



*Allen-Bradley*

*1336 FORCE™  
PLC®  
Communications  
Adapter*

*(Cat. No. 1336T-GT1EN)*

# Function Block Programming Manual



## Important User Information

Solid state equipment has operational characteristics differing from those of electromechanical equipment. “Safety Guidelines for the Application, Installation and Maintenance of Solid State Controls” (Publication SGI-1.1) describes some important differences between solid state equipment and hard-wired electromechanical devices. Because of this difference, and also because of the wide variety of uses for solid state equipment, all persons responsible for applying this equipment must satisfy themselves that each intended application of this equipment is acceptable.

In no event will the Allen-Bradley Company be responsible or liable for indirect or consequential damages resulting from the use or application of this equipment.

The examples and diagrams in this manual are included solely for illustrative purposes. Because of the many variables and requirements associated with any particular installation, the Allen-Bradley Company cannot assume responsibility or liability for actual use based on the examples and diagrams.

No patent liability is assumed by Allen-Bradley Company with respect to use of information, circuits, equipment, or software described in this manual.

Reproduction of the contents of this manual, in whole or in part, without written permission of the Allen-Bradley Company is prohibited.

Throughout this manual we use notes to make you aware of safety considerations.



**ATTENTION:** Identifies information about practices or circumstances that can lead to personal injury or death, property damage, or economic loss.

---

Attentions help you:

- identify a hazard
- avoid the hazard
- recognize the consequences

**Important:** Identifies information that is especially important for successful application and understanding of the product.

# Table of Contents

---

## Preface

Manual Overview .....	P-1
Product Overview .....	P-1
Terminology .....	P-3
Function Block Components .....	P-4

## Getting Started

### Chapter 1

Chapter Objectives .....	1-1
Sawtooth Application Operation .....	1-2
Getting Started Using DriveBlockEditor .....	1-3
Getting Started Using a PLC .....	1-15

## System Component Detail

### Chapter 2

Chapter Objectives .....	2-1
Execution List Overview .....	2-1
Creating an Execution List .....	2-4
Adding Events to the Execution List .....	2-4
NO-OP Events .....	2-4
Example Execution Lists .....	2-5
Linking Events .....	2-5
Link Operation During Execution .....	2-6
Deleting Events from the Execution List .....	2-7
Downloading and Compiling the Execution List .....	2-8
Understanding Function Block I/O Nodes .....	2-9
DriveBlockEditor Node References .....	2-10
Understanding PLC and Drive Node References .....	2-10
Examples of Function Block I/O Node References .....	2-11
Node Data Types .....	2-12
Creating Links Between Nodes .....	2-12

## System Interactions

### Chapter 3

Chapter Objectives .....	3-1
The Function Block BRAM Functions .....	3-1
The Function Block Init Command .....	3-2
The Function Block Store Command .....	3-2
The Function Block Recall Command .....	3-3
Linear Parameter BRAM Functions and Links .....	3-3
Power Up Sequence .....	3-5
Compiler Modes and Terminal Operation Differences .....	3-6
Compiler Modes .....	3-6
Initial Compile Mode .....	3-6
Subsequent Compile Mode .....	3-7

DriveTools' DriveBlockEditor Download and Compile Operation . . . .	3-9
Graphic Programming Terminal . . . . .	3-10
PLC Block Transfer . . . . .	3-10
Understanding Multiple Execution List Copies . . . . .	3-10
Task Status Service . . . . .	3-11
Link Processing Faults . . . . .	3-12
Performance Issues Involving Links . . . . .	3-13
Link Processing Sequence . . . . .	3-14

## Function Block Library

### Chapter 4

Chapter Objectives . . . . .	4-1
Function Block Overview . . . . .	4-1
Double Word Function Block Caution . . . . .	4-2
Function Block Index . . . . .	4-3
BIN2DEC . . . . .	4-6
COMPHYST . . . . .	4-8
DEC2BIN . . . . .	4-10
DELAY . . . . .	4-12
DERIV . . . . .	4-14
DIVIDE . . . . .	4-16
FILTER . . . . .	4-20
FUNCTION . . . . .	4-24
INTEGRATOR . . . . .	4-27
MULTIPLY . . . . .	4-36
PI CTRL . . . . .	4-39
PULSE CNTR . . . . .	4-43
RATE LIMITER . . . . .	4-45
SCALE) . . . . .	4-47
UP/DWN CNTR . . . . .	4-53

## Block Transfer Services

### Chapter 5

Chapter Objectives . . . . .	5-1
Block Transfer Descriptions . . . . .	5-1
Block Transfer Status Word . . . . .	5-2
Application Status Services: Event List Checksum . . . . .	5-4
Event List Checksum . . . . .	5-5
Application Status Services:	
Read Task Name . . . . .	5-6
Write Task Name . . . . .	5-7
Total Number of Events in Application . . . . .	5-8
Total Number of I/O Nodes . . . . .	5-9
Read Task Status . . . . .	5-10
Fault Status Read . . . . .	5-12

Program Limits Information:	
Library Description .....	5-14
Scheduled Task Interval (mS) .....	5-15
Maximum Number of Events per Application .....	5-16
Number of Function Block Task Files in Product .....	5-17
Maximum Number of I/O Nodes Allowed per Application .....	5-18
Application Control Commands:	
BRAM Functions: Store, Recall, and Initialize .....	5-19
Download and Compile .....	5-21
Read Single Event .....	5-26
Clear/Process Links .....	5-28
Download Service Init .....	5-30
Node Adjustment:	
Read Block Value .....	5-31
Write Block Value .....	5-33
Read Block Link .....	5-35
Write Block Link .....	5-36
Read Full Node Information .....	5-38
Read Node Value .....	5-41
Write Node Value .....	5-42
Read Node Link .....	5-43
Write Node Link .....	5-45

## Handling Exceptions -- Faults and Warnings

### Chapter 6

Chapter Objectives .....	6-1
Handling Function Block Faults and Warnings .....	6-1
Accessing the System Fault and Warning Queues .....	6-3
Handling Download Service Errors .....	6-3
Handling Compile Faults .....	6-4
Link Processing Fault .....	6-5
I/O Node Limit Fault .....	6-5
Memory Limit Fault .....	6-5
BRAM Checksum Fault .....	6-6
Using the Task Status Service .....	6-7
Using the Fault Status Service .....	6-8
Download Errors .....	6-8
Invalid Link Fault Condition .....	6-8
Clear Faults Command .....	6-9
Fault Codes .....	6-10

*End of Table of Contents*

## Preface

### Manual Overview

This manual attempts to accommodate users who are unfamiliar with the function block system as well as more experienced users. When read from front to back, this manual provides an increasing level of detail, with each chapter building upon information presented in the previous chapter.

**Chapter 1** is an introductory chapter. It provides general information on the function block system by walking you through a sample application. The application is represented by an event list and its associated function nodes and links.

**Chapters 2 and 3** discuss the pieces of the function block system — How function blocks operate and how they interact with the rest of the drive.

**Chapters 4 and 5** provide a function block library and explain the block transfer services provided for programming & maintaining applications.

Beginning users should be able to learn the function block system by reading the Product Overview and using one of the Getting Started examples in **Chapter 1**. More experienced users may want to skip **Chapter 1** and begin directly with **Chapter 2** or **3** to obtain detailed information.

**Important:** Due to their complexity and use, certain concepts will be purposely repeated in this manual.

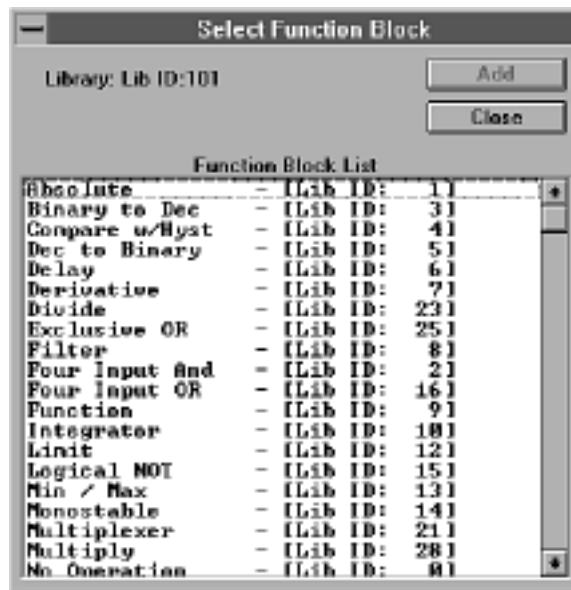
### Product Overview

The function block system allows you to customize drive operation to your specific application. The function block software contained in the PLC Comm Adapter Board provides several advantages.

- On larger system applications the loading of the PLC control system will be reduced.
- On smaller stand-alone operations, programming will be carried out completely within the drive, redefining the term *standalone drive*.

Function blocks are integral to drive operation and can be combined together to operate on almost any part of the drive functionality. The flexibility of the function block system allows blocks to be used with the drive's velocity or current control parameters, drive-to-drive parameters, as well as analog and remote I/O parameters.

Shown below are a portion of the function blocks that are available viewed through DriveTools DriveBlockEditor. By scrolling forward, the 28 different function blocks that currently make up the function block library may be viewed. Functions range from logical function blocks (AND, OR, XOR and NOT) — to math function blocks (ADD, SUB, MULT and SCALE) — to more involved functions, including Proportional/Integral Control and Rate Limiter. Control functions such as Monostable, Compare with Hysteresis, Delay, Multiplexer, and Pulse Counter are available, as well as conversion functions like Binary-to-Decimal and Decimal-to-Binary.



Currently any combination of function blocks up to a maximum of 128 events are executed with a 20mS task interval. A function block application can be created and set up by any of the three terminals compatible with the PLC Communications Adapter Board. These terminals are a PC using DriveTools DriveBlockEditor, a Graphic Programming Terminal, or a PLC.

The function block application is created by programming an execution list of function blocks, and then downloading the execution list to the drive where it is compiled into a function block program. When the drive compiles the function block program, it also creates the functionality and data sets within the drive. Once the execution list has been successfully downloaded, I/O nodes at each function block can be further manipulated to control the function block application.

The 1336FORCE when equipped with a PLC Comm Board has 497 fixed parameters which are referred to as linear parameters. The function block program allows 799 new dynamic node parameters. Dynamic parameters are not fixed and can be modified and manipulated to meet the needs of your particular application.



---

## Terminology

**Application** — An application is represented by an event list and its associated function, nodes and links.

**Block Type Number** — The block type number specifies one of the 28 different types of function blocks currently installed in the function block library. You can use each type of function block as many times as required in an execution list.

**Block ID#** — A block ID# is a unique number assigned to a function block when it is entered into an execution list. The number is used to identify each individual function block.

**BRAM** — This is the function block's *hard memory* storage which is battery backed up. This is often referred to as EPROM or EE storage. EE functions and BRAM functions are synonymous.

**Compiling** — Compiling creates the program and data sets within the drive. This is a background operation in the drive that involves a series of checks before the drive accepts a downloaded function block execution list.

**Event** — An event is a function block that has been assigned both a block ID and a block type number. Both are required to enter a function block into an execution list.

**Execution List** — An execution list is the list of events that will be sent to the drive in a predetermined sequence. A maximum of 128 events are allowed in an execution list.

**Input** — Input refers to the data provided for a function block operation.

**Linear Parameter** — A linear parameter is a fixed parameter from 1-497 that resides in the drive parameter table. These parameters always exist and cannot be deleted from the drive, as opposed to function blocks which can be created within the drive and subsequently deleted from the drive.

**Linking** — Linking refers to the software connections between function block nodes, or fixed drive parameters and function block nodes.

**Node or Node Parameter** — A node is a dynamic, non-fixed parameter that can be created and manipulated using the function block program.

**RAM** — This is the function block's *scratch pad* memory where the application is compiled and runs. Random Access Memory is not backed up and clears each time there is a power loss or a BRAM initialization.

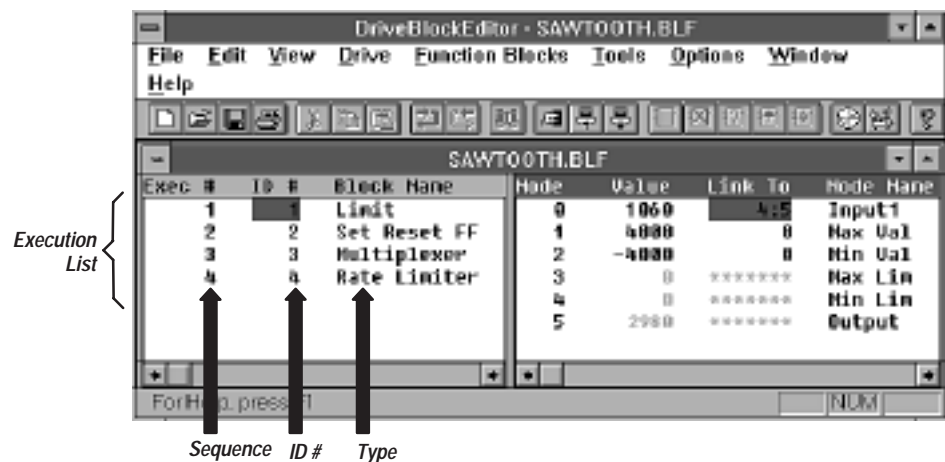
**Output** — The result of a function block operation.

## Function Block Components

Developing and successfully entering a new function block application in your 1336FORCE involves four distinct steps. These steps are shown below and on the following pages of this chapter using DriveTools.

### Step 1 — Create an Execution List.

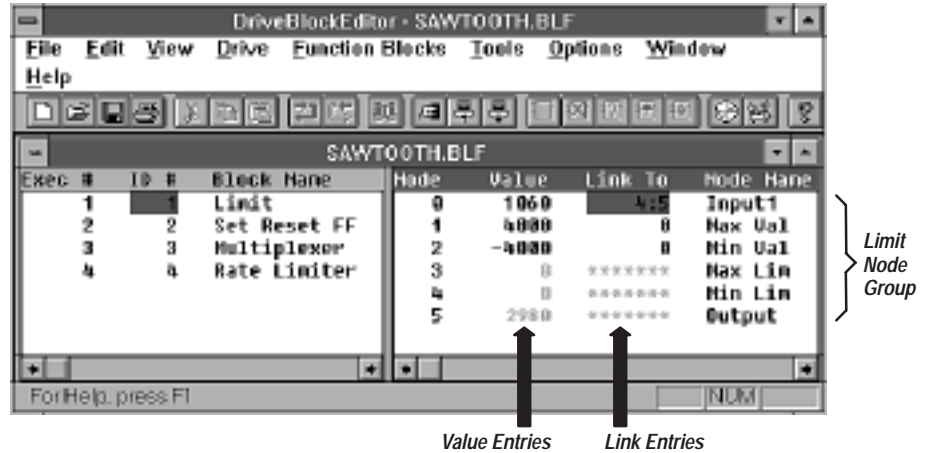
You can create an execution list by entering function blocks into an on screen display. The execution list entries are shown on the left side of the DriveBlockEditor screen shown below.



These events or function blocks (Limit, Set Reset FF, Multiplexer, etc.) are chosen from the function block library. You may enter any combination of events up to a total of 128 in your execution list. Events are executed in the order in which you enter them in the list. A full description of each available function block can be found in **Chapter 4**.

### Step 2 — Enter Block Values

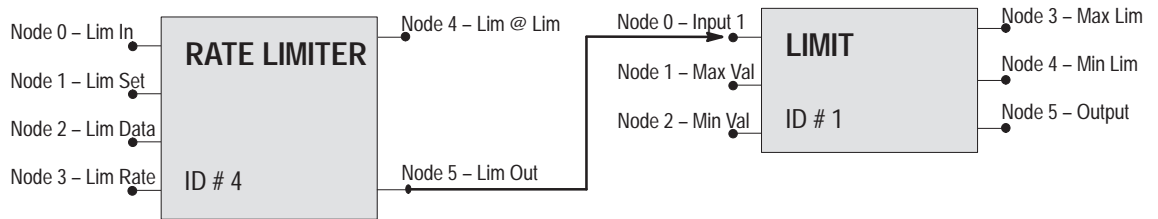
Once you have entered all events into your execution list, you may want to adjust the values of the node parameters of the function blocks. These values are entered in the Value column on the right side of the DriveBlockEditor screen as shown on the next page. Node values must be within the range specified by the maximum and minimum limits. **Chapter 2** provides detailed examples on entering function block nodes.



**Step 3 — Enter Links**

You can now use links to alter an application by connecting function block inputs and outputs to other nodes or linear parameters in the drive. Links are accomplished by entering block ID's and nodes in the Link To column on the right side of the DriveBlockEditor screen.

In the example shown below, Rate Limit output node 5 (4:5) is linked to Limit Block node 0 (1:0).



**Step 4 — Download and Compile**

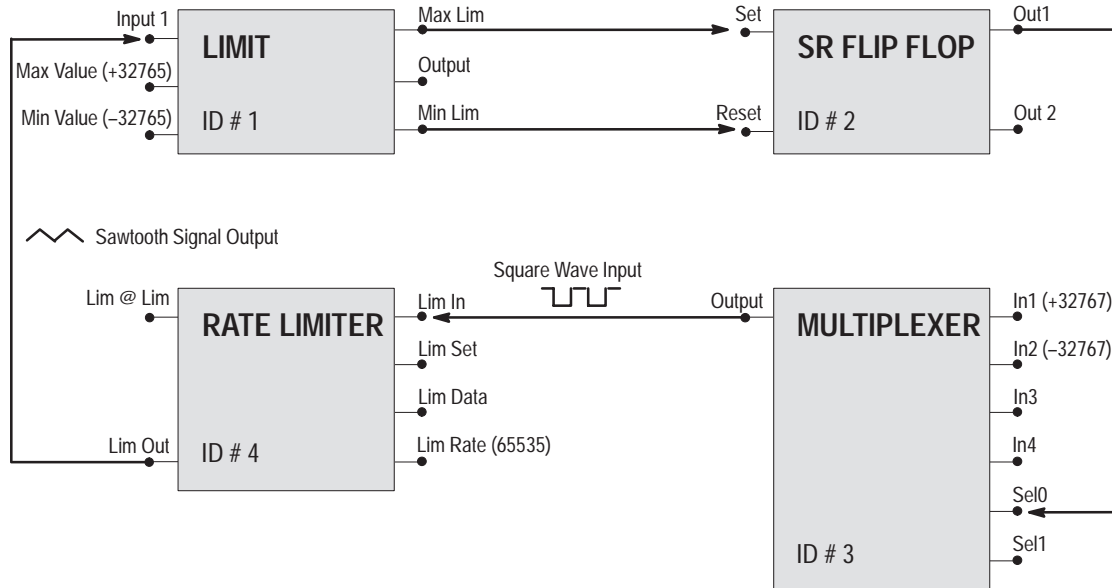
Once you have established your execution list with all values and links, you can download this list from the PC to the drive. The service will run a series of tests on your execution list before accepting and compiling the function block task you have established. Compiling will create the program data sets within the drive. The DriveBlockEditor will wait until the compile is complete before sending node values and link connections to the drive.

***End of Preface***

## Getting Started

### Chapter Objectives

This chapter introduces you to an application using function block programming. The exercises in this chapter take you through the programming of the sawtooth generator application shown below.



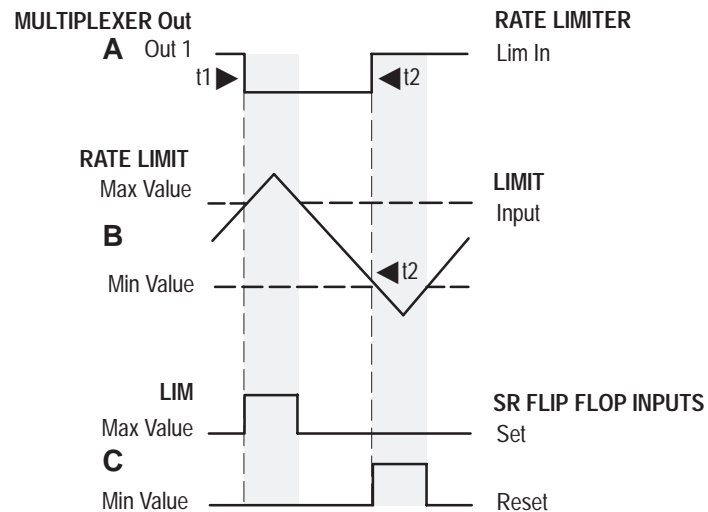
The first exercise begins on page 1-3. **Getting Started Using DriveTools DriveBlockEditor** creates the application using the DriveTools DriveBlockEditor Program.

The second exercise begins on page 1-15. **Getting Started Using a PLC** creates the same application using PLC Block Transfer Services.

These two exercises are solely step-by-step basic programming instructions. The following two chapters explain in detail the pieces of the system, their operation, and their interactions with the rest of the drive.

## Sawtooth Application Operation

The output from the **RATE LIMITER** function block will be a sawtooth signal. The value of the **RATE LIMITER** output will ramp up to the value specified by the **MULTIPLEXER** input #1 (+32767). When the **RATE LIMITER** output reaches the maximum value specified by the **LIMIT** block  $t1$  (+32765), the max limit flag will set the **SR FLIP FLOP** output, which in turn selects the **MULTIPLEXER** input #2. The **RATE LIMITER** output will then ramp down to this new value of -32767.



When the **RATE LIMITER** output reaches the minimum value specified by the **LIMIT** block (-32765), the min limit flag will clear the **SR FLIP FLOP** output ( $t2$ ), which in turn selects the **MULTIPLEXER** input #1. The **RATE LIMITER** output will continue to ramp up and down between the **LIMIT** block minimum and maximum values.

## Getting Started Using DriveBlockEditor

To start DriveTools:

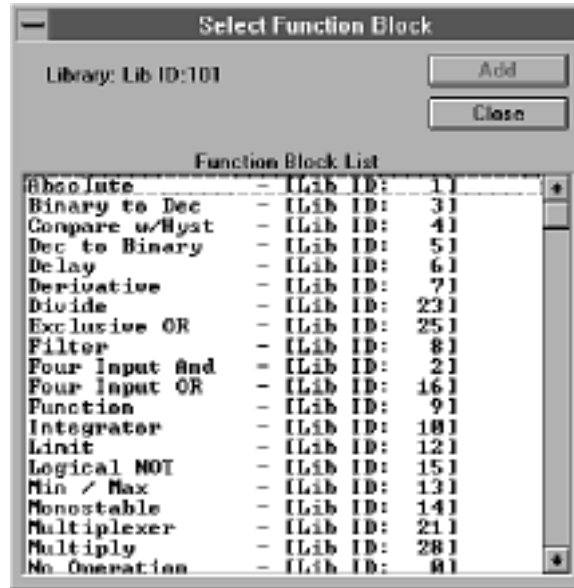
- Enter DriveTools.
- Select DriveBlockEditor.
- Select the **New** option from the DriveBlockEditor's pull-down **File** menu to create a new execution list. The display shows a function block library list similar to the one shown on the next page.



- Select the file.
- Click on **OK**.

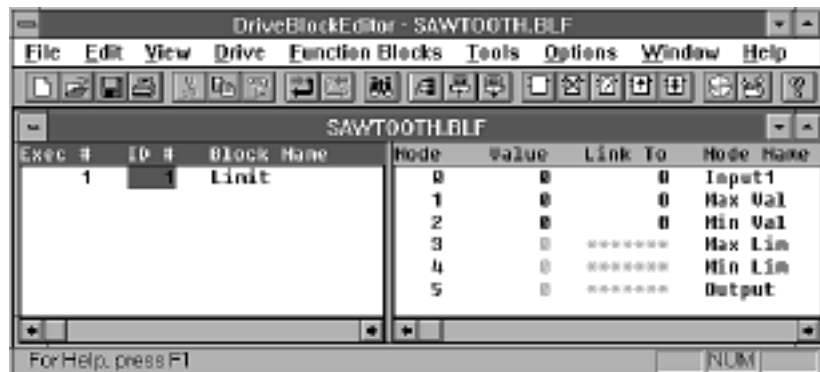
### Step 1 — Add a Limit Block

1. Select the **Add Block** option from the **Function Blocks** pull-down menu.
2. Double click on **Limit** — [Lib ID: 12].



- Click on **CLOSE**.

As shown below, the DriveBlockEditor software now enters one Limit function block in your new execution list with an **ID#** of 1.



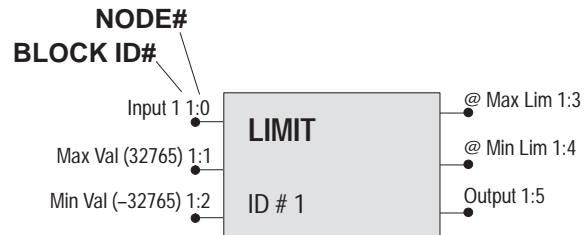
3. Enter the maximum value by clicking on the **Value** field for **Node 1** and entering **+32765**. Press enter to save the value as shown below.





4. Enter the minimum value by clicking on the **Value** field for **Node 2** and entering **-32765**. Press enter to save the value as shown above.

You have now created the first function block and set the input node value limits as shown below.



### Step 2 — Enter a Set/Reset FF Block

To add a Set Reset FF function block:

1. Move the cursor to the left side of the execution list and click.
2. Select the **Add Block** option from the **Function Blocks** pull-down menu.
3. Double click on **Set Reset FF — [Lib ID: 22]**.
4. Click on **Close**.

As shown below, the DriveBlockEditor software now enters a Set Reset Flip Flop function block and gives it an **ID #** of 2.

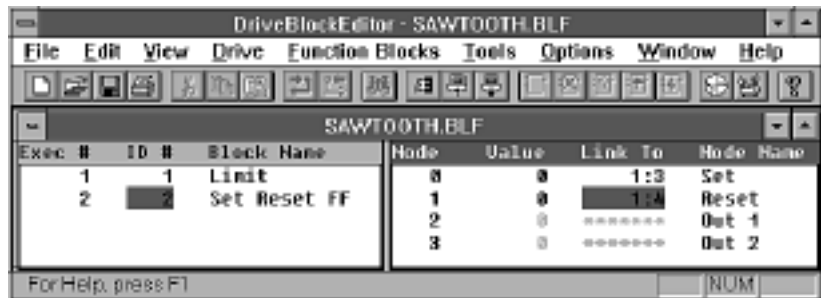


You can now link the Set Reset Flip Flop inputs to the outputs of the Limit function block entered in **Step 1**.

5. Link the Set Reset FF's set input (node 0), to the maximum limit flag of the Limit function block (node 3).

Click on the **Link To** field for **Node 0** and enter **1:3**.

Press enter to save the value as shown below.

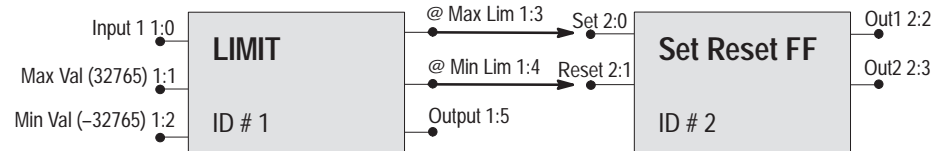


6. Link the Set Reset FF's reset input (node 1), to the minimum limit flag of the Limit function block (node 4).

Click on the **Link To** field for **Node 1** and enter **1:4**.

Press enter to save the value as shown above.

With the Set Reset FF function block and links added, the function block diagram now appears like this.



### Step 3 — Enter a Multiplexer Block

To add a Multiplexer function block:

1. Move the cursor to the left side of the execution list and click.
2. Select the **Add Block** option from the **Function Blocks** pull-down menu.
3. Double click on **Multiplexer** — [Lib ID: 21].
4. Click on **CLOSE**.

As shown below, the DriveBlockEditor software now enters a Multiplexer function block and gives it an **ID # of 3**.



