylinder.

c) When the cylinder is fully retracted a limit switch (LS3) is activated. The cycle
may not begin again until this limit switch is active. This is one way to ensure
that a new part is present, is there another?

d) A cycle counter should also be included to allow counts of parts produced.
When this value exceeds some variable amount (from 1 to 5000) the machine
should shut down, and a job done light lit up.

e) A safety check should be included. If the cylinder solenoid has been on for more
than 5 seconds, it suggests that the cylinder is jammed, or the machine has a
fault. If this is the case the machine should be shut down, and a maintenance
light turned on.

f) Implement the ladder diagram on a PLC in the laboratory.

g) Fully document the ladder logic and prepare a short report - This should be of
use to another engineer that will be maintaining the system.
13. NUMBERS AND DATA

Topics:
- Number bases; binary, octal, decimal, hexadecimal
- Binary calculations; 2s compliments, addition, subtraction and Boolean operations
- Encoded values; BCD and ASCII
- Error detection; parity, gray code and checksums

Objectives:
- To be familiar with binary, octal and hexadecimal numbering systems.
- To be able to convert between different numbering systems.
- To understand 2s compliment negative numbers.
- To be able to convert ASCII and BCD values.
- To be aware of basic error detection techniques.

13.1 INTRODUCTION

Base 10 (decimal) numbers developed naturally because the original developers (probably) had ten fingers, or 10 digits. Now consider logical systems that only have wires that can be on or off. When counting with a wire the only digits are 0 and 1, giving a base 2 numbering system. Numbering systems for computers are often based on base 2 numbers, but base 4, 8, 16 and 32 are commonly used. A list of numbering systems is given in Figure 13.1. An example of counting in these different numbering systems is shown in Figure 13.2.

<table>
<thead>
<tr>
<th>Base</th>
<th>Name</th>
<th>Data Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Binary</td>
<td>Bit</td>
</tr>
<tr>
<td>8</td>
<td>Octal</td>
<td>Nibble</td>
</tr>
<tr>
<td>10</td>
<td>Decimal</td>
<td>Digit</td>
</tr>
<tr>
<td>16</td>
<td>Hexadecimal</td>
<td>Byte</td>
</tr>
</tbody>
</table>

*Figure 13.1*  Numbering Systems
The effect of changing the base of a number does not change the actual value, only how it is written. The basic rules of mathematics still apply, but many beginners will feel disoriented. This chapter will cover basic topics that are needed to use more complex programming instructions later in the book. These will include the basic number systems, conversion between different number bases, and some data oriented topics.

### 13.2 NUMERICAL VALUES

#### 13.2.1 Binary

Binary numbers are the most fundamental numbering system in all computers. A single binary digit (a bit) corresponds to the condition of a single wire. If the voltage on the wire is true the bit value is 1. If the voltage is off the bit value is 0. If two or more wires are used then each new wire adds another significant digit. Each binary number will have an equivalent digital value. Figure 13.3 shows how to convert a binary number to a decimal equivalent. Consider the digits, starting at the right. The least significant digit is 1, and
is in the 0th position. To convert this to a decimal equivalent the number base (2) is raised to the position of the digit, and multiplied by the digit. In this case the least significant digit is a trivial conversion. Consider the most significant digit, with a value of 1 in the 6th position. This is converted by the number base to the exponent 6 and multiplying by the digit value of 1. This method can also be used for converting the other number system to decimal.

\[
\begin{align*}
2^6 &= 64 \\
2^5 &= 32 \\
2^4 &= 16 \\
2^3 &= 8 \\
2^2 &= 4 \\
2^1 &= 2 \\
2^0 &= 1
\end{align*}
\]

Decimal numbers can be converted to binary numbers using division, as shown in Figure 13.4. This technique begins by dividing the decimal number by the base of the new number. The fraction after the decimal gives the least significant digit of the new number when it is multiplied by the number base. The whole part of the number is now divided again. This process continues until the whole number is zero. This method will also work for conversion to other number bases.

**Figure 13.3** Conversion of a Binary Number to a Decimal Number
start with decimal number 932

for binary (base 2)

\[
\frac{932}{2} = 466.0
\]
\[
\frac{466}{2} = 233.0
\]
\[
\frac{233}{2} = 116.5
\]
\[
\frac{116}{2} = 58.0
\]
\[
\frac{58}{2} = 29.0
\]
\[
\frac{29}{2} = 14.5
\]
\[
\frac{14}{2} = 7.0
\]
\[
\frac{7}{2} = 3.5
\]
\[
\frac{3}{2} = 1.5
\]
\[
\frac{1}{2} = 0.5
\]

multiply places after decimal by division base, in this case it is 2 because of the binary.

1110100100

* This method works for other number bases also, the divisor and multipliers should be changed to the new number bases.

**Figure 13.4** Conversion from Decimal to Binary

Most scientific calculators will convert between number bases. But, it is important to understand the conversions between number bases. And, when used frequently enough the conversions can be done in your head.

Binary numbers come in three basic forms - a bit, a byte and a word. A bit is a single binary digit, a byte is eight binary digits, and a word is 16 digits. Words and bytes are
shown in Figure 13.5. Notice that on both numbers the least significant digit is on the right hand side of the numbers. And, in the word there are two bytes, and the right hand one is the least significant byte.

![Figure 13.5 Bytes and Words](image)

Figure 13.5 Bytes and Words

Binary numbers can also represent fractions, as shown in Figure 13.6. The conversion to and from binary is identical to the previous techniques, except that for values to the right of the decimal the equivalents are fractions.

binary: 101.011

\[
1(2^2) = 4 \quad 0(2^1) = 0 \quad 1(2^0) = 1 \quad 0(2^{-1}) = 0 \quad 1(2^{-2}) = \frac{1}{4} \quad 1(2^{-3}) = \frac{1}{8}
\]

\[
= 4 + 0 + 1 + 0 + \frac{1}{4} + \frac{1}{8} = 5.375 \text{  decimal}
\]

Figure 13.6 A Binary Decimal Number

13.2.1.1 - Boolean Operations

In the next chapter you will learn that entire blocks of inputs and outputs can be used as a single binary number (typically a word). Each bit of the number would correspond to an output or input as shown in Figure 13.7.
There are three motors $M_1$, $M_2$ and $M_3$ represented with three bits in a binary number. When any bit is on the corresponding motor is on.

100 = Motor 1 is the only one on  
111 = All three motors are on  
in total there are $2^n$ or $2^3$ possible combinations of motors on.

Figure 13.7 Motor Outputs Represented with a Binary Number

We can then manipulate the inputs or outputs using Boolean operations. Boolean algebra has been discussed before for variables with single values, but it is the same for multiple bits. Common operations that use multiple bits in numbers are shown in Figure 13.8. These operations compare only one bit at a time in the number, except the shift instructions that move all the bits one place left or right.

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>0010 * 1010</td>
<td>0010</td>
</tr>
<tr>
<td>OR</td>
<td>0010 + 1010</td>
<td>1010</td>
</tr>
<tr>
<td>NOT</td>
<td>0010</td>
<td>1101</td>
</tr>
<tr>
<td>EOR</td>
<td>0010 eor 1010</td>
<td>1000</td>
</tr>
<tr>
<td>NAND</td>
<td>0010 * 1010</td>
<td>1101</td>
</tr>
<tr>
<td>shift left</td>
<td>111000</td>
<td>110001         (other results are possible)</td>
</tr>
<tr>
<td>shift right</td>
<td>111000</td>
<td>011100         (other results are possible)</td>
</tr>
</tbody>
</table>

dot=

Figure 13.8 Boolean Operations on Binary Numbers

13.2.1.2 - Binary Mathematics

Negative numbers are a particular problem with binary numbers. As a result there are three common numbering systems used as shown in Figure 13.9. Unsigned binary numbers are common, but they can only be used for positive values. Both signed and 2s compliment numbers allow positive and negative values, but the maximum positive values is reduced by half. 2s compliment numbers are very popular because the hardware and software to add and subtract is simpler and faster. All three types of numbers will be found in PLCs.
Examples of signed binary numbers are shown in Figure 13.10. These numbers use the most significant bit to indicate when a number is negative.

An example of 2s compliment numbers are shown in Figure 13.11. Basically, if the number is positive, it will be a regular binary number. If the number is to be negative, we start the positive number, compliment it (reverse all the bits), then add 1. Basically when these numbers are negative, then the most significant bit is set. To convert from a negative 2s compliment number, subtract 1, and then invert the number.
Using 2s compliments for negative numbers eliminates the redundant zeros of signed binaries, and makes the hardware and software easier to implement. As a result most of the integer operations in a PLC will do addition and subtraction using 2s compliment numbers. When adding 2s compliment numbers, we don’t need to pay special attention to negative values. And, if we want to subtract one number from another, we apply the twos compliment to the value to be subtracted, and then apply it to the other value.

Figure 13.12 shows the addition of numbers using 2s compliment numbers. The three operations result in zero, positive and negative values. Notice that in all three operation the top number is positive, while the bottom operation is negative (this is easy to see because the MSB of the numbers is set). All three of the additions are using bytes, this is important for considering the results of the calculations. In the left and right hand calculations the additions result in a 9th bit - when dealing with 8 bit numbers we call this bit the carry C. If the calculation started with a positive and negative value, and ended up with a carry bit, there is no problem, and the carry bit should be ignored. If doing the calculation on a calculator you will see the carry bit, but when using a PLC you must look elsewhere to find it.

<table>
<thead>
<tr>
<th>decimal</th>
<th>binary byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>00000010</td>
</tr>
<tr>
<td>1</td>
<td>00000001</td>
</tr>
<tr>
<td>0</td>
<td>00000000</td>
</tr>
<tr>
<td>-1</td>
<td>11111111</td>
</tr>
<tr>
<td>-2</td>
<td>11111110</td>
</tr>
</tbody>
</table>

METHOD FOR MAKING A NEGATIVE NUMBER

1. write the binary number for the positive
   00011110 becomes 11100011
2. Invert (compliment) the number
3. Add 1
   11100001 + 00000001 = 11100010
The integers have limited value ranges, for example a 16 bit word ranges from -32,768 to 32,767. In some cases calculations will give results outside this range, and the Overflow $O$ bit will be set. (Note: an overflow condition is a major error, and the PLC will probably halt when this happens.) For an addition operation the Overflow bit will be set when the sign of both numbers is the same, but the sign of the result is opposite. When the signs of the numbers are opposite an overflow cannot occur. This can be seen in Figure 13.13 where the numbers two of the three calculations are outside the range. When this happens the result goes from positive to negative, or the other way.

