Chapter 12  HANDLING DATA

Introduction

In this chapter are found a number of output commands that handle data. These are instructions that tend to be accepted with most PLC manufacturers as a core of instructions that accomplish a simple task and provide a useful function to the larger program.

These instructions include boolean instructions, instructions for handling simple math conversions, file manipulation instructions, queuing instructions, and instructions for bit shifting.

Many of these instructions were created after earlier PLCs reported some functions being performed again and again at the cost of a great amount of PLC code. For instance, the queuing operations of FIFO or LIFO could be created in PLC code using a number of other instructions. This became too confusing to the PLC programmer and the FIFO or LIFO instruction was provided as a result.

Some of the instructions came about as the result of the PLC emulating the microprocessor instruction set. Word-length boolean instructions such as AND and moving operations such as bit shifting were copied and installed in the PLC instruction set from similar microprocessor instructions.

The most important point of the chapter is that most instructions are given with an example from industry. Each of these examples has been programmed by many different programmers using techniques similar to the examples shown. Instructions such as these have been used to provide control of automation for a wide variety of industry and solve many complex problems.
Siemens Word Logic and Shift/Rotate Instructions

Instructions from Siemens are given first. They are word logic operations and shift/rotate instructions. The instructions are shown in the groups below and their operations are explained in the following pages.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>AND logic operation</td>
</tr>
<tr>
<td>OR</td>
<td>OR logic operation</td>
</tr>
<tr>
<td>XOR</td>
<td>EXCLUSIVE OR logic operation</td>
</tr>
<tr>
<td>INV</td>
<td>Create ones complement</td>
</tr>
<tr>
<td>DECO</td>
<td>Decode</td>
</tr>
<tr>
<td>ENCO</td>
<td>Encode</td>
</tr>
<tr>
<td>SEL</td>
<td>Select</td>
</tr>
<tr>
<td>MUX</td>
<td>Multiplex</td>
</tr>
<tr>
<td>DEMUX</td>
<td>Demultiplex</td>
</tr>
</tbody>
</table>

Fig. 12-1 Siemens Word Logic, Shift/Rotate Instructions

AND

You can use the AND logic operation instruction to combine the value at the IN1 input and the value at the IN2 input bit-by-bit by AND logic and query the result at the OUT output. When the instruction is executed, bit 0 of the value at the IN1 input and bit 0 of the value at the IN2 input are logically ANDed. The result is stored in bit 0 of the OUT output. The same logic operation is executed for all other bits of the specified values.

The number of inputs can be expanded in the instruction box. The added inputs are numbered in ascending order in the box. When the instruction is executed, the values of all available input parameters are combined with AND logic. The result is stored in the OUT output.

The result bit has the signal state 1 only when both of the bits in the logic operation also have signal state 1. If one of the two bits of the logic operation has the signal state 0, the corresponding result bit is reset.

The instruction is only executed if the signal state is 1 at the EN enable input. In this case, the ENO output also has the signal state 1.

If the signal state at the EN enable input is 0, the ENO enable output also has the signal state 0.
The following table shows how the instruction works using specific operand values:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Operand</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN1</td>
<td>Tag_Value1</td>
<td>0101 0101 0101 0101</td>
</tr>
<tr>
<td>IN2</td>
<td>Tag_Value2</td>
<td>0000 0000 0000 1111</td>
</tr>
<tr>
<td>OUT</td>
<td>Tag_Result</td>
<td>0000 0000 0000 0101</td>
</tr>
</tbody>
</table>

If operand Tag_IN has the signal state 1, the AND logic operation instruction is executed. The value of operand Tag_Value1 and the value of operand Tag_Value2 are ANDed. The result is mapped bit-for-bit and output in operand Tag_Result. The ENO output and the TagOut output are set to signal state 1.

**OR**

You can use the OR logic operation instruction to combine the value at the IN1 input and the value at the IN2 input bit-by-bit by OR logic and query the result at the OUT output. When the instruction is executed, bit 0 of the value at the IN1 input and bit 0 of the value at the IN2 input are logically ORed. The result is stored in bit 0 of the OUT output. The same logic operation is executed for all bits of the specified tags.

The number of inputs can be expanded in the instruction box. The added inputs are numbered in ascending order in the box. When the instruction is executed, the values of all available input parameters are combined with OR logic. The result is stored in the OUT output. The result bit has the signal state 1 when at least one of the two bits in the logic operation has the signal state 1. If both of the bits of the logic operation have signal state 0, the corresponding result bit is reset. The instruction is only executed if the signal state is 1 at the EN enable input. In this case, the ENO output also has the signal state 1.

If the signal state at the EN enable input is 0, the ENO enable output also has the signal state 0.

The following example shows how the instruction works:
If operand TagIn has the signal state 1, the OR logic operation instruction is executed. The value of operand Tag_Value1 and the value of operation Tag_Value2 are ORed. The result is mapped bit-for-bit and output in operand Tag_Result. The ENO output and the TagOut output are set to signal state 1.

**XOR**

You can use the EXCLUSIVE OR logic operation to combine the value at the IN1 input and the value at the IN2 input bit-by-bit by EXCLUSIVE OR logic and query the result at the OUT output.

When the instruction is executed, bit 0 of the value at the IN1 input and bit 0 of the value at the IN2 input are logically EXCLUSIVELY ORed. The result is stored in bit 0 of the OUT output. The same logic operation is executed for all other bits of the specified value.