5. Enter a value for input 1 by clicking on the **Value** field for **Node 0** and entering **+32767**.

Press enter to save the value as shown below.



6. Enter a value for input 2 by clicking on the **Value** field for **Node 1** and entering **-32767**.

Press enter to save the value as shown above.

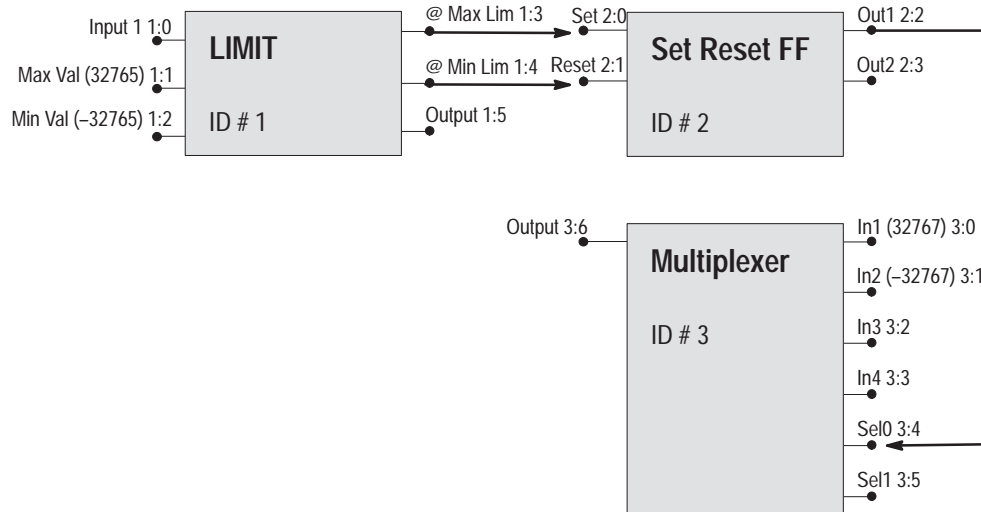
- Link the Multiplexer's sel0 input (node 4), to the Set Reset FF's output (node 2).

Click on the **Link To** field for **Node 4** and enter **2:2**.

Press enter to save the value as shown below.



With the Multiplexer function block values and link added, the function block diagram now appears like this.



### Step 4 — Enter a Rate Limit Block

To add a Rate Limit function block:

1. Move the cursor to the left side of the execution list and click.
2. Select the **Add Block** option from the **Function Blocks** pull-down menu.
3. Double click on **Rate Limiter** — [Lib ID: 19].
4. Click on **CLOSE**.

As shown below, the DriveBlockEditor software now enters a Rate Limiter function block and gives it an **ID #** of 4.

The screenshot shows the DriveBlockEditor interface for a file named SAWTOOTH.BLF. The main window displays a table with the following data:

Exec #	ID #	Block Name	Node	Value	Link To	Node Name
1	1	Limit	0	0	0	Lin In
2	2	Set Reset FF	1	0	0	Lin Set
3	3	Multiplexor	2	0	0	Lin Data
4	4	Rate Limiter	3	0	0	Lin Rate
			4	0	0	Lin DLin
			5	0	0	Lin Out

5. Enter a value for the rate by clicking in the **Value** field for **Node 3** and entering **65535**.

Press enter to save the value as shown below.

The screenshot shows the DriveBlockEditor interface for a file named SAWTOOTH.BLF. The main window displays a table with the following data:

Exec #	ID #	Block Name	Node	Value	Link To	Node Name
1	1	Limit	0	0	0	Lin In
2	2	Set Reset FF	1	0	0	Lin Set
3	3	Multiplexor	2	0	0	Lin Data
4	4	Rate Limiter	3	65535	0	Lin Rate
			4	0	0	Lin DLin
			5	0	0	Lin Out

You can now link the Rate Limiter input to the output of the Multiplexor function block entered in **Step 3**.

6. Link the Rate Limiter's input (node 0), to the Multiplexor's output (node 6).

Click on the **Link To** field for **Node 0** and enter **3:6**.

Press the enter key to save the value as shown below.

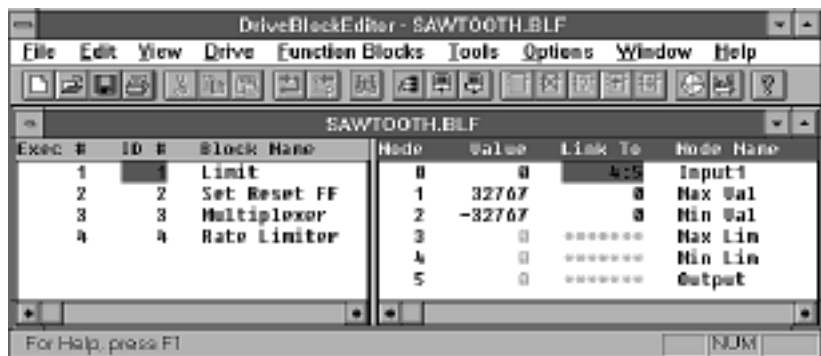


**Step 5 — Modify the Limit Block**

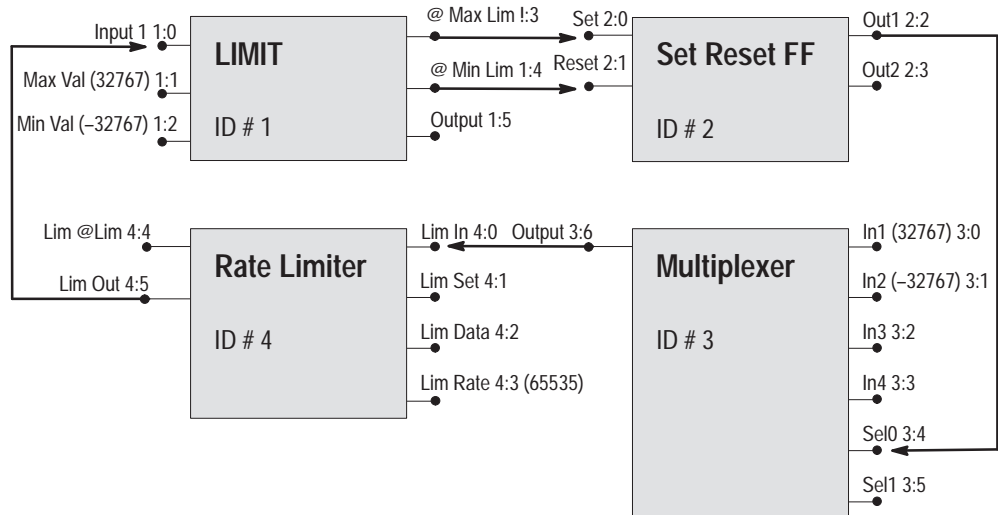
1. Move to the Limit function block node entry field by clicking on **ID # 1**. Link the Limit's input (node 0), to the Rate Limiter's output (node 5).

Click on the **Link To** field for **Node 0** and enter **4:5**.

Press the enter key to save the value as shown below.



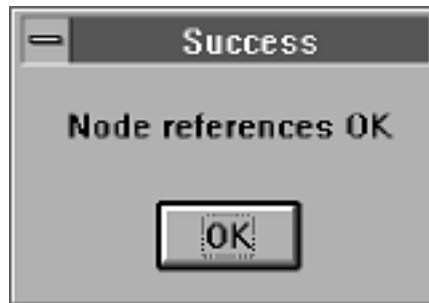
The block diagram is now complete and should appear as shown below with the Rate Limiter function block added.



### Step 6 — Check Links

Once all function blocks and their links have been established, node connections in the program should be validated by using a **Check Node Connections** command from the **Function Blocks** pull-down menu. This function is performed by the DriveBlockEditor, not the drive.

If all links are correct, the following display will be shown.

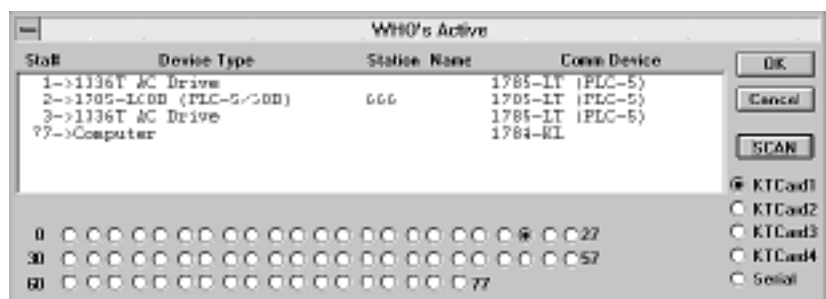


If any errors were made, a Connection Errors Dialog Box will detail the errors.

### Step 7 — Download the Program

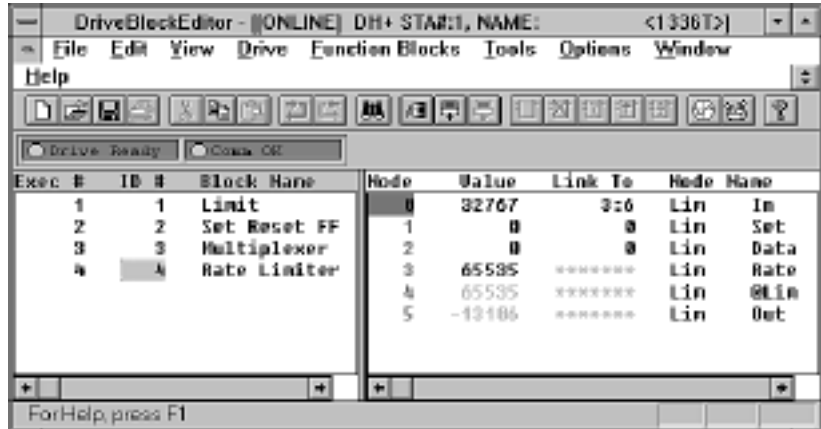
Once the links have been checked, the execution list must be downloaded to the drive to enable the function block program.

1. Select the **Download to Drive** option from the DriveBlockEditor's pull-down **Drive** menu. If you know the station number, enter it at this time. If the Station number is unknown, use the **WHO** menu option shown below to scan for active DH+ stations.



2. During the download process, the drive checks function block links and node values. A message will appear telling you whether the download was successful.

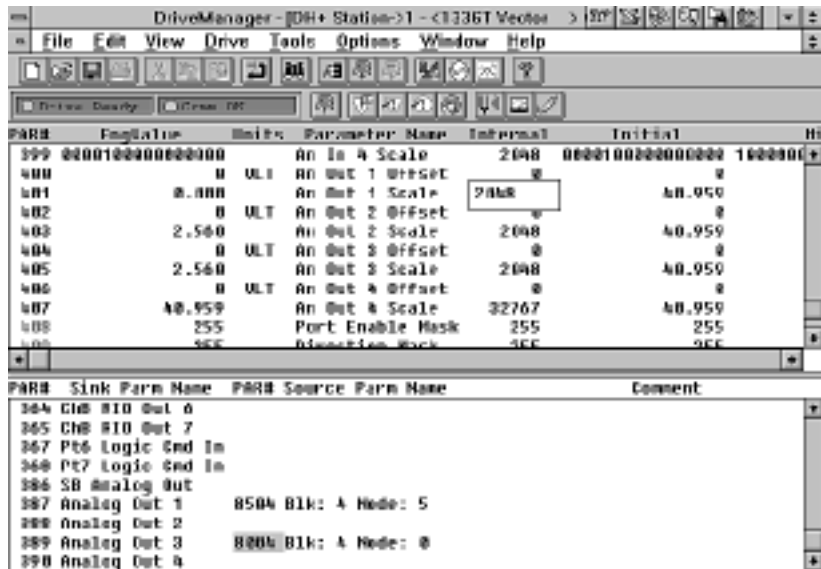
3. Upon completion, select the **Connect to Drive** option from the DriveBlockEditor's pull-down **Drive** menu and re-enter the station number to go online.
4. Once online, verify that values are changing at the Rate Limiter function block's output node.



### Step 8 — Link Analog Output Parameters to Function Block Nodes

Enter the DriveManager program to link linear parameters.

1. Enter a scale factor value of 2048 for Parameters 401 and 405. The value 2048 will be shown in the **Internal** units field shown below.





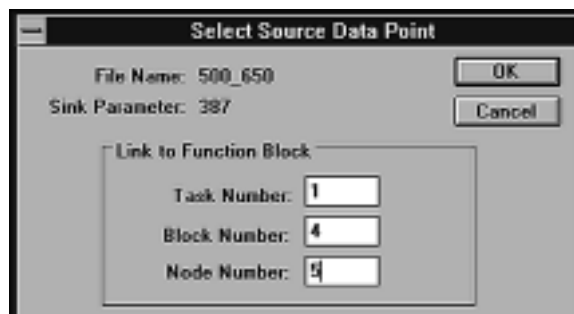
- Enter an offset value of **0** for Parameters **400** and **404**. This allows a value of  $\pm 32767$  to traverse the entire  $\pm 10V$  range for both analog outputs.



- Within the link window at the bottom of the screen, double click on the Par # field associated with Parameter 387.

Link analog output #1, Parameter 387, to Rate Limiter output node Output (4:5).

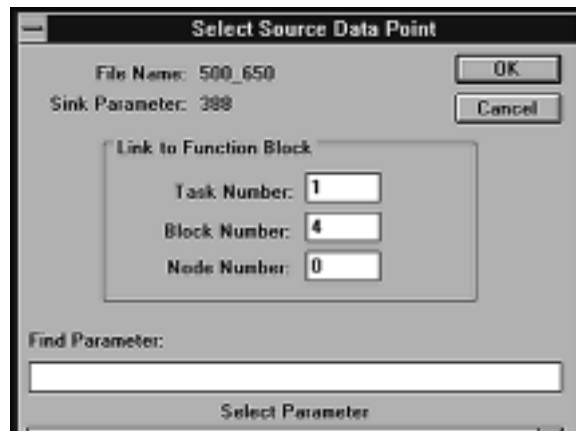
The window shown below should appear with entry boxes.



- Enter a **Task Number** of **1**.  
Enter a **Block Number** of **4**.  
Enter a **Node Number** of **5**.  
Click on **OK**.

Link analog output #3, Parameter 389, to Rate Limiter input node Input 1 (4:0).

- Double click on the Par # field associated with Parameter 389.  
The window shown below should appear with entry boxes.



- Enter a **Task Number** of 1.  
Enter a **Block Number** of 4.  
Enter a **Node Number** of 0.  
Click on **OK**.

If desired, you can now use DriveMonitor or an oscilloscope to view the analog outputs in a graphic format.

**Important:** DriveMonitor can be used to monitor any function block node directly. Function block nodes do not need to be linked to analog outputs when using DriveManager.

### **Step 9 — Modify Node Values**

Return to DriveBlockEditor to modify function block node values in an on-line application.

- Adjust the Limit function block values.  
Set the Max Lim input node value to 200.  
Set the Min Lim input node value to -200.

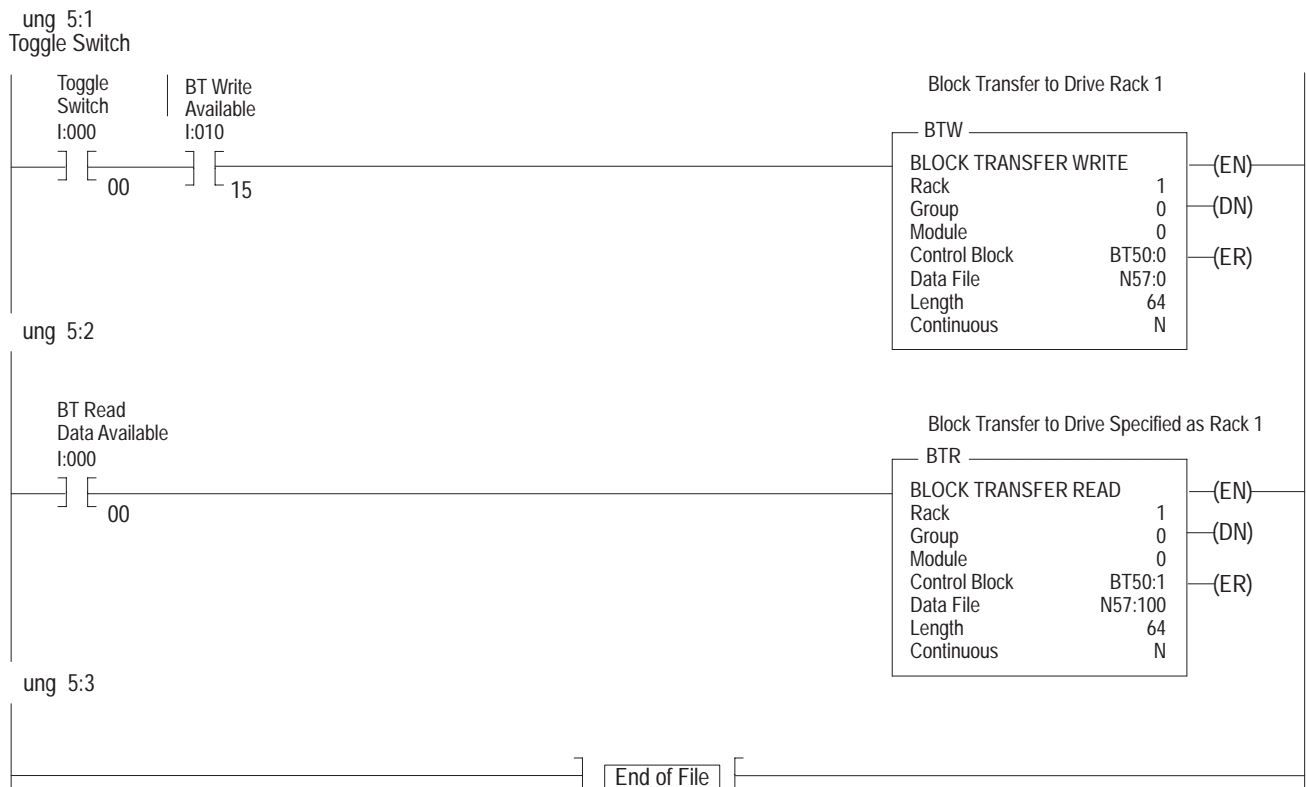
This will cause the output of the Rate Limiter function block to ramp up and down between 200 and -200.

- Adjust the value of the Rate Limiter Rate (block 3, node 4) to change the slope of the sawtooth signal.

## Getting Started Using a PLC

Shown below is a sample program that will transfer data to a drive that is set up as Rack 1. The block transfers are executed by toggling input I:00/00. The Block Transfer Write sends the information in data file N57:0 to the drive. The data in these addresses determines what type of operation is performed. The Block Transfer Read instruction receives information from the drive and places it in data file N57:100. This data contains the status of the operation being performed and any data (if applicable) that is returned from the drive.

Toggle switch #0 to initiate the Block Transfer Read/Write pair. File N57:0 contains the data that is transferred to the drive.



**Important:** If a PLC 5/15 or 5/25 is used, the control block must use an integer data type, not the Block Transfer (BT) data type.

### Step 1 — Initialize the Function Blocks

Initialize the function block BRAM to clear the current function block application.

1. Toggle bit I:00/00 to indicate the block transfer.
2. Verify that the initialization was successful.
3. If N57:101 = 0F02<sub>hex</sub> (Block Transfer Read Data), there are no errors.

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N57:0	0000	8F02	0000	0003	0000	0000	0000	0000	0000	0000

Words N57:0 – N57:3 will be sent to the drive. Values are displayed in hexadecimal format.

### Step 2 — Download and Compile the Program

Thirty-two events can be downloaded in each block transfer. Because this example consists of only (4) events, only (1) block transfer routine is required to download the execution list.

1. Type the data shown in the table below into addresses N57:0 — N57:9. The Block Transfer Write Data specifies a download operation and contains the events in the execution list. Values are displayed in hexadecimal format.

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N57:0	0000	8F03	4000	0000	0004	0A4A	010C	0216	0315	0413

Words N57:0 – N57:5 are the Block Transfer Header Information

Words N57:6 – N57:9 are the Execution List

2. Toggle bit I:00/00 to initiate the block transfer routine which downloads block transfer data and the execution list.
3. Verify that the write was successful. If N57:101 = 0F03<sub>hex</sub> (Block Transfer Read data), there are no errors.

### Step 3 — Write Node Values

One node value is downloaded in each block transfer routine. The same block transfer routine is used in each download, but information in the data file is changed for each node value that is transferred. As shown below, the data in word N57:2 specifies the block and node that is being written to, while word N57:3 specifies the value that is being sent. Data is entered in hexadecimal format.

#### — Enter the 1st Value at the Function Block Node

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N57:0	0000	8F01	8101	7FFD	0	0	0	0	0	0

1. Set the Limit block Max value (block 1, node 1) to  $7FFD_{\text{hex}} = 32765_{\text{dec}}$ .
2. Toggle bit I:00/00 to initiate the block transfer routine which downloads the node value.
3. Verify that the write was successful. If  $N57:101 = 0F01_{\text{hex}}$  (Block Transfer Read data), there are no errors.

#### — Enter the 2nd Value at the Function Block Node

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N57:0	0000	8F01	8201	8003	0	0	0	0	0	0

4. Set the Limit block Min value (block 1, node 2) to  $8003_{\text{hex}} = -32765_{\text{dec}}$ .
5. Toggle bit I:00/00 to initiate the block transfer routine which downloads the node value.
6. Verify that the write was successful. If  $N57:101 = 0F01_{\text{hex}}$  (Block Transfer Read data), there are no errors.

**— Enter the 3rd Value at the Function Block Node**

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N57:0	0000	8F01	8003	7FFF	0	0	0	0	0	0

7. Set the Multiplexer block Input 1 value (block 3, node 0) to  $7FFF_{\text{hex}} = 32767_{\text{dec}}$ .
8. Toggle bit I:00/00 to initiate the block transfer routine which downloads the node value.
9. Verify that the write was successful. If  $N57:101 = 0F01_{\text{hex}}$  (Block Transfer Read data), there are no errors.

**— Enter the 4th Value at the Function Block Node**

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N57:0	0000	8F01	8103	8001	0	0	0	0	0	0

10. Set the Multiplexer block Input 3 value (block 3, node 1) to  $8001_{\text{hex}} = -32767_{\text{dec}}$ .
11. Toggle bit I:00/00 to initiate the block transfer routine which downloads the node value.
12. Verify that the write was successful. If  $N57:101 = 0F01_{\text{hex}}$  (Block Transfer Read data), there are no errors.

**— Enter the 5th Value at the Function Block Node**

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N57:0	0000	8F01	8304	FFFF	0	0	0	0	0	0

13. Set the Rate Limit block Rate value (block 4, node 3) to  $FFFF_{\text{hex}} = 65535_{\text{dec}}$ .
14. Toggle bit I:00/00 to initiate the block transfer routine which will download the node value.
15. Verify that the write was successful. If  $N57:101 = 0F01_{\text{hex}}$  (Block Transfer Read data), there are no errors.

### Step 4 — Write Links

One node link is downloaded in each block transfer routine. The same block transfer routine is used in each download, but information in the data file is changed for each node link that is transferred. The data in word N57:2 specifies the block and node that receives the data, while word N57:3 specifies the block and node that provides the data. Data is entered in hexadecimal format.

#### — Link the Limit Block Input to the Rate Limit Output

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N57:0	0000	8F04	8001	8504	0	0	0	0	0	0

1. Link the Limit block Input (block 1, node 0) to the Rate Limit Output (block 4, node 5).
2. Toggle bit I:00/00 to initiate the block transfer routine which will download the link.
3. Verify that the write was successful. If N57:101 = 0F04<sub>hex</sub> (Block Transfer Read data), there are no errors.

#### — Link the SR FF Block Set to the Limit Max Limit Flag

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N57:0	0000	8F04	8002	8301	0	0	0	0	0	0

4. Link the SR FF block Set (block 2, node 0) to the Limit Max Limit Flag (block 1, node 3).
5. Toggle bit I:00/00 to initiate the block transfer routine which will download the link.
6. Verify that the write was successful. If N57:101 = 0F04<sub>hex</sub> (Block Transfer Read data), there are no errors.

**— Link the SR FF Block Reset to the Limit Min Limit Flag**

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N57:0	0000	8F04	8102	8401	0	0	0	0	0	0

7. Link the SR FF block Reset (block 2, node 1) to the Limit Min Limit Flag (block 1, node 4).
8. Toggle bit I:00/00 to initiate the block transfer routine which will download the link.
9. Verify that the write was successful. If N57:101 = 0F04<sub>hex</sub> (Block Transfer Read data), there are no errors.

**— Link the Multiplexer Block Sel 0 to the SR FF Output 1**

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N57:0	0000	8F04	8403	8202	0	0	0	0	0	0

10. Link the Multiplexer block Sel 0 (block 3, node 4) to the SR FF Output 1 (block 2, node 2).
11. Toggle bit I:00/00 to initiate the block transfer routine which will download the link.
12. Verify that the write was successful. If N57:101 = 0F04<sub>hex</sub> (Block Transfer Read data), there are no errors.

**— Link the Rate Limiter Input to the Multiplexer Output 1**

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N57:0	0000	8F04	8004	8603	0	0	0	0	0	0

13. Link the Rate Limiter block Input (block 4, node 0) to Multiplexer Output 1 (block 3, node 6).
14. Toggle bit I:00/00 to initiate the block transfer routine which will download the link.
15. Verify that the write was successful. If N57:101 = 0F04<sub>hex</sub> (Block Transfer Read data), there are no errors.



### Step 5 — View Node Values

Drive analog outputs can be linked to function block nodes. Analog scale factors can be set and the analog outputs can be linked to the function block nodes by using the same block transfer routine. A device such as an oscilloscope can be connected to the analog outputs to monitor the operation of the function block program.

#### — Set the Analog Output 1 Scale Factor and Download to Drive

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N57:0	4	8301	191	800	0	0	0	0	0	0

Words N57:2 contains the parameter number in hexadecimal format. Word N57:3 contains the desired value in hexadecimal.

1. Set the Analog Output 1 Scale Factor (Parameter 401) to a value of 2048.
2. Toggle bit I:00/00 to initiate the block transfer routine which processes all function block links.
3. Verify that the write was successful. If N57:101 = 0301<sub>hex</sub> (Block Transfer Read data), there are no errors.

#### — Set the Analog Output 3 Scale Factor and Download to Drive

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N57:0	4	8301	195	800	0	0	0	0	0	0

1. Set the Analog Output 3 Scale Factor (Parameter 405) to a value of 2048.
2. Toggle bit I:00/00 to initiate the block transfer routine which processes all function block links.
3. Verify that the write was successful. If N57:101 = 0301<sub>hex</sub> (Block Transfer Read data), there are no errors.

**— Link Analog Output 1 to the Rate Limit Output**

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N57:0	4	8900	183	8504	0	0	0	0	0	0

1. Link Analog Output 1 (Parameter 387) to the Rate Limit output (Block 4, Node 5).
2. Toggle bit I:00/00 to initiate the block transfer routine which processes all function block links.
3. Verify that the write was successful. If N57:101 = 0900<sub>hex</sub> (Block Transfer Read data), there are no errors.

**— Link Analog Output 3 to the Rate Limit Input**

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N57:0	4	8900	185	8004	0	0	0	0	0	0

1. Link Analog Output 3 (Parameter 389) to the Rate Limit Input (block 4, node 0).
2. Toggle bit I:00/00 to initiate the block transfer routine which processes all function block links.
3. Verify that the write was successful. If N57:101 = 0900<sub>hex</sub> (Block Transfer Read data), there are no errors.

## System Component Detail

### Chapter Objectives

This chapter provides information about the following system component concepts:

- Execution lists and their events
- Downloading and compiling function block applications
- Understanding function block I/O nodes
- Connecting or linking blocks

### Execution List Overview

An execution list provides a way for you to organize the function blocks, or events, in the order that you want the drive to execute the events. Within an execution list, you may have up to 128 events in any combination. Each event is defined by a block type number and a block ID number.