Note: Normally the carry bit is ignored during the operation, but some additional logic is required to make sure that the number has not overflowed and moved outside of the range of the numbers. Here the 2s complement byte can have values from -128 to 127.

Figure 13.12  Adding 2s Compliment Numbers

These bits also apply to multiplication and division operations. In addition the PLC will also have bits to indicate when the result of an operation is zero $Z$ and negative $N$. 

Note: If an overflow bit is set this indicates that a calculation is outside and acceptable range. When this error occurs the PLC will halt. Do not ignore the limitations of the numbers.

Figure 13.13  Carry and Overflow Bits
13.2.2 Other Base Number Systems

Other number bases are typically converted to and from binary for storage and mathematical operations. Hexadecimal numbers are popular for representing binary values because they are quite compact compared to binary. (Note: large binary numbers with a long string of 1s and 0s are next to impossible to read.) Octal numbers are also popular for inputs and outputs because they work in counts of eight; inputs and outputs are in counts of eight.

An example of conversion to, and from, hexadecimal is shown in Figure 13.14 and Figure 13.15. Note that both of these conversions are identical to the methods used for binary numbers, and the same techniques extend to octal numbers also.

---

**Figure 13.14  Conversion of a Hexadecimal Number to a Decimal Number**

\[16^3 = 4096 \quad 16^2 = 256 \quad 16^1 = 16 \quad 16^0 = 1\]

\[\begin{align*}
15(16^3) &= 61440 \\
8(16^2) &= 2048 \\
10(16^1) &= 160 \\
3(16^0) &= 3
\end{align*}\]

\[63651\]

---

**Figure 13.15  Conversion from Decimal to Hexadecimal**

\[\frac{5724}{16} = 357.75 \quad \rightarrow \quad 16(0.75) = 12 \ 'c'\]

\[\frac{357}{16} = 22.3125 \quad \rightarrow \quad 16(0.3125) = 5\]

\[\frac{22}{16} = 1.375 \quad \rightarrow \quad 16(0.375) = 6\]

\[\frac{1}{16} = 0.0625 \quad \rightarrow \quad 16(0.0625) = 1\]

---
13.2.3 BCD (Binary Coded Decimal)

Binary Coded Decimal (BCD) numbers use four binary bits (a nibble) for each digit. (Note: this is not a base number system, but it only represents decimal digits.) This means that one byte can hold two digits from 00 to 99, whereas in binary it could hold from 0 to 255. A separate bit must be assigned for negative numbers. This method is very popular when numbers are to be output or input to the computer. An example of a BCD number is shown in Figure 13.16. In the example there are four digits, therefore 16 bits are required. Note that the most significant digit and bits are both on the left hand side. The BCD number is the binary equivalent of each digit.

![Figure 13.16 A BCD Encoded Number](image)

Most PLCs store BCD numbers in words, allowing values between 0000 and 9999. They also provide functions to convert to and from BCD. It is also possible to calculations with BCD numbers, but this is uncommon, and when necessary most PLCs have functions to do the calculations. But, when doing calculations you should probably avoid BCD and use integer mathematics instead. Try to be aware when your numbers are BCD values and convert them to integer or binary value before doing any calculations.

13.3 DATA CHARACTERIZATION

13.3.1 ASCII (American Standard Code for Information Interchange)

When dealing with non-numerical values or data we can use plain text characters and strings. Each character is given a unique identifier and we can use these to store and interpret data. The ASCII (American Standard Code for Information Interchange) is a very common character encryption system is shown in Figure 13.17 and Figure 13.18. The table includes the basic written characters, as well as some special characters, and some control codes. Each one is given a unique number. Consider the letter A, it is readily recognized by most computers world-wide when they see the number 65.
<table>
<thead>
<tr>
<th>decimal</th>
<th>hexadecimal</th>
<th>ASCII</th>
<th>decimal</th>
<th>hexadecimal</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000</td>
<td>NUL</td>
<td>20</td>
<td>00100000</td>
<td>space</td>
</tr>
<tr>
<td>1</td>
<td>00000001</td>
<td>SOH</td>
<td>21</td>
<td>00100001</td>
<td>!</td>
</tr>
<tr>
<td>2</td>
<td>00000010</td>
<td>STX</td>
<td>22</td>
<td>00100010</td>
<td>‘</td>
</tr>
<tr>
<td>3</td>
<td>00000011</td>
<td>ETX</td>
<td>23</td>
<td>00100011</td>
<td>#</td>
</tr>
<tr>
<td>4</td>
<td>00000100</td>
<td>EOT</td>
<td>24</td>
<td>00100100</td>
<td>$</td>
</tr>
<tr>
<td>5</td>
<td>00000101</td>
<td>ENQ</td>
<td>25</td>
<td>00100101</td>
<td>%</td>
</tr>
<tr>
<td>6</td>
<td>00000110</td>
<td>ACK</td>
<td>26</td>
<td>00100110</td>
<td>&amp;</td>
</tr>
<tr>
<td>7</td>
<td>00000111</td>
<td>BEL</td>
<td>27</td>
<td>00100111</td>
<td>‘</td>
</tr>
<tr>
<td>8</td>
<td>00001000</td>
<td>BS</td>
<td>28</td>
<td>00101000</td>
<td>(</td>
</tr>
<tr>
<td>9</td>
<td>00001001</td>
<td>HT</td>
<td>29</td>
<td>00101001</td>
<td>)</td>
</tr>
<tr>
<td>10</td>
<td>00001010</td>
<td>LF</td>
<td>30</td>
<td>00101010</td>
<td>*</td>
</tr>
<tr>
<td>11</td>
<td>00001011</td>
<td>VT</td>
<td>31</td>
<td>00101100</td>
<td>+</td>
</tr>
<tr>
<td>12</td>
<td>00001100</td>
<td>FF</td>
<td>32</td>
<td>00101101</td>
<td>,</td>
</tr>
<tr>
<td>13</td>
<td>00001101</td>
<td>CR</td>
<td>33</td>
<td>00111011</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>00001110</td>
<td>S0</td>
<td>34</td>
<td>00111010</td>
<td>.</td>
</tr>
<tr>
<td>15</td>
<td>00001111</td>
<td>SI</td>
<td>35</td>
<td>00111101</td>
<td>/</td>
</tr>
<tr>
<td>16</td>
<td>00010000</td>
<td>DLE</td>
<td>36</td>
<td>00111110</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>00010001</td>
<td>DC1</td>
<td>37</td>
<td>00111111</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>00010010</td>
<td>DC2</td>
<td>38</td>
<td>00110000</td>
<td>2</td>
</tr>
<tr>
<td>19</td>
<td>00010011</td>
<td>DC3</td>
<td>39</td>
<td>00110001</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>00010100</td>
<td>DC4</td>
<td>40</td>
<td>00110010</td>
<td>4</td>
</tr>
<tr>
<td>21</td>
<td>00010101</td>
<td>NAK</td>
<td>41</td>
<td>00110011</td>
<td>5</td>
</tr>
<tr>
<td>22</td>
<td>00010110</td>
<td>SYN</td>
<td>42</td>
<td>00110100</td>
<td>6</td>
</tr>
<tr>
<td>23</td>
<td>00010111</td>
<td>ETB</td>
<td>43</td>
<td>00110101</td>
<td>7</td>
</tr>
<tr>
<td>24</td>
<td>00011000</td>
<td>CAN</td>
<td>44</td>
<td>00110110</td>
<td>8</td>
</tr>
<tr>
<td>25</td>
<td>00011001</td>
<td>EM</td>
<td>45</td>
<td>00111000</td>
<td>9</td>
</tr>
<tr>
<td>26</td>
<td>00011010</td>
<td>SUB</td>
<td>46</td>
<td>00111001</td>
<td>:</td>
</tr>
<tr>
<td>27</td>
<td>00011011</td>
<td>ESC</td>
<td>47</td>
<td>00111100</td>
<td>;</td>
</tr>
<tr>
<td>28</td>
<td>00011100</td>
<td>FS</td>
<td>48</td>
<td>00111101</td>
<td>&lt;</td>
</tr>
<tr>
<td>29</td>
<td>00011101</td>
<td>GS</td>
<td>49</td>
<td>00111110</td>
<td>=</td>
</tr>
<tr>
<td>30</td>
<td>00011110</td>
<td>RS</td>
<td>50</td>
<td>00111111</td>
<td>&gt;</td>
</tr>
<tr>
<td>31</td>
<td>00011111</td>
<td>US</td>
<td>51</td>
<td>00011000</td>
<td>?</td>
</tr>
</tbody>
</table>

*Figure 13.17  ASCII Character Table*
This table has the codes from 0 to 127, but there are more extensive tables that contain special graphics symbols, international characters, etc. It is best to use the basic codes, as they are supported widely, and should suffice for all controls tasks.

**Figure 13.18  ASCII Character Table**
An example of a string of characters encoded in ASCII is shown in Figure 13.19.

e.g. The sequence of numbers below will convert to

<table>
<thead>
<tr>
<th>A</th>
<th>W</th>
<th>e</th>
<th>e</th>
<th>T</th>
<th>e</th>
<th>s</th>
<th>t</th>
</tr>
</thead>
</table>
| A  | 65
| space | 32
| W  | 87
| e  | 101
| e  | 101
| space | 32
| T  | 84
| e  | 101
| s  | 115
| t  | 116

*Figure 13.19  A String of Characters Encoded in ASCII*

When the characters are organized into a string to be transmitted and LF and/or CR code are often put at the end to indicate the end of a line. When stored in a computer an ASCII value of zero is used to end the string.

13.3.2 Parity

Errors often occur when data is transmitted or stored. This is very important when transmitting data in noisy factories, over phone lines, etc. Parity bits can be added to data as a simple check of transmitted data for errors. If the data contains error it can be retransmitted, or ignored.

A parity bit is normally a 9th bit added onto an 8 bit byte. When the data is encoded the number of true bits are counted. The parity bit is then set to indicate if there are an even or odd number of true bits. When the byte is decoded the parity bit is checked to make sure it that there are an even or odd number of data bits true. If the parity bit is not satisfied, then the byte is judged to be in error. There are two types of parity, even or odd. These are both based upon an even or odd number of data bits being true. The odd parity bit is true if there are an odd number of bits on in a binary number. On the other hand the Even parity is set if there are an even number of true bits. This is illustrated in Figure 13.20.
Parity bits are normally suitable for single bytes, but are not reliable for data with a number of bits.