The number of inputs can be expanded in the instruction box. The added inputs are numbered in ascending order in the box. When the instruction is executed, the values of all available input parameters are combined with EXCLUSIVE OR logic. The result is stored in the OUT output.

The result bit has the signal state 1 when one of the two bits in the logic operation has the signal state 1. If both of the bits of the logic operation have signal state 1 or 0, the corresponding result bit is reset.

The instruction is only executed if the signal state is 1 at the EN enable input. In this case, the ENO output also has the signal state 1.

If the signal state at the EN enable input is 0, the ENO enable output also has the signal state 0. The following example shows how the instruction works:

If operand TagIn has the signal state 1, the EXCLUSIVE OR logic operation instruction is executed. The value of operand Tag_Value1 and the value of operand Tag_Value2 are EXCLUSIVELY ORed. The result is mapped bit-for-bit and output in operand Tag_Result. The ENO output and the TagOut output are set to signal state 1.

**INV**

You can use the Create ones complement instruction to invert the signal state of the bits at the IN input. When the instruction is processed, the value at the IN input and a hexadecimal template (W#16#FFFF for 16-bit numbers or DW#16#FFFF FFFF for 32-bit numbers) are logically EXCLUSIVELY ORed. As a result, the signal state of the individual bits is inverted and sent to the OUT output.
The instruction is only executed if the signal state is 1 at the EN enable input. In this case, the ENO output also has the signal state 1.

If the signal state at the EN enable input is 0, the ENO enable output also has the signal state 0.

The following example shows how the instruction works:

![Diagram](image)

If operand TagIn has the signal state 1, then the Create ones complement instruction will be executed. The instruction inverts the signal state of the individual bits at the TagIn_Value input and writes the result to the TagOut_Value output. The ENO output and the TagOut output are set to signal state 1.

**DECO**

You can use the Decode instruction to set a bit in the output value specified by the input value. The Decode instruction reads the value at the IN input and sets the bit in the output value whose bit position corresponds to the read value. The other bits in the output value will be overwritten with zeroes. When the value at the IN input is greater than 31, a modulo-32 instruction is executed.

The Decode instruction is only started when the signal state at the EN enable input is 1. If no errors occur during execution, the ENO output also has the signal state 1.

If the signal state at the EN enable input is 0, the ENO enable output also has the signal state 0.

The following example shows how the instruction works:

![Diagram](image)

The following figure shows how the instruction works using specific operand values:

```
"TagIn_Value"  3
31 ... 16 15 ... 3 ... 0
"TagOut_Value" 0000 0000 0000 0000 0000 0000 0000 1000
```
If operand TagIn has the signal state 1, the Decode instruction is executed. The instruction reads bit number 3 from the value at the TagIn_Value input and sets the third bit in the value at the TagOut_Value output.

If the instruction is executed without errors, the ENO output has the signal state 1 and the TagOut output is set.

ENCO

You can use the Encode instruction to read the bit number of the least significant bit in the input value and to send it to the OUT output. The Encode instruction selects the least significant bit of the value at the IN input and writes its bit number to the tag in the OUT output.

The Encode instruction is only started when the signal state at the EN enable input is 1. If no errors occur during execution, the ENO output also has the signal state 1. If the signal state at the EN enable input is 0, the ENO enable output also has the signal state 0.

The following example shows how the instruction works:

![ENCO Diagram]

The following figure shows how the instruction works using specific operand values:

```
<table>
<thead>
<tr>
<th>TagIn_Value</th>
<th>31 ...</th>
<th>16, 15 ...</th>
<th>3 ... 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000</td>
<td>1111</td>
<td>0000</td>
</tr>
</tbody>
</table>

| TagOut_Value | 3       |
```

If operand TagIn has the signal state 1, the Encode instruction is executed. The instruction selects the least significant bit at the TagIn_Value input and writes bit position 3 to the tag in the TagOut_Value output.

If the instruction is executed without errors, the ENO output has the signal state 1 and the TagOut output is set.

SEL

Depending on a switch (G input), the Select instruction selects one of the IN0 or IN1 inputs and copies its content to the OUT output. When the G input has the signal state 0, the value at the IN0 input is moved. When the G input has the signal state 1, the value at the IN1 input is copied to the OUT output.
Execution of the instruction assumes the signal state 1 at the EN enable input and that the tags are of the same data type at all parameters. If the instruction is executed without errors, the ENO enable output also has the signal state 1.

The ENO enable output is reset when the EN enable input has the signal state 0 or errors occur during execution of the instruction.

The following example shows how the instruction works:

![Fig. 12-8 Siemens SELECT Instruction](image)

The following table shows how the instruction works using specific operand values:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Operand</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>TagIn_G</td>
<td>0</td>
</tr>
<tr>
<td>IN0</td>
<td>TagIn_Value0</td>
<td>W#16#0000</td>
</tr>
<tr>
<td>IN1</td>
<td>TagIn_Value1</td>
<td>W#16#FFFF</td>
</tr>
<tr>
<td>OUT</td>
<td>TagOut_Value</td>
<td>W#16#0000</td>
</tr>
</tbody>
</table>

If operand TagIn has the signal state 1, the Select instruction is executed. Based on the signal state at the TagIn_G input, the value at the TagIn_Value0 or TagIn_Value1 input is selected and copied to the TagOut_Value output. If the instruction is executed without errors, enable output ENO has the signal state 1 and the TagOut output is set.