- The block type number specifies one of the 28 function types to create and execute. Chapter 4 provides information about the available function types.
- The block ID identifies each event as being unique. The block ID does not indicate when the event will be executed. Instead, the drive uses the block ID to differentiate one event from another event with the same block type number. The block ID must be between 1 and 254.

For example, you could have an event that has a block type number of 8, which would specify a **FILTER** function block, and a block ID of 12. If you include a second **FILTER** function block with different input parameters, the second entry requires a new block ID; such as 27. By doing this, the compiler can distinguish between the **FILTER** function blocks, even if you later change the position of the events within the execution list.

Once a block ID is assigned to a certain event with a specific type, that ID number cannot be used again within the same list with a different type number. In the same execution list, you could not assign a block ID of 12 to an event with a block type of 20 (specifying a **SCALE** function).

The position of each event in the execution list implies an associated execution sequence number. The execution number specifies the order in which the event is to be executed. When you use a PLC, the execution numbers are not visible, but the events are executed in the order that they are listed in the PLC data table.

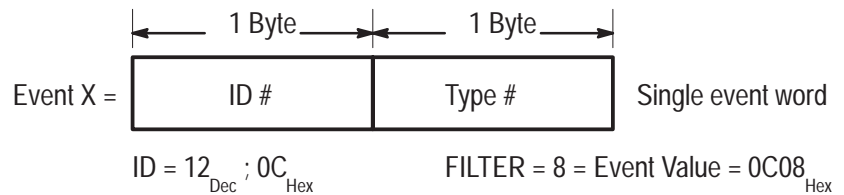
When you use DriveTools, block names are displayed in place of block type numbers. Basically, a block name uses words to identify the block type. Therefore, the block name always corresponds to the same block type number.

DriveTools shows the execution order number in the left column. The following is an example execution list from the DriveTools' DriveBlockEditor.

Exec #	ID #	Block Name	Node	Value	Link To	Node Name
1	1	Limit	0	1060	4:5	Input1
2	2	Set Reset FF	1	4000	0	Max Val
3	3	Multiplexer	2	-4000	0	Min Val
4	4	Rate Limiter	3	0	*****	Max Lin
			4	0	*****	Min Lin
			5	2980	*****	Output

Event values are easier to understand when they are represented by a hexadecimal value. Hexadecimal (or hex) representation is a base 16 numerical system, where the letters A through F represent the numbers 10 through 15.

An event is stored in the drive memory as a word. The **FILTER** event would be stored as the following:



Within the drive, the execution list is stored as an array of words. Internally, the execution list for the sawtooth example can be represented as follows:

				Hex Value Stored in drive
Event 1	ID = 01	Type = Limit	=12 <sub>Dec</sub> =0C <sub>Hex</sub>	010C
Event 2	ID = 02	Type = SRFF	=22 <sub>Dec</sub> =16 <sub>Hex</sub>	0216
Event 3	ID = 03	Type = Multiplexer	=21 <sub>Dec</sub> =15 <sub>Hex</sub>	0315
Event 4	ID = 04	Type = Ratelim	=19 <sub>Dec</sub> =13 <sub>Hex</sub>	0413
	0	0		0
	0	0		0
	⋮	⋮		⋮
	0	0		0

The same execution list using a PLC data table is shown here:

Block Transfer Header Information							Event 1	Event 2	Event 3	Event 4
	0	1	2	3	4	5	6	7	8	9
N57:0	0000	8F03	4000	0000	0004	0A4A	010C	0216	0315	0413

ID
Type

Once you complete an execution list, you need to download it to the drive and compile it to create an application. Only one execution list, or application, is operating in a drive at any given time.

When the drive enables a function block application, the events are executed every 20 milliseconds, regardless of how long it actually takes to execute the application. For example, if it takes the processor 5 milliseconds to execute your application, the drive's processor will not start to execute the application again until the full 20 milliseconds have elapsed. This is referred to as a 20 millisecond task interval.

## Creating an Execution List

The steps needed to create an execution list vary depending on the type of terminal you are using. You should refer to the appropriate documentation for information specific to your terminal. However, general steps for creating an execution list are included here.

- If you are using the DriveTools software, you can create an execution list offline by selecting the **New** option from the DriveBlockEditor's pull-down **File** menu.
- If you are using a PLC terminal, you can create an execution list by developing a block transfer routine. Chapter 5 provides information about block transfer routines.

## Adding Events to the Execution List

Once you have created your execution list, you can add events to it. When adding events to your execution list, keep the following information in mind:

- Events are executed in the order in which they appear in the list. Therefore, you need to add an event at the point in the list where you want the function block to be executed.
- Each event that you add requires a unique block ID.

In DriveTools, you can add multiple events or a single event to an execution list by selecting the name(s) to add from the Function Block Library window.

## NO-OP Events

You can specify that an event in your execution list have both an ID number and a type number of zero. This is called a NO-OP event, or no-operation event. NO-OPs are typically used as placed markers that place a NULL event within the executed application.

If you assign a non-zero value to either the ID number or the type number, you must also assign a non-zero value to the other number. For example, if you assign an ID number of 25 to a function block, you cannot assign it a type number of 0. Likewise, if you assign a valid type number to a function block, you cannot assign the function block an ID number of 0.

**Important:** If you insert a NO-OP event when using the DriveTools' DriveBlockEditor, the DriveBlockEditor appears to assign an ID number to a NO-OP event block when added to an off-line file. However, the ID number is not sent to the drive during download for NO-OP blocks. When an on-line file is uploaded, all NO-OP blocks have an ID number of 0.

### Example Execution Lists

**Important:** In the following examples, the block type text is used in place of the block type number for clarity.

The following example shows a valid execution list, with each block type having a unique block ID.

Exec#	Block ID	Block Type
1	22	ABS
2	23	AND4
3	24	BIN2DEC
4	25	COMPHY
5	30	DEC2BIN
6	27	FILTER

The following example shows an invalid execution list. The execution list is invalid because block ID 22 cannot be assigned to both the **ABS** function and the **DELAY** function in the same execution list. Keep in mind that you cannot assign the same block ID to more than one block type or block name within any given execution list.

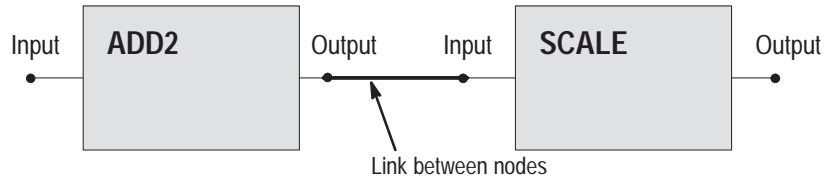
Exec#	Block ID	Block Type
1	22	ABS
2	23	AND4
3	24	BIN2DEC
4	25	COMPHY
5	30	DEC2BIN
6	22	DELAY

Chapter 3 contains additional examples of execution lists.

### Linking Events

To use the output of one function block as the input of another function block, you can create a link between the two function blocks. A link is a software connection between two data points. You can also use links if you want to use the same input values for two different function blocks, or if you want to link drive linear parameters to function block nodes.

For example, if you want to use the output, or result, of an **ADD2** function block as an input to a **SCALE** function block, you can create a link between the two function blocks as shown here.



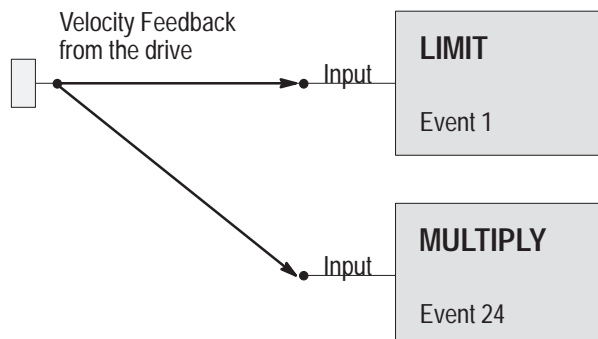
You can also create a function block that is linked to a linear parameter, such as the velocity feedback parameter. In addition, if you have two function blocks that both use the drive's velocity feedback parameter, you can link their input parameters together so that both function blocks receive the same input data.

When you link two function blocks, the information about the link (source reference number) is stored with the function block that receives the information (destination) and not with the function block that provides the information (source). In the example above, the information about the link is stored at the input to the **SCALE** function block.

### Link Operation During Execution

During execution, the drive processes the function block links one function block at a time. If two function block inputs are both linked to the same linear drive parameter, the drive transfers the data from the required parameter twice. Because the drive parameters are updated every one to two milliseconds, the values for the same drive parameter may be different during the same pass through the application.

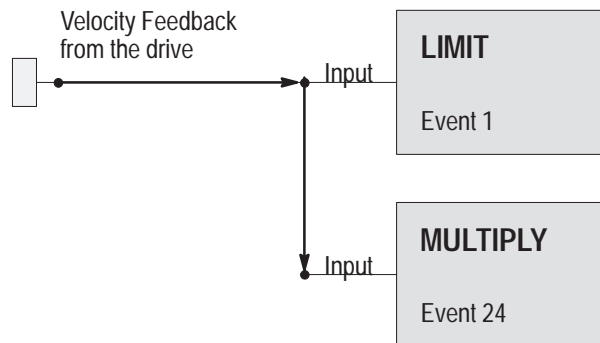
In the following example, the **LIMIT** and **MULTIPLY** function blocks both receive input from the same drive parameter. However, they may receive a different value from the drive parameter during the same pass through the application.





If you want both function blocks to receive the same value for a drive parameter, you should link the first function block's input node to the drive parameter. You should link all subsequent inputs using this drive parameter should be linked to the first function block node that is linked to the drive parameter.

In this second example, you want the **MULTIPLY** function block to receive the same value from the drive parameter as the **LIMIT** function block. Here, you would link the input to the **MULTIPLY** function block to the input to the **LIMIT** function block instead of linking the **MULTIPLY** function block to the drive parameter itself.



Refer to Chapter 3 for additional information about links and performance.

### Deleting Events from the Execution List

If you delete an event from an execution list, you need to remove all links that reference the block that is being deleted. You need to do this because the link information is stored with the function block that receives the information.

Referring to the second example, if you delete the **LIMIT** function block, you need to remove the link that is stored with the **MULTIPLY** function block and re-establish the input to the **MULTIPLY** function.

If you do not remove the link, you will receive an error when you download the execution list if you are using DriveTools, or the drive will generate a fault if you are using a PLC.

## Downloading and Compiling the Execution List

While you are creating your execution list, you are generally working on a terminal using DriveTools, a PLC, or a GPT. At this point, the execution list is an array of words that the software you are using can understand. You need to download and compile your execution list before the drive can execute it.

The download process sends a copy of the execution list array from the terminal to the drive. The compile process uses the execution list in the drive to create an application which contains the functionality and data within the drive. The drive can then execute the application.

The process that takes place when you download and compile your execution list is as follows:

1. The terminal device (DriveBlockEditor, GPT, or PLC) writes or downloads a new execution list.
2. The drive software checks the execution list for errors.
3. After the initial service checks have been completed satisfactorily, the drive acknowledges the download service and prepares the execution list to compile as a background task.
4. The drive disables any currently active (executing) function block application and calls the function block compiler.
5. The compiler moves sequentially through the execution list creating and initializing the function block objects. A function block event object associates a certain functional operation to be performed with the appropriate information and data.
6. All links associated with the function blocks are processed.
7. If no errors are encountered, the drive begins executing the application.

When the drive enables a function block application, the events are executed every 20 milliseconds, regardless of how long it actually takes to execute the application. For example, if it takes the processor 5 milliseconds to execute your application, the drive's processor does not start to execute the application again until the full 20 milliseconds have elapsed. This is referred to as a 20 millisecond task interval.

Refer to Chapter 3 for more information about the compile process. Chapter 5 provides more information on the PLC block transfer service for the download and compile operation.

## Understanding Function Block I/O Nodes

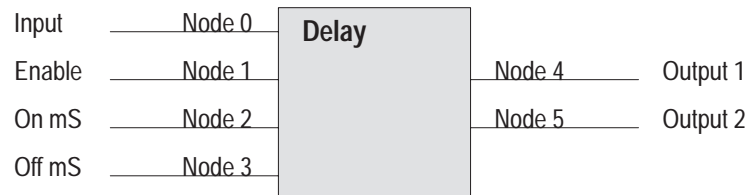
Once you have properly downloaded and compiled your execution list, you can access the I/O (input/output) nodes associated with each function block. An I/O node is a parameter that provides information to or from a function block.

The function block I/O nodes are different from the standard linear parameters. While the linear parameters always reference the same information, the I/O nodes are dynamic. The drive allocates memory for the function block parameters (I/O nodes) depending on the execution list. Thus, the drive only allocates as much memory as you need to execute your application.

Because the I/O nodes are dynamic, you cannot use fixed numbers (such as parameter 723) to refer to function block nodes. Instead, function block parameters are referenced by block ID number and node number. The block ID number and node number are also application dependent.

As the execution list is compiled, the drive allocates the I/O nodes associated with each event as a group. You can have a maximum of 799 I/O nodes per execution list.

The function block type defines the required number of I/O nodes and the characteristics of each node for a particular function block. I/O nodes are numbered from zero up to the proper number of nodes, with the input nodes numbered first. A function block with six nodes, numbered from zero to five, is shown below.



To reference a specific node of a particular function block, you need the block ID number and the node number. Using the figure shown above, if you want to access the I/O node for Output 1, the node number would be 4.

The way you reference the block ID and node number depends on whether you are using DriveTools or a PLC. DriveTools allows you to use a decimal format to reference nodes, while PLC block transfer uses a single word value.

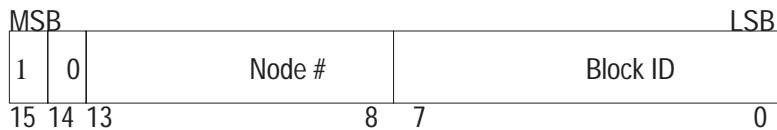
## DriveBlockEditor Node References

If you are using DriveTools, you can reference a specific node of a particular function block by specifying the *block ID:node number*. For example, to reference block number 6, node 2, you would enter 6:2.

Even though the DriveTools' DriveBlockEditor allows you to reference I/O nodes as block ID:node number, the DriveBlockEditor software converts the decimal information into the single word reference number. DriveTools uses emulated block transfer services because it uses the Data Highway Plus protocol. Block transfer services are covered in Chapter 5.

## Understanding PLC and Drive Node References

Node references are easier to understand when you use hexadecimal values. The drive and PLC block transfers both reference the node number and block ID as a single word in the following form:



MSB = Most Significant Bit

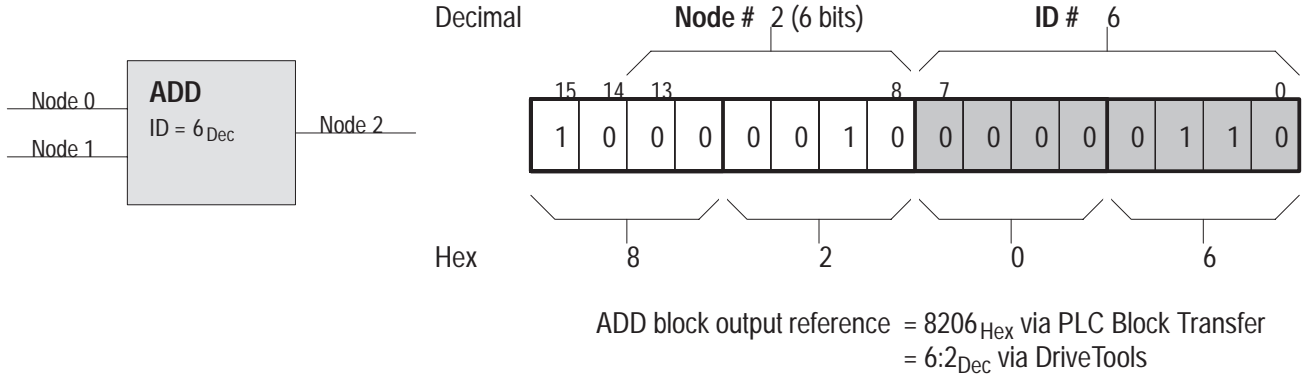
LSB = Least Significant Bit

Bit	Description
15	0 – The value or link reference represents a standard linear Motor Control board or a PLC Communication board file parameter number. 1 – The value or link reference represents a function block I/O node reference.
14	0
8–13	Contains the I/O node reference.
0–7	Contains the block ID number.

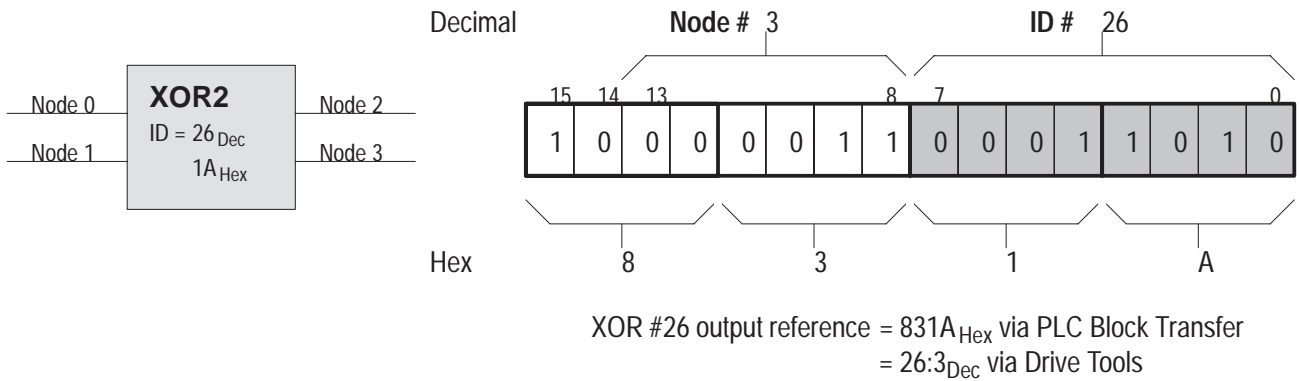
The upper-most four bits, bits 12 – 15, typically have a value of 8<sub>Hex</sub> for most function block node references. The value will not exceed 8<sub>Hex</sub> unless you reference a node number of 16 or greater.

### Examples of Function Block I/O Node References

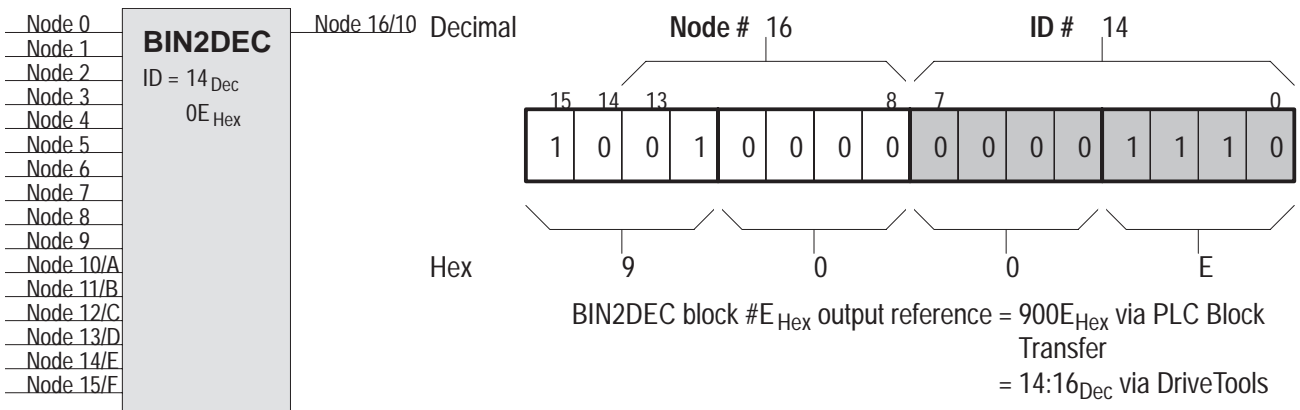
The first example represents output node 2 of an **ADD** function block that has a block ID of 6. You can convert this information to either a decimal value or a hexadecimal value.



The second example represents output node 3 of an **XOR2** function block that has a block ID of 26.



The third example represents output node 16 of the **BIN2DEC** function block that has a block ID of 14. Notice that the first number in hex for this example is 9. Normally, you can recognize a function block by an initial hex number of 8, unless the seventeenth I/O node (node number 16) is being referenced.



## Node Data Types

The value of a function block I/O node will be one of the following types:

- A signed decimal integer with a value range of  $\pm 32767$ .
- An unsigned decimal integer with a value range of 0 – 65535.
- A logical value where 0 = False and any non-zero value = True.

Some nodes have additional range restrictions. For example, a node may be a signed integer with a range of  $\pm 16383$  instead of  $\pm 32767$ .

In addition, nodes may be linkable or non-linkable. A linkable node is a node which is able to receive information from another source, while a non-linkable node cannot receive information from another source. Input nodes may be either linkable or non-linkable. Output nodes are not linkable. However, you can use output nodes to provide data for inputs to other function blocks or to drive linear parameters.

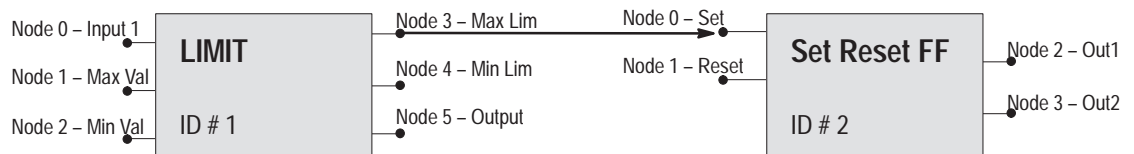
When you connect function blocks, you need to be careful. Linkable inputs can get data from any function block node or linear parameter, regardless of its data type. As an example, you could link a logical input to a signed decimal output.

The characteristics of the destination node determine how the input value is interpreted. In the case where a signed decimal output is linked to a logical input, the value would be interpreted as a true value unless the source value (such as a velocity or position feedback) was equal to zero.

## Creating Links Between Nodes

When you create a link between two function blocks, you are actually creating a connection between a node on one function block and a node on another function block. The information about the link is stored at the destination node, which is the node that receives the data.

In the following example, the link between the **LIMIT** function block and the **Set Reset FF** function block is located between Node 3 of the **LIMIT** function block and Node 0 of the **Set Reset FF** function block. The information about the link is stored with Node 0 of the **Set Reset FF** function block. Therefore, when you create the link, you need to create it at Node 0 (2:0), not Node 3 (1:3).



If you are using DriveTools, you would create this same link by doing the following:

1. Click on the **Set Reset FF** function block ID number. The nodes for the **Set Reset FF** function block are displayed on the right side of the DriveTools screen as shown here.



2. Click on the **Link To** field for the **Set** node.
3. Enter 1:3 to specify that you are linking Node 3 of the function block having a block ID of 1 (in this case, the **LIMIT** function block) to this node.
4. Press the enter key to save the value.

If you are using a PLC, you would create this same link by doing the following:

1. Set up your block transfer read and write blocks.
2. Create your data table. For this example, your data table would look like the following:

	0	1	2	3	4	5	6	7	8	9
BTW Data File	N10:10	0	<u>8F04</u>	<u>8002</u>	<u>8301</u>					
		Write Link Service Request	Destination Node	Source Node						

The 8002 in column 2 specifies that Node 0 of Block ID 2 (the **Set Reset FF** function block) is receiving information. The 8301 specifies that Node 3 of Block ID 1 (the **LIMIT** function block) is providing the information.

3. Perform a link processing service request.



## System Interactions

### Chapter Objectives

This chapter provides information about the following topics:

- The function block BRAM functions
- The power up sequence
- The compiler modes and terminal operation differences
- Multiple execution list copies
- The task status services
- The link processing faults
- The performance issues that concern links

### The Function Block BRAM Functions

Function block applications use two kinds of memory: RAM and BRAM.

- RAM, or Random Access Memory, is the working memory area where information is stored while the system is powered up. Any information that is in RAM is lost when you remove the power, perform a system reset, or performs a function block **Init**. When a function block application is properly set up, the application executes partially out of RAM and manipulates data stored in RAM.
- BRAM, or Battery backed up RAM (also known as EEPROM), is memory which is retained when the power is removed from the system. You can copy your function block application from RAM to BRAM by saving it. If you save your function block application in BRAM, it is transferred from BRAM to RAM when the power is cycled or a drive reset occurs.

The following are descriptions of the function block **Init** (initialize), function block **Store** (save), and function block **Recall** (restore) operations.

## The Function Block Init Command

A function block **Init** operation effectively removes any previous function block application from the working RAM area. However, it does not actually clear out the BRAM itself; it only clears the function block application out of the working RAM area. To truly initialize the BRAM area, you need to perform both an initialization service and a function block **Store** operation.

When requested, a function block **Init** does the following:

1. Clears out the application that is currently executing from RAM.
2. Releases all allocated system RAM back to the system.
3. Dissolves any previous links to or from function block nodes.

A function block **Init** goes through the linear drive parameter link reference table and dissolves those links for any linear parameter inputs that are using information from a function block node. The function block **Init** does not otherwise influence the linear parameter data.

When the **Init** operation is complete, no function block application will exist in the drive.

After you initialize the function block system, you cannot access function blocks or I/O nodes until you either recall the application stored in BRAM or download a new execution list from a terminal device. If you try to read data from or write data to an I/O node before you place another execution list in memory, your request will be rejected. Trying to link a linear parameter to a function block I/O node will also be rejected.

## The Function Block Store Command

A function block **Store** writes the function block application in active RAM memory to the drive's BRAM. When requested, a function block **Store** does the following:

1. Stores the current valid execution list.
2. Stores the function block node values. The linear parameter values are *not* stored.
3. Stores the function block link references. The linear parameter links in the linear parameter reference table are *not* stored.
4. Calculates and stores a new function block application checksum.



**Note:** In the DriveBlockEditor, the function block **Store** operation is referred to as **EEPROM SAVE**.

## The Function Block Recall Command

A function block **Recall** copies the function block application that is currently stored in BRAM into RAM. This application is then stored in RAM and is available for execution. When requested, a function block **Recall** does the following:

1. Verifies the function block data checksum before performing a function block RAM initialization.
2. Restores the execution list values, all node values, and link references to the appropriate RAM data tables from their associated counterparts within BRAM.
3. Activates the function block compiler.
4. Processes links first and then goes over the linear parameters and adjusts links of linear parameter inputs to function block nodes.
5. If there are no function block soft faults after this processing, execution of the 20 millisecond function block application is activated, and drive enable is allowed.

After all **Recall** and power up operations, automatic recompilation and processing of function block links occurs.