<table>
<thead>
<tr>
<th></th>
<th>data bits</th>
<th>parity bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Odd Parity</td>
<td>10101110</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>10111000</td>
<td>0</td>
</tr>
<tr>
<td>Even Parity</td>
<td>00101010</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>10111101</td>
<td>1</td>
</tr>
</tbody>
</table>

*Figure 13.20 Parity Bits on a Byte*

Parity bits are normally suitable for single bytes, but are not reliable for data with a number of bits.

Note: Control systems perform important tasks that can be dangerous in certain circumstances. If an error occurs there could be serious consequences. As a result error detection methods are very important for control systems. When error detection occurs the system should either be robust enough to recover from the error, or the system should fail-safe. If you ignore these design concepts you will eventually cause an accident.

### 13.3.3 Checksums

Parity bits are suitable for a few bits of data, but checksums are better for larger data transmissions. These are simply an algebraic sum of all of the data transmitted. Before data is transmitted the numeric values of all of the bytes are added. This sum is then transmitted with the data. At the receiving end the data values are summed again, and the total is compared to the checksum. If they match the data is accepted as good. An example of this method is shown in Figure 13.21.
Checksums are very common in data transmission, but these are also hidden from the average user. If you plan to transmit data to or from a PLC you will need to consider parity and checksum values to verify the data. Small errors in data can have major consequences in received data. Consider an oven temperature transmitted as a binary integer (1023d = 0000 0100 0000 0000b). If a single bit were to be changed, and was not detected the temperature might become (0000 0110 0000 0000b = 1535d) This small change would dramatically change the process.

13.3.4 Gray Code

Parity bits and checksums are for checking data that may have any value. Gray code is used for checking data that must follow a binary sequence. This is common for devices such as angular encoders. The concept is that as the binary number counts up or down, only one bit changes at a time. Thus making it easier to detect erroneous bit changes. An example of a gray code sequence is shown in Figure 13.22. Notice that only one bit changes from one number to the next. If more than a single bit changes between numbers, then an error can be detected.

ASIDE: When the signal level in a wire rises or drops, it induces a magnetic pulse that excites a signal in other nearby lines. This phenomenon is known as cross-talk. This signal is often too small to be noticed, but several simultaneous changes, coupled with background noise could result in erroneous values.
13.4 SUMMARY

- Binary, octal, decimal and hexadecimal numbers were all discussed.
- 2s compliments allow negative binary numbers.
- BCD numbers encode digits in nibbles.
- ASCII values are numerical equivalents for common alphanumeric characters.
- Gray code, parity bits and checksums can be used for error detection.

13.5 PRACTICE PROBLEMS

1. Why are binary, octal and hexadecimal used for computer applications?

2. Is a word is 3 nibbles?

3. What are the specific purpose for Gray code and parity?

4. Convert the following numbers to/from binary
5. Convert the BCD number below to a decimal number,

0110 0010 0111 1001

6. Convert the following binary number to a BCD number,

0100 1011

7. Convert the following binary number to a Hexadecimal value,

0100 1011

8. Convert the following binary number to a octal,

0100 1011

9. Convert the decimal value below to a binary byte, and then determine the odd parity bit,

97

10. Convert the following from binary to decimal, hexadecimal, BCD and octal.

a) 101101  
   b) 11011011  
   c) 1000000001  
   d) 0010110110101
11. Convert the following from decimal to binary, hexadecimal, BCD and octal.

a) 1
b) 17

c) 20456
d) -10

12. Convert the following from hexadecimal to binary, decimal, BCD and octal.

a) 1
b) 17

c) ABC
d) -A

13. Convert the following from BCD to binary, decimal, hexadecimal and octal.

a) 1001
b) 1001 0011

c) 0011 0110 0001
d) 0000 0101 0111 0100

14. Convert the following from octal to binary, decimal, hexadecimal and BCD.

a) 7
b) 17

c) 777
d) 32634

15.

a) Represent the decimal value thumb wheel input, 3532, as a Binary Coded Decimal (BCD) and a Hexadecimal Value (without using a calculator).
   i) BCD
   ii) Hexadecimal

b) What is the corresponding decimal value of the BCD value, 1001111010011011?

16. Add/subtract/multiply/divide the following numbers.

a) binary 101101101 + 01010101111011
b) hexadecimal 101 + ABC
c) octal 123 + 777
d) binary 110110111 - 010111

e) hexadecimal ABC - 123
f) octal 777 - 123

g) binary 0101111 - 110110111
h) hexadecimal 123-ABC

i) octal 123 - 777
j) 2s complement bytes 101101101 + 00000011
k) 2s complement bytes 00111011 + 00000011
l) binary 101101101 * 10101
m) octal 123 * 777

n) octal 777 / 123

o) binary 101101101 / 10101

p) hexadecimal ABC / 123
17. Do the following operations with 8 bit bytes, and indicate the condition of the overflow and carry bits.
   a) 10111011 + 00000011
   b) 00111011 + 00000011
   c) 11011011 + 11011111
   d) 110110111 - 01011111
   e) 01101011 + 01111011
   f) 10110110 - 11101110

18. Consider the three BCD numbers listed below.
    
    1001 0110 0101 0001  
    0010 0100 0011 1000  
    0100 0011 0101 0001  
    a) Convert these numbers to their decimal values.
    b) Convert the decimal values to binary.
    c) Calculate a checksum for all three binary numbers.
    d) What would the even parity bits be for the binary words found in b).

19. Is the 2nd bit set in the hexadecimal value F49?

20. Explain where grey code occurs when creating Karnaugh maps.

21. Convert the decimal number 1000 to a binary number, and then to hexadecimal.

**13.6 PRACTICE PROBLEM SOLUTIONS**

1. base 2, 4, 8, and 16 numbers translate more naturally to the numbers stored in the computer.

2. no, it is four nibbles

3. Both of these are coding schemes designed to increase immunity to noise. A parity bit can be used to check for a changed bit in a byte. Gray code can be used to check for a value error in a stream of continuous values.

4. a) 1101 0100 0011 0001, b) 3117

5. 6279

6. 0111 0101

7. 4B

8. 113
9. 1100001 odd parity bit = 1

10.

<table>
<thead>
<tr>
<th></th>
<th>binary</th>
<th>101101</th>
<th>11011011</th>
<th>10000000001</th>
<th>0010110110101</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCD</td>
<td></td>
<td>0100 0101</td>
<td>0010 0001</td>
<td>1001 0001 0010 0101</td>
<td>0001 0100 0110 0001</td>
</tr>
<tr>
<td>decimal</td>
<td>45</td>
<td>219</td>
<td>1025</td>
<td>1461</td>
<td></td>
</tr>
<tr>
<td>hex</td>
<td>2D</td>
<td>DB</td>
<td>401</td>
<td>5B5</td>
<td></td>
</tr>
<tr>
<td>octal</td>
<td>55</td>
<td>333</td>
<td>2001</td>
<td>2665</td>
<td></td>
</tr>
</tbody>
</table>

11.

<table>
<thead>
<tr>
<th></th>
<th>decimal</th>
<th>1</th>
<th>17</th>
<th>20456</th>
<th>-10</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCD</td>
<td>0001</td>
<td>0001 0111</td>
<td>0010 0000 0100 0101 0110</td>
<td>-0001 0000</td>
<td></td>
</tr>
<tr>
<td>binary</td>
<td>1</td>
<td>10001</td>
<td>0100 1111 1110 1000</td>
<td>1111 1111 1111 0110</td>
<td></td>
</tr>
<tr>
<td>hex</td>
<td>1</td>
<td>11</td>
<td>4FE8</td>
<td>FFF6</td>
<td></td>
</tr>
<tr>
<td>octal</td>
<td>1</td>
<td>21</td>
<td>47750</td>
<td>177766</td>
<td></td>
</tr>
</tbody>
</table>

12.

<table>
<thead>
<tr>
<th></th>
<th>hex</th>
<th>1</th>
<th>17</th>
<th>ABC</th>
<th>-A</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCD</td>
<td>0001</td>
<td>0010 0011</td>
<td>0010 0111 0100 1000</td>
<td>-0001 0000</td>
<td></td>
</tr>
<tr>
<td>binary</td>
<td>1</td>
<td>10111</td>
<td>0000 1010 1011 1100</td>
<td>1111 1111 1111 0110</td>
<td></td>
</tr>
<tr>
<td>decimal</td>
<td>1</td>
<td>23</td>
<td>2748</td>
<td>-10</td>
<td></td>
</tr>
<tr>
<td>octal</td>
<td>1</td>
<td>27</td>
<td>5274</td>
<td>177766</td>
<td></td>
</tr>
</tbody>
</table>

13.

<table>
<thead>
<tr>
<th></th>
<th>BCD</th>
<th>1001</th>
<th>1001 0011</th>
<th>0011 0110 0001</th>
<th>0000 0101 0111 0100</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary</td>
<td>1001</td>
<td>101 1101</td>
<td>1 0110 1001</td>
<td>10 0011 1110</td>
<td></td>
</tr>
<tr>
<td>decimal</td>
<td>9</td>
<td>93</td>
<td>361</td>
<td>0574</td>
<td></td>
</tr>
<tr>
<td>hex</td>
<td>9</td>
<td>5D</td>
<td>169</td>
<td>23E</td>
<td></td>
</tr>
<tr>
<td>octal</td>
<td>11</td>
<td>135</td>
<td>551</td>
<td>1076</td>
<td></td>
</tr>
</tbody>
</table>
14.

<table>
<thead>
<tr>
<th>octal</th>
<th>7</th>
<th>17</th>
<th>777</th>
<th>32634</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary</td>
<td>111</td>
<td>1111</td>
<td>1111 1111</td>
<td>0011 0101 1001 1100</td>
</tr>
<tr>
<td>decimal</td>
<td>7</td>
<td>15</td>
<td>511</td>
<td>13724</td>
</tr>
<tr>
<td>hex</td>
<td>7</td>
<td>F</td>
<td>1FF</td>
<td>359C</td>
</tr>
<tr>
<td>BCD</td>
<td>0111</td>
<td>0001 0101</td>
<td>0101 0001 0001</td>
<td>0001 0011 0111 0010 0100</td>
</tr>
</tbody>
</table>

15. a) $3532 = 0011 0101 0011 0010 = DCC$, b0 the number is not a valid BCD.