**MUX**

You can use the Multiplex instruction to copy the content of a selected input to the OUT output. The number of selectable inputs of the instruction box can be expanded. The inputs are automatically numbered in the box. Numbering starts at IN0 and continues consecutively with each new input. You use the K parameter to define the input whose content is to be copied to the OUT output. If the value of the K parameter is greater than the number of available inputs, the content of the ELSE parameter is copied to the OUT output and the ENO enable output is assigned the signal state 0.

The Multiplex instruction can only be executed, when the tags in all inputs and in the OUT output have the same data type. The K parameter is an exception, since only integers can be specified for it.

The instruction is only executed if the signal state is 1 at the EN enable input. If no errors occur during execution, the ENO output also has the signal state 1.
The ENO enable output is reset if one of the following conditions is fulfilled:

- The EN enable input has the signal state 0.
- The value of the K parameter is greater than the number of available inputs.
- Errors occurred during execution of the instruction.

The following table shows how the instruction works using specific operand values:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Operand</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>Tag_Number</td>
<td>1</td>
</tr>
<tr>
<td>IN0</td>
<td>Tag_Value_0</td>
<td>DW#16#00000000</td>
</tr>
<tr>
<td>IN1</td>
<td>Tag_Value_1</td>
<td>DW#16#3E4A7D</td>
</tr>
<tr>
<td>ELSE</td>
<td>Tag_Value_2</td>
<td>DW#16#FFFF0000</td>
</tr>
<tr>
<td>OUT</td>
<td>Tag_Result</td>
<td>DW#16#3E4A7D</td>
</tr>
</tbody>
</table>

If operand Tag_Input has the signal state 1, the Multiplex instruction is executed. Depending in the value of operand Tag_Number, the value at the Tag_Value_1 input is copied and assigned to the operand at the Tag_Result output. If the instruction is executed without errors, the ENO and Tag_Output outputs are set.

**DEMUX**

You can use the Demultiplex instruction to copy the content of the IN input to a selected output. The number of selectable outputs can be expanded in the instruction box. The outputs are automatically numbered in the box. Numbering starts at OUT0 and continues consecutively with each new input. You use the K parameter to define the output to which the content of the IN input is to be copied. The other outputs are not changed. If the value of the K parameter is greater than the number of available outputs, then the content of the IN input will be copied to the ELSE parameter and the signal state 0 is assigned to the ENO enable output.

The Demultiplex instruction can only be executed if the tags in the IN input and in all outputs have the same data type. The K parameter is an exception, since only integers can be specified for it. The instruction is only executed if the signal state is 1 at the EN enable input. If no errors occur during execution, the ENO output also has the signal state 1.

The ENO enable output is reset if one of the following conditions is fulfilled:
• The `EN` enable input has the signal state 0.
• The value of the `K` parameter is greater than the number of available outputs.
• Errors occurred during execution of the instruction.

The following example shows how the instruction works.

```
<table>
<thead>
<tr>
<th>Tag_Input</th>
<th>DEMUX</th>
<th>Tag_Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DWORD</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EN</td>
<td>ENO</td>
</tr>
<tr>
<td></td>
<td>K</td>
<td>OUT0</td>
</tr>
<tr>
<td></td>
<td>IN</td>
<td>OUT1</td>
</tr>
<tr>
<td>ELSE</td>
<td>Tag_Output_0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Tag_Output_1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Tag_Output_2</td>
<td></td>
</tr>
</tbody>
</table>
```

Fig. 12-10 Siemens DEMUX Instruction

The following table shows how the instruction works using specific operand values:

**Input values of the Demultiplex instruction before network execution:**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Operand</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>K</code></td>
<td>Tag_Number</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td><code>IN</code></td>
<td>Tag_Value</td>
<td>DW#16#FFFFFFFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DW#16#3E4A7D</td>
</tr>
</tbody>
</table>

**Output values of the Demultiplex instruction after network execution:**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Operand</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>OUT0</code></td>
<td>Tag_Output_0</td>
<td>Unchanged</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unchanged</td>
</tr>
<tr>
<td><code>OUT1</code></td>
<td>Tag_Output_1</td>
<td>DW#16#FFFFFFFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unchanged</td>
</tr>
<tr>
<td><code>ELSE</code></td>
<td>Tag_Output_2</td>
<td>Unchanged</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DW#16#3E4A7D</td>
</tr>
</tbody>
</table>

If the `Tag_Input` input has the signal state 1, the Demultiplex instruction is executed. Depending on the value of operand `Tag_Number`, the value at the `IN` input is copied to the corresponding output.

**SHR**

You can use the Shift right instruction to shift the content of the operand at the `IN` input bit-by-bit to the right and query the result at the `OUT` output. You use the `N` parameter to specify the number of bit positions by which the specified value is to be shifted.

When the value at the `N` parameter is 0, the value at the `IN` input is copied to the operand at the `OUT` output. When the value at the `N` parameter is greater than the number of available bit positions, the operand value at the `IN` input is shifted by the available number of bit positions to the right.

In the case of unsigned values, the freed bit positions in the left area of the operand are filled with zeroes when shifting occurs. If the specified value has a sign, the free bit positions are filled with the signal state of the sign bit.
The following figure shows how the content of an operand of integer data type is shifted four bit positions to the right:

![Diagram showing shifting of integer operand](image)

The Shift right instruction is only executed if the signal state is 1 at the EN enable input. In this case, the ENO enable output also has the signal state 1.

If the signal state at the EN enable input is 0, the ENO enable output also has the signal state 0.