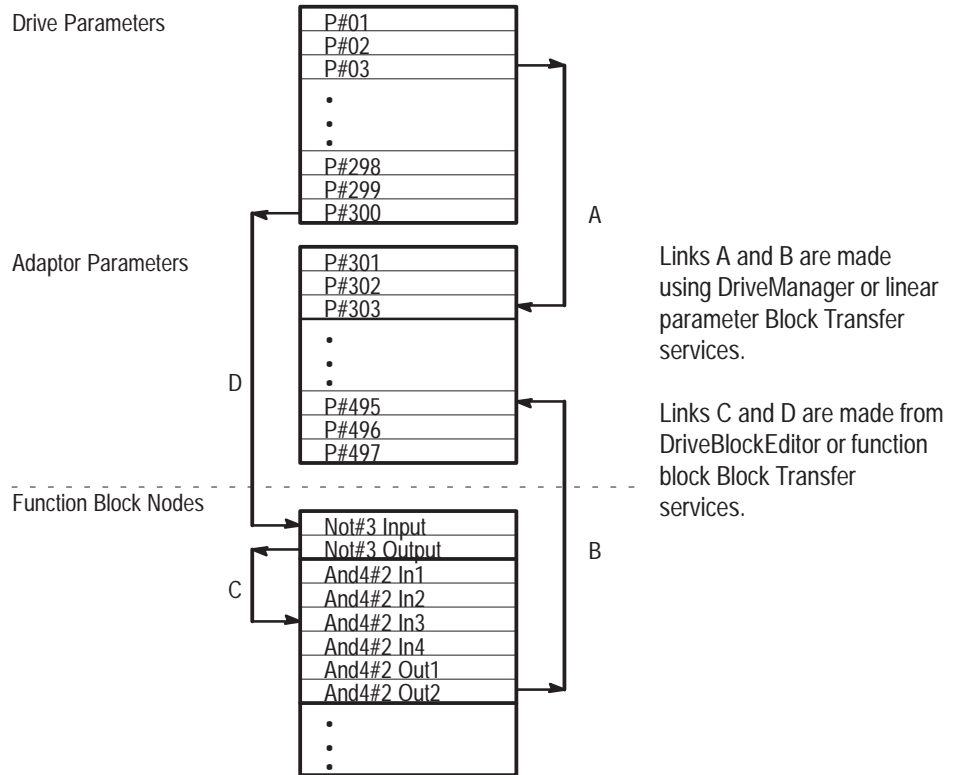
**Important:** You cannot perform a **Recall** while the drive is enabled.

## Linear Parameter BRAM Functions and Links

In a 1336FORCE drive with a PLC Communication Board, the 497 linear parameters are stored separately from the function block node parameters. Therefore, the functions for initializing, saving, and restoring data for the function blocks are separate from the linear parameter BRAM functions.

This effects the link information. The information about a link is stored with the parameter or node that is receiving the information. Therefore, if you have a link between a function block node and a linear parameter and you perform a BRAM function for either the function blocks or the linear parameters, you may create an invalid link. This is due to the way that the **Init** and **Recall** functions work between the function blocks and the linear parameters.

For example, if your RAM area is as shown below, a linear parameter BRAM **Init** would clear links A and B, while a function block BRAM **Init** would clear all function blocks as well as links B, C, and D.



If a linear parameter is receiving information from a function block node (as shown by link B) and you request a function block **Init**, the link information exists in the linear parameter area of BRAM, but the function block area was initialized and all function blocks were cleared. Link B is removed.

In this example, if you initialize both the linear parameter area and the function block area, you need to be careful when you restore the information. When you do a **Recall** on the linear parameter area, links A and B are restored. When you do a **Recall** on the function block area, links C and D are restored along with the function blocks. With this type of link information, you need to restore the function block area *before* restoring the linear parameter area. If you do a function block **Recall** first, the function blocks and links C and D are restored. You can then execute a linear parameter **Recall** to restore links A and B.

If you do a **Recall** on the linear parameter area before doing a **Recall** on the function block area, no function blocks will exist in RAM. When the linear parameter **Recall** is executed, it cannot build link B because the function block that provides the data to the linear parameters does not yet exist in RAM. This will cause a drive soft fault to occur.

The system does not automatically clear a link that was valid but has become invalid. You need to clear the link or adjust the link to point to a valid parameter or node before you can clear the fault.

The function block fault read service returns the reference of the first node with an invalid link reference. You may clear links individually, or you can clear all links at once.

When using DriveTools, you can access the function block BRAM functions from DriveBlockEditor EEPROM selection and the linear parameter BRAM functions from DriveManager. For additional information about the linear parameter value, link, and BRAM services, refer to the PLC Communications manual.

## Power Up Sequence

Whenever you power up the system, the drive does the following:

1. Performs a linear parameter BRAM **Recall** function to restore the linear links and parameter values from BRAM to RAM
2. Builds the linear parameter link scanner. While building this list, links which reference function block nodes are skipped. No function blocks exist yet.
3. Performs a function block **Init**.
4. Performs a function block **Recall** to restore the execution list values, all node values, and link references to the appropriate RAM data tables from their associated counterparts within BRAM.
5. Activates the function block compiler.
6. Processes all function block links when the compilation is complete.
7. Goes over the linear parameters and adjust links from linear parameters to function block nodes. This step processes any linear destination parameters that were not linked in step 2 because they referenced function blocks.
8. Activates the 20 millisecond function block application and allows drive enable if no function block soft faults occur during processing.

## Compiler Modes and Terminal Operation Differences

You can use any of the three supported terminal devices to create and update your function block applications:

- DriveTools' DriveBlockEditor
- Graphic Programming Terminal (or GPT)
- PLC block transfer

The three terminal devices use different compiler modes due to differences in the amount of their available RAM. Depending on how you change your execution list, you may receive an error when using one terminal device, but not when you use another terminal device. This section describes the compiler modes that are used and information specific to the individual terminal devices to help you use function blocks more effectively.

Regardless of which terminal device you are using, an application is created when you compile your execution list. You should also note the following information:

- The application executes partially out of RAM.
- The application executes within a 20 millisecond task interval and is integral to the system operation.
- The compiler resides on the PLC Communication Adapter board as part of its AP or Application Processor code PROM.

## Compiler Modes

The two basic compiler modes used for function blocks are the initial compile mode and the subsequent, or comparison, compile mode. You do not select which mode to use; the compile mode is automatically determined by the terminal device and whether or not an application already exists in the RAM area of the drive.

### Initial Compile Mode

With the initial compile mode, no events exist in the drive when the execution list is downloaded, and all the objects are created from scratch.

The DriveTools' DriveBlockEditor always uses the initial compile mode for its download and compile service. The DriveBlockEditor forces an initial compile mode by initializing the function block's RAM before downloading the new execution list for compilation.

## Subsequent Compile Mode

With the subsequent compile mode, a new execution list is compared against the current application in the drive to selectively create and delete function block objects. This compile mode is automatically enabled when a previously valid application exists in the drive.

Common event blocks will retain previous node values and links. Only the new event blocks need to be adjusted.

The GPT uses the subsequent compile mode for its download and compile service.

The subsequent mode of operation is a bit more involved and requires that you remove any links from blocks that receive input from an object to be deleted before downloading and compiling the new list. Processing links is the compiler's last step. If you do not remove links to objects that are to be deleted, you will get an error.

## Examples of Subsequent Compile Mode Operations

**Important:** In the following examples, the block type name is used in place of the block type number for clarity.

The following example shows a valid subsequent compile. In this example, no new blocks were created and no existing block was deleted; only the execution sequence was changed.

Existing, valid application			New (subsequent) event list		
Exec#	Block ID	Block Type	Exec#	Block ID	Block Type
1	1	ABS	1	6	DELAY
2	2	AND4	2	5	DEC2BIN
3	3	BIN2DEC	3	4	COMPHY
4	4	COMPHY	4	3	BIN2DEC
5	5	DEC2BIN	5	2	AND4
6	6	DELAY	6	1	ABS

The following example is also valid. Block ID's 23 and 26 were removed from the original program, and blocks 38 and 46 were newly created and need to be set up. The blocks which are common to both lists will retain all previous links and values.

Existing, valid application			New (subsequent) event list		
Exec#	Block ID	Block Type	Exec#	Block ID	Block Type
1	21	ABS	1	21	ABS
2	22	AND4	2	22	AND4
3	23	BIN2DEC	3	38	Integral
4	24	COMPHY	4	24	COMPHY
5	25	DEC2BIN	5	25	DEC2BIN
6	26	DELAY	6	46	DELAY

The following example is not valid because block ID 23 was re-used for a new event when it was already assigned to a **BIN2DEC** function block.

#### Existing, valid application

Exec#	Block ID	Block Type
1	21	ABS
2	22	AND4
3	23	BIN2DEC
4	24	COMPHY
5	25	DEC2BIN
6	26	DELAY

#### New (subsequent) event list

Exec#	Block ID	Block Type
1	21	ABS
2	22	AND4
3	23	Integral
4	24	COMPHY
5	25	DEC2BIN
6	26	DELAY

In this last example, the second execution list is invalid only in the subsequent compile mode. If the first application was cleared with a function block **Init**, the second (new) execution list would be valid during an initial mode compile. This is a major difference between using the DriveBlockEditor and a GPT to compile your execution list. The following sections contain additional information about the different terminals.

You can do either an initial compile mode or a subsequent compile mode when you use the PLC block transfer mechanism.

## DriveTools' DriveBlockEditor Download and Compile Operation

Because a PC running DriveTools has significantly more available RAM than a hand-held GPT, the PC running DriveTools can store more information. Besides maintaining its own copy of an execution list (when in ONLINE mode), a PC running DriveTools also holds a one word value for every node to be created and a one word link reference for every linkable input node.

The DriveTools' DriveBlockEditor uses the Data Highway Plus protocol to perform emulated block transfer commands. It uses the same block transfer services available to a PLC from one of the adapter's RIO ports.



During a download operation, the DriveTools' DriveBlockEditor does the following:

1. Performs a function block **Init** which effectively removes the function block application from the system. This initialization goes over the linear parameter link reference table and clears any links to function block source nodes. These links are not rebuilt until you make a call to the linear parameter **Recall** function.

**Important:** In DriveTools, use DriveBlockEditor to access the function block BRAM operations. You must use DriveManager to access the linear parameter BRAM operations.

2. Downloads the new execution list. An initial mode compile is performed because the initialization step removed the existing function block application from the drive.
3. Reads the Task Status byte until the execution list has finished compiling.
4. Downloads all function block node values.
5. Downloads all function block node links.
6. If an ONLINE window is open and the execution list has changed, the ONLINE window detects a difference in the executing checksum and prompts you for an upload.



**Note:** If an error occurs while downloading the new list and this operation is terminated, there will be no function block application in RAM because it was initialized.

## Graphic Programming Terminal

Because the Graphical Programming Terminal (GPT) does not have as much RAM memory as the DriveTools' PC, the GPT cannot retain all the possible node value and link information. Therefore, the GPT relies on a subsequent or comparison mode of the drive's compiler to retain the node value and link information for common event blocks between subsequent compiles.

When using a GPT, you need to remove the links that refer to the blocks to be deleted before downloading and compiling to prevent an invalid function block link fault.

## PLC Block Transfer

The PLC has the flexibility to use either mode and perform every bit of functionality the other terminals use via the block transfer services. PLC block transfers are explained in more detail in Chapter 5 of this manual.

## Understanding Multiple Execution List Copies

Only one function block application is active within the drive at any one time. However, multiple execution lists can be present in the terminal devices attached to the drive. You can connect up to seven terminal devices to the PLC Communications Adapter board, and each terminal device can have its own copy of an execution list. In addition, the DriveBlockEditor can even have multiple OFFLINE execution lists in RAM or stored in the PC's hard drive.

Even though multiple execution lists can appear to be on the system, only one function block application is active within the drive at any one time. The drive's execution list reflects what is currently running. The only other execution list stored within the drive is a copy in BRAM, which is used during BRAM recall or power up.

Terminal devices such as the DriveTools' DriveBlockEditor and the GPT maintain their own copies of the execution list. Because you can initiate download and compile operations from any of the terminal devices, do *not* initiate download and compile operations from different terminal devices at the same time.

The drive uses a checksum value to differentiate between the running application and the application stored in BRAM. If the checksum values are the same, then the current application has not been modified.

## Task Status Service

The compile operation performed during the function block **Recall** and the download and compile services are performed as background tasks. Even though you can perform other service requests while the execution list is compiling, you should avoid making node value write requests and link requests during the compilation process.

You can use the Task Status service to determine the current state of the compiler and the application execution status. The following are possible states for the compiler:

Value	Task Status	Description
0	Run Mode	The application is executing within the 20 millisecond task interval. No faults have occurred within the function block portion of functionality.
1	Download in Progress	The previously compiled application is still enabled and executing within the function block task interval. One or more downloaded packets have been received for a new function block program and the function block system is waiting for more data. The currently active application is not interrupted until all packets have been received and the data has been verified for the new function block program before compilation.
2	Compilation in Progress	All packets have been downloaded and the data verified. The service has initiated a compile. Compilation can take seconds when a large application is used.
3	Link Processing	The application is disabled and links between function blocks and drive parameters are being established
4	Recall in Progress	A <b>Recall</b> is in progress.
0x00FF	Fault Mode	A function block application has a faulted status. Function block compile time errors create a soft fault condition within the drive. The 1336T system architecture contains a system fault queue that describes the nature of the fault. SCANport provides two fault reporting values should the Task Status word indicate a faulted mode.  The previous application is disabled and will not run until you correct the fault. You cannot clear function block compiler faults with the clear faults command until you correct the function block fault.

You can read the fault queue via GPT, PLC block transfer, or DriveManager.

**Important:** A function block **BRAM Init** is recommended after any function block fault other than a function block link processing fault. If you do perform an **Init**, you must also perform a **Recall** or download a new program.

## Link Processing Faults

Even though the links appear to be processed as part of the download and compile operation, the links are actually processed after the compilation is complete. The two processes, compiling and linking, are separate. Therefore, if the drive finds a function block link fault after a compile, and no other function block compiler faults are indicated in the fault queue, you can adjust the link without recompiling the execution list. The rest of your application will still be valid.

A link processing error is indicated by the first function block fault status word having bit 1 set or a value of 0x002<sub>Hex</sub>. If a link processing error occurs, the second fault word, which is designated the code identifier, holds a reference to the first input parameter or node it found which has an invalid link reference. You can read the status words from the drive via the block transfer services.

Chapters 5 and 6 provide additional information about the clear function block links service and reading status words.

You cannot clear function block faults with a clear fault operation without first addressing the problem. The function block system will not make an assumption about what to do with an illegal link. You are forced to either clear the link to this node or reconnect the node to a valid node. After you do this, the clear faults mechanism can clear the faults and allow the drive to run.

If multiple link faults occur, you can either remove all the function block links with the clear function block links service and then clear the faults, or you can continue reading the code identifier to find the individual link errors and correct each link, one at a time.

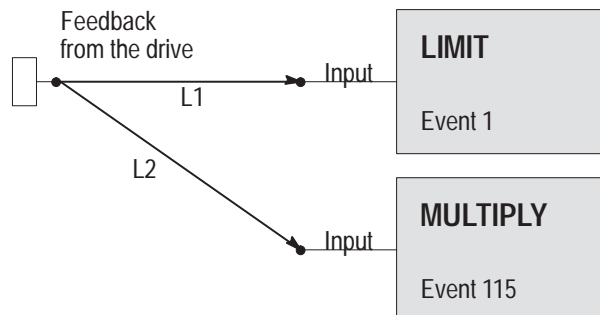
## Performance Issues Involving Links

A drive with a connected PLC Communication Board has two link processing mechanisms. One link processing mechanism operates specifically upon linear parameter links, and the other mechanism processes the function block links.

As the application executes, the function block links are processed block by block. The inputs for each individual block are checked for links. If a link is found, the link processor goes to the source parameter or node and copies the data from the source to the destination node. Once all data for the links to one function block is gathered, the function block algorithm is executed. The system can then begin processing the next function block. Because the function block application is executed every 20 milliseconds, the data for an individual link is also updated every 20 milliseconds.

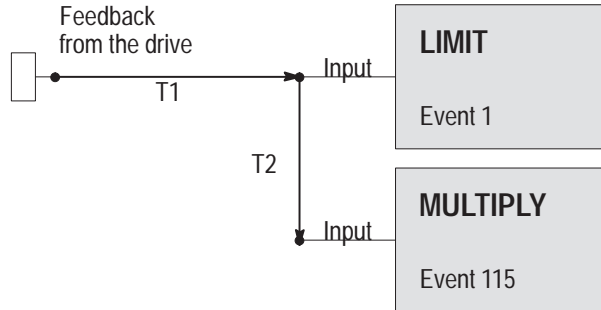
Currently, the link mechanism for the linear parameters executes every 1 to 2 milliseconds. This updates all of the linear parameter links (up to the maximum of 50) every 1 to 2 milliseconds. Therefore, two function block nodes that receive input from the same linear parameter may receive different data values within the same task interval.

For example, if you have an application with 115 events that takes 12 milliseconds to execute and the input for event 1 and event 115 are linked to the feedback, event 115 may receive a value that is different than the value event 1 received during the same execution task interval pass.



In this example, L1 represents the first transfer of data, which occurs when Event 1 is processed. L2 represents the second transfer of data, which occurs when Event 115 is processed. The data transferred in L1 and L2 may have different values.

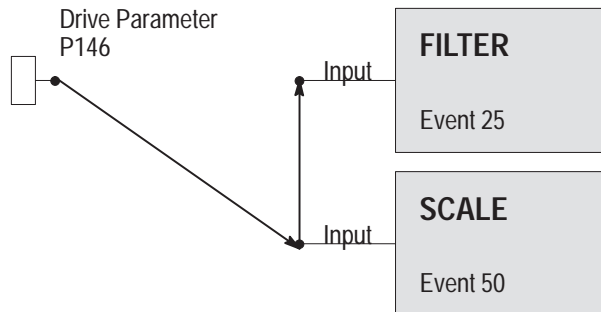
To make nodes linked to a common linear parameter operate upon the same value every 20 milliseconds, you can link the input of the second event to the input of the first event. As shown here, you could link the input of event 115 to the input of event 1.



T1 in this example represents the first transfer of data, and T2 represents the second transfer of data. The data transferred during T2 is the same value that was captured and transferred during T1.

## Link Processing Sequence

When you create your links, you do not always need to transfer data from a function block that is executed before the function block that is receiving the data. For example, if you have a **FILTER** function block that has an execution number of 25, the **FILTER** function block could receive data from a **SCALE** function block with an execution number of 50:



In this example, you should be aware that even though the **FILTER** function block receives data from the **SCALE** function block, the drive executes the **FILTER** function block before the **SCALE** function block. The first time the drive executes the **FILTER** function block, it uses the input node's initial or default value because the **SCALE** function block has not yet been executed. During subsequent passes, the **FILTER** function block always receives the data that the **SCALE** function block received during the previous pass. For example, if the **SCALE** function block received a value of 56 from the drive during the fourth time that the drive executes the application, the **FILTER** function block will not receive a value of 56 until the fifth time it is executed.

## Function Block Library

### Chapter Objectives

Detailed in this chapter are the (28) function blocks that make up the PLC Comm Board function block set.

### Function Block Overview




Each function block is a firmware subroutine stored in PLC Comm Board Memory. Each type of function block has a unique Block type number that identifies the functionality and the nodes that are associated with the block. Function blocks can be linked together to perform the same functions as equivalent analog or digital circuits. Function blocks are executed in the order in which they are entered in the execution list. Each function block type can be used any number of times.

For each function block shown, the value of an I/O node will be one of the following:

1. A signed decimal integer with a value range of  $\pm 32767$ .
2. An unsigned decimal integer with a value range of 0 – 65535.
3. A logical value where 0 = False and any non-zero value = True.

**Important:** The use of integer math causes the truncation of any fractional remainder resulting from a divide operation.

In addition, nodes used for input may be either linkable or non-linkable.

4. Linkable input nodes are indicated by a  in the function block diagram.
5. Non-linkable input nodes are indicated by a  in the function block diagram.
6. The nodes used for output are not linkable and are indicated by a  in the function block diagram. However, you can use output nodes to provide data for inputs to other function blocks or to drive linear parameters.

When you connect function blocks, you need to be careful. Linkable inputs can get data from any function block node or linear parameter, regardless of its data type. As an example, you could link a signed decimal output to a logical input. The characteristics of the destination node determine how the input value will be interpreted. In the case where a signed decimal output is linked to a logical input, the value would be interpreted as a true value unless the source value (such as a velocity or position feedback) passed through zero.

## Double Word Function Block Caution

The only library function blocks that have double word input or output nodes are Multiply, Divide and Scale. These three function blocks are intended to be used together. Special handling may be required when using any of these three blocks with other function blocks.

Double word nodes can present difficulties since the system architecture does not have 32 bit integrity. When double word parameters are manipulated by DriveTools or a PLC, each 16 bit node must be handled separately. However the function algorithm uses both words together when interpreting these double word values.

The output range of multiplied input values can be critical. The range of a double word (32 bit) value is  $\pm 2,147,483,647$ . The range of the more common, signed single-word 16 bit node is  $\pm 32767$ . If only one word of the Multiply or Scale function block output were used, the output would appear to roll over or under should the product exceed  $+32767$  or go negative.



As shown above, the sign bit for 32 bit values is the most significant bit, #31. The sign bit for a single word value is the most significant bit, #15.

Should the DIVIDE function block's LSW input node (Node 0) be used without manipulating the MSW node (node 1), difficulties can occur should a signed word be linked to the inputs. False results may be output if the signed input value goes negative. Range checking and possible limiting may need to be performed. Special manipulation of the MSW (Most Significant Word) may be required. The double word characteristic of each of these three blocks is detailed further in the individual block descriptions.

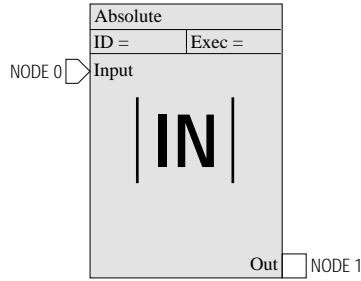


## Function Block Index

FUNCTION	BLOCK TYPE	DESCRIPTION	PAGE
ABS	1	An absolute value function block whose output is the positive value.	4-4
BIN2DEC	3	A binary to decimal function block that takes (16) input words and produces (1) decimal output word.	4-5
COMPHYST	4	Compare with hysteresis function block that checks for input = preset value with a hysteresis around the value.	4-7
DEC2BIN	5	A decimal to binary function block that takes (1) decimal input word and produces (16) binary output words.	4-9
DELAY	6	A time delay function block that echoes a logic input after a delay.	4-11
DERIV	7	A derivative function block that calculates the change in input per second.	4-13
DIVIDE	23	A divide function block that divides (2) signed integers.	4-15
EXOR2	25	An exclusive OR function that takes (2) inputs and provides (2) output values — The XOR of those values and the NOT of the output value.	4-18
FILTER	8	A first order low pass algorithm filter, with a programmable bandwidth in tenths of radians per second.	4-19
4AND	2	An and function that takes (4) inputs and performs a logical and.	4-21
4OR	16	An or function that takes the logical or of (4) inputs.	4-22
FUNCTION	9	A "function" function block that with a user approximation for a function, linearly interpolates between (2) of (5) possible points.	4-23
INTEGRATOR	10	An integrator function block that does trapezoidal integration.	4-26
LIMIT	12	A limiter function block that limits an input to programmed minimum and maximum values.	4-30
LNOT	15	A logical not function.	4-31
MINMAX	13	A minimum or maximum function block that can be programmed to take the minimum or maximum of two input values.	4-32
MONOSTABLE	14	A one shot monostable function block that elongates a rising edge signal for a specified time duration.	4-33
MULTIPLEXER	21	A select function block that multiplexes one of four inputs based on the state of the selector inputs.	4-34
MULTIPLY	28	A multiply function block that multiplies (2) signed integers.	4-35
NO-OP	0	A PLC space holder.	4-37
PI CTRL	17	A proportional/integral control function block that takes the difference between two inputs and performs a PI control with a proportional and integral gains.	4-38
PULSE CNTR	18	A pulse counter function block that counts rising edges of an input value.	4-42
RATE LIMITER	19	A "ramp" function block that limits the rate of change of an input value	4-44
SCALE	20	A scale function block that uses the following formula: $IN1 \times (MULTI/DIV)$ .	4-46
SR FF	22	A set-reset flip-flop.	4-48
SUB	27	A subtract function block that subtracts (2) signed numbers.	4-49
T-FF	11	A toggle flip flop function block, that changes the state of the input.	4-50
2ADD	26	An add function block that adds (2) signed numbers.	4-51
UP/DWN CNTR	24	An up/down counter function block that increments or decrements to a specified value in a specified amount of time.	4-52

**ABS**

BLOCK TYPE 1 decimal 1 hexadecimal



**DEFINITION**

An absolute (+) output value *Out* derived from a 16-bit signed (+ or -) 2's complement input *Input*.

**INPUT**

*Input* — A signed Integer.

**OUTPUT**

*Out* — An unsigned integer that is the absolute value of *Input*.

**FUNCTION**

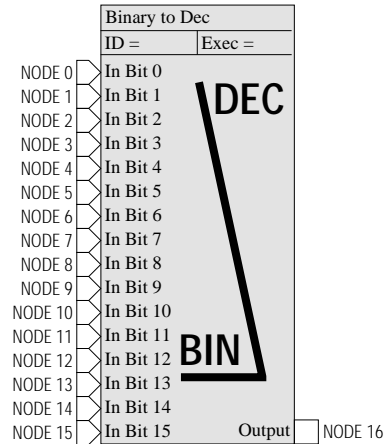
$Out = |Input|$ .

PARAMETERS	DATA TYPE	LINKABLE	DEFAULT VALUE	RANGE
<i>Input</i>	Signed Integer	Yes	0	±32767
<i>Out</i>	Signed Integer	Yes	—	0 to +32767

EXAMPLES	EXAMPLE 1	EXAMPLE 2
<i>Input</i>	4	-4
<i>Out</i>	4	4

**BIN2DEC**

BLOCK TYPE 3 decimal 3 hexadecimal

**DEFINITION**

Combines 16 logical input words *In Bit 0* – *In Bit 15* into 1 decimal output word *Output*.

**INPUTS**

*In Bit 0* – *In Bit 15* — Logical input words.

**OUTPUT**

*Output* — A decimal output word.

**FUNCTION**

If *In Bit 0* = 0, *Output* bit 0 = 0.

If *In Bit 0* ≠ 0, *Output* bit 0 = 1.

•  
•  
•

If *In Bit 15* = 0, *Output* bit 16 = 0.

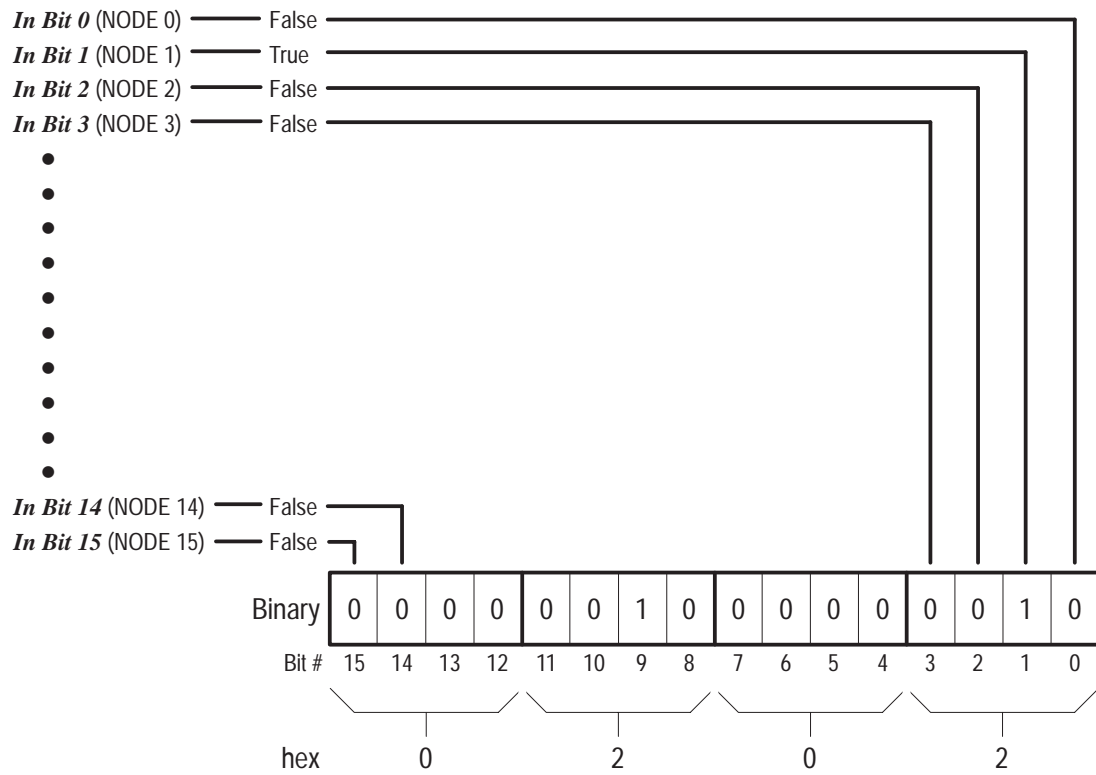
If *In Bit 15* ≠ 0, *Output* bit 16 = 1.