16.

a) $0001 0110 1110 1000$

b) BBD

c) 1122

d) 0000 0001 1000 1000

e) 999

f) 654

g) 1111 1110 0111 1000

h) -999

i) -654

j) 0000 0001 0111 1010

k) 0000 0000 0011 1110

l) 0001 1101 1111 0001

m) 122655

n) 6

o) 0000 0000 0001 0001

p) 9

17.

a) $10111011 + 00000011 = 1011 1110$

b) $00111011 + 00000011 = 0011 1110$

c) $11011011 + 11011111 = 1011 1010 + C + O$

d) $11011011 - 01011111 = 0101 1000 + C + O$

e) $01101011 + 01111011 = 1110 0110$

f) $10110110 - 11101110 = 1100 1000$

18. a) 9651, 2438, 4351, b) 0010 0101 1011 0011, 0000 1001 1000 0110, 0001 0000 1111 1111, c) 16440, d) 1, 0, 0

19. The binary value is 1111 0100 1001, so the second bit is 0.

20. when selecting the sequence of bit changes for Karnaugh maps, only one bit is changed at a time. This is the same method used for grey code number sequences. By using the code the bits in the map are naturally grouped.
1000_{10} = 1111101000_{2} = 3e8_{16}

13.7 ASSIGNMENT PROBLEMS

1. Why are hexadecimal numbers useful when working with PLCs?
14. PLC MEMORY

Topics:
- PLC-5 memory types; program and data
- Data types; output, input, status, bit, timer, counter, integer, floating point, etc.
- Memory addresses; words, bits, data files, expressions, literal values and indirect.

Objectives:
- To know the basic memory types available
- To be able to use addresses for locations in memory

14.1 INTRODUCTION

Advanced ladder logic functions allow controllers to perform calculations, make decisions and do other complex tasks. Timers and counters are examples of ladder logic functions. They are more complex than basic input contacts and output coils and they rely upon data stored in the memory of the PLC. The memory of the PLC is organized to hold different types of programs and data.

14.2 MEMORY ADDRESSES

The memory in a PLC is organized by data type as shown in Figure 14.1. There are two fundamental types of memory used in Allen-Bradley PLCs - Program and Data memory. Memory is organized into blocks of up to 1000 elements in an array called a file. The Program file holds programs, such as ladder logic. There are eight Data files defined by default, but additional data files can be added if they are needed.
14.3 PROGRAM FILES

In a PLC-5 the first three program files, from 0 to 2, are defined by default. File 0 contains system information and should not be changed, and file 1 is reserved for SFCs. File 2 is available for user programs and the PLC will run the program in file 2 by default. Other program files can be added from file 3 to 999. Typical reasons for creating other

---

**Figure 14.1** PLC Memory

These are a collection of up to 1000 slots to store up to 1000 programs. The main program will be stored in program file 2. SFC programs must be in file 1, and file 0 is used for program and password information. All other program files from 3 to 999 can be used for subroutines.
programs are for subroutines.

When a user creates a ladder logic program with programming software, it is converted to a mnemonic-like form, and then transferred to the PLC, where it is stored in a program file. The contents of the program memory cannot be changed while the PLC is running. If, while a program was running, it was overwritten with a new program, serious problems could arise.

14.4 DATA FILES

Data files are used for storing different information types, as shown in Figure 14.2. These locations are numbered from 0 to 999. The letter in front of the number indicates the data type. For example, F8: is read as floating point numbers in data file 8. Numbers are not given for O: and I:, but they are implied to be O0: and I1:. The number that follows the : is the location number. Each file may contain from 0 to 999 locations that may store values. For the input I: and output O: files the locations are converted to physical locations on the PLC using rack and slot numbers. The addresses that can be used will depend upon the hardware configuration. The status S2: file is more complex and is discussed later. The other memory locations are simply slots to store data in. For example, F8:35 would indicate the 36th value in the 8th data file which is floating point numbers.

![Figure 14.2 Data Files for an Allen Bradley PLC-5](image-url)
Only the first three data files are fixed \(O:1\) and \(S2:\), all of the other data files can be moved. It is also reasonable to have multiple data files with the same data type. For example, there could be two files for integer numbers \(N7:\) and \(N10:\). The length of the data files can be from 0 up to 999 as shown in Figure 14.3. But, these files are often made smaller to save memory.

![Data File Locations](image)

**Figure 14.3** Locations in a Data File

Figure 14.2 shows the default data files for a PLC-5. There are many additional data types, a full list is shown in Figure 14.4. Some of the data types are complex and contain multiple data values, including \(BT\), \(C\), \(MG\), \(PD\), \(R\), \(SC\), and \(T\). Recall that timers require integers for the accumulator and preset, and \(TT\), \(DN\) and \(EN\) bits are required. Other data types are based on single bits, 8 bit bytes and 16 bit words.
When using data files and functions we need to ask for information with an address. The simplest data addresses are data bits (we have used these for basic inputs and outputs already). An example of Address bits is shown in Figure 14.5. Memory bits are normally indicated with a forward slash followed by a bit number /n. The first example is from an input card I:000, the third input is indicated with the bit address /02. The second example is for a counter C5: done bit /DN. This could also be replaced with C5:4/15 to get equivalent results. The DN notation, and others like it are used to simplify the task of programming. The example B3/4 will get the fourth bit in bit memory B3. For bit memory the slash is not needed, because the data type is already bits.

<table>
<thead>
<tr>
<th>Type</th>
<th>Length (words)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A - ASCII</td>
<td>1/2</td>
</tr>
<tr>
<td>B - bit</td>
<td>1/16</td>
</tr>
<tr>
<td>BT - block transfer</td>
<td>6</td>
</tr>
<tr>
<td>C - counter</td>
<td>3</td>
</tr>
<tr>
<td>D - BCD</td>
<td>1</td>
</tr>
<tr>
<td>F - floating point</td>
<td>2</td>
</tr>
<tr>
<td>MG - message</td>
<td>56</td>
</tr>
<tr>
<td>N - integer (signed, unsigned, 2s compliment, BCD)</td>
<td>1</td>
</tr>
<tr>
<td>PD - PID controller</td>
<td>82</td>
</tr>
<tr>
<td>R - control</td>
<td>3</td>
</tr>
<tr>
<td>SC - SFC status</td>
<td>3</td>
</tr>
<tr>
<td>ST - ASCII string</td>
<td>42</td>
</tr>
<tr>
<td>T - timer</td>
<td>3</td>
</tr>
</tbody>
</table>

NOTE: Memory is a general term that refers to both files and locations. The term *file* is specific to PLC manufacturers and is not widely recognized elsewhere.
bit - individual bits in accessed - this is like addressing a single output as a data bit.

\[ I:000/02 \text{ - the third input bit from input card } I:000 \]
\[ C5:4/DN \text{ - the DN bit of a counter} \]
\[ B3/4 \text{ - the fourth bit in bit memory} \]

NOTE: Some bit addresses, especially inputs and outputs are addressed using octal. This often leads to careless errors and mistakes. For example if you want the 11th output bit, or bit 10, you would need to use 12 in octal to address it properly.

<table>
<thead>
<tr>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
<th>4th</th>
<th>5th</th>
<th>6th</th>
<th>7th</th>
<th>8th</th>
<th>9th</th>
<th>10th</th>
<th>11th</th>
<th>12th</th>
<th>13th</th>
<th>14th</th>
<th>15th</th>
<th>16th</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
</tbody>
</table>

Figure 14.5 Bit Level Addressing

Entire words can be addressed as shown in Figure 14.6. These values will normally be assumed to be 2s compliment, but some functions may assume otherwise. The first example shows a simple integer memory value. The next example gets up to inputs (from card 0 in rack zero) as a single word. The last two examples are more complex and they access the accumulator and preset values for a timer. Here a '.' is used as the '/' was used for bit memory to indicate it is an integer. The first two examples don’t need the '.' because they are both integer value types. Other types of word addressing are possible, including floating point numbers.

integer word - 16 bits can be manipulated as an integer.

\[ N7:8 \text{ - the 9th value from integer memory} \]
\[ I:000 \text{ - an integer with all input values from an input card} \]
\[ T4:7.ACC \text{ - the accumulator value for a timer} \]
\[ T4:7.PRE \text{ - the preset value for a timer} \]

Figure 14.6 Integer Word Addressing

Data values do not always need to be stored in memory, they can be define literally. Figure 14.7 shows an example of two different data values. The first is an integer, the second is a real number. Hexadecimal numbers can be indicated by following the number with \( H \), a leading zero is also needed when the first digit is \( A, B, C, D, E \) or \( F \). A binary number is indicated by adding a \( B \) to the end of the number.
literal data value - a data value can be provided without storing it in memory.

8 - an integer
8.5 - a floating point number
08FH - a hexadecimal value 8F
01101101B - a binary number 01101101

*Figure 14.7* Literal Data Values

Sometimes we will want to refer to an array of values, as shown in Figure 14.8. This data type is indicated by beginning the number with a pound or hash sign pound'. The first example describes an array of floating point numbers staring in file 8 at location 5. The second example is for an array of integers in file 7 starting at location 0. The length of the array is determined elsewhere.

file - the first location of an array of data values.

#F8:5 - indicates a group of values starting at F8:5
#N7:0 - indicates a group of values starting at N7:0

*Figure 14.8* File Addressing

Indirect addressing is a method for allowing a variable in a data address, as shown in Figure 14.9. The indirect (variable) part of the address is shown between brackets '[' and ']' . If a PLC is looking at an address and it finds an indirect address it will look in the specified memory location, and put that number in place of the indirect address. Consider the first example below I:000/[N7:2], if the value in the integer memory location N7:2 is 45, then the address becomes I:000/45. The other examples are very similar. This type of technique is very useful when making programs that can be adapted for different recipes - by changing a data value in one memory location the program can follow a new set of data.
indirect - another memory location can be used in the description of a location.

I:000/[N7:2] -If N7:2 location contains 5 this will become I:000/05
I:[N7:1]/03 -If the integer memory location contains 2 this will become I:002/03
#I:[N7:1] -If the integer memory location contains 2 the file will start at I:002
N[N7:0]:8 - If the number in N7:0 is 10 the data address becomes N10:8

*Figure 14.9  Indirect Addressing*

Expressions allow addresses and functions to be typed in and interpreted when the program is run. The example in Figure 14.10 will get a floating point number from file 8, location 3, perform a sine transformation, and then add 1.3. The text string is not interpreted until the PLC is running, and if there is an error, it may not occur until the program is running - so use this function cautiously.

expression - a text string that describes a complex operation.

“sin(F8:3) + 1.3” - a simple calculation

*Figure 14.10  Expression Data Values*

These data types and addressing modes will be discussed more as applicable functions are presented later in this chapter and book. Floating point numbers, expressions and indirect addressing may not be available on older or lower cost PLCs.