The following example shows how the instruction works:

![Figure 12-11 Siemens SHIFT RT Instruction](image)

The following table shows how the instruction works using specific operand values:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Operand</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>TagIn_Value</td>
<td>0011 1111 1010 1111</td>
</tr>
<tr>
<td>N</td>
<td>Tag_Number</td>
<td>3</td>
</tr>
<tr>
<td>OUT</td>
<td>TagOut_Value</td>
<td>0000 0111 1111 0101</td>
</tr>
</tbody>
</table>

If operand TagIn has the signal state 1, the Shift right instruction is executed. The content of operand TagIn_Value is shifted three bit positions to the right. The result is sent to the TagOut_Value output. If the instruction is executed without errors, the ENO output has the signal state 1 and the TagOut output is set.
SHL

You can use the Shift left instruction to shift the content of the operand at the IN input bit-by-bit to the left and query the result at the OUT output. You use the N parameter to specify the number of bit positions by which the specified value is to be shifted. When the value at the N parameter is 0, the value at the IN input is copied to the operand at the OUT output.

When the value at the N parameter is greater than the number of available bit positions, the operand value at the IN input is shifted by the available number of bit positions to the left. The bit positions in the right part of the operand freed by shifting are filled with zeros. The following figure shows how the content of an operand of WORD data type is shifted six bit positions to the left:

The Shift left instruction is only executed if the signal state is 1 at the EN enable input. In this case, the ENO enable output also has the signal state 1.

If the signal state at the EN enable input is 0, the ENO enable output also has the signal state 0.

The following example shows how the instruction works:

The following table shows how the instruction works using specific operand values:
### Parameters | Operand   | Value                   
-------------|------------|-------------------------
IN           | TagIn_Value| 0011 1111 1010 1111     
N            | Tag_Number | 4                       
OUT          | TagOut_Value| 1111 1010 1111 0000     

If operand TagIn has the signal state 1, the Shift left instruction is executed. The content of operand TagIn_Value is shifted four bit positions to the left. The result is sent to the TagOut_Value output. If the instruction is executed without errors, the ENO output has the signal state 1 and the TagOut output is set.

**ROR**

The Rotate right instruction rotates the content of the operand at the IN input bit-by-bit to the right and queries the result at the OUT output. You use the N parameter to specify the number of bit positions by which the specified value is to be rotated. The bit positions freed by rotating are filled with the bit positions that are pushed out.

When the value at the N parameter is 0, the value at the IN input is copied to the operand at the OUT output.

When the value at the N parameter is greater than the number of available bit positions, the operand value at the IN input is nevertheless rotated by the specified number of bit positions.

The following figure shows how the content of an operand of DWORD data type is rotated three positions to the right:

![Diagram of DWORD rotation](image)

The Rotate right instruction is only executed if the signal state is 1 at the EN enable input. In this case, the ENO enable output also has the signal state 1.

If the signal state at the EN enable input is 0, the ENO enable output also has the signal state 0. The following example shows how the instruction works:
The following table shows how the instruction works using specific operand values:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Operand</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>TagIn_Value</td>
<td>0011 1111 1001 0101</td>
</tr>
<tr>
<td>N</td>
<td>Tag_Number</td>
<td>5</td>
</tr>
<tr>
<td>OUT</td>
<td>TagOut_Value</td>
<td>1010 1000 0111 1100</td>
</tr>
</tbody>
</table>

If operand TagIn has the signal state 1, the Rotate right instruction is executed. The content of operand TagIn_Value is rotated five bit positions to the right. The result is sent to the TagOut_Value output. If the instruction is executed without errors, the ENO output has the signal state 1 and the TagOut output is set.

**ROL**

The Rotate left instruction rotates the content of the operand at the IN input bit-by-bit to the left and queries the result at the OUT output. You use the N parameter to specify the number of bit positions by which the specified value is to be rotated. The bit positions freed by rotating are filled with the bit positions that are pushed out.

When the value at the N parameter is 0, the value at the IN input is copied to the operand at the OUT output.

When the value at the N parameter is greater than the number of available bit positions, the operand value at the IN input is nevertheless rotated by the specified number of bit positions. The following figure shows how the content of an operand of DWORD data type is rotated three positions to the left:

![Diagram showing Rotate left instruction](image-url)
The Rotate left instruction is only executed if the signal state is 1 at the EN enable input. In this case, the ENO enable output also has the signal state 1.

If the signal state at the EN enable input is 0, the ENO enable output also has the signal state 0.

The following example shows how the instruction works:

The following table shows how the instruction works using specific operand values:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Operand</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>TagIn_Value</td>
<td>1010 1000 1111 0110</td>
</tr>
<tr>
<td>N</td>
<td>Tag_Number</td>
<td>5</td>
</tr>
<tr>
<td>OUT</td>
<td>TagOut_Value</td>
<td>0001 1110 1101 0101</td>
</tr>
</tbody>
</table>

If the TagIn input has the signal state 1, the Rotate left instruction is executed. The content of operand TagIn_Value is rotated five bit positions to the left. The result is sent to the TagOut_Value output. If the instruction is executed without errors, the ENO output has the signal state 1 and the TagOut output is set.
Allen-Bradley Logical Instructions

When enabled, the AND instruction performs a bitwise AND operation on SourceA and SourceB and places the result in Dest.