PARAMETERS	DATA TYPE	LINKABLE	DEFAULT VALUE	RANGE
<i>In Bit 0 – 15</i>	Logic Input	Yes	0	True/False
<i>Out Output</i>	Signed Integer	No	—	±32767

**BIN2DEC**  
(continued)

EXAMPLES	EXAMPLE 1	Example 2	EXAMPLE 3
<i>In Bit 0</i>	False	False	False
<i>In Bit 1</i>	True	True	True
<i>In Bit 2</i>	False	False	False
<i>In Bit 3</i>	False	False	True
<i>In Bit 4</i>	False	False	False
<i>In Bit 5</i>	False	False	True
<i>In Bit 6</i>	False	False	True
<i>In Bit 7</i>	False	False	True
<i>In Bit 8</i>	False	False	True
<i>In Bit 9</i>	False	True	True
<i>In Bit 10</i>	False	False	True
<i>In Bit 11</i>	False	False	True
<i>In Bit 12</i>	False	False	True
<i>In Bit 13</i>	False	False	True
<i>In Bit 14</i>	False	False	True
<i>In Bit 15</i>	False	False	True
<i>Output</i>	2	514	-22

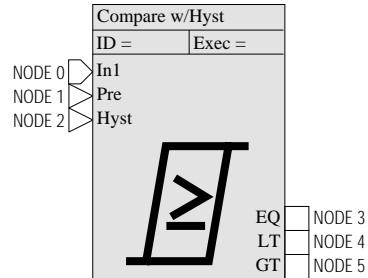
Example 2 — Output = 514<sub>dec</sub> = 0202<sub>hex</sub> = 0000 0010 0000 0010<sub>binary</sub>



**Output** decimal value (NODE 16) = 514

**COMPHYST**

BLOCK TYPE 4 decimal 4 hexadecimal

**DEFINITION**

Compares the input value *In1* against a preset value *Pre* with an associated hysteresis band *Hyst* and sets the appropriate indicator flags.

**EQ Output****INPUTS**

*In1* — Input value signed integer.

*Pre* — Preset value signed integer.

*Hyst* — Hysteresis band unsigned integer between 0 and +32767.

**OUTPUTS**

*EQ* — Equal flag set to true when the input is within the hysteresis band.

*LT* — Less than flag is set to true when  $In1 < Pre$ .

*GT* — Greater than flag is set to true when  $In1 > Pre$ .

**FUNCTION**

1. If  $In1 \leq Pre + Hyst$  and  $\geq Pre - Hyst$ ,  
then *EQ* = true, else *EQ* = false.
2. If  $In1 > Pre$ ,  
then *GT* = true and *LT* = false.
3. If  $In1 < Pre$ ,  
then *LT* = true and *GT* = false.

**COMPHYST**

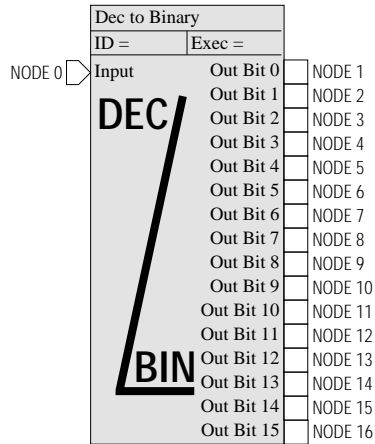
(continued)

PARAMETERS	DATA TYPE	LINKABLE	DEFAULT VALUE	RANGE
<i>In1</i>	Signed Integer	Yes	0	±32767
<i>Pre</i>	Signed Integer	No	0	±32767
<i>Hyst</i>	Unsigned Integer	No	0	0 to +32767
<i>EQ</i>	Logic Output	No	—	True/False
<i>LT</i>	Logic Output	No	—	True/False
<i>GT</i>	Logic Output	No	—	True/False

EXAMPLES	EXAMPLE 1	EXAMPLE 2
<i>In1</i>	8	15
<i>Pre</i>	10	10
<i>Hyst</i>	3	3
<i>EQ</i>	True	False
<i>LT</i>	True	False
<i>GT</i>	False	True

**DEC2BIN**

BLOCK TYPE 5 decimal 5 hexadecimal

**DEFINITION**

Takes 1 unsigned decimal input word *Input* and produces 16 logical output words *Out Bit 0* to *Out Bit 15*.

**INPUT**

*Input* — A decimal input word.

**OUTPUT**

*Out Bit 0* – *Out Bit 15* — Logical output words.

**FUNCTION**

If *Input* bit 0 = 0, *Out Bit 0* = 0.

If *Input* bit 0 = 1, *Out Bit 0* = 65535.

•  
•  
•

If *Input* bit 15 = 0, *Out Bit 15* = 0.

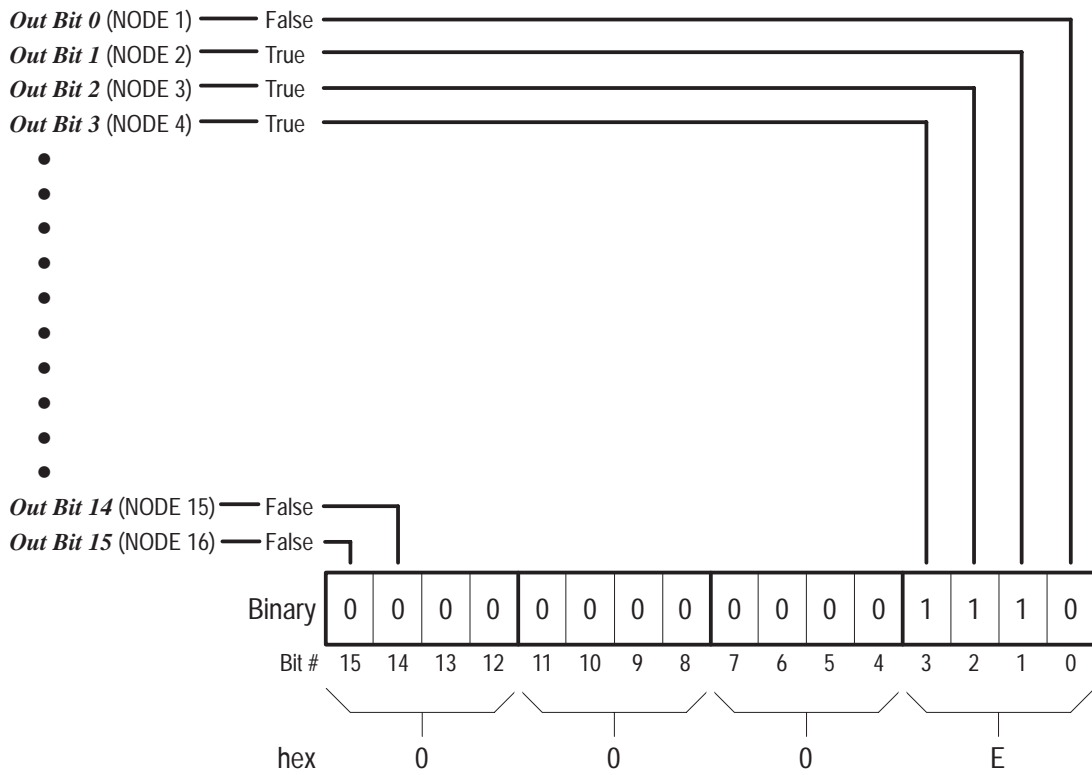
If *Input* bit 15 = 1, *Out Bit 15* = 65535.

PARAMETERS	DATA TYPE	LINKABLE	DEFAULT VALUE	RANGE
<i>Input</i>	Unsigned Integer	Yes	0	0 – 65535
<i>Out Bit 0 – 15</i>	Logic Output	No	—	True/False

**DEC2BIN**  
(continued)

EXAMPLES	Example 1	EXAMPLE 2
<b>Input</b>	14	65584
<b>Out Bit 0</b>	False	False
<b>Out Bit 1</b>	True	False
<b>Out Bit 2</b>	True	False
<b>Out Bit 3</b>	True	False
<b>Out Bit 4</b>	False	True
<b>Out Bit 5</b>	False	True
<b>Out Bit 6</b>	False	False
<b>Out Bit 7</b>	False	False
<b>Out Bit 8</b>	False	False
<b>Out Bit 9</b>	False	False
<b>Out Bit 10</b>	False	False
<b>Out Bit 11</b>	False	False
<b>Out Bit 12</b>	False	False
<b>Out Bit 13</b>	False	False
<b>Out Bit 14</b>	False	False
<b>Out Bit 15</b>	False	True

Example 1 — Input = 14<sub>dec</sub> = 000E<sub>hex</sub> = 0000 0000 0000 1110<sub>binary</sub>

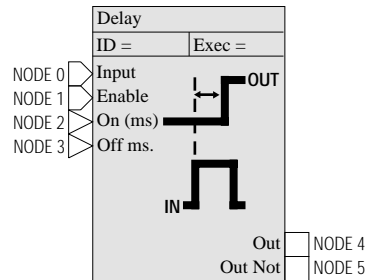


**Input** decimal value (NODE 0) = 14



**DELAY**

BLOCK TYPE 6 decimal 6 hexadecimal

**DEFINITION**

The output echoes the logical input after a specified time delay. Separate time delays of *On (ms)* and *Off ms.* are provided for rising and falling edges. The resolution of the on and off delay times are calculated and limited by the 20mS task interval.

**INPUTS**

**Input** — A logic input.

**Enable** — When true, enables the delay function. When false, holds the output at it's last state.

**On (ms)** — On time delay, entered in 20mS increments.

**Off ms.** — Off time delay, entered in 20mS increments.

**OUTPUTS**

**Out** — A logical output that follows **Input** if **Enable** is true.

**Out Not** — A logical output that is the complement of **Out**.

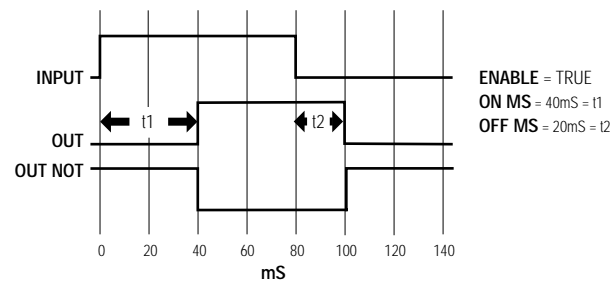
**FUNCTION**

1. If **Enable**  $\neq$  0:
  - For the rising edge of **Input**, the on delay is in progress and the **On (ms)** counter =  $mS \div 20$ .
  - For the falling edge of **Input**, the off delay is in progress and the **Off ms.** counter =  $mS \div 20$ .
2. For the **On (ms)** delay:
  - Decrement the **On (ms)** counter
  - If **On (ms)** counter = 0, then **Out** is true and **Out Not** is false.
3. For the **Off ms.** delay:
  - Decrement the **Off ms.** counter
  - If **Off ms.** counter = 0, then **Out** is false and **Out Not** is true.

**DELAY**

(continued)

PARAMETERS	DATA TYPE	LINKABLE	DEFAULT VALUE	RANGE
<i>Input</i>	Logic Input	Yes	0	True/False
<i>Enable</i>	Logic Input	Yes	0	True/False
<i>On (ms)</i>	Unsigned Integer	No	0	0 to 65535
<i>Off ms.</i>	Unsigned Integer	No	0	0 to 65535
<i>Out</i>	Logic Output	No	—	True/False
<i>Out Not</i>	Logic Output	No	—	True/False

**Example****Important:**

The on and off timers operate independently.

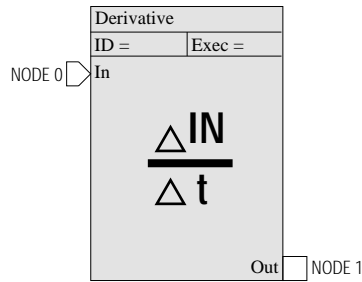
Once the on delay timer (t1) is initiated, subsequent *rising* edges are ignored.

Once the off delay timer (t2) is initiated, subsequent *falling* edges are ignored.

1. The on duration time of input pulses should be checked when the **ON MS** > **OFF MS**.
2. The off duration time of input pulses should be checked when the **OFF MS** > **ON MS**.

**DERIV**

BLOCK TYPE 7 decimal 7 hexadecimal



**DEFINITION**

The rate of change of input *In* over a single sample interval. The sample interval  $\Delta t = .020$  seconds. Output *Out* is clamped at  $\pm 32767$  and not allowed to over or under flow.

**INPUT**

*In* — A signed Integer.

**OUTPUT**

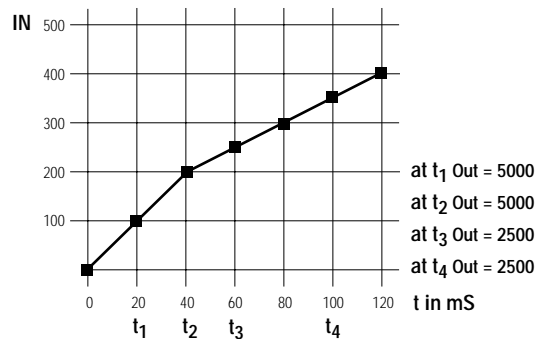
*Out* — A signed integer representing the derivative or change in input in units per second.

**FUNCTION**

$$Out = 50 \times [In - In \text{ (previous)}].$$

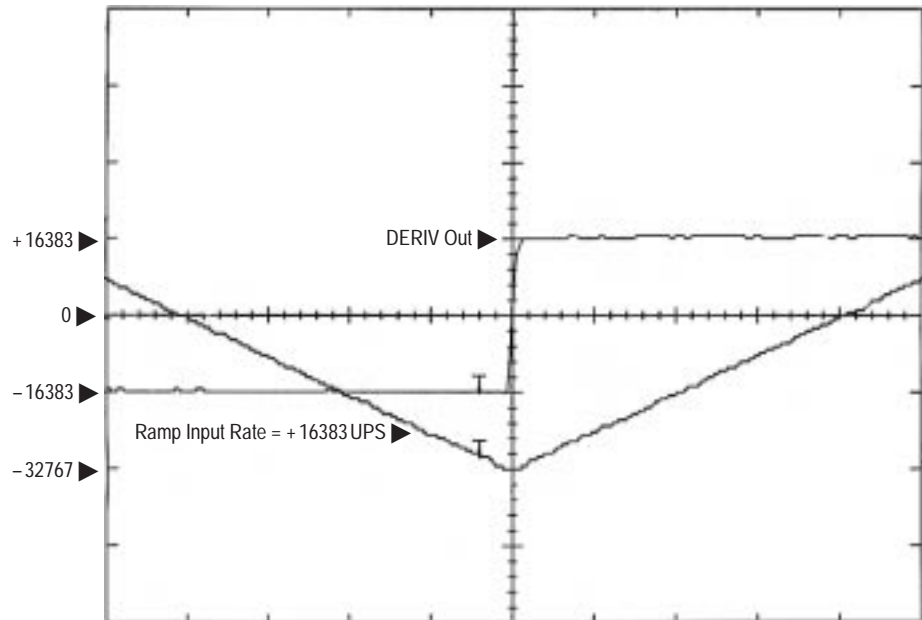
PARAMETERS	DATA TYPE	LINKABLE	DEFAULT VALUE	RANGE
<i>In</i>	Signed Integer	Yes	0	$\pm 32767$
<i>Out</i>	Signed Integer	Yes	—	$\pm 32767$

**Example 1**



**DERIV**

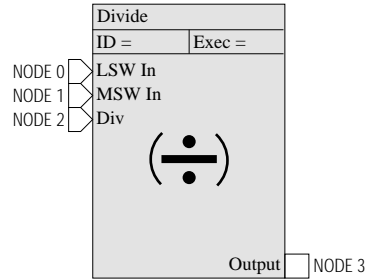
(continued)

**Example 2 — Rate = 16383 Units-per-Second (UPS)**

In the example above, the rate limit input to the derivative function is set to allow a 16383 UPS change of output. A constant rate-of-change to the input, yields a constant output level of  $\pm 16383$ , with the output sign changing with the input slope.

**DIVIDE**

BLOCK TYPE 23 decimal 17 hexadecimal

**DEFINITION**

Divides a 32 bit signed integer by a 16 bit signed integer. Any remainder is truncated. If *Div* = 0, the calculation is not performed and *Output* = 0. The *Output* is clamped to  $\pm 32767$  should the result exceed the limits.

**INPUTS**

*LSW In* — A least significant dividend word value representing bits 0–15 of a 32 bit dividend value.

*MSW In* — A most significant dividend signed integer value representing bits 16–31 of a 32 bit dividend value.

*Div* — The signed integer divisor.

**OUTPUT**

*Output* — The result of dividing the dividend by the divisor.

**FUNCTION**

$Output = LSW\ In, MSW\ In \div Div.$

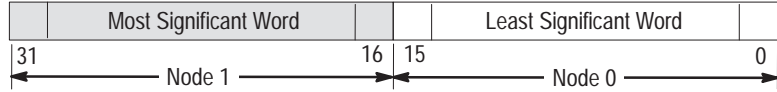
PARAMETERS	DATA TYPE	LINKABLE	DEFAULT VALUE	RANGE
<i>LSW In</i>	Unsigned Integer	Yes	0	0 to 65535
<i>MSW In</i>	Signed Integer	Yes	0	$\pm 32767$
<i>Div</i>	Signed Integer	Yes	0	$\pm 32767$
<i>Output</i>	Signed Integer	No	—	$\pm 32767$

**DIVIDE**

(continued)

**DOUBLE WORD VALUES**

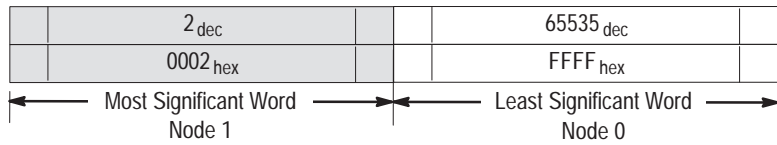
Both the Most Significant Word and the Least Significant Word are interpreted together by the function algorithm as one word when using 32 bit values. The range of a double word value is  $\pm 2,147,483,647$ . The range of the more common, signed single-word 16 bit node is  $\pm 32767$ .



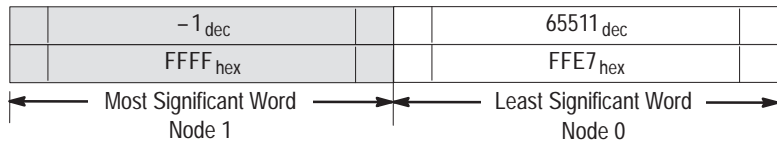
Bit 31 is the sign bit, the most significant bit for 32 bit values. For 16 bit values, bit 15 (the sign bit) is the most significant bit.

EXAMPLES	EXAMPLE 1	EXAMPLE 2	Example 3	Example 4
<i>LSW In</i>	15	20	65535	65511
<i>MSW In</i>	0	0	2	-1
INPUT VALUE	15	20	196607	-25
<i>Div</i>	4	0	42	5
<i>Output</i>	3	0	4681	-5

**Example 3 — Double Word Input Value = +196607<sub>dec</sub> = 0002 FFFF<sub>hex</sub>**



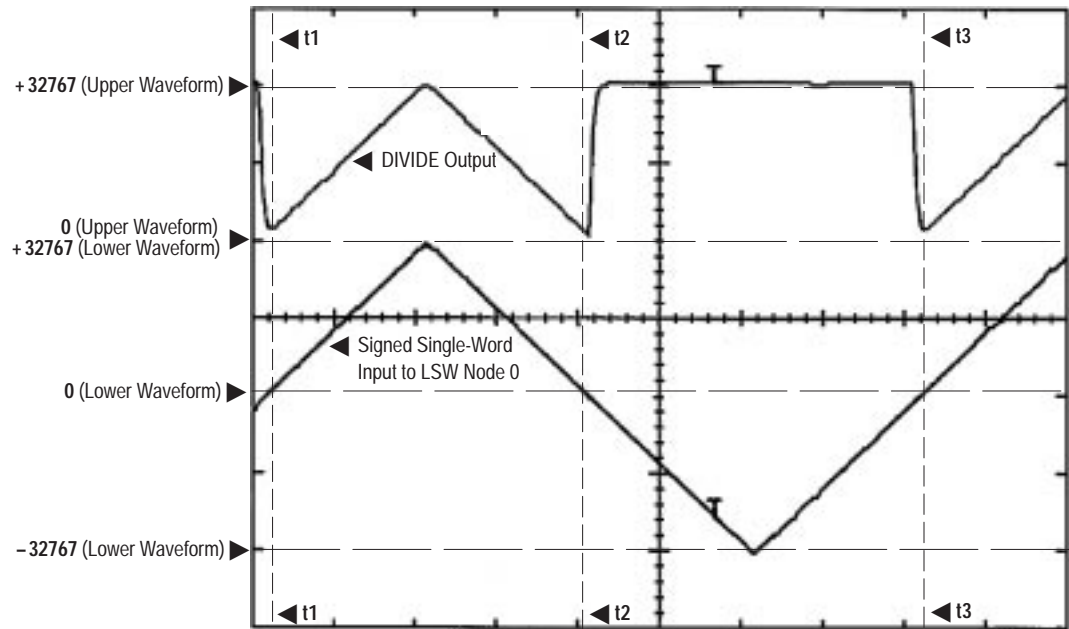
**Example 4 — Double Word Input Value = -25<sub>dec</sub> = FFFF FFE7<sub>hex</sub>**



## DIVIDE

(continued)

**Important:** If the DIVIDE function block's LSW input node (node 0) is used without manipulating the MSW node (node 1), difficulties can occur should a signed word be linked to the inputs. False results may be output if the signed input value goes negative.



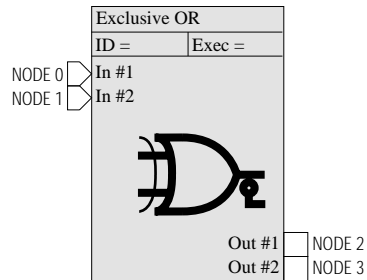
The Lower Waveform is a signed single word input traversing a range of  $\pm 32767$  units. This is linked to the LSW input node of the DIVIDE block. The MSW node is unlinked and is set at a value of 0. The divisor (node 2), has a value of 1. Points t1, t2 and t3 denote where the signed input signal crosses zero. Note that between time t1 and t2 the output follows the positive input value exactly. As the input moves negative (at time t2), the output goes to the positive limit of + 32767.

The signed single word value which represents  $-1_{dec}$  is  $FFFF_{hex}$ . The signed double word value representing  $-1_{dec}$  is  $FFFF FFFF_{hex}$ . Without the upper MSW word (node 1) being manipulated, the DIVIDE block interprets the double word input as being  $0000 FFFF_{hex}$  which equals  $65535_{dec}$ .

With the MSW (node 1) always 0, the lower LSW (node 0) is always interpreted as an unsigned decimal word with a range of 0 to 65535. As the signed single word RATE LIM Output (which is linked to the DIVIDE LSW input) goes negative, its sign (bit 15) is always set. While bit 15 is set, the DIVIDE LSW input value is always interpreted as being greater than + 32768 because the LSW is always positive when the DIVIDE MSW input is 0.

**EXOR2**

BLOCK TYPE 25 decimal 19 hexadecimal

**DEFINITION**

An exclusive OR function that takes (2) inputs *In #1* and *In #2*., and provides the XOR and XNOR — *Out #1* and *Out #2*.

**INPUTS**

*In #1* — A logical input value.

*In #2* — A logical input value.

**OUTPUTS**

*Out #1* — A logical output value.

*Out #2* — A logical output value.

PARAMETERS	DATA TYPE	LINKABLE	DEFAULT VALUE	RANGE
<i>In #1</i>	Logic Input	Yes	False	True/False
<i>In #2</i>	Logic Input	Yes	False	True/False
<i>Out #1</i>	Logic Output	No	—	True/False
<i>Out #2</i>	Logic Output	No	—	True/False

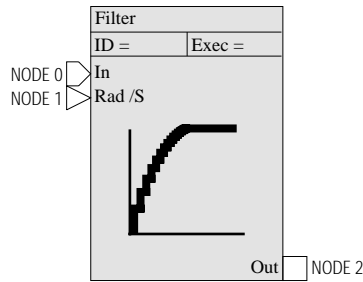
**Example**

<i>In #1</i>	<i>In #2</i>	<i>Out #1</i>	<i>Out #2</i>
False	False	False	True
False	True	True	False
True	False	True	False
True	True	False	True



**FILTER**

BLOCK TYPE 8 decimal 8 hexadecimal



**DEFINITION**

A first order low pass algorithmic filter with a programmable bandwidth in increments of .1 radians/second.

**INPUTS**

*In* — The signed integer to be filtered.

*Rad/S* — The bandwidth in .1 radians/second, with a maximum value of 400 (40 radians).

**OUTPUT**

*Out* — A signed, filtered, output integer that is clamped to ±32767.

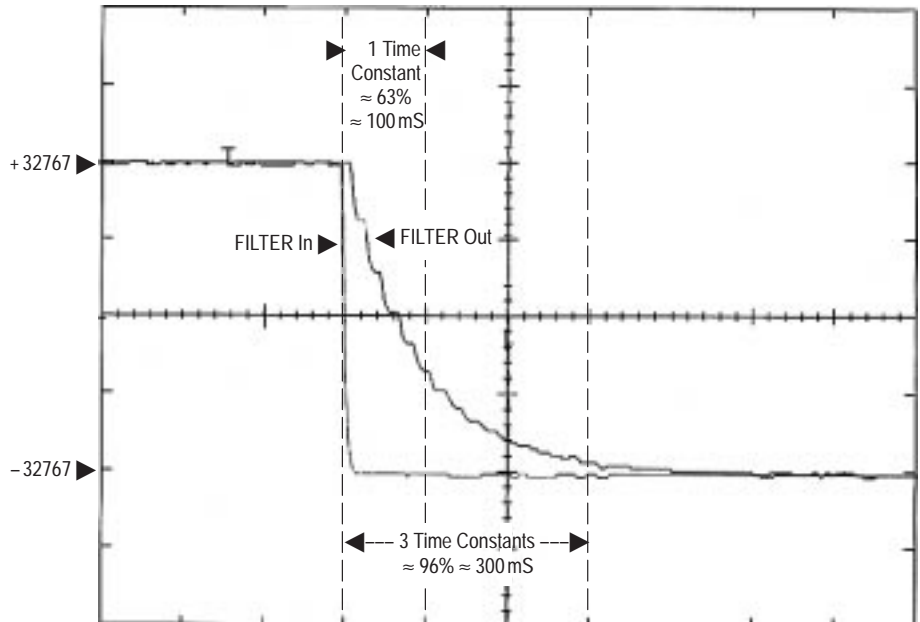
**FUNCTION**

If *Rad/S* = 0, then *Out* = *In*.

If *Rad/S* ≠ 0, then *Out* = *Out*<sub>T-1</sub> + k(*In* - *Out*<sub>T-1</sub>).