Figure 14.11 shows a simple example ladder logic with functions. The basic operation is such that while input 4 is true the functions will be performed. The first statement will move (MOV) the literal value of 130 into integer memory N7:0. The next move function will copy the value from N7:0 to N7:1. The third statement will add integers value in N7:0 and N7:1 and store the results in N7:2.
14.4.1 User Bit Memory

Individual data bits can be accessed in the bit memory. These can be very useful when keeping track of internal states that do not directly relate to an output or input. The bit memory can be accessed with individual bits or with integer words. Examples of bit addresses are shown in Figure 14.12. The single blackened bit is in the third word $B3:2$ and it is the 4th bit $03$, so it can be addressed with $B3:2/03$. Overall, it is the 35th bit, so it could also be addressed with $B3/35$. 

Figure 14.11  An Example of Ladder Logic Functions

```
MOV source 130
destination N7:0

MOV source N7:0
destination N7:1

ADD sourceA N7:0
sourceB N7:1
destination N7:2
```
This method can also be used to access bits in integer memory also.

14.4.2 Timer Counter Memory

Previous chapters have discussed the operation of timers and counters. The ability to address their memory directly allows some powerful tools. Recall that by default timers are stored in the \textit{T4}: file. The bits and words for timers are;

- EN - timer enabled bit (bit 15)
- TT - timer timing bit (bit 14)
- DN - timer done bit (bit 13)
- PRE - preset word
- ACC - accumulated time word

Counter are stored in the \textit{C5}: file and they have the following bits and words.

\textbf{Figure 14.12} Bit Memory

This bit is B3:2/3 or B3/35. 
\((2) \times 16 + (3) = (35)\)
CU - count up bit (bit 15)
CD - count down bit (bit 14)
DN - counter done bit (bit 13)
OV - overflow bit (bit 12)
UN - underflow bit (bit 11)
PRE - preset word
ACC - accumulated count word

As discussed before we can access timer and counter bits and words using the proper notation. Examples of these are shown in Figure 14.13. The bit values can only be read, and should not be changed. The presets and accumulators can be read and overwriten.

Words

T4:0.PRE - the preset value for timer T4:0
T4:0.ACC - the accumulated value for timer T4:0
C5:0.PRE - the preset value for counter C5:0
C5:0.ACC - the accumulated value for counter C5:0

Bits

T4:0/EN - indicates when the input to timer T4:0 is true
T4:0/TT - indicates when the timer T4:0 is counting
T4:0/DN - indicates when timer T4:0 has reached the maximum
C5:0/CU - indicates when the count up instruction is true for C5:0
C5:0/CD - indicates when the count down instruction is true for C5:0
C5:0/DN - indicates when the counter C5:0 has reached the preset
C5:0/OV - indicates when the counter C5:0 has reached the maximum value (32767)
C5:0/UN - indicates when the counter C5:0 has reached the minimum value (-32768)

**Figure 14.13  Examples of Timer and Counter Addresses**

Consider the simple ladder logic example in Figure 14.14. It shows the use of a timer timing \( TT \) bit to seal on the timer when a door input has gone true. While the timer is counting, the bit will stay true and keep the timer counting. When it reaches the 10 second delay the \( TT \) bit will turn off. The next line of ladder logic will turn on a light while the timer is counting for the first 10 seconds.
14.4.3 PLC Status Bits (for PLC-5s and Micrologix)

Status memory allows a program to check the PLC operation, and also make some changes. A selected list of status bits is shown in Figure 14.15 for Allen-Bradley Micrologic and PLC-5 PLCs. More complete lists are available in the manuals. For example the first four bits $S2:0/x$ indicate the results of calculations, including carry, overflow, zero and negative/sign. The $S2:1/15$ will be true once when the PLC is turned on - this is the first scan bit. The time for the last scan will be stored in $S2:8$. The date and clock can be stored and read from locations $S2:18$ to $S2:23$. 

![Door Light Example Diagram](image-url)
The other status words allow more complex control of the PLC. The watchdog timer allows a time to be set in $S2:28$ so that if the PLC scan time is too long the PLC will give a fault condition - this is very important for dangerous processes. When a fault occurs the program number in $S2:29$ will run. For example, if you have a divide by zero fault, you can run a program that recovers from the error, if there is no program the PLC will halt. The locations from $S2:30$ to $S2:55$ are used for interrupts. Interrupts can be used to run programs at fixed time intervals, or when inputs change.

### 14.4.4 User Function Control Memory

Simple ladder logic functions can complete operations in a single scan of ladder logic. Other functions such as timers and counters will require multiple ladder logic scans to finish. While timers and counters have their own memory for control, a generic type of control memory is defined for other function. This memory contains the bits and words in Figure 14.16. Any given function will only use some of the values. The meaning of particular bits and words will be described later when discussing specific functions.

---

**Figure 14.15** Status Bits and Words for Micrologix and PLC-5s

- $S2:0/0$ carry in math operation
- $S2:0/1$ overflow in math operation
- $S2:0/2$ zero in math operation
- $S2:0/3$ sign in math operation
- $S2:1/15$ first scan of program file
- $S2:8$ the scan time (ms)
- $S2:18$ year
- $S2:19$ month
- $S2:20$ day
- $S2:21$ hour
- $S2:22$ minute
- $S2:23$ second
- $S2:28$ watchdog setpoint
- $S2:29$ fault routine file number
- $S2:30$ STI (selectable timed interrupt) setpoint
- $S2:31$ STI file number
- $S2:46$-$S2:54$, $S2:55$-$S2:56$ PII (Programmable Input Interrupt) settings
- $S2:55$ STI last scan time (ms)
- $S2:77$ communication scan time (ms)
14.4.5 Integer Memory

Integer memory is 16 bit words that are normally used as 2s compliment numbers that can store data values from -32768 to +32767. When decimal fractions are supplied they are rounded to the nearest number. These values are normally stored in N7:xx by default, but new blocks of integer memory are often created in other locations such as N9:xx. Integer memory can also be used for bits.

14.4.6 Floating Point Memory

Floating point memory is available in newer and higher cost PLCs, it is not available on the Micrologix. This memory stores real numbers in 4 words, with 7 digits of accuracy over a range from +/-1.1754944e-38 to +/-3.4028237e38. Floating point memory is stored in F8:xx by default, but other floating point numbers can be stored in other locations. Bit level access is not permitted (or useful) for these numbers.

14.5 SUMMARY

- Program files store users programs in files 2 - 999.
- Data files are available to users and will be 0-999 locations long.
- Default data types on a PLC-5 include Output (O0:), Input (I1:), Status (S2:), Bit (B3:), Timer (T4:), Counter (C5:), Control (R6:), Integer (N7:) and Float (F8:).
- Other memory types include Block Transfer (BT), ASCII (A), ASCII String (ST), BCD (D), Message (MG), PID Control (PD), SFC Status (SC).
• In memory locations a ‘/’ indicates a bit, ‘.’ indicates a word.
• Indirect addresses will substitute memory values between ‘[’, ‘]’.
• Files are like arrays and are indicated with ‘#’.
• Expressions allow equations to be typed in.
• Literal values for binary and hexadecimal values are followed by B and H.

14.6 PRACTICE PROBLEMS

1. Can PLC outputs can be set with Bytes instead of bits?
2. How many types of memory can a PLC-5 have?
3. What are the default program memory locations?
4. How many types of number bases are used in PLC memory?
5. How are timer and counter memory similar?
6. What types of memory cannot be changed?
7. Develop Ladder Logic for a car door/seat belt safety system. When the car door is open, or the seatbelt is not done up, a buzzer will sound for 5 seconds if the key has been switched on. A cabin light will be switched on when the door is open and stay on for 10 seconds after it is closed, unless a key has started the ignition power.
8. Look at the manuals for the status memory in your PLC and find the first scan location
9. Write ladder logic for the following problem description. When button A is pressed a value of 1001 will be stored in N7:0. When button B is pressed a value of -345 will be stored in N7:1, when it is not pressed a value of 99 will be stored in N7:1. When button C is pressed N7:0 and N7:1 will be added, and the result will be stored in N7:2.
10. Using the status memory locations, write a program that will flash a light for the first 15 seconds after it has been turned on. The light should flash once a second.
11. How many words are required for timer and counter memory?

14.7 PRACTICE PROBLEM SOLUTIONS

1. yes, for example the output word would be addressed as O:000
2. There are 13 different memory types, 10 of these can be defined by the user for data files
3. Program files 0 and 1 are reserved for system functions. File 2 is the default ladder logic program, and files 3 to 999 can be used for other programs.

4. binary, octal, BCD, 2s compliment, signed binary, floating point, bits, hexadecimal

5. both are similar. The timer and counter memories both use words for the accumulator and presets, and they use bits to track the status of the functions. These bits are somewhat different, but parallel in function.

6. Inputs cannot be changed by the program, and some of the status bits/words cannot be changed by the user.

7. S2:1/14 for micrologix, S2:1/15 for PLC-5.

9.

A

MOV
Source 1001
Dest N7:0

B

MOV
Source -345
Dest N7:1

B

MOV
Source 99
Dest N7:1

C

ADD
Source A N7:0
Source B N7:1
Dest N7:2
11. three memory words are used for a timer or a counter.

14.8 ASSIGNMENT PROBLEMS

1. Briefly list and describe the different methods for addressing values (e.g., word, bit, literal, etc.).

2. Could timer ‘T’ and counter ‘C’ memory types be replaced with control ‘R’ memory types? Explain your answer.
15. LADDER LOGIC FUNCTIONS

Topics:
- Functions for data handling, mathematics, conversions, array operations, statistics, comparison and Boolean operations.
- Design examples

Objectives:
- To understand basic functions that allow calculations and comparisons
- To understand array functions using memory files

15.1 INTRODUCTION

Ladder logic input contacts and output coils allow simple logical decisions. Functions extend basic ladder logic to allow other types of control. For example, the addition of timers and counters allowed event based control. A longer list of functions is shown in Figure 15.1. Combinatorial Logic and Event functions have already been covered. This chapter will discuss Data Handling and Numerical Logic. The next chapter will cover Lists and Program Control and some of the Input and Output functions. Remaining functions will be discussed in later chapters.
Most of the functions will use PLC memory locations to get values, store values and track function status. Most function will normally become active when the input is true. But, some functions, such as TOF timers, can remain active when the input is off. Other functions will only operate when the input goes from false to true, this is known as positive edge triggered. Consider a counter that only counts when the input goes from false to true, the length of time the input is true does not change the function behavior. A negative edge triggered function would be triggered when the input goes from true to false. Most functions are not edge triggered: unless stated assume functions are not edge triggered.
15.2 DATA HANDLING

15.2.1 Move Functions

There are two basic types of move functions;

MOV(value,destination) - moves a value to a memory location
MVM(value,mask,destination) - moves a value to a memory location, but with a
mask to select specific bits.