OR

When enabled, the OR instruction performs a bitwise OR operation on SourceA and SourceB and places the result in dest.
XOR

When enabled, the XOR instruction performs a bitwise XOR operation on SourceA and SourceB and places the result in the destination tag.

NOT

When enabled, the NOT instruction performs a bitwise NOT operation on value_1 and places the result in value_result_not.
Swap Byte – SWPB

The three SWPB instructions each reorder the bytes of DINT_1 according to a different order mode. The display style is ASCII, and each character represents one byte. Each instruction places the bytes, in the new order, in a different Destination.

Example 2: Swap the bytes in all elements of an array

The following example reverses the bytes in each element of an array.

1. Initialize the tags.
   The SIZE instruction finds the number of elements in array and stores that value in array_length. A subsequent instruction uses this value to determine when the routine has acted on all the elements in the array.

2. Reverse the bytes in one element of array.
   The SWPB instruction reverses the bytes of the element number that is indicated by the value of index. For example, when index equals 0, the SWPB instruction acts on array[0].

   The ADD instruction increments index. The next time the rung executes, the SWPB instruction will act on the next element in array.
3. Determine when the SWPB instruction has acted on all the elements in the array:
   If index is less than the number of elements in the array (array_length), then execution
   jumps to the rung that has the swap_bytes label (the previous rung). This lets the routine
   act on the next element in the array.

   If index equals the number of elements in the array (array_length), then the SWPB has acted on all
   the elements in the array. Execution goes to the next rung.

---

**Fig. 12-20** A-B Swap Byte Example

**Clear – CLR**

Clear all the bits of value to 0.
**Bit Field Distribute (BTD)**

When enabled, the BTD instruction copies a group of bits from the Source to the Destination. The group of bits is identified by the Source bit (lowest bit number of the group) and the Length (number of bits to copy). The Destination bit identifies the lowest bit number bit to start with in the Destination. The Source remains unchanged. If the length of the bit field extends beyond the Destination, the instruction does not save the extra bits. Any extra bits do not wrap to the next word.

If you mix integer data types, the instruction fills the upper bits of the smaller integer data types with 0's so that they are the same size as the largest data type.

---

**Example 1**

When enabled, the BTD instruction moves bits within value_1.

**Fig. 12-22a** A-B Bit Field Distribution (BTD) Instruction

**Example 2**

When enabled, the BTD instruction moves 10 bits from value_1 to value_2.

**Fig. 12-22b** A-B Bit Field Distribution (BTD) Instruction
The shaded boxes show the bits that changed in value_2.

**Allen-Bradley Array/Shift Instructions**

**Bit Shift Left (BSL)**

The BSL instruction shifts the specified bits within the Array one position left.

When enabled, the BSL instruction starts at bit 0 in array_dint[0]. The instruction unloads array_dint[0].9 into the .UL bit, shifts the remaining bits, and loads input_1 into array_dint[0].0. The remaining bits (10-31) are invalid.
Example 2

When enabled, the BSL instruction starts at bit 0 in array_dint[0]. The instruction unloads array_dint[1].25 into the .UL bit, shifts the remaining bits, and loads input_1 into array_dint[0].0. The remaining bits (31-26) in array_dint[1] are invalid.

Bit Shift Right (BSR)

The BSR instruction shifts the specified bits within the Array one position right.

Example 1

When enabled, the BSR instruction starts at bit 9 in array_dint[0]. The instruction unloads array_dint[0].0 into the .UL bit, shifts the remaining bits right, and loads input_1 into array_dint[0].9. The remaining bits (10-31) are invalid.

Example 2

When enabled, the BSR instruction starts at bit 25 in array_dint[1]. The instruction unloads array_dint[0].0 into the .UL bit, shifts the remaining bits right, and loads input_1 into array_dint[1].25. The remaining bits (31-26 in dint_array[1]) are invalid. Note how array_dint[1].0 shifts across words into array_dint[0].31.
**FIFO Load (FFL)**

**Ladder Diagram**

Fig. 12-24b  A-B Bit Shift Right (BSR) Instruction

These bits shift left

**array_dint[0]**

```
011110000111000011100001110000
```

These bits shift

**array_dint[1]**

```
1000000000111000011100001110000
```

These bits shift

**input_1**

---

Fig. 12-25  A-B This FFL instruction copies the Source value to the FIFO.

When enabled, the FFL instruction loads value_1 into the next available position in the FIFO, which is array_dint[5] in this example.
**FIFO Unload (FFU)**

The FFU instruction unloads the value from position 0 (first position) of the FIFO and stores that value in the Destination. The remaining data in the FIFO shifts down one position.

![Ladder Diagram](image)

**LIFO Load (LFL)**

The LFL instruction copies the Source value to the LIFO.

![Ladder Diagram](image)
LIFO Unload (LFU)

The LFL instruction unloads the value at .POS of the LIFO and stores 0 in that location.
Queueing Operations

If a queueing operation is needed in a process, consider using the FIFO or LIFO instructions. FIFO is short for First in, First out. LIFO is short for Last in, First out. Both refer to queueing of a part or entity for use in a stage of a machine. For instance, consider the following batching system using the FIFO instruction:

Start with the Mix System empty and all Process Tanks A-F full:

The system above consists of 6 processes that are filled separately with a liquid made in the Mix System tank. Tanks A-F call for a new batch of liquid when down to ¼ full. If the Mix System is in the process of making a batch for a particular tank, the tank that just fell less than ¼ full must wait for its batch to be made. Since batches are unique for each Tank A-F, the system must completely drain, go through a flush cycle, and then start mixing for the next tank. Multiple tanks may call for a batch but no tank may call more than once.