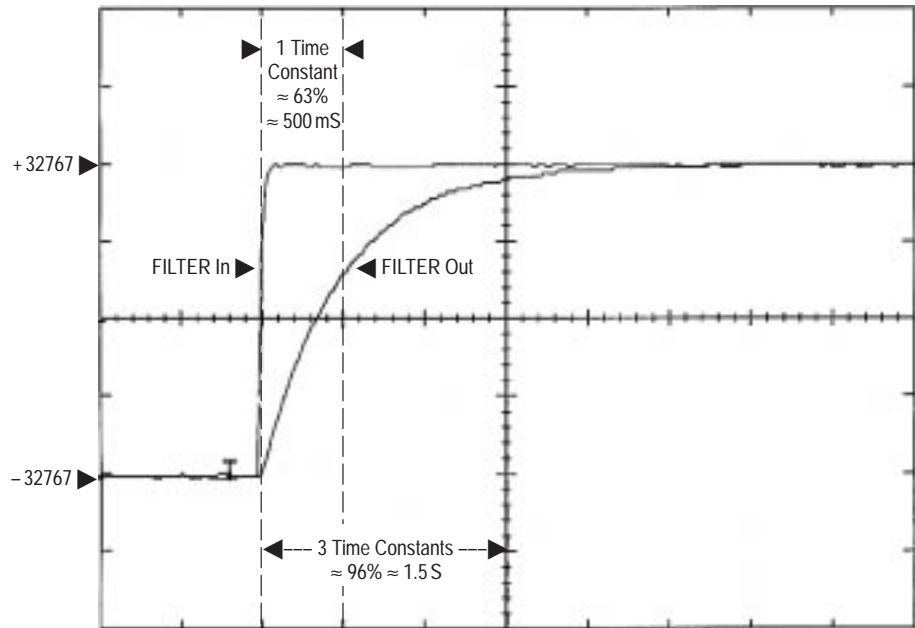
PARAMETERS	DATA TYPE	LINKABLE	DEFAULT VALUE	RANGE
<i>In</i>	Signed Integer	Yes	0	±32767
<i>Rad/S</i>	Unsigned Integer	No	0	0 to 400
<i>Out</i>	Signed Integer	No	—	±32767

Example 1 — (10) Radians-per-Second — Horizontal Scale = 100mS/Division

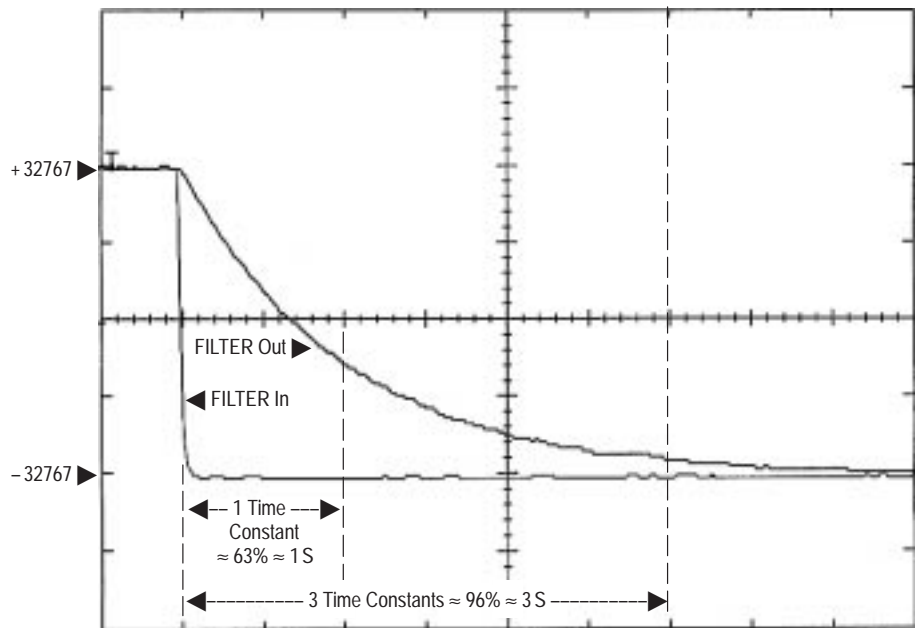


**FILTER**  
(continued)

**Example 2 — (2) Radians-per-Second — Horizontal Scale = 500 mS/Division**

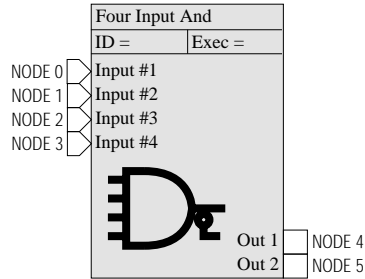


**Example 3 — (1) Radian-per-Second — Horizontal Scale = 500 mS/Division**



**4AND**

BLOCK TYPE 2 decimal 2 hexadecimal

**DEFINITION**

Performs a logic AND and NAND of *Input #1*, *Input #2*, *Input #3*, and *Input #4*.

**INPUTS**

*Input #1* — A logic input value.

*Input #2* — A logic input value.

*Input #3* — A logic input value.

*Input #4* — A logic input value.

**OUTPUTS**

*Out 1* — A logic output value.

*Out 2* — A logic output value.

**FUNCTION**

$Out\ 1 = Input\ \#1 \ \& \ Input\ \#2 \ \& \ Input\ \#3 \ \& \ Input\ \#4.$

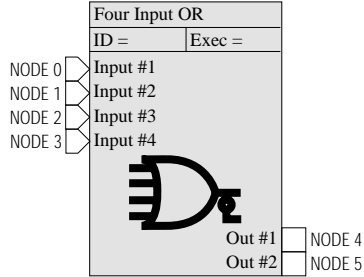
$Out\ 2 = (Not)Out\ 1.$

PARAMETERS	DATA TYPE	LINKABLE	DEFAULT VALUE	RANGE
<i>Input #1</i>	Logic Input	Yes	True	True/False
<i>Input #2</i>	Logic Input	Yes	True	True/False
<i>Input #3</i>	Logic Input	Yes	True	True/False
<i>Input #4</i>	Logic Input	Yes	True	True/False
<i>Out 1</i>	Logic Output	No	—	True/False
<i>Out 2</i>	Logic Output	No	—	True/False

EXAMPLES	EXAMPLE 1	EXAMPLE 2
<i>Input #1</i>	True	True
<i>Input #2</i>	False	True
<i>Input #3</i>	False	True
<i>Input #4</i>	False	True
<i>Out 1</i>	False	True
<i>Out 2</i>	True	False

**40R**

BLOCK TYPE 16 decimal 10 hexadecimal



**DEFINITION**

Performs a logic OR and NOR on four input words.

**INPUTS**

- Input #1* — A logic input value.
- Input #2* — A logic input value.
- Input #3* — A logic input value.
- Input #4* — A logic input value.

**OUTPUTS**

- Out #1* — A logic output value.
- Out #2* — A logic output value.

**FUNCTION**

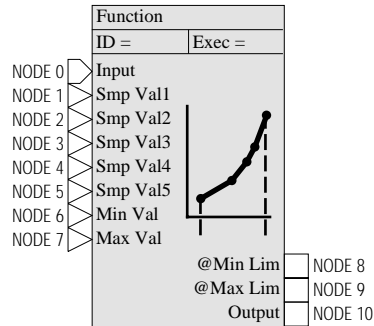
- Out #1* = *Input #1* or *Input #2* or *Input #3* or *Input #4*.
- Out #2* = (Not)*Out #1*.

PARAMETERS	DATA TYPE	LINKABLE	DEFAULT VALUE	RANGE
<i>Input #1</i>	Logic Input	Yes	False	True/False
<i>Input #2</i>	Logic Input	Yes	False	True/False
<i>Input #3</i>	Logic Input	Yes	False	True/False
<i>Input #4</i>	Logic Input	Yes	False	True/False
<i>Out #1</i>	Logic Output	No	—	True/False
<i>Out #2</i>	Logic Output	No	—	True/False

EXAMPLES	EXAMPLE 1	EXAMPLE 2	EXAMPLE 3
<i>Input #1</i>	False	False	True
<i>Input #2</i>	True	False	True
<i>Input #3</i>	False	False	True
<i>Input #4</i>	False	False	True
<i>Out #1</i>	True	False	True
<i>Out #2</i>	False	True	False

**FUNCTION**

BLOCK TYPE 9 decimal 9 hexadecimal

**DEFINITION**

A function generator that uses:

- Sample values *Smp Val1*—*Smp Val5* to define the y-axis components.
- (2) signed integers *Min Val* & *Max Val* to describe the x-axis components spaced in  $(Max Val - Min Val)/4$  increments.
- Interpolation between the y-components to calculate the output *Output*.

**INPUTS**

*Input* — A signed integer specifying an x-axis coordinate.

*Smp Val1*—*Smp Val5* — Signed integers representing the y-axis components.

*Min Val* — A signed integer associated with *Smp Val1* that defines the smallest x-axis component.

*Max Val* — A signed integer associated with *Smp Val5* that defines the largest x-axis component.

**OUTPUTS**

*@Min Lim* — A logic value = true when *Input* < *Min Val*.

*@Max Lim* — A logic value = true when *Input* > *Max Val*.

*Output* — A signed integer representing the y-axis value, that corresponds to the x-axis value specified by *Input*.

**FUNCTION**

1. If *Input* > *Max Val*, *@Max Lim* = true, *@Min Lim* = false, and *Output* = *Smp Val5*.
2. If *Input* < *Min Val*, *@Max Lim* = false, *@Min Lim* = true, and *Output* = *Smp Val1*.

**FUNCTION**  
(continued)

**3. If *Min Val* < *Input* and < *Max Val*:**

Calculate  $x_i, x_{i+1}$  from *Input*, where  $x_i \leq \text{Input} \leq x_{i+1}$ .

Calculate  $y_i, y_{i+1}$  from  $x_i, x_{i+1}$ .

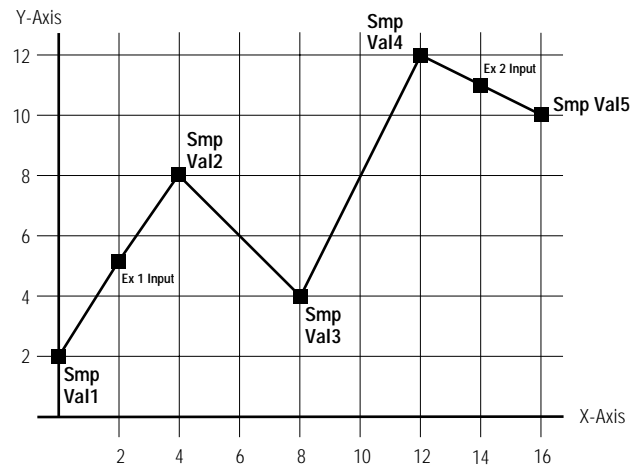
$$\text{Output} = \left\{ \frac{[(y_{i+1} - y_i) \times (\text{Input} - x_i)]}{(x_{i+1} - x_i)} \right\} + y_i$$

**@Max Lim = @Min Lim = false.**

PARAMETERS	DATA TYPE	LINKABLE	DEFAULT VALUE	RANGE
<i>Input</i>	Signed Integer	Yes	0	±32767
<i>Smp Val1 - 5</i>	Signed Integer	No	0	±32767
<i>Min Val</i>	Signed Integer	No	0	±32767
<i>Max Val</i>	Signed Integer	No	4	±32767
@ <i>Min Lim</i>	Logic Output	No	—	True/False
@ <i>Max Lim</i>	Logic Output	No	—	True/False
<i>Output</i>	Signed Integer	No	—	±32767

EXAMPLES	EXAMPLE 1	EXAMPLE 2	EXAMPLE 3	EXAMPLE 4
<i>Input</i>	2	14	±32767	±32767
<i>Smp Val1</i>	2	2	16383	16383
<i>Smp Val2</i>	8	8	32767	32767
<i>Smp Val3</i>	4	4	0	0
<i>Smp Val4</i>	12	12	8192	8192
<i>Smp Val5</i>	10	10	-16383	-16383
<i>Min Val</i>	0	0	-16383	-32767
<i>Max Val</i>	16	16	+16383	+32767
@ <i>Min Lim</i>	0	0	-16383	-32767
@ <i>Max Lim</i>	16	16	+16383	+32767
<i>Output</i>	5	11	See plot	See plot

**Examples 1 & 2**



Y-Axis coordinate **Smp Val1** is located at X-Axis coordinate **Min Val**.

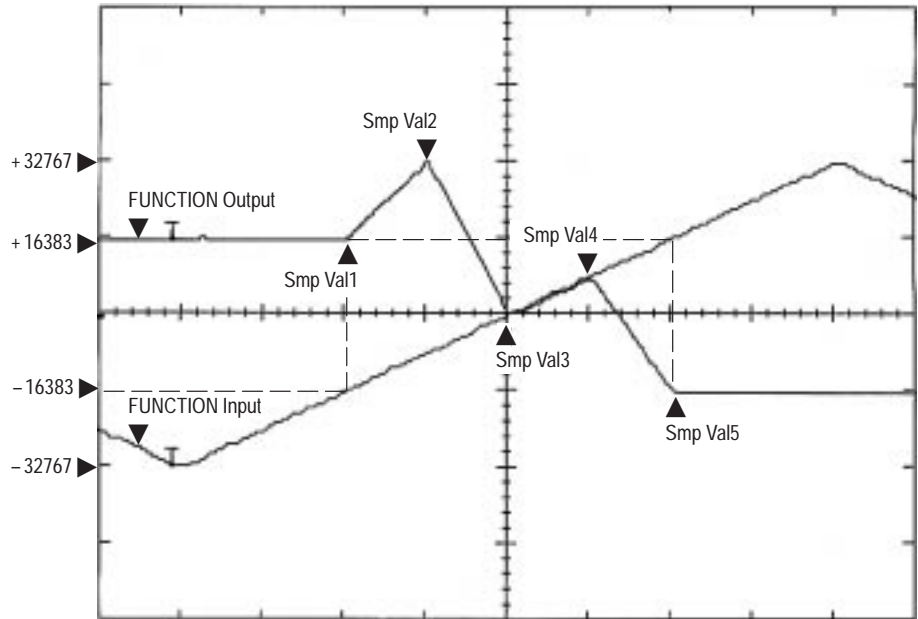
Y-Axis coordinate **Smp Val5** is located at X-Axis coordinate **Max Val**.

**Smp Val2, Smp Val3, & Smp Val4**, are located at equal X-Axis coordinates between **Smp Val1 & Smp Val5**.

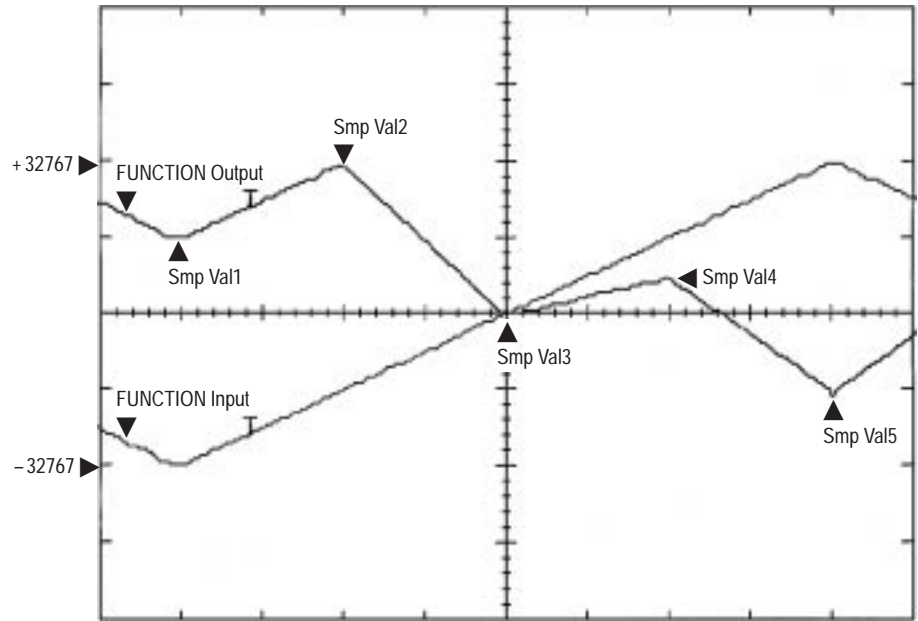
FUNCTION interpolates to determine the Y-Axis coordinate of the **Input**.

**FUNCTION**  
(continued)

**Example 3**

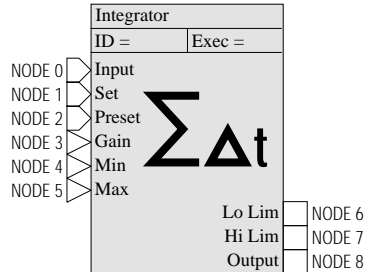


**Example 4**



**INTEGRATOR**

BLOCK TYPE 10 decimal 0A hexadecimal

**DEFINITION**

Integrates an input value *Input* over a period of time using trapezoidal integration. The *Output* is limited to *Min* and *Max* values that are defined by the user. The integrator can be set to the *Preset* value when the *Set* input is true.

**INPUTS**

*Input* — A signed integer signal that will be integrated.

*Set* — A logic input that forces the *Output* and integrated internal variable to the *Preset* value when not equal to 0.

*Preset* — A signed integer that is loaded into the integrator when *Set* is true.

*Gain* — A scaled, signed integer that is multiplied by the sum of the current and last inputs — Scaling 256 = effective gain of 1.

*Min* — A signed integer that is the lower limit on the integrated *Output*.

*Max* — A signed integer that is the upper limit on the integrated *Output*.

**OUTPUTS**

*Lo Lim* — A logic value = true when the integrated value < *Min*.

*Hi Lim* — A logic value = true when the integrated value > *Max*.

*Output* — A signed value that is the integral of *Input* with respect to time.

**Important:** The *Output* is clamped between the *Hi* and *Lo Lim* values and cannot over or under flow. When the *Output* reaches either of these limits, the internal accumulator clamps and will not integrate beyond that limit.



**INTEGRATOR**

(continued)

**FUNCTION**

$$\text{Denominator} = 1/2 \times \Delta t \times 1/(\text{divisor gain}) = 25600$$

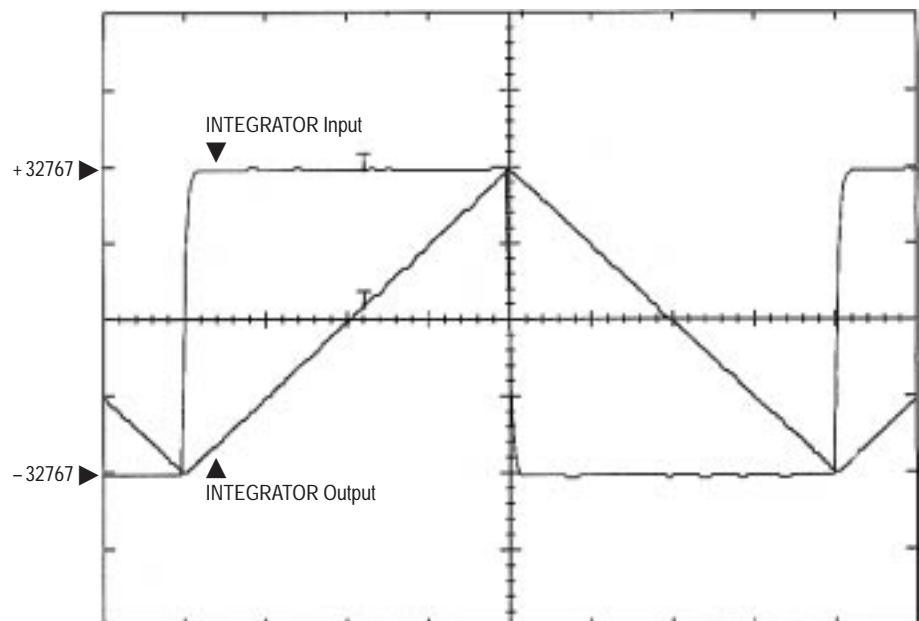
where:  $\Delta t$  = a task interval of .020 seconds = 1/50 samples/second.

divisor gain = 256.

1. If *Set* = true, accumulator = *Preset* × denominator.
2. If *Set* = false,  
accumulator = [*Gain* × (*Input*<sub>i-1</sub> + *Input*)] +  
previous accumulated value.
3. If accumulator ÷ denominator > *Max*,  
*Output* = *Max*, *Hi Lim* = true, *Lo Lim* = false.
4. If accumulator ÷ denominator < *Min*,  
*Output* = *Min*, *Hi Lim* = false, *Lo Lim* = true.
5. If accumulator ÷ denominator < *Max* and > *Min*,  
*Output* = accumulator ÷ denominator, *Hi Lim* = false,  
*Lo Lim* = false.

PARAMETERS	DATA TYPE	LINKABLE	DEFAULT VALUE	RANGE
<i>Input</i>	Signed Integer	Yes	0	±32767
<i>Set</i>	Logic Input	Yes	False	True/False
<i>Preset</i>	Signed Integer	Yes	0	±32767
<i>Gain</i>	Signed Integer	No	256	±16383
<i>Min</i>	Signed Integer	No	0	0 to -32767
<i>Max</i>	Signed Integer	No	0	0 to +32767
<i>Lo Lim</i>	Logic Output	No	—	True/False
<i>Hi Lim</i>	Logic Output	No	—	True/False
<i>Output</i>	Signed Integer	No	—	±32767

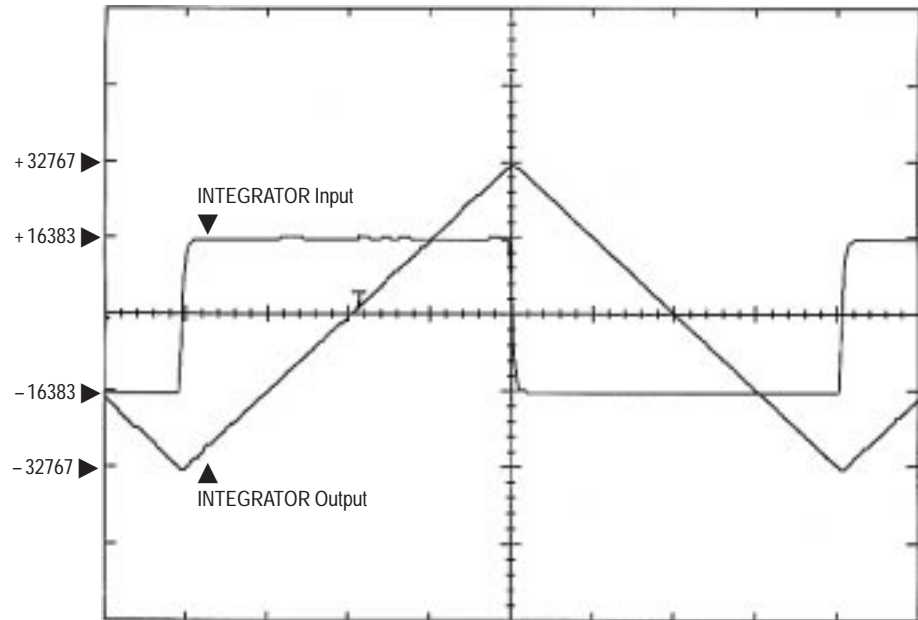
**Example 1 — Gain = 256 = 1× — Horizontal Scale = 500 mS/Division**



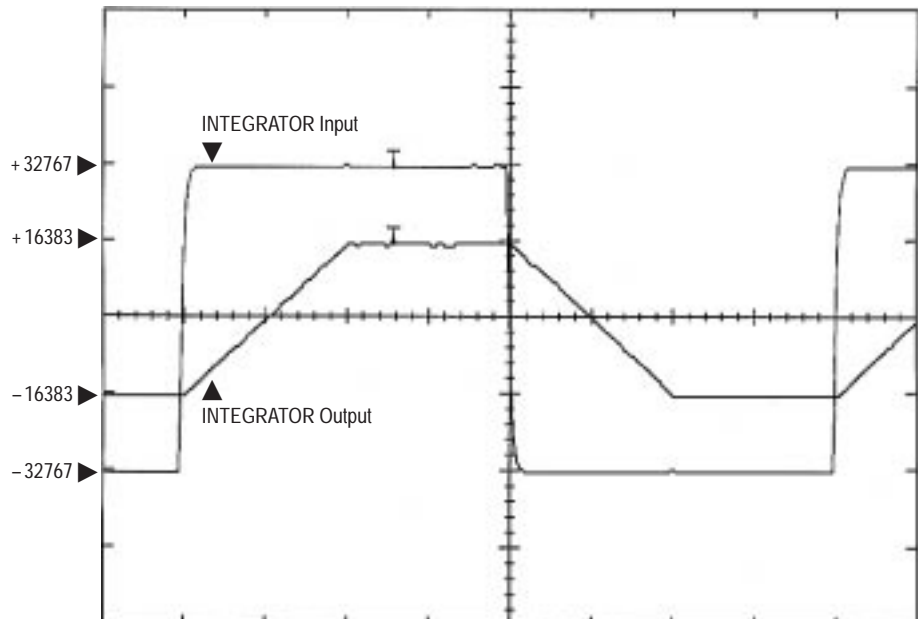
In Example 1, the INTEGRATOR Output accumulates at a rate of 32767 units-per-second, an amount equal to its constant INTEGRATOR Input level. It takes (2) seconds to traverse the entire range.

**INTEGRATOR**

(continued)

**Example 2 — Gain = 512 = 2× — Horizontal Scale = 500mS/Division**

In Example 2, the Output also changes at a rate of 32767 units-per-second. The Input level here is only 16383, but input Gain has doubled.

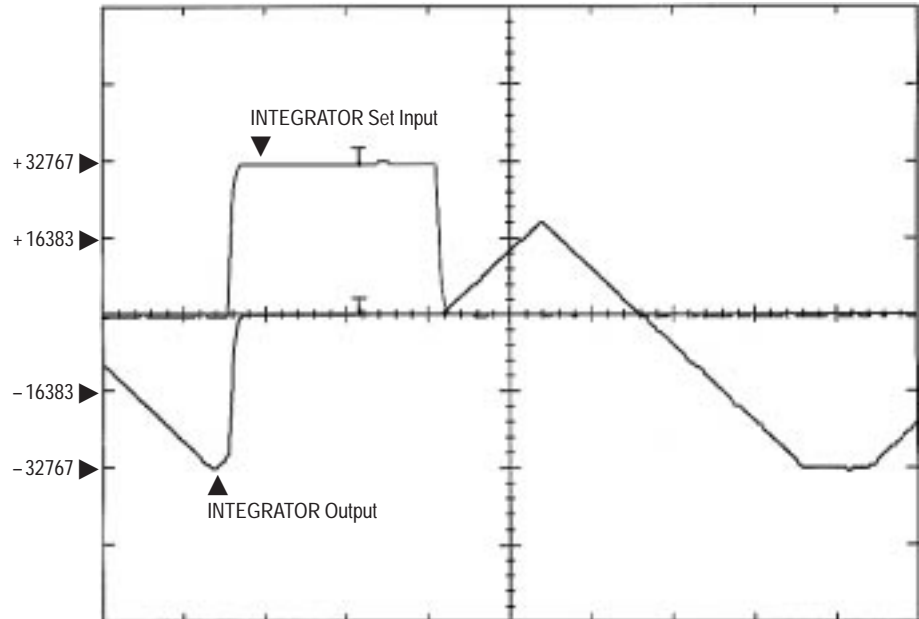
**Example 3 — Min & Max = ±16383 — Horizontal Scale = 500mS/Division**

In Example 3, the Max and Min input limits have been set at  $\pm 16383$  and the Output is clamped accordingly. The internal accumulator is also held at these limits and not allowed to accumulate beyond these limits. This allows the Output to come out of its limits as soon as the input moves in the negative direction.