The simple MOV will take a value from one location in memory and place it in
another memory location. Examples of the basic MOV are given in Figure 15.2. When $A$
is true the MOV function moves a floating point number from the source to the destination
address. The data in the source address is left unchanged. When $B$ is true the floating point
number in the source will be converted to an integer and stored in the destination address
in integer memory. The floating point number will be rounded up or down to the nearest
integer. When $C$ is true the integer value of 123 will be placed in the integer file $N7:23$. 

NOTE: I do not draw functions exactly as they appear in manuals and programming soft-
ware. This helps save space and makes the instructions somewhat easier to read. All of
the necessary information is given.
A more complex example of move functions is given in Figure 15.3. When \( A \) becomes true the first move statement will move the value of 130 into \( N7:0 \). And, the second move statement will move the value of -9385 from \( N7:1 \) to \( N7:2 \). (Note: The number is shown as negative because we are using 2s compliment.) For the simple MOVs the binary values are not needed, but for the MVM statement the binary values are essential. The statement moves the binary bits from \( N7:3 \) to \( N7:5 \), but only those bits that are also on in the mask \( N7:4 \), other bits in the destination will be left untouched. Notice that the first bit \( N7:5/0 \) is true in the destination address before and after, but it is not true in the mask. The MVM function is very useful for applications where individual binary bits are to be manipulated, but they are less useful when dealing with actual number values.

\[
\begin{align*}
\text{MOV} & \quad \text{Source F8:07} \\
& \quad \text{Destination F8:23} \\
\text{MOV} & \quad \text{Source F8:07} \\
& \quad \text{Destination N7:23} \\
\text{MOV} & \quad \text{Source 123} \\
& \quad \text{Destination N7:23}
\end{align*}
\]

NOTE: when a function changes a value, except for inputs and outputs, the value is changed immediately. Consider Figure 15.2, if \( A \), \( B \) and \( C \) are all true, then the value in \( F8:23 \) will change before the next instruction starts. This is different than the input and output scans that only happen before and after the logic scan.

**Figure 15.2** Examples of the MOV Function
15.2.2 Mathematical Functions

Mathematical functions will retrieve one or more values, perform an operation and
store the result in memory. Figure 15.4 shows an ADD function that will retrieve values from N7:4 and F8:35, convert them both to the type of the destination address, add the floating point numbers, and store the result in F8:36. The function has two sources labelled source A and source B. In the case of ADD functions the sequence can change, but this is not true for other operations such as subtraction and division. A list of other simple arithmetic function follows. Some of the functions, such as the negative function are unary, so there is only one source.

![diagram]

```
ADD(source A N7:04
 source B F8:35
 destination F8:36)
```

ADD(value,value,destination) - add two values
SUB(value,value,destination) - subtract
MUL(value,value,destination) - multiply
DIV(value,value,destination) - divide
NEG(value,destination) - reverse sign from positive/negative
CLR(value) - clear the memory location

NOTE: To save space the function types are shown in the shortened notation above. For example the function ADD(value, value, destination) requires two source values and will store it in a destination. It will use this notation in a few places to reduce the bulk of the function descriptions.

Figure 15.4 Arithmetic Functions

An application of the arithmetic function is shown in Figure 15.5. Most of the operations provide the results we would expect. The second ADD function retrieves a value from N7:3, adds 1 and overwrites the source - this is normally known as an increment operation. The first DIV statement divides the integer 25 by 10, the result is rounded to the nearest integer, in this case 3, and the result is stored in N7:6. The NEG instruction takes the new value of -10, not the original value of 0, from N7:4 inverts the sign and stores it in N7:7.
A list of more advanced functions are given in Figure 15.6. This list includes basic trigonometry functions, exponents, logarithms and a square root function. The last function CPT will accept an expression and perform a complex calculation.
Figure 15.7 shows an example where an equation has been converted to ladder logic. The first step in the conversion is to convert the variables in the equation to unused memory locations in the PLC. The equation can then be converted using the most nested calculations in the equation, such as the LN function. In this case the results of the LN function are stored in another memory location, to be recalled later. The other operations are implemented in a similar manner. (Note: This equation could have been implemented in other forms, using fewer memory locations.)

ACS(value,destination) - inverse cosine
COS(value,destination) - cosine
ASN(value,destination) - inverse sine
SIN(value,destination) - sine
ATN(value,destination) - inverse tangent
TAN(value,destination) - tangent
XPY(value,value,destination) - X to the power of Y
LN(value,destination) - natural log
LOG(value,destination) - base 10 log
SQR(value,destination) - square root
CPT(destination,expression) - does a calculation
given

\[ A = \sqrt{\ln B + e^C \cos(D)} \]

assign

\[ \begin{align*}
A &= F8:0 \\
B &= F8:1 \\
C &= F8:2 \\
D &= F8:3
\end{align*} \]

The same equation in Figure 15.7 could have been implemented with a CPT function as shown in Figure 15.8. The equation uses the same memory locations chosen in Figure 15.7. The expression is typed directly into the PLC programming software.

*Figure 15.7 An Equation in Ladder Logic*
Figure 15.8  Calculations with a Compute Function

Math functions can result in status flags such as overflow, carry, etc. Care must be taken to avoid problems such as overflows. These problems are less common when using floating point numbers. Integers are more prone to these problems because they are limited to the range from -32768 to 32767.

15.2.3 Conversions

Ladder logic conversion functions are listed in Figure 15.9. The example function will retrieve a BCD number from the \( D \) type (BCD) memory and convert it to a floating point number that will be stored in \( F8:2 \). The other function will convert from 2s complement binary to BCD, and between radians and degrees.

Figure 15.9  Conversion Functions

| Source | Function | Destination
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>CPT</td>
<td>Dest. F8:0</td>
</tr>
<tr>
<td></td>
<td>Expression</td>
<td>SQR(LN(F8:1)+XPY(2.718,F8:2)*ACS(F8:3))</td>
</tr>
</tbody>
</table>

TOD(value,destination) - convert from BCD to 2s complement
FRD(value,destination) - convert from 2s complement to BCD
DEG(value,destination) - convert from radians to degrees
RAD(value,destination) - convert from degrees to radians

Examples of the conversion functions are given in Figure 15.10. The functions load in a source value, do the conversion, and store the results. The TOD conversion to BCD could result in an overflow error.
Arrays allow us to store multiple data values. In a PLC this will be a sequential series of numbers in integer, floating point, or other memory. For example, assume we are measuring and storing the weight of a bag of chips in floating point memory starting at #F8:20 (Note the ‘#’ for a data file). We could read a weight value every 10 minutes, and once every hour find the average of the six weights. This section will focus on techniques that manipulate groups of data organized in arrays, also called blocks in the manuals.
15.2.4.1 - Statistics

Functions are available that allow statistical calculations. These functions are listed in Figure 15.11. When \( A \) becomes true the average (AVE) conversion will start at memory location \( F8:0 \) and average a total of 4 values. The control word \( R6:1 \) is used to keep track of the progress of the operation, and to determine when the operation is complete. This operation, and the others, are edge triggered. The operation may require multiple scans to be completed. When the operation is done the average will be stored in \( F8:4 \) and the \( R6:1/DN \) bit will be turned on.

\[
\begin{array}{c|c}
\text{AVE} & \\
\text{File} & \#F8:0 \\
\text{Dest} & F8:4 \\
\text{Control} & R6:1 \\
\text{length} & 4 \\
\text{position} & 0 \\
\end{array}
\]

AVE(start value,destination,control,length) - average of values
STD(start value,destination,control,length) - standard deviation of values
SRT(start value,control,length) - sort a list of values

Figure 15.11 Statistic Functions

Examples of the statistical functions are given in Figure 15.12 for an array of data that starts at \( F8:0 \) and is 4 values long. When done the average will be stored in \( F8:4 \), and the standard deviation will be stored in \( F8:5 \). The set of values will also be sorted in ascending order from \( F8:0 \) to \( F8:3 \). Each of the function should have their own control memory to prevent overlap. It is not a good idea to activate the sort and the other calculations at the same time, as the sort may move values during the calculation, resulting in incorrect calculations.
ASIDE: These function will allow a real-time calculation of SPC data for control limits, etc. The only PLC function missing is a random function that would allow random sample times.

15.2.4.2 - Block Operations

A basic block function is shown in Figure 15.13. This COP (copy) function will
copy an array of 10 values starting at \(N7:50\) to \(N7:40\). The \textit{FAL} function will perform mathematical operations using an expression string, and the \textit{FSC} function will allow two arrays to be compared using an expression. The \textit{FLL} function will fill a block of memory with a single value.

\begin{figure}
\centering
\begin{tabular}{|l|c|}
\hline
A & COP  \\
\hline
Source \#N7:50 & Dest \#N7:40  \\
\hline
Length 10 &  \\
\hline
\end{tabular}
\caption{Block Operation Functions}
\end{figure}

COP\text{(start value,destination,length)} - copies a block of values  
FAL\text{(control,length,mode,destination,expression)} - will perform basic math operations to multiple values.  
FSC\text{(control,length,mode,expression)} - will do a comparison to multiple values  
FLL\text{(value,destination,length)} - copies a single value to a block of memory

Figure 15.14 shows an example of the \textit{FAL} function with different addressing modes. The first FAL function will do the following calculations 
\[N7:5 = N7:0 + 5, \ N7:6 = N7:1 + 5, \ N7:7 = N7:2 + 5, \ N8:7 = N7:3 + 5, \ N7:9 = N7:4 + 5.\]
The second FAL statement does not have a file ‘#’ sign in front of the expression value, so the calculations will be 
\[N7:5 = N7:0 + 5, \ N7:6 = N7:0 + 5, \ N7:7 = N7:0 + 5, \ N8:7 = N7:0 + 5, \ N7:9 = N7:0 + 5.\] With a mode of 2 the instruction will do two of the calculations when there is a positive edge from B (i.e., a transition from false to true). The result of the last FAL statement will be 
\[N7:5 = N7:0 + 5, \ N7:5 = N7:1 + 5, \ N7:5 = N7:2 + 5, \ N7:5 = N7:3 + 5, \ N7:5 = N7:4 + 5.\] The last operation would seem to be useless, but notice that the mode is \textit{incremental}. This mode will do one calculation for each positive transition of C. The \textit{all} mode will perform all five calculations in a single scan whenever there is a positive edge on the input. It is also possible to put in a number that will indicate the number of calculations per scan. The calculation time can be long for large arrays and trying to do all of the calculations in one scan may lead to a watchdog time-out fault.
15.3 LOGICAL FUNCTIONS

15.3.1 Comparison of Values

Comparison functions are shown in Figure 15.15. Previous function blocks were outputs, these replace input contacts. The example shows an EQU (equal) function that compares two floating point numbers. If the numbers are equal, the output bit $B3:5/1$ is true, otherwise it is false. Other types of equality functions are also listed.
EQU(value,value) - equal
NEQ(value,value) - not equal
LES(value,value) - less than
LEQ(value,value) - less than or equal
GRT(value,value) - greater than
GEQ(value,value) - greater than or equal
CMP(expression) - compares two values for equality
MEQ(value,mask,threshold) - compare for equality using a mask
LIM(low limit,value,high limit) - check for a value between limits

Figure 15.15  Comparison Functions

The example in Figure 15.16 shows the six basic comparison functions. To the right of the figure are examples of the comparison operations.
The ladder logic in Figure 15.16 is recreated in Figure 15.17 with the CMP function that allows text expressions.
Expressions can also be used to do more complex comparisons, as shown in Figure 15.18. The expression will determine if $F8:1$ is between $F8:0$ and $F8:2$.