After some time passes, the tank levels lower due to the demand from the processes needing the liquids. The first one to fall to its low level switch calls for a new batch to be made. That tank’s request is entered into the FIFO stack and immediately taken from the FIFO stack to the recipe program since the Mix System tank is not active.
Tank F requests a batch be made. Tank F’s request enters the FIFO stack. Since the batching program is not running, Tank F’s request is immediately acted on by the batching program and a new batch starts in the Mix System for Tank F. The stack is emptied.

A mix is being made for Tank F. Other tanks continue to drop but no tanks have called for a new mix. The Stack continues to remain empty.
While the batch is being mixed for F, two other tanks reach their low tank limit and enter the stack. They enter in the order that they reached their low limit. The following shows the stack with Tank B and Tank D in the same order as the tanks went low.

As soon as the batch for Tank F is made and emptied, the request for Tank B is taken using the FIFO unload and the mix system starts making a batch for Tank B. Tank D moves to the top of the stack.
While the process continues supplying mix to each tank as needed, the FIFO stack will continue to control the queueing operation. The FIFO controls the operation.
An Aside on Simulation

The various scenarios on queuing can be planned and implemented and the process not run at all as planned. It cannot always be predicted when a bottleneck will appear so care must be taken before the tanks are designed and the process is built to predict whether the various processes will work as advertised. It is not acceptable to build the process only to tear it out and add equipment in a few months based on observed bottlenecks. Computer simulation may be a necessary component of the design of a process. It may be required to size equipment and predict flows in order to maximize the manufacturer’s investment.

With a good software simulation package, the user can adequately predict flow rates and equipment throughput efficiently enough to size the equipment for the process. Not all processes must be simulated but those with queues in them are more apt to need a good simulation than other types of processes.

The simulation becomes an experiment in which as much as is known about the process is introduced and the various parts are combined to run a model of the process. The model is constructed of elements that act the role of machine parts and the final product is assumed to contain all elements of the process involved. Statistics play a role in the successful simulation. If a rate is not known, a random variable may be substituted. If a rate is known, it is used in the simulation. Better than a predictable rate is a rate and a statistical distribution of the rate. With this information, a process may be seen to not fail under most circumstances but if a number of entities in the model have statistically long manufacturing times or high fail rates, the process may react differently than at the average. If this is the case, bottlenecks may spoil the overall throughput and cause the engineer to re-think the design.

The use of sophisticated simulation software with its statistical prediction algorithms will predict bottlenecks not necessarily predicted by a simpler analysis.

Using common sense is also a tool to be used. For example, if the loading rate of a part can be determined, the operator should not be allowed to make more of that product per unit of time than can be off-loaded from a manufacturing line. If rules such as this are not demanded in the control strategy, chaos is as predictable as snow in Toledo in January. So, the best plan of a designed queuing strategy may fall apart if the process is not adequately designed and perhaps a software simulation tried to predict failure and future bottlenecks. After-the-fact programming on the PLC cannot compensate for a design failure in this area. Do not be surprised if you are asked to try, however, when this happens.

Books on the theory of queuing may also be explored but, unless the process is very simple, the process is better served with a simulation.
Application Specific Instructions

The other instruction type is BSL and BSR. These instructions will be reviewed here. Although similar to other Data Handling Instructions, they are used for a specific type of application.

BSL and BSR instructions are used primarily as shift register functions, tracking parts down conveyors.

Definition of BSL and BSR (from the Reference Manual):

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSL</td>
<td>Bit Shift Left</td>
<td>Loads a bit of data into a bit array, shifts the pattern of data through the array, and unloads the last bit of data in the array. BSL shifts data to the left.</td>
</tr>
<tr>
<td>BSR</td>
<td>Bit Shift Right</td>
<td>Loads a bit of data into a bit array, shifts the pattern of data through the array, and unloads the last bit of data in the array. BSR shifts the data to the right.</td>
</tr>
</tbody>
</table>

An Application using BSL or BSR:

If a conveyor is installed in a system, the BSL or BSR is used to track pieces on the conveyor (if the piece does not slide or scoot on the conveyor).

To move the bits through the shift register requires an input pulse signal. Bits are shifted with each leading edge of the BSR or BSL. For example, if an input watched the sprocket of a conveyor as the conveyor turned, the input would pulse on and off as the conveyor’s sprocket rotated. Proximity switches work well for this application. Bits in the shift register move each time the input from the proximity switch activates the BSR or BSL instruction.
With each new leading edge of the proximity switch signal, the bits shift one place in the shift register.

Each type of product needing to be tracked needs a separate shift register. In the example above, in order to track yellow boxes separately from pink boxes, the bit present in one shift register must represent yellow boxes. Same position bits in a second shift register must represent pink boxes.

Input into the shift register of 1’s come from the Bit Address. Usually this address is an input address but it may represent the result of logic as well. For instance, a photo-eye is usually the device used to see a box prior to introduction onto a conveyer. If the box is bar-coded, entrance into a shift register requires the successful read of a bar code and the photo-eye seeing the product. Both must be present to turn on the input to the Bit Address.

The shift register works equally well with paint applications. Moving parts down a conveyer to paint poses the requirement that no sensor can be placed immediately in the paint area since paint can cover and make inoperable most types of presence sensors. Parts must be tracked to the paint area and painted with various colors based on the process requirement. For instance, pink parts are painted at station A and yellow parts at station B. If a part is to be painted pink, it is entered in the pink shift register. If the part is to be painted yellow, it is entered in the yellow shift register.