## INTEGRATOR

(continued)

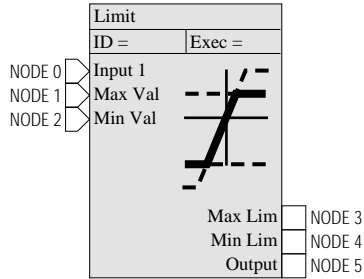
Example 4 — Preset Verification — Preset = 0 — Horizontal Scale = 500mS/Division



Example 4 demonstrates the Preset functionality. When the Set input is raised the Output is preset to a value of zero. When the Set node is cleared, the integral begins accumulating from the Preset value, 0.

**LIMIT**

BLOCK TYPE 12 decimal 0C hexadecimal



**DEFINITION**

Limits an input *Input 1* to the programmed maximum *Max Val* and minimum *Min Val* values.

**INPUTS**

*Input 1* — A signed integer that is limited.

*Max Val* — A signed integer that represents a maximum input value.

*Min Val* — A signed integer that represents a minimum input value.

**OUTPUTS**

*Max Lim* — High limit flag that is true when *Input 1* > *Max Val*.

*Min Lim* — Low limit flag that is true when *Input 1* < *Min Val*.

*Output* — A signed integer that results from limiting *Input 1*.

**FUNCTION**

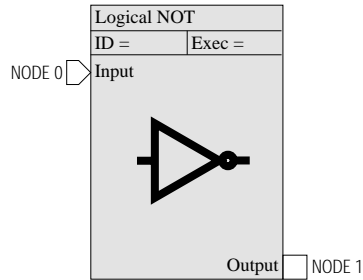
1. If *Input 1* < *Min Val*, then *Output* = *Min Val* and *Min Lim* is true.
2. If *Input 1* > *Max Val*, then *Output* = *Max Val* and *Max Lim* is true.
3. If *Input 1* < *Max Val* and > *Min Val*, then both are false and *Output* = *Input 1*.

PARAMETERS	DATA TYPE	LINKABLE	DEFAULT VALUE	RANGE
<i>Input 1</i>	Signed Integer	Yes	0	±32767
<i>Max Val</i>	Signed Integer	Yes	0	±32767
<i>Min Val</i>	Signed Integer	Yes	0	±32767
<i>Max Lim</i>	Logic Output	No	—	True/False
<i>Min Lim</i>	Logic Output	No	—	True/False
<i>Output</i>	Signed Integer	No	—	±32767

EXAMPLES	EXAMPLE 1	EXAMPLE 2	EXAMPLE 3
<i>Input 1</i>	5	11	5
<i>Max Val</i>	10	10	-10
<i>Min Val</i>	-25	-25	10
<i>Max Lim</i>	False	True	True
<i>Min Lim</i>	False	False	False
<i>Output</i>	5	10	-10

**LNOT**

BLOCK TYPE 15 decimal 0F hexadecimal

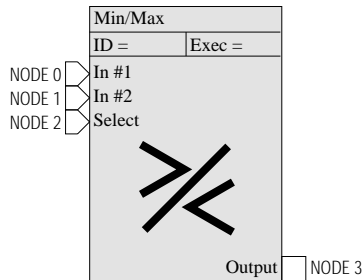
**DEFINITION**Performs a logic inversion of the *Input*.**INPUT***Input* — A logic input value.**OUTPUT***Output* — A logic output value.**FUNCTION** $Output = (Not)Input.$ 

PARAMETERS	DATA TYPE	LINKABLE	DEFAULT VALUE	RANGE
<i>Input</i>	Logic Input	Yes	False	True/False
<i>Output</i>	Logic Output	No	—	True/False

EXAMPLES	EXAMPLE 1	EXAMPLE 2
<i>Input</i>	False	True
<i>Output</i>	True	False

**MINMAX**

BLOCK TYPE 13 decimal 0D hexadecimal

**DEFINITION**

Chooses the minimum or maximum of two input values *In #1*, *In #2* according to the *Select* Input.

**INPUTS**

*In #1* — A signed integer that is compared to *In #2*.

*In #2* — A signed integer that is compared to *In #1*.

*Select* — When = 0 will select the minimum function.  
When ≠ 0 will select the maximum function.

**OUTPUT**

*Output* — A signed integer that is the minimum or maximum of *In #1* and *In #2*.

**FUNCTION**

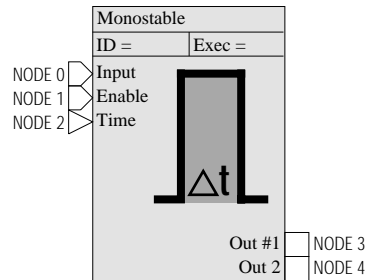
1. If *Select* is false and  $In\ #1 < In\ #2$ , then  $Output = In\ #1$ .
2. If *Select* is false and  $In\ #1 > In\ #2$ , then  $Output = In\ #2$ .
3. If *Select* is true and  $In\ #1 < In\ #2$ , then  $Output = In\ #2$ .
4. If *Select* is true and  $In\ #1 > In\ #2$ , then  $Output = In\ #1$ .

PARAMETERS	DATA TYPE	LINKABLE	DEFAULT VALUE	RANGE
<i>In #1</i>	Signed Integer	Yes	0	±32767
<i>In #2</i>	Signed Integer	Yes	0	±32767
<i>Select</i>	Logic Input	Yes	0	True/False
<i>Output</i>	Signed Integer	No	—	±32767

EXAMPLES	EXAMPLE 1	EXAMPLE 2
<i>In #1</i>	5	5
<i>In #2</i>	-8	-8
<i>Select</i>	0	1
<i>Output</i>	-8	5

**MONOSTABLE**

BLOCK TYPE 14 decimal 0E hexadecimal

**DEFINITION**

Elongates a rising edge input signal *Input* for a duration *Time*. The output signal *Out #1* is true for the duration set by *Time*. The *Time* resolution is limited by the task interval and calculated by counting 20mS task intervals.

**INPUTS**

*Input* — A signal that triggers the monostable function.

*Enable* — A logic input that when  $\neq 0$  enables the monostable function, and when  $= 0$  forces *Out #1* = 0.

*Time* — Represents the time in mS that the input signal *Input* is held high. *Time* value must be entered in 20mS increments.

**OUTPUTS**

*Out #1* — A logic output value.

*Out 2* — The inverse of *Out #1*.

**FUNCTION**

If delay = 0 then,

    If rising edge input

        Set delay =  $Time \div 20$

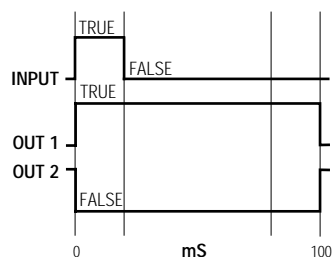
*Out #1* = true.

Else delay  $\neq 0$ , decrement delay.

If delay = 0, *Out #1* = 0, and *Out 2* is the compliment of *Out #1*.

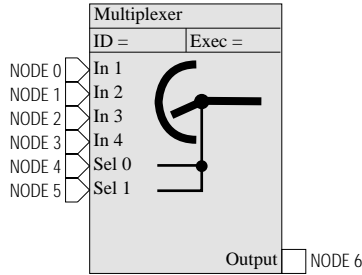
**Example** ENABLE = TRUE

TIME = 100



**MULTIPLEXER**

BLOCK TYPE 21 decimal 15 hexadecimal



**DEFINITION**

Selects one of (4) input values *In 1 – In 4* based on selector *Sel 0* and *Sel 1*.

**INPUTS**

*In 1 – In 4* — A signed input integer.

*Sel 0* and *Sel 1* — Selector inputs that form a two-bit binary value used to select one of (4) inputs *In 1 – In 4*.

**OUTPUT**

*Output* — A signed integer.

**FUNCTION**

When <i>Sel 1</i> is	and <i>Sel 0</i> is	<i>Output</i> =
0	0	<i>In 1</i>
0	≠ 0	<i>In 2</i>
≠ 0	0	<i>In 3</i>
≠ 0	≠ 0	<i>In 4</i>

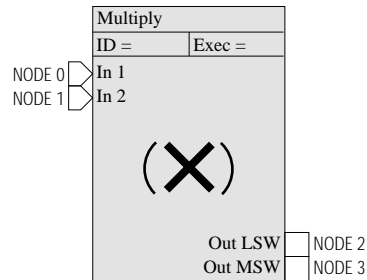
PARAMETERS	DATA TYPE	LINKABLE	DEFAULT VALUE	RANGE
<i>In 1</i>	Signed Integer	Yes	0	±32767
<i>In 2</i>	Signed Integer	Yes	0	±32767
<i>In 3</i>	Signed Integer	Yes	0	±32767
<i>In 4</i>	Signed Integer	Yes	0	±32767
<i>Sel 0</i>	Logic Input	Yes	0	True/False
<i>Sel 1</i>	Logic Input	Yes	0	True/False
<i>Output</i>	Signed Integer	No	—	±32767

EXAMPLES	EXAMPLE 1	EXAMPLE 2	EXAMPLE 3	EXAMPLE 4
<i>In 1</i>	10	10	10	10
<i>In 2</i>	25	25	25	25
<i>In 3</i>	42	42	42	42
<i>In 4</i>	-60	-60	-60	-60
<i>Sel 0</i>	False	True	False	True
<i>Sel 1</i>	False	False	True	True
<i>Output</i>	10	25	42	-60



**MULTIPLY**

BLOCK TYPE 28 decimal 1C hexadecimal

**DEFINITION**

Multiplies two 16 bit inputs. This block calculates a 32 bit result that is stored in two words *Out LSW* and *Out MSW*.

**INPUTS**

*In 1* — A signed input integer.

*In 2* — A signed input integer.

**OUTPUTS**

*Out LSW* — A least significant result word value representing bits 0–15 of a 32 bit output value.

*Out MSW* — A most significant result word value representing bits 16–31 of a 32 bit output value.

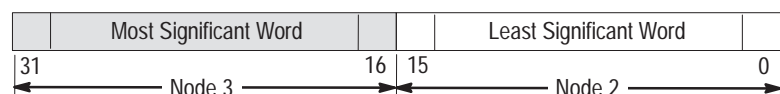
**FUNCTION**

$Out\ LSW, Out\ MSW = In\ 1 \times In\ 2.$

PARAMETERS	DATA TYPE	LINKABLE	DEFAULT VALUE	RANGE
<i>In 1</i>	Signed Integer	Yes	0	±32767
<i>In 2</i>	Signed Integer	Yes	0	±32767
<i>Out LSW</i>	Unsigned Integer	No	—	0 to 65535
<i>Out MSW</i>	Signed Integer	No	—	±32767

**DOUBLE WORD VALUES**

Both the Most Significant Word and the Least Significant Word are interpreted together by the function algorithm as one word when using 32 bit values. The range of a double word value is ±2,147,483,647. The range of the more common, signed single-word 16 bit node is ±32767.

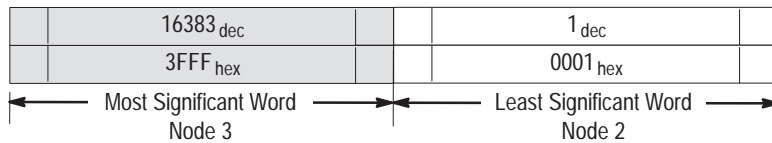


**MULTIPLY**  
(continued)

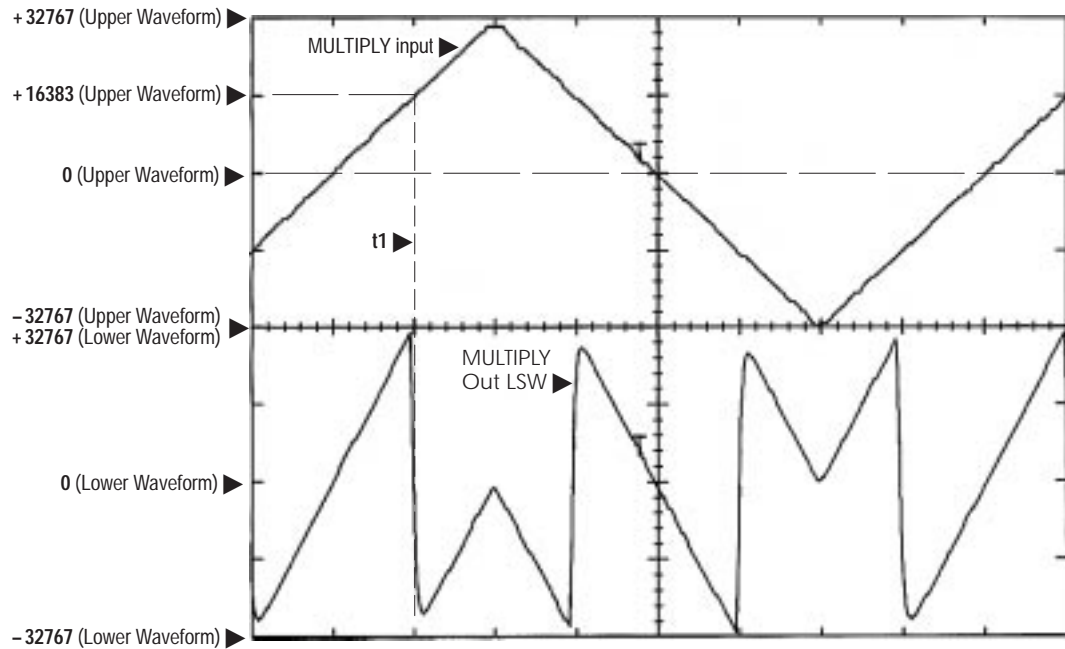
Bit 31 is the sign bit, the most significant bit for 32 bit values. For 16 bit values, bit 15 (the sign bit) is the most significant bit.

EXAMPLES	Example 1	EXAMPLE 2
<i>In 1</i>	32767	3
<i>In 2</i>	32767	32767
<i>Out LSW</i>	1	32765
<i>Out MSW</i>	16383	1
OUTPUT VALUE	<b>1073676289</b>	<b>98301</b>

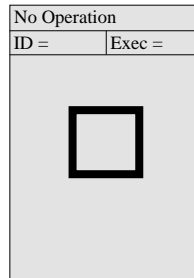
**Example 1 — Double Word Output Value = 1073676289 = 32767 × 32767**



**Important:** The output range of the multiplied values can be critical. Should only one output word (LSW node) of the MULTIPLY block be used, it will appear to roll over or under if the product exceeds ±32767.



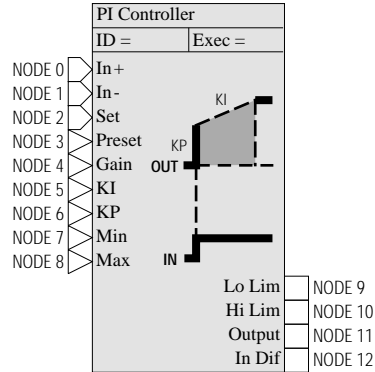
In the Upper Waveform above, the MULTIPLY In1 input (node 0), is multiplied by an In 2 (node 1) value of 2. As the In1 value increases beyond +32767 (time t1), the product increases beyond +32767. Although the Out LSW output (node 2) has a range of 0 to 65535 (0 to FFFF<sub>hex</sub>), the analog input has a single word signed range of ±32767. When the input reaches 16385, the multiplied output of 32770 (8002<sub>hex</sub>) is interpreted by the signed analog output parameter as having a value of -32766 which makes it appear to roll over.

**NO-OP****BLOCK TYPE** 0 decimal 0 hexadecimal**DEFINITION**

A PLC place holder.

## PI CTRL

BLOCK TYPE 17 decimal 11 hexadecimal



### DEFINITION

A proportional/integral control function block that uses trapezoidal integration.

### INPUTS

**INI+** — A signed input integer to the PI block.

**INI-** — A signed input integer to the PI block.

**Set** — A logic signal that when  $\neq 0$  sets the integral term's accumulator to the **Preset** value.

**Preset** — A signed integer that is preloaded into the integrator's accumulator when **Set** = true.

**Gain** — A scaled, signed integer that adjusts the **In+** & **In-** summing node. Scaling: 2048 = effective gain of 1.

**KI** — A word value that represents integral gain.  
Scaling: 4096 = effective gain of 1, max effective gain of 8.

**KP** — A scaled word value that represents proportional gain.  
Scaling: 4096 = effective gain of 1, max effective gain of 8.

**Min** — A signed integer that limits the lower value of the output and the integral accumulator.

**Max** — A signed integer that limits the upper value of the output and the integral accumulator.

### OUTPUTS

**Lo Lim** — Low limit flag that is true when the calculated **Output** < **Min**.

**Hi Lim** — Hi limit flag that is true when the calculated **Output** > **Max**.

**Output** — A signed integer representing the output of the PI controller.

**In Dif** — A signed integer that represents the difference between **In+** & **In-**. This value is limited to  $\pm 32767$  and will not represent the internal difference if it is not within these limits.

## PI CTRL

(continued)

## FUNCTION

1. If set = true

$$KI_{out_{i-1}} = \mathit{Preset}$$

$$KI_{val} = KI_{val_{i-1}} = 0.$$

2. If set = false

$$\text{sum (limited to } \pm 32767) = \mathit{Gain} \times (\mathit{In+} - \mathit{In-})/2048$$

$$KP_{out} = \text{sum} \times KP/4096$$

$$KI_{val} = \text{sum} \times KI/4096.$$

$$\mathit{In\ Difout} = \mathit{In+} - \mathit{In-}$$

$\mathit{In\ Difout}$  is clamped to  $\pm 32767$ .

$$KI_{out} = KI_{out_{i-1}} + [(KI_{val} + KI_{val_{i-1}}) \times (\Delta t/2 = 1/100)]$$

1. If  $KI_{out} + KP_{out} > \mathit{Max}$  then,

$$\mathit{Output} = \mathit{Max}, \mathit{Hi\ Lim} = \text{true}, \mathit{Lo\ Lim} = \text{false}.$$

2. If  $KI_{out} + KP_{out} < \mathit{Min}$  then,

$$\mathit{Output} = \mathit{Min}, \mathit{Hi\ Lim} = \text{false}, \mathit{Lo\ Lim} = \text{true}.$$

3. If  $KI_{out} + KP_{out}$  is neither **1.** nor **2.** then,

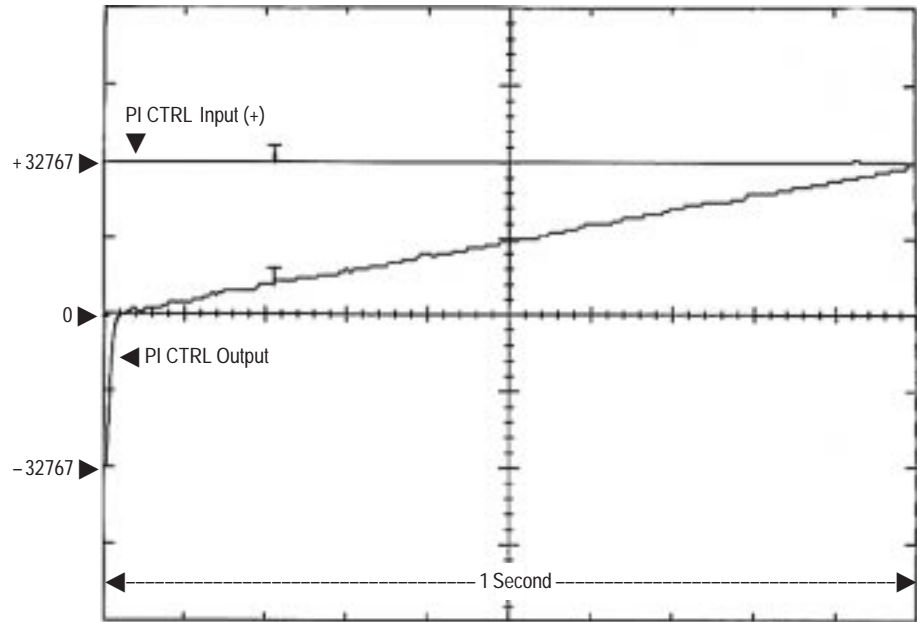
$$\mathit{Output} = KI_{out} + KP_{out}.$$

PARAMETERS	DATA TYPE	LINKABLE	DEFAULT VALUE	RANGE
<i>In+</i>	Signed Integer	Yes	0	$\pm 32767$
<i>In-</i>	Signed Integer	Yes	0	$\pm 32767$
<i>Set</i>	Logic Input	Yes	False	True/False
<i>Preset</i>	Signed Integer	No	0	$\pm 32767$
<i>Gain</i>	Signed Integer	No	2048	$\pm 32767$
<i>KI</i>	Unsigned Integer	No	4096	0 to + 32767
<i>KP</i>	Unsigned Integer	No	4096	0 to + 32767
<i>Min</i>	Signed Integer	No	0	0 to - 32767
<i>Max</i>	Signed Integer	No	0	0 to + 32767
<i>Lo Lim</i>	Logic Output	No	—	True/False
<i>Hi Lim</i>	Logic Output	No	—	True/False
<i>Output</i>	Signed Integer	No	—	$\pm 32767$
<i>In Dif</i>	Signed Integer	No	—	$\pm 32767$

**PI CTRL**  
(continued)

**Example 1 — KI & KP = 1 — Horizontal Scale = 100mS/Division**

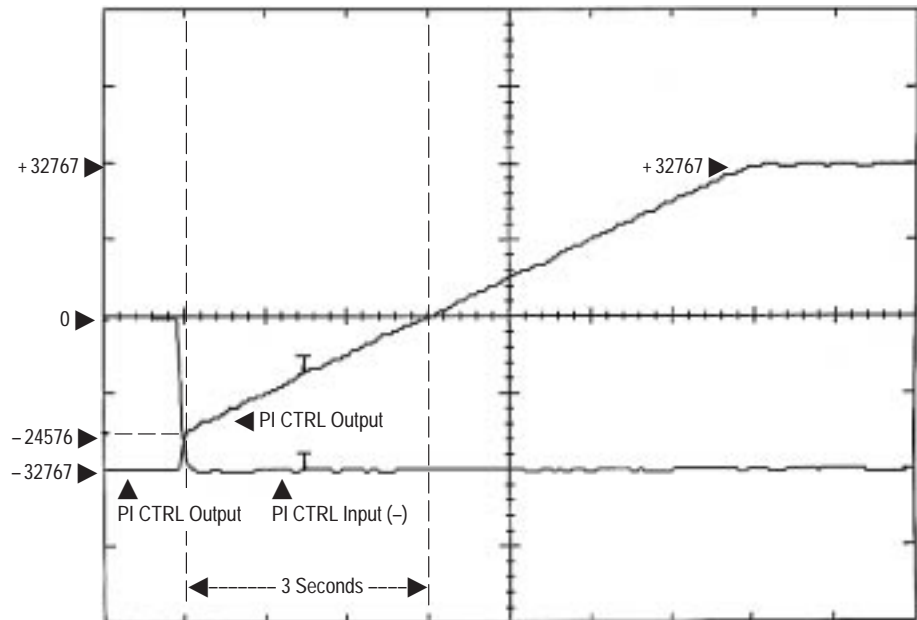
In Example 1, all gains are at unity — KI and KP are set to 4096, Gain is set to 2048. Preset is set to -32767.



When the input difference steps from 0 to +32767, the proportional term responds by going from -32767 to zero within one task interval. The Output then takes approximately one second to integrate up to the Max setting of +32767.

**Example 2 — KI & KP = 1/4 — Horizontal Scale = 1 S/Division**

In Example 2, the KI and KP gains are at 1/4 (1024), while Gain is set to unity (2048). Preset has been set to -32767.

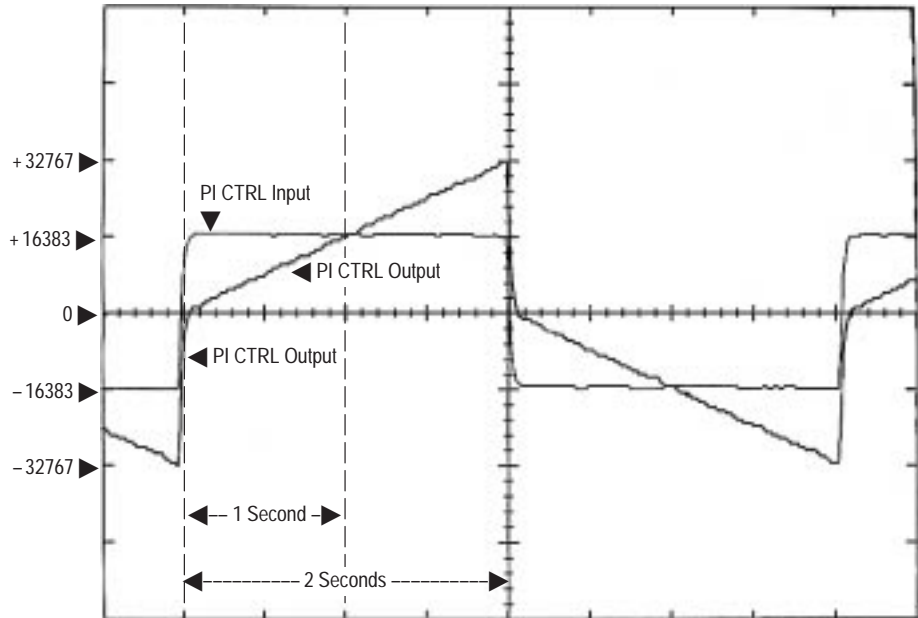


When the input difference steps from 0 to -32767, the proportional term responds by immediately going to a value of -24576. After 3 seconds, the Output integrates to zero. With KI at 1/4 gain, it will accumulate at a rate of approximately 8191 units-per-second with an input difference of +32767. Note that it takes (4) seconds to integrate from 0 to +32767.

## PI CTRL (continued)

### Example 3 — KI & KP = 1 — Horizontal Scale = 500mS/Division

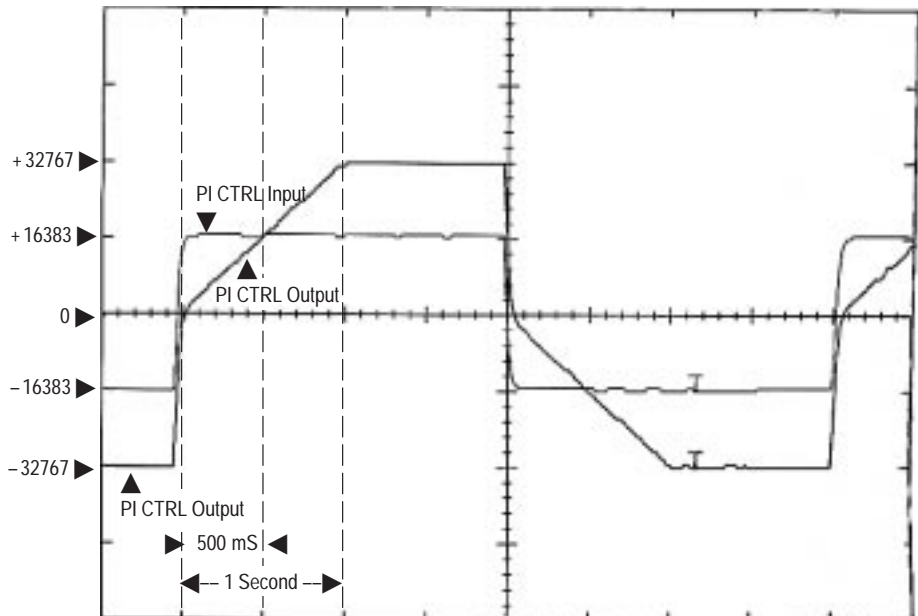
In Example 3, all gains are at unity — KI and KP are set to 4096, Gain is set to 2048.



The input difference in this example alternates between  $\pm 16383$ . The proportional Output response immediately jumps to 0. The Output takes one second to accumulate to +16383 and two seconds to move from -32767 to +32767.