The LIM and MEQ functions are shown in Figure 15.19. The first three functions will compare a test value to high and low limits. If the high limit is above the low limit and the test value is between or equal to one limit, then it will be true. If the low limit is above
the high limit then the function is only true for test values outside the range. The masked equal will compare the bits of two numbers, but only those bits that are true in the mask.

---

**Figure 15.19** Complex Comparison Functions
Figure 15.20 shows a numberline that helps determine when the LIM function will be true.

![A Number Line for the LIM Function](image)

File to file comparisons are also permitted using the FSC instruction shown in Figure 15.21. The instruction uses the control word R6:0. It will interpret the expression 10 times, doing two comparisons per logic scan (the Mode is 2). The comparisons will be \( F8:10 < F8:0 \), \( F8:11 < F8:0 \) then \( F8:12 < F8:0 \), \( F8:13 < F8:0 \) then \( F8:14 < F8:0 \), \( F8:15 < F8:0 \) then \( F8:16 < F8:0 \), \( F8:17 < F8:0 \) then \( F8:18 < F8:0 \), \( F8:19 < F8:0 \). The function will continue until a false statement is found, or the comparison completes. If the comparison completes with no false statements the output \( A \) will then be true. The mode could have also been All to execute all the comparisons in one scan, or Increment to update when the input to the function is true - in this case the input is a plain wire, so it will always be true.

![File Comparison Using Expressions](image)

**Figure 15.21** File Comparison Using Expressions
15.3.2 Boolean Functions

Figure 15.22 shows Boolean algebra functions. The function shown will obtain data words from bit memory, perform an and operation, and store the results in a new location in bit memory. These functions are all oriented to word level operations. The ability to perform Boolean operations allows logical operations on more than a single bit.

\[
\begin{array}{c|c|c}
A & & \text{AND} \\
& & \text{source A B3:0} \\
& & \text{source B B3:1} \\
& & \text{dest. B3:2}
\end{array}
\]

\begin{itemize}
  \item AND(value, value, destination) - Binary and function
  \item OR(value, value, destination) - Binary or function
  \item NOT(value, value, destination) - Binary not function
  \item XOR(value, value, destination) - Binary exclusive or function
\end{itemize}

Figure 15.22  Boolean Functions

The use of the Boolean functions is shown in Figure 15.23. The first three functions require two arguments, while the last function only requires one. The AND function will only turn on bits in the result that are true in both of the source words. The OR function will turn on a bit in the result word if either of the source word bits is on. The XOR function will only turn on a bit in the result word if the bit is on in only one of the source words. The NOT function reverses all of the bits in the source word.
15.4 DESIGN CASES

15.4.1 Simple Calculation

Problem: A switch will increment a counter on when engaged. This counter can be reset by a second switch. The value in the counter should be multiplied by 2, and then displayed as a BCD output using (O:0.0/0 - O:0.0/7)
15.4.2 For-Next

Problem: Design a for-next loop that is similar to ones found in traditional programming languages. When $A$ is true the ladder logic should be active for 10 scans, and the scan number from 1 to 10 should be stored in N7:0.

Solution:
15.4.3 Series Calculation

Problem: Create a ladder logic program that will start when input A is turned on and calculate the series below. The value of n will start at 1 and with each scan of the ladder logic n will increase until n=100. While the sequence is being incremented, any change in A will be ignored.

\[ x = 2(n - 1) \]

\[ A = I:000/00 \]
\[ n = N7:0 \]
\[ x = N7:1 \]

Solution:

\[ x = 2(n - 1) \]

\[ A = I:000/00 \]
\[ n = N7:0 \]
\[ x = N7:1 \]

Figure 15.26 A Series Calculation Example
15.4.4 Flashing Lights

Problem: We are designing a movie theater marquee, and they want the traditional flashing lights. The lights have been connected to the outputs of the PLC from O:001/00 to O:001/17. When the PLC is turned, every second light should be on. Every half second the lights should reverse. The result will be that in one second two lights side-by-side will be on half a second each.

Solution:

Figure 15.27  A Flashing Light Example

15.5 SUMMARY

- Functions can get values from memory, do simple operations, and return the results to memory.
- Scientific and statistics math functions are available.
- Masked function allow operations that only change a few bits.
- Expressions can be used to perform more complex operations.
- Conversions are available for angles and BCD numbers.
- Array oriented file commands allow multiple operations in one scan.
plc basic functions - 15.26

• Values can be compared to make decisions.
• Boolean functions allow bit level operations.
• Function change value in data memory immediately.

15.6 PRACTICE PROBLEMS

1. Do the calculation below with ladder logic,
   \[ N7:2 = -(5 - N7:0 / N7:1) \]

2. Implement the following function,
   \[ x = \frac{\log(y) + \log(y)}{y + 1} \]

3. A switch will increment a counter on when engaged. This counter can be reset by a second switch. The value in the counter should be multiplied by 5, and then displayed as a binary output using (O:000)

4. Create a ladder logic program that will start when input \( A \) is turned on and calculate the series below. The value of \( n \) will start at 0 and with each scan of the ladder logic \( n \) will increase by 2 until \( n = 20 \). While the sequence is being incremented, any change in \( A \) will be ignored.
   \[ x = 2(\log(n) - 1) \]
   \[ A = B3/10 \]
   \[ n = N7:0 \]
   \[ x = F8:15 \]

5. The following program uses indirect addressing. Indicate what the new values in memory will be when button A is pushed after the first and second instructions.

<table>
<thead>
<tr>
<th>A</th>
<th>ADDR 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>ADDR 2</td>
</tr>
<tr>
<td>addr before</td>
<td>after 1st</td>
</tr>
<tr>
<td>N7:0</td>
<td>1</td>
</tr>
<tr>
<td>N7:1</td>
<td>2</td>
</tr>
<tr>
<td>N7:2</td>
<td>3</td>
</tr>
</tbody>
</table>
6. A thumbwheel input card acquires a four digit BCD count. A sensor detects parts dropping down a chute. When the count matches the BCD value the chute is closed, and a light is turned on until a reset button is pushed. A start button must be pushed to start the part feeding. Develop the ladder logic for this controller. Use a structured design technique such as a state diagram.

<table>
<thead>
<tr>
<th>INPUT</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>I:000 - BCD input card</td>
<td>O:002/00 - chute open</td>
</tr>
<tr>
<td>I:001/00 - part detect</td>
<td>O:002/01 - light</td>
</tr>
<tr>
<td>I:001/01 - start button</td>
<td></td>
</tr>
<tr>
<td>I:001/02 - reset button</td>
<td></td>
</tr>
</tbody>
</table>

7. Describe the difference between incremental, all and a number for file oriented instruction, such as FA L.

8. What is the maximum number of elements that moved with a file instruction? What might happen if too many are transferred in one scan?

9. Write a ladder logic program to do the following calculation. If the result is greater than 20.0, then the output O:012/15 will be turned on.

\[ A = D - Be^{T/C} \]

where,
\[ A = F8:0 \]
\[ B = F8:1 \]
\[ C = F8:2 \]
\[ D = F8:3 \]
\[ T = F8:4 \]

10. Write ladder logic to reset an RTO counter (T4:0) without using the RES instruction.

11. Write a program that will use Boolean operations and comparison functions to determine if bits 9, 4 and 2 are set in the input word I:001. If they are set, turn on output bit O:000/4.

12. Explain how the mask works in the following MVM function. Develop a Boolean equation.

<table>
<thead>
<tr>
<th>MVM Source N7:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mask N7:1</td>
</tr>
<tr>
<td>Dest N7:2</td>
</tr>
</tbody>
</table>
15.7 PRACTICE PROBLEM SOLUTIONS

1. 

DIV
Source A N7:0
Source B N7:1
Dest N7:2

SUB
Source A 5
Source B N7:2
Dest N7:2

NEG
Source N7:2
Dest N7:2

2. 

LOG
Source F8:0
Dest F8:1

ADD
Source A F8:0
Source B F8:1
Dest F8:2

ADD
Source A F8:0
Source B 1
Dest F8:3

DIV
Source A F8:2
Source B F8:3
Dest F8:4

MUL
Source A F8:0
Source B F8:4
Dest F8:5

ATN
Source F8:3
Dest F8:6

y = F8:0
x = F8:6
3.

CTU
Counter C5:0
Preset 1234

RES
C5:0

MUL
Source A 5
Source B C5:0.ACC
Dest O:000

4.

A
active

A
active

A
active

EQ
Source A 20
Source B N7:0

MOV
Source -2
Dest N7:0

LEQ
Source A N7:0
Source B 20

ADD
Source A N7:0
Source B 2
Dest N7:0

LOG
Source N7:0
Dest F8:15

SUB
Source A F8:15
Source B 1
Dest F8:15

MUL
Source A F8:15
Source B 2
Dest F8:15
5.

<table>
<thead>
<tr>
<th>addr</th>
<th>before</th>
<th>after 1st</th>
<th>after 2nd</th>
</tr>
</thead>
<tbody>
<tr>
<td>N7:0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>N7:1</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>N7:2</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

6.
7. An incremental mode will do one calculation when the input to the function is a positive edge - goes from false to true. The all mode will attempt to complete the calculation in a single scan. If a number is used, the function will do that many calculations per scan while the input is true.