Pink Part Shift Register

Movement of bits ➔ Bit moves from bit 0 of word 0 to bit 15 of word 1.

<table>
<thead>
<tr>
<th>0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0</th>
<th>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

Yellow Part Shift Register

| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 |
|----------------------------------|----------------------------------|
| 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
Paint Application:

Parts on Chain:
Movement of parts

Fig. 12-32 Multiple Types of Parts Tracked through Spray Booth

Pink Spray Shift Register

| B3:4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| B3:5 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| B3:6 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| B3:7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Yellow Spray Shift Register

| B3:8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| B3:9 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| B3:10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| B3:11 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
Logic for Shift Registers

I: Move bits with pulse

I: B3:4/15
Introduce pink part to shift register

I: B3:8/15
Introduce yellow part to shift register

B3:6/8
Spray Pink (may include multiple bits)

B3:10/2
Spray Yellow (may include multiple bits)

Logic from above is used to control the output of the shift register and control down-stream operations in a tracking application.

While the shift right (or left) instruction is done for many tracking applications, it should be considered for only the simplest applications. For more involved applications, an algorithmic approach including a FIFO table of leading edges with a counter to index through a zone should be used. The approach should include a pointer to a recipe of choices in which the recipe includes specific control bits to turn on or off an output when the parts’ leading edge passes an action point. Since shift instructions take significant time to execute, the algorithm must be considered where multiple zones are involved and multiple parts are included in any specific zone. Pulse tachometers can be used that are significantly faster than the scan of the PLC with no loss of motion with the more advanced algorithm. The shift instruction depends on
Which SLC

With a number of different applications being presented, it is appropriate to ask the question as to which language to use. The title above “Which SLC” attempts to ask a question as to the programming languages of A-B and Siemens commonly referred to as SLC or Sequential Logic. The programming languages are procedural instead of object-oriented. The two languages discussed to date are object-oriented. You may be interested to discuss how these languages fit the object-oriented category if desired. The language introduced below is a procedural language and looks very much like “C” or Visual Basic. The languages are useful in looping or iterative applications and may be useful as well in a number of other control schemes.

What other SLC are we referring to? The SLC processor of Allen-Bradley is a favorite choice. These processors formed the back-bone of A-B market share for years and still are useful for the addressing schemes using B3’s and N7’s. You are familiar with them since you have been using the nomenclature in many examples in the text.

The examples below are from Siemens but the A-B equivalent language is very similar and supports many of the same procedural operations.

Siemens SLC Language

The instructions shown here are part of the SLC or procedural language from Siemens’ S7-1200.
Explanations of some of the more important SLC instructions with examples are given below:

**IF: Run conditionally**

**Description**

The instruction "Run conditionally" branches the program flow depending on a condition. The condition is an expression with Boolean value ((TRUE or FALSE). Logical expression or comparative expressions can be stated as conditions.

When the instruction is executed, the stated expressions are evaluated. If the value of an expression is TRUE, the condition is fulfilled; if the value is FALSE, it is not fulfilled.

**Syntax**

Depending on the type of branch, you can program the following forms of the instruction:

- **Branch through IF:**

  \[
  \text{IF <Condition> THEN <Instructions>}
  \]

  \[
  \text{END_IF}
  \]
If the condition is satisfied, the instructions programmed after the THEN are executed. If the condition is not satisfied, the execution of the program continues with the next instruction after the END_IF.

- Branch through IF and ELSE:

```
IF <Condition> THEN <Instructions1>
ELSE <Instructions0>;
END_IF
```

If the condition is satisfied, the instructions programmed after the THEN are executed. If the condition is not satisfied, the instructions programmed after the ELSE are executed. Then the execution of the program continues with the next instruction after the END_IF.

- Branch through IF, ELSIF and ELSE:

```
IF <Condition1> THEN <Instructions1>
ELSIF <Condition2> THEN <Instruction2>
ELSE <Instructions0>;
END_IF;
```

If the first condition (<Condition1>) is satisfied, the instructions (<Instructions1>) after the THEN are executed. After execution of the instructions, the execution of the program continues after the END_IF.

If the first condition is not satisfied, the second condition (<Condition2>) is checked. If the second condition (<Condition2>) is fulfilled, the instructions (<Instruction2>) after the THEN are executed. After execution of the instructions, the execution of the program continues after the END_IF.

If none of the conditions are fulfilled, the instructions (<Instructions0>) after ELSE are executed followed by the execution of the program after END_IF.

You can nest as many combinations of ELSIF and THEN as you like within the IF instruction. The programming of an ELSE branch is optional.
Example

The following example shows how the instruction works:

IF "Tag_1" = 1 THEN "Tag_Value" := 10;
ELSIF "Tag_2" = 1 THEN "Tag_Value" := 20;
ELSIF "Tag_3" = 1 THEN "Tag_Value" := 30;
ELSE "Tag_Value" := 0;
END_IF;

FOR: Run in counting loop

Description

The instruction "Run in counting loop" causes repeated execution of a program loop until a run variable lies within a specified value range.

Program loops can also be nested. Within a program loop, you can program additional program loops with other run variables.

The current continuous run of a program loop can be ended by the instruction "Recheck loop condition" (CONTINUE). The instruction "Exit loop immediately" (EXIT) ends the entire loop execution. For additional information on this topic refer to "See also."