8. The maximum number is 1000. If the instruction takes too long the instruction may be paused and continued the next scan, or it may lead to a PLC fault because the scan takes too long.
9.  

```
NEG  
Source F8:4  
Dest F8:0  

DIV  
Source A F8:0  
Source B F8:2  
Dest F8:0  

XPY  
Source A 2.718  
Source B F8:0  
Dest F8:0  

MUL  
Source A F8:1  
Source B F8:0  
Dest F8:0  

SUB  
Source A F8:3  
Source B F8:0  
Dest F8:0  

GRT  
Source A F8:0  
Source B 20.0  

O:012/15  
```

10.  

```
reset  

MOV  
Source 0  
Dest T4:0.ACC  
```
11. Write a ladder logic program that will implement the function below, and if the result is greater than 100.5 then the output O:0.0/0 will be turned on.

12. Use an FAL instruction to average the values in N7:0 to N7:20 and store them in F8:0.

3. Write some simple ladder logic to change the preset value of a counter. When the input ‘A’ is active the preset should be 13, otherwise it will be 9.

4. The 16 input bits from I:000 are to be read and XORed with the inputs from I:001. The result is to be written to the output card O:002. If the binary pattern of the outputs is 1010 0101 0111 0110 then the output O:003/0 will be set. Write the ladder logic.

5. A machine ejects parts into three chutes. Three optical sensors (A, B and C) are positioned in each of the slots to count the parts. The count should start when the reset (R) button is pushed. The count will stop, and an indicator light (L) turned on when the average number of parts counted as 100 or greater.

**15.8 ASSIGNMENT PROBLEMS**

1. Write a ladder logic program that will implement the function below, and if the result is greater than 100.5 then the output O:0.0/0 will be turned on.

\[
X = 6 + Ae^B \cos(C + 5) \quad \text{where,}
\]
\[
A = N7:0 \\
B = F8:0 \\
C = N7:1 \\
X = F8:1
\]

2. Use an FAL instruction to average the values in N7:0 to N7:20 and store them in F8:0.

3. Write some simple ladder logic to change the preset value of a counter. When the input ‘A’ is active the preset should be 13, otherwise it will be 9.

4. The 16 input bits from I:000 are to be read and XORed with the inputs from I:001. The result is to be written to the output card O:002. If the binary pattern of the outputs is 1010 0101 0111 0110 then the output O:003/0 will be set. Write the ladder logic.

5. A machine ejects parts into three chutes. Three optical sensors (A, B and C) are positioned in each of the slots to count the parts. The count should start when the reset (R) button is pushed. The count will stop, and an indicator light (L) turned on when the average number of parts counted as 100 or greater.
6. Write ladder logic to calculate the average of the values from N10:0 to N10:99. The operation should start after a momentary contact push button A is pushed. The result should be stored in N7:0. If button B is pushed, all operations should be complete in a single scan. Otherwise, only ten values will be calculated each scan. (Note: this means that it will take 10 scans to complete the calculation if A is pushed.)

7. Write and simplify a Boolean equation that implements the masked move (MVM) instruction. The source is S, the mask is M and the destination is D.

8. a) Write ladder logic to calculate and store the binary sequence in integer memory starting at N7:0 up to N7:200 so that N7:0 = 1, N7:1 = 2, N7:2 = 4, N7:3 = 8, N7:4 = 16, etc. b) Will the program operate as expected?
16. ADVANCED LADDER LOGIC FUNCTIONS

Topics:
- Shift registers, stacks and sequencers
- Program control; branching, looping, subroutines, temporary ends and one shots
- Interrupts; timed, fault and input driven
- Immediate inputs and outputs
- Block transfer
- Conversion of State diagrams using program subroutines
- Design examples

Objectives:
- To understand shift registers, stacks and sequencers.
- To understand program control statements.
- To understand the use of interrupts.
- To understand the operation of immediate input and output instructions.
- To be prepared to use the block transfer instruction later.
- Be able to apply the advanced function in ladder logic design.

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 regis-
The 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 16.3 Buffers and Stack Types
Figure 16.4 FIFO Stack Instructions

The LIFO stack commands are shown in Figure 16.5. 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
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.

![Figure 16.6 A Single Cam in a Drum Sequencer](image)

As the cam rotates it makes contact with none, one, or two terminals, as determined by the depressions and rises in the rotating cam.

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 $0000000000001111b$ 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.
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.
Figure 16.8  A Sequencer For Traffic Light Control

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.
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 16.9 Sequencer Instruction Examples

These instructions are well suited to processes with a single flow of execution, such as traffic lights.
16.10. 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.

**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, \( N7:0 \) and \( 123 \). The subroutine (SBR) function receives these two arguments and puts them in \( N10:0 \) and \( N10:1 \). 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 \( N10:1 \) to the calling program where it is put in location \( N7:1 \). 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 \textit{FOR} and ends at the \textit{NXT} function. In this example there is an \textit{ADD} function that will add 1 to the value of \textit{N7:1}. So when this for-next statement is complete the value of \textit{N7:1} will be larger. Notice that the label number is the same in the \textit{FOR} and \textit{NXT}, 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 \textit{NXT} statement. When \( A \) is true, all 5 loops will be completed in a single program scan. If \( B \) is true the \textit{NXT} statement will no longer return the program scan to the \textit{FOR} 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.

\textit{Figure 16.11} Subroutines
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 $D$.

Figure 16.12 A For-Next Loop
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 the rung status.

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.

**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 the rung status.

**Figure 16.14** One Shot Instruction
16.3.2 Fault Detection and Interrupts

The PLC can be set up to run programs automatically using interrupts. This is rou-
tinely 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 S2 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 inter-
rupt 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 inter-
rupt 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 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 milliseconds. 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.
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 0100b, or input 02.) When the input goes positive 10 times the bit B3/100 will be set.
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 $N7:4$. 

*Figure 16.17  An Input Driven Interrupt*
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.
The normal operation of the PLC is:

- Fast [input scan]
  - Input values scanned

- Slow [ladder logic is checked]
  - Outputs are updated in memory only, as the ladder logic is scanned

- Fast [outputs updated]
  - Output values are updated to match values in memory

Figure 16.19  Input, Program and Output Scan

Figure 16.20 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.
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
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
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](image)

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 STA=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.
Figure 16.24  The Main Program for the State Diagram (Program File 2)

The ladder logic for each of the state subroutines is shown in Figure 16.25. These blocks of logic examine the transitions and change states as required. Note that state STB includes logic to give state C higher priority, by blocking A when C is active.
The arrangement of the subroutines in Figure 16.24 and Figure 16.25 could experience problems with *racing* 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 STA. 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 B3/0-STA will turn off the wait B3/10-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.

Figure 16.27 Subroutines for State STA to Prevent Racing
16.6 DESIGN CASES

16.6.1 If-Then

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

```java
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;
}
```

Solution:
16.6.2 Traffic Light

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 second for both yellow lights. Use either stacks or sequencers.

Solution: The sequencer is the best solution to this problem.
16.7 SUMMARY

- 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.
16.8 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.

<table>
<thead>
<tr>
<th>time sequence in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>O:000/00 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16</td>
</tr>
<tr>
<td>O:000/00 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1</td>
</tr>
<tr>
<td>O:000/01 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0</td>
</tr>
<tr>
<td>O:000/02 1 0 0 1 0 0 0 0 0 1 1 0 0 0 1 0 0 0</td>
</tr>
<tr>
<td>O:000/03 0 0 0 0 1 0 0 0 0 0 1 0 1 0 0 0 1 0</td>
</tr>
<tr>
<td>O:000/04 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>O:000/05 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>O:000/06 0 0 0 0 0 1 1 0 0 0 0 0 1 0 0 0 0 0</td>
</tr>
<tr>
<td>O:000/07 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

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.

12. Implement the following state diagram using subroutines.
16.9 PRACTICE PROBLEM SOLUTIONS

1.

T4:0/DN

TON
Timer T4:0
Delay 4s

BSR
File B3:0
Control R6:0
Bit address R6:0/UL
Length 10

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

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

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

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

BSR
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)
2.

N7:0 = 0000 0000 0000 0000
N7:1 = 0000 0000 0000 0110
N7:2 = 0000 0000 0001 0000
N7:3 = 0000 0000 0001 0000
N7:4 = 0000 0000 0000 0100
N7:5 = 0000 0000 0000 1000
N7:6 = 0000 0000 0100 0000
N7:7 = 0000 0000 0110 0000
N7:8 = 0000 0000 0000 0001
N7:9 = 0000 0000 1000 0000
N7:10 = 0000 0000 0000 0100
N7:11 = 0000 0000 0000 1100
N7:12 = 0000 0000 0000 0000
N7:13 = 0000 0000 0100 1000
N7:14 = 0000 0000 0000 0010
N7:15 = 0000 0000 0000 0100
N7:16 = 0000 0000 0000 1000
N7:17 = 0000 0000 0000 0001
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.

7. 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
8.

PROGRAM 2

S2:1/15 - first scan

MOV
Source 3
Dest S2:31

MOV
Source 30000
Dest S2:30

PROGRAM 3

I:000/0

L B3/37

9.

\[
\begin{align*}
T_1 &= S_1 \cdot B3/1 \\
T_2 &= S_2 \cdot B3/2 \\
T_3 &= S_2 \cdot \overline{B3/2} \\
T_4 &= S_3 \cdot T4:0/DN \\
T_5 &= S_5 \cdot T4:1/DN \\
T_6 &= S_1 \cdot I:000/3 \\
T_7 &= S_4 \cdot B3/0
\end{align*}
\]

\[
\begin{align*}
S_1 &= (S_1 + T_2 + T_5 + FS) \cdot \overline{T_1} \cdot \overline{T_6} \\
S_2 &= (S_2 + T_1 \cdot \overline{T_6} + T_4) \cdot \overline{T_2} \cdot \overline{T_3} \\
S_3 &= (S_3 + T_3) \cdot \overline{T_4} \\
S_4 &= (S_4 + T_6) \cdot \overline{T_7} \\
S_5 &= (S_5 + T_7) \cdot \overline{T_5}
\end{align*}
\]
S3

S5

B3/0

S2

EQU
SourceA N7:1
SourceB N7:2

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

O:001/0

O:001/1

B3/2

B3/1

MOV
Source 1
Dest N7:2

MOV
Source 2
Dest N7:2

MOV
Source 3
Dest 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
12.

File 2

| FS |

File 3

| ST0 |

| ST1 |

| ST2 |

| JSR File 3 |

File 4

| ST0 |

| ST1 |

| ST2 |

| JSR File 4 |

| JSR File 5 |

File 5

| ST1 |

| ST2 |

| RET |

| RET |

| RET |
16.10 ASSIGNMENT PROBLEMS

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 an interrupt driven 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 on an indicator light on the operator panel.

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.