Syntax

The following syntax is used for the "Run in counting loop" instruction:

FOR <Run_tag> := <Start_value> TO <End_value> BY <Increment> DO <Instructions>
END_FOR

Example

The following example shows how the instruction works:

FOR i := 2 TO 8 BY 2
DO "a_array[i] := "Tag_Value"*"b_array[i]";

END_FOR;

The operand "Tag_Value" is multiplied with the elements (2, 4, 6, 8) of the ARRAY tag "b_array". The result is read in to the elements (2, 4, 6, 8) of the ARRAY tag "a_array".

WHILE: Run if condition is met

Description

The instruction "Run if condition is met" causes a program loop to be repeatedly executed until the implementation condition is satisfied. The condition is an expression with Boolean value ((TRUE or FALSE). Logical expression or comparative expressions can be stated as conditions.

When the instruction is executed, the stated expressions are evaluated. If the value of an expression is TRUE, the condition is fulfilled; if the value is FALSE, it is not fulfilled.

Program loops can also be nested. Within a program loop, you can program additional program loops with other run variables.

The current continuous run of a program loop can be ended by the instruction "Recheck loop condition" (CONTINUE). The instruction "Exit loop immediately" (EXIT) ends the entire loop execution. For additional information on this topic refer to "See also."

Syntax

The following syntax is used for the "Run if condition is met" instruction:

WHILE <Condition> DO <Instructions>

END_WHILE

Example

The following example shows how the instruction works:

WHILE "Tag_Value1" <> "Tag_Value2"

DO "Tag_Result" := "Tag_Input";

END_WHILE;
As long as the values of the operands "Tag_Value1" and "Tag_Value2" do not match, the value of the operand "Tag_Input" is allocated to the operand "Tag_Result".

EXIT: Exit loop immediately

Description

The instruction "Exit loop immediately" cancels the execution of a FOR, WHILE or REPEAT loop at any point regardless of conditions. The execution of the program is continued after the end of the loop (END_FOR, END_WHILE, END_REPEAT).

The instruction affects the program loop which directly contains the instruction.

Syntax

The following syntax is used for the "Exit loop immediately" instruction:

EXIT

Example

The following example shows how the instruction works:

FOR i := 15 TO 1 BY -2 DO
IF (i < 5) THEN EXIT;
END_IF;

"DB10".Test[i] := 1;
END_FOR;

For additional information on valid data types, refer to "See also".

If the condition i < 5 is satisfied, then the execution of the loop will be cancelled. Program execution resumes after the END_FOR.

If the condition i < 5 is not satisfied, the subsequent value allocation ("DB10".Test[i] := 1) is executed and a new loop is started. The run tag (i) is decreased by the increment of "-2" and it is checked whether its current value lies in the programmed value range. If the (i) run variable lies within the value range, the IF condition is evaluated again.
Example of Procedural Tracking (Similar to Shift Register)

The use of a shift register to track parts on a conveyor is a very good visual way to track items. However, it may not be efficient and capable of more sophisticated tracking problems. If there are a number of different parts being tracked and there are a number of parts in an area, then consider using a table similar to the one below and track the part and its position in a table. If the part has an action that is configured for a position in the table, the program can ask whether the part is to have the particular action take place or not. The sophistication of tracking multiple parts and keeping close track of each part lends itself to advanced tracking more sophisticated to either state diagram tracking or shift register tracking and with a small amount of computer code that can be very efficiently executed.

<table>
<thead>
<tr>
<th>Part one’s Pattern</th>
<th>Part one’s Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part two’s Pattern</td>
<td>Part two’s Position</td>
</tr>
<tr>
<td>empty</td>
<td>empty</td>
</tr>
<tr>
<td>empty</td>
<td>empty</td>
</tr>
<tr>
<td>empty</td>
<td>empty</td>
</tr>
</tbody>
</table>

The tracking program of the parts above is more sophisticated than the tracking using shift registers described earlier. Either may be chosen and the programmer must decide which is best in a specific case.
Summary

This chapter explains a number of data handling instructions and provides applications for their use in factory automation. Instructions are provided for a number of operations that would otherwise have required a significant programming effort to provide. Included are the queuing operations using FIFO instructions. Other interesting instructions in data handling include the bit-shift instructions. These instructions are used for shift register part tracking. Instructions used in data manipulation were shown with examples included for each type of instruction.

Examples of different tracking options are discussed and the methods of programming are examined.

Exercises:

1. Discuss alternative methods of writing tracking programs other than the shift register. Which would you prefer? Under what conditions would you prefer the alternative? What advantages/disadvantages does the shift register approach have?

2. Why would an instruction called “SEL” or select be useful in a PLC? How else could the programmer execute this operation if the instruction were not present?
Lab 12.1

Set up the Shift Register for tracking two different parts similar to the example explained in the chapter. Use inputs for introducing the two different boxes into two different shift registers and the prox switch. You can also use a timer to introduce pulses for a simulated prox switch. Make a variable distance that can be inserted at a location to vary the point of extraction. Turn on an output to signal an extractor solenoid turning on.
Lab 12.2

Set up the shift register for tracking multiple parts on a conveyor using SCL similar to the application described in Fig. 12-35.

Lab 12.3

Set up the queuing operation similar to the mix tank example in the chapter. Demonstrate a tank level switch going low and the subsequent operation of the mix tank to make a mix, drain it to the appropriate tank and continue with the next queued operation.