### Example 4 — KP = 1, KI = 2 — Horizontal Scale = 500mS/Division

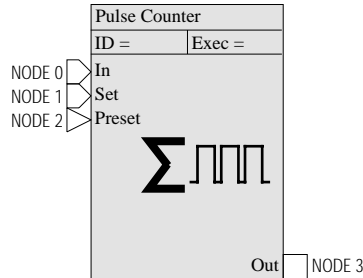
Example 4 is similar to Example 3, except that the KI gain is now at 8192, or  $2\times$  the value in Example 3. KP gain is set to unity (4096), and Gain is set to unity (2048).



The Output reaches 16383 within 500mS and +32767 within one second. In this Example, the Output is clamped at the positive and negative limits. The internal accumulator is also clamped at these limits, and cannot accumulate beyond these limits. This allows the PI CTRL function to immediately get out of saturation when the function's input difference changes polarity.

**PULSE CNTR**

BLOCK TYPE 18 decimal 12 hexadecimal

**DEFINITION**

Counts the rising edges of input signal *In*.

**INPUTS**

*In* — A logic input signal.

*Set* — A logic level value that preloads the pulse counter accumulator with the *Preset* value when  $\neq 0$ .

*Preset* — A signed integer that is preloaded into the pulse counter accumulator when *Set*  $\neq 0$ .

**OUTPUT**

*Out* — A signed integer that is incremented on every rising edge of input signal *In*. *Out* increments to the positive limit but is not allowed to roll over.

**FUNCTION**

If *Set* is true, then *Out* = *Preset* value.

If *Set* is false, then *Out* = *Out* + 1 on rising edge of *In*.

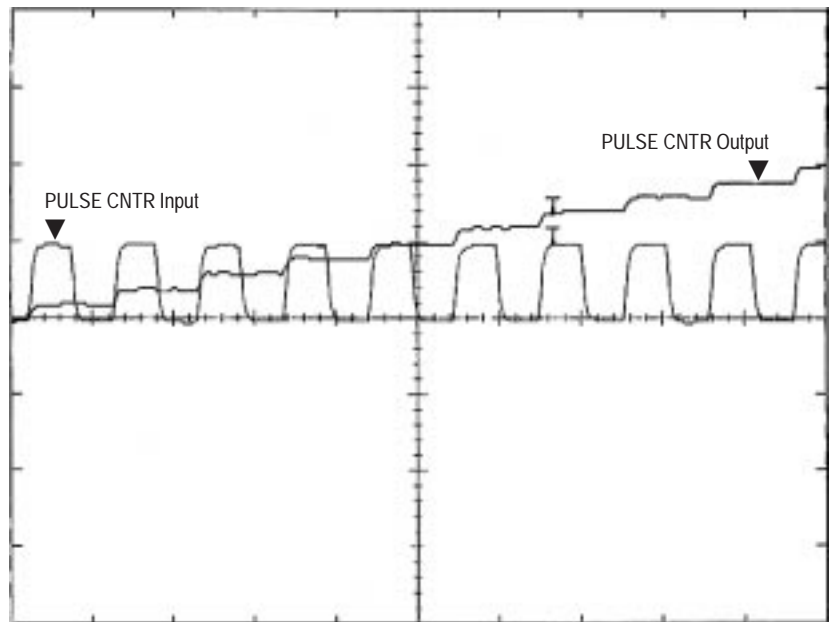
PARAMETERS	DATA TYPE	LINKABLE	DEFAULT VALUE	RANGE
<i>In</i>	Logic Input	Yes	0	True/False
<i>Set</i>	Logic Input	Yes	0	True/False
<i>Preset</i>	Signed Integer	No	0	$\pm 32767$
<i>Out</i>	Signed Integer	No	0	$\pm 32767$



## PULSE CNTR

(continued)

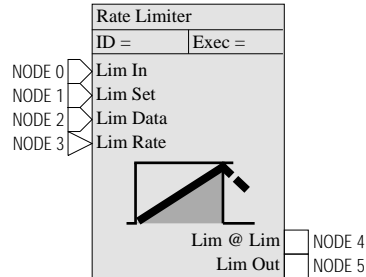
Example 1 — Preset = 0 — Horizontal Scale = 500 mS/Division



The plot shown in Example 1 shows the pulse counter Output incrementing with every rising edge of the input signal.

**RATE LIMITER**

BLOCK TYPE 19 decimal 13 hexadecimal

**DEFINITION**

Limits the rate of change of the input value *Lim In* by the value of rate *Lim Rate*.

**INPUTS**

*Lim In* — A signed integer that is rate limited in the positive or negative direction.

*Lim Set* — A logic value that preloads the output with the *Lim Data* value when *Lim Set*  $\neq$  0.

*Lim Data* — A signed integer that the output will be set to when *Lim Set*  $\neq$  0.

*Lim Rate* — A word that specifies the *Lim Out* maximum rate of change in units/second.

**OUTPUTS**

*Lim @ Lim* — A logic flag that indicates *Lim Out* is limited by *Lim Rate*.

*Lim Out* — A signed integer, rate limited to the *Lim Rate* value.

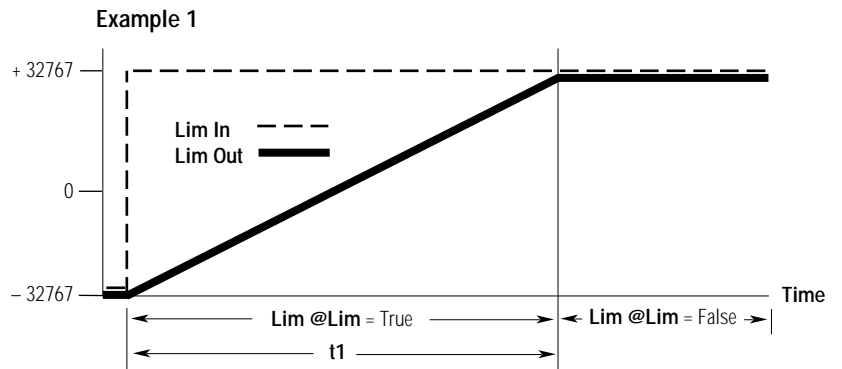
**FUNCTION**

1. If *Lim Set* is true, *Lim Out* = *Lim Data*.
2. If *Lim Set* is false,  $\Delta = \text{Lim In} - \text{Lim Out}$ .  
 If  $\Delta$  is + and  $> \text{Lim Rate}$ , *Lim @ Lim* is true and  $\Delta = \text{Lim Rate}$ .  
 If  $\Delta$  is + and  $< \text{Lim Rate}$ , *Lim @ Lim* is false.  
  
 If  $\Delta$  is - and  $< -\text{Lim Rate}$ , *Lim @ Lim* is true.  
 If  $\Delta$  is - and  $> -\text{Lim Rate}$ , *Lim @ Lim* is false.  
*Lim Out* = *Lim Out* +  $\Delta$ .

## RATE LIMITER

(continued)

PARAMETERS	DATA TYPE	LINKABLE	DEFAULT VALUE	RANGE
<i>Lim In</i>	Signed Integer	Yes	0	$\pm 32767$
<i>Lim Set</i>	Logic Input	Yes	0	True/False
<i>Lim Data</i>	Signed Integer	Yes	0	$\pm 32767$
<i>Lim Rate</i>	Unsigned Integer	No	0	0 to 65535
<i>Lim @ Lim</i>	Logic Output	No	—	True/False
<i>Lim Out</i>	Signed Integer	No	—	$\pm 32767$

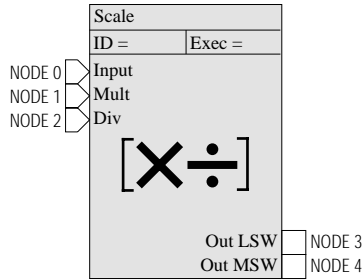


When **Lim Rate** = 65535,  $t_1$  = 1 sec

When **Lim Rate** = 32767,  $t_1$  = 2 sec

**SCALE**

BLOCK TYPE 20 decimal 14 hexadecimal



**DEFINITION**

Multiplies the input *Input* by *Mult* and divides by *Div*. The result is a two word output consisting of a least significant word *Out LSW*, and a most significant signed integer *Out MSW*. If the result is a value between 0 and 65535, *Out LSW* contains the value and *Out MSW* is 0. For values less than 0 the sign bit that indicates the negative value is bit 15 of *Out MSW*. When *Div* is equal to 0, the result of the scale block is 0.

**INPUTS**

- Input* — A signed integer between ±32767.
- Mult* — A signed integer between ±32767.
- Div* — A signed integer between ±32767.

**OUTPUTS**

- Out LSW* — A least significant result word value representing bits 0–15 of a 32 bit output value.
- Out MSW* — A most significant result word value representing bits 16–31 of a 32 bit output value.

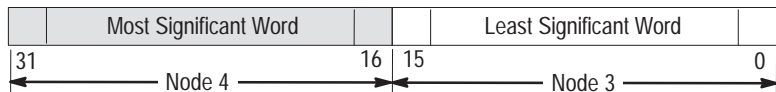
**FUNCTION**

$Out\ LSW, Out\ MSW = (Input \times Mult) \div Div.$

PARAMETERS	DATA TYPE	LINKABLE	DEFAULT VALUE	RANGE
<i>Input</i>	Signed Integer	Yes	0	±32767
<i>Mult</i>	Signed Integer	Yes	0	±32767
<i>Div</i>	Signed Integer	Yes	1	±32767
<i>Out LSW</i>	Unsigned Integer	No	—	0 to 65535
<i>Out MSW</i>	Signed Integer	No	—	±32767

**DOUBLE WORD VALUES**

Both the Most Significant Word and the Least Significant Word are interpreted together by the function algorithm as one word when using 32 bit values. The range of a double word value is ±2,147,483,647. The range of the more common, signed single-word 16 bit node is ±32767.

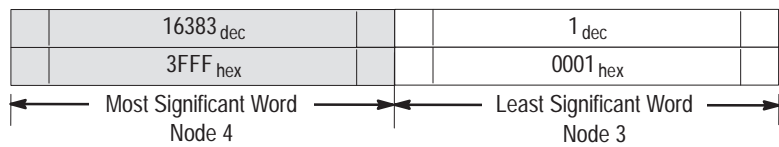


**SCALE**  
(continued)

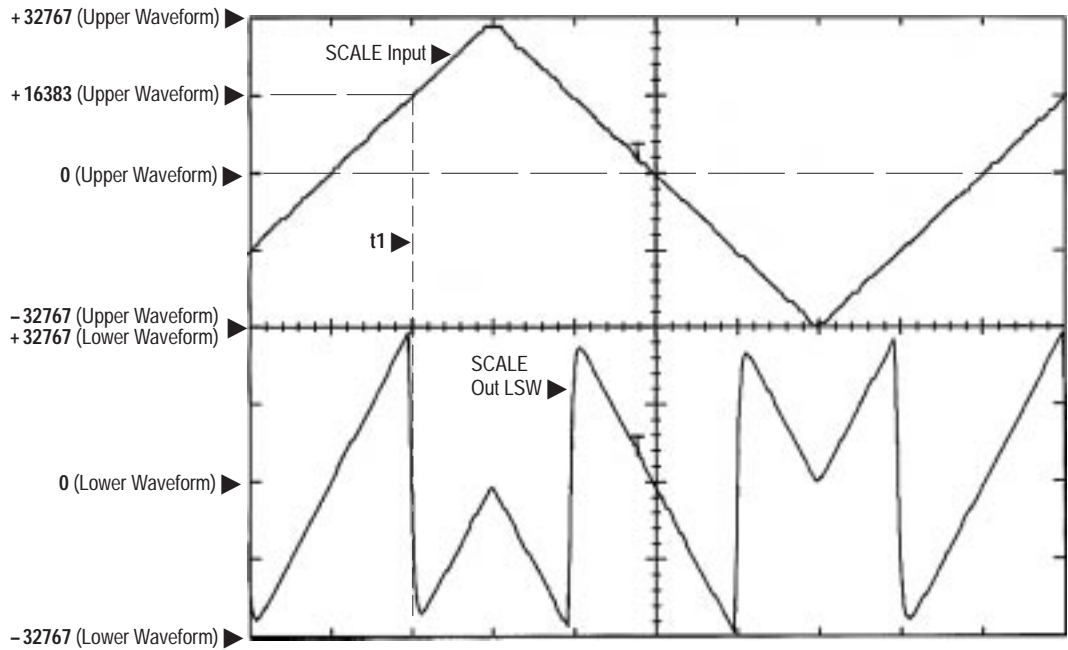
Bit 31 is the sign bit, the most significant bit for 32 bit values. For 16 bit values, bit 15 (the sign bit) is the most significant bit.

EXAMPLES	Example 1	EXAMPLE 2	EXAMPLE 2
<i>Input</i>	32767	56	-2
<i>Mult</i>	32767	128	16383
<i>Div</i>	1	10	1
<i>Out LSW</i>	1	716	32770
<i>Out MSW</i>	16383	0	-1
OUTPUT VALUE	<b>1073676289</b>	<b>716</b>	<b>-32766</b>

**Example 1 — Double Word Output Value = 1073676289 = 32767 × 32767 ÷ 1**



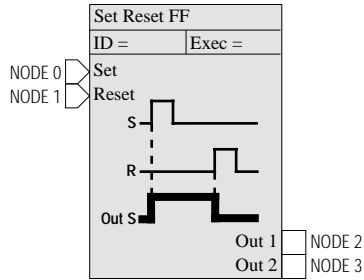
**Important:** The output range of the multiplied values can be critical. Should only one output word (LSW node) of the SCALE block be used, it will appear to roll over or under if the product exceeds ±32767.



In the Upper Waveform above, the SCALE Input (node 0), is multiplied by a Mult (node 1) value of 2 and divided by a Div (node 2) value of 1. As the input value increases beyond 16383 (time t1), the product increases beyond +32767. Although the Out LSW output (node 3) has a range of 0 to 65535 (0 to FFFF<sub>hex</sub>), the analog input has a single word signed range of ±32767. When the input reaches 16385, the multiplied output of 32770 (8002<sub>hex</sub>) is interpreted by the signed analog output parameter as having a value of -32766 which makes it appear to roll over.

**SR FF**

BLOCK TYPE 22 decimal 16 hexadecimal



**DEFINITION**

A set/reset function block where:

*Out 1* is false if *Reset* is true.

*Out 1* is true if *Reset* is false and *Set* is true.

*Out 2* is the inverse of *Out 1*.

**INPUTS**

*Set* — A logic input value

*Reset* — A logic input value.

**OUTPUTS**

*Out 1* — A logic output value

*Out 2* — A logic output value.

**FUNCTION**

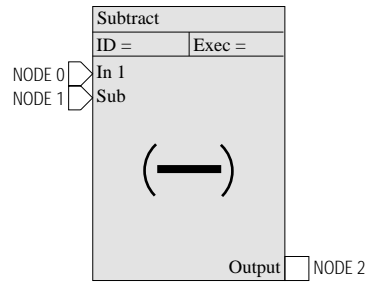
*Out 1* is *Set* or *Reset*.

PARAMETERS	DATA TYPE	LINKABLE	DEFAULT VALUE	RANGE
<i>Set</i>	Logic Input	Yes	False	True/False
<i>Reset</i>	Logic Input	Yes	False	True/False
<i>Out 1</i>	Logic Output	No	—	True/False
<i>Out 2</i>	Logic Output	No	—	True/False

EXAMPLES	EXAMPLE 1	EXAMPLE 2	EXAMPLE 3	EXAMPLE 4
<i>Set</i>	False	True	False	True
<i>Reset</i>	True	False	False	True
<i>Out 1</i>	False	True	(Last State)	False
<i>Out 2</i>	True	False	(Last State)	True

**SUB**

BLOCK TYPE 27 decimal 1B hexadecimal

**DEFINITION**

Subtracts two signed integers. *Output* will be clamped to  $\pm 32767$  and not allowed to over or under flow.

**INPUTS**

*In 1* — A signed integer between  $\pm 32767$ .

*Sub* — A signed integer between  $\pm 32767$ .

**OUTPUT**

*Output* — A signed integer between  $\pm 32767$ .

**FUNCTION**

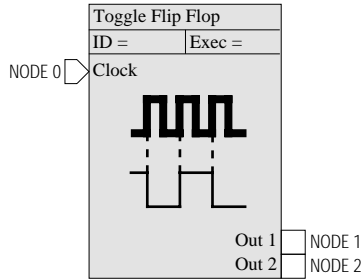
$Output = In\ 1 - Sub.$

PARAMETERS	DATA TYPE	LINKABLE	DEFAULT VALUE	RANGE
<i>In 1</i>	Signed Integer	Yes	0	$\pm 32767$
<i>Sub</i>	Signed Integer	Yes	0	$\pm 32767$
<i>Output</i>	Signed Integer	No	—	$\pm 32767$

EXAMPLES	EXAMPLE 1	EXAMPLE 2	EXAMPLE 3
<i>In 1</i>	+10	-32765	-5
<i>Sub</i>	+15	+2	-23
<i>Output</i>	-5	-32767	+18

**T-FF**

BLOCK TYPE 11 decimal 0B hexadecimal



**DEFINITION**

A flip-flop function block that toggles the rising edge of the *Clock* input.

**INPUT**

*Clock* — A logic input.

**OUTPUTS**

*Out 1* — A logic output.

*Out 2* — A logic output.

**FUNCTION**

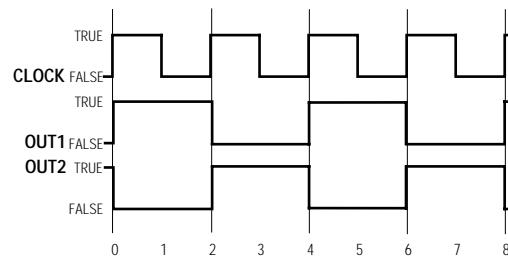
When *Clock* signal changes from false to true:

*Out 1* changes to the opposite state.

*Out 2* is the inverse of *Out 1*.

PARAMETERS	DATA TYPE	LINKABLE	DEFAULT VALUE	RANGE
<i>Clock</i>	Logic Input	Yes	False	True/False
<i>Out 1</i>	Logic Output	No	False	True/False
<i>Out 2</i>	Logic Output	No	—	True/False

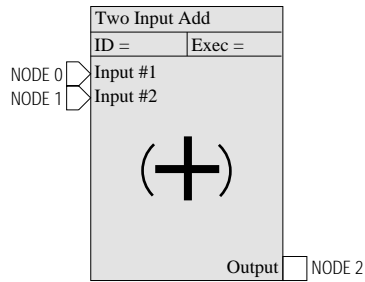
**Example**





**2ADD**

BLOCK TYPE 26 decimal 1A hexadecimal

**DEFINITION**

Sums two signed numbers *Input #1* and *Input #2*. The output *Output* is clamped to  $\pm 32767$  should the sum exceed these limits.

**INPUT**

*Input #1* — A signed integer between  $\pm 32767$ .

*Input #2* — A signed integer between  $\pm 32767$ .

**OUTPUT**

*Output* — A signed integer between  $\pm 32767$ .

**FUNCTION**

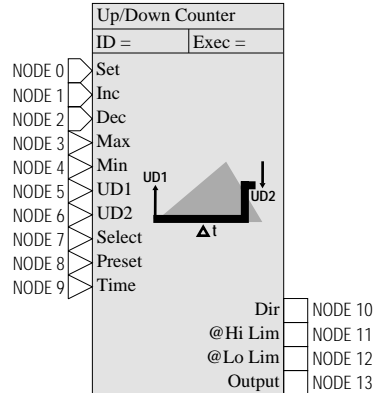
$Output = Input\ #1 + Input\ #2.$

PARAMETERS	DATA TYPE	LINKABLE	DEFAULT VALUE	RANGE
<i>Input #1</i>	Signed Integer	Yes	0	$\pm 32767$
<i>Input #2</i>	Signed Integer	Yes	0	$\pm 32767$
<i>Output</i>	Signed Integer	No	—	$\pm 32767$

EXAMPLES	EXAMPLE 1	EXAMPLE 2	EXAMPLE 3
<i>Input #1</i>	1	32767	-32767
<i>Input #2</i>	2	4	-10
<i>Output</i>	3	32767	-32767

**UP/DWN CNTR**

BLOCK TYPE 24 decimal 18 hexadecimal

**DEFINITION**

An up/down counter function block that can be programmed to count up or down in a specified, programmable time period.

**INPUTS**

**Set** — A logic signal that when  $\neq 0$  sets the counter output to the value of **Preset**.

**Inc** — A logical word that specifies the counter is to increment the accumulator.

**Dec** — A logical word that specifies the counter is to decrement the accumulator.

**Max** — A signed integer that limits the upper value of the output.

**Min** — A signed integer that limits the lower value of the output.

**UD1** — A signed integer that together with **Time** determines the **Inc/Dec** rate.

**UD2** — A signed integer that together with **Time** determines the **Inc/Dec** rate.

**Select** — An input word value if  $= 0$  selects **UD1**, if  $\neq 0$  selects **UD2**.

**Preset** — A signed integer that can be preloaded into the output if **Set**  $\neq 0$ .

**Time** — An unsigned word that determines the period where the **UD1/UD2** increment is to be added or subtracted from the accumulator counter value. The **Time** value is entered in mS, with a resolution limited by the task interval —  $(\text{Time in}) / 20\text{mS}$ .

## UP/DWN CNTR

(continued)

## OUTPUTS

**Dir** — A signed integer that indicates counter direction —  
-1 if decrementing, 0 if no change, 1 if incrementing.

**@Hi Lim** — A logic value = true only when the accumulator >  
**Max**.

**@Lo Lim** — A logic value = true only when the accumulator <  
**Min**.

**Output** — A signed integer that is the counter's output.

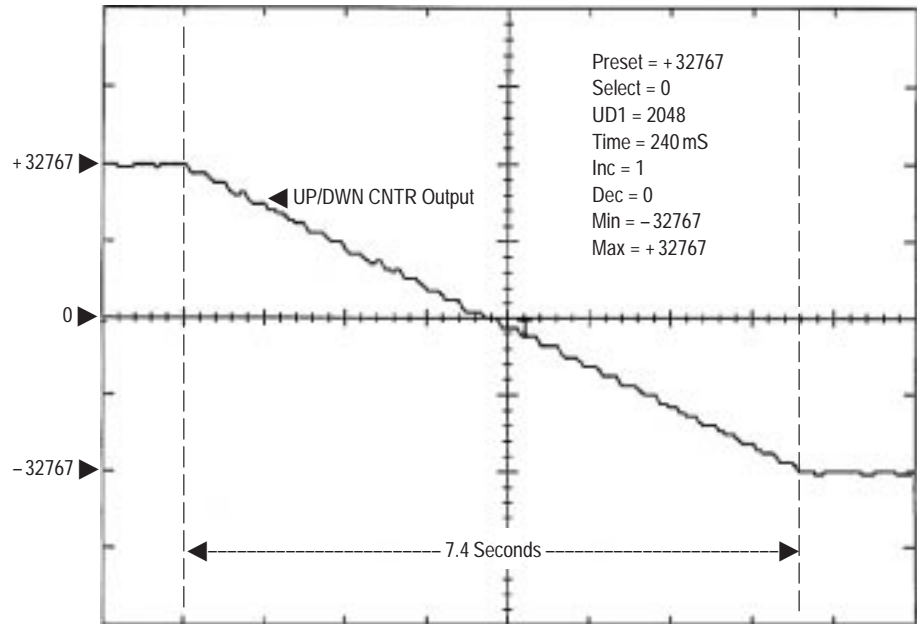
## FUNCTION

1. If **Set** ≠ 0, the accumulator = **Preset**.
2. When **Select** = 0 and **Time** = time since last adjustment of accumulator:  
If **Inc** ≠ 0, the accumulator = accumulator + **UD1**, and **Dir** = 1.  
If **Dec** ≠ 0, The accumulator = accumulator - **UD1**, and **Dir** = -1.
3. When **Select** ≠ 0 and **Time** = time since last adjustment of accumulator:  
If **Inc** ≠ 0, the accumulator = accumulator + **UD2**, and **Dir** = 1  
If **Dec** ≠ 0, The accumulator = accumulator - **UD2**, and **Dir** = -1.
4. When the accumulator > **Max**:  
The accumulator = **Max**, **@Hi Lim** = true, and **@Lo Lim** = false.
5. When the accumulator < **Min**:  
The accumulator = **Min**, **@Hi Lim** = false, and **@Lo Lim** = true.
6. When the accumulator is < **Max** and > **Min**:  
**@Lo Lim** = **@Hi Lim** = false.
7. When **Output** = accumulator, **Dir** = 0.

PARAMETERS	DATA TYPE	LINKABLE	DEFAULT VALUE	RANGE
<b>Set</b>	Logic Input	Yes	False	True/False
<b>Inc</b>	Logic Input	Yes	False	True/False
<b>Dec</b>	Logic Input	Yes	False	True/False
<b>Max</b>	Signed Integer	No	0	±32767
<b>Min</b>	Signed Integer	No	0	±32767
<b>UD1</b>	Signed Integer	No	0	0 to +32767
<b>UD2</b>	Signed Integer	No	0	0 to +32767
<b>Select</b>	Logic Input	No	False	True/False
<b>Preset</b>	Signed Integer	No	0	±32767
<b>Time</b>	Unsigned Integer	No	0	0 to 65535
<b>Dir</b>	Signed Integer	No	—	±32767
<b>@Hi Lim</b>	Logic Input	No	—	True/False
<b>@Lo Lim</b>	Logic Input	No	—	True/False
<b>Output</b>	Signed Integer	No	—	±32767

**UP/DWN CNTR**  
(continued)

**Example 1 — Horizontal Scale = 1 S/Division**



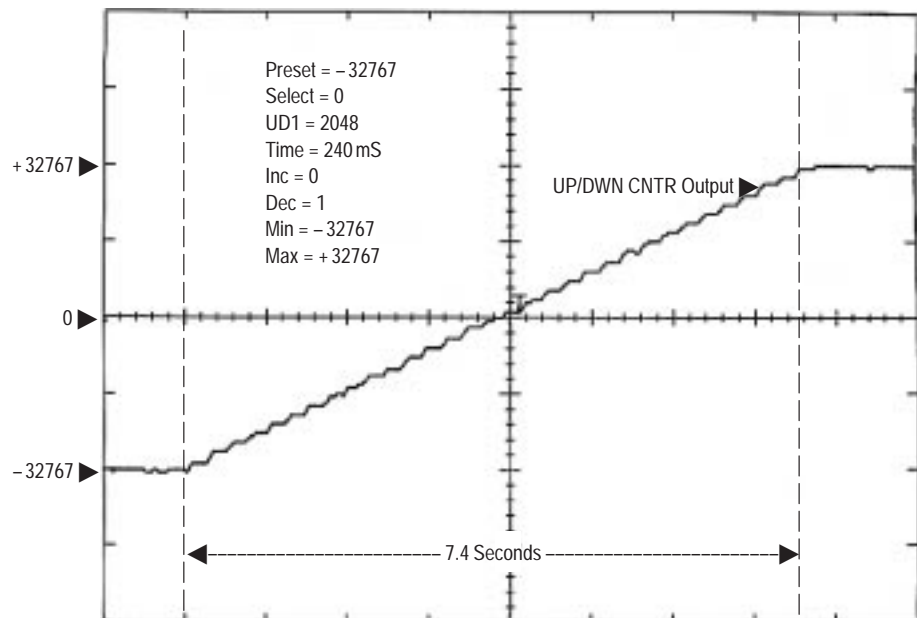
The number of iterations required to decrement the output from it's Preset value of +32767 to the Min value of -32767 would be:

$$Total\ Iterations = Range \div Increment = 65534 \div 2048 = 31.99 \approx 31.$$

The time it takes to traverse from a Max setting of +32767 to -32767 would be:

$$Total\ Time = Total\ Iterations \div Time\ Input = 31 \times 240\ mS = 7.44\ S\ or\ approximately\ 7.4\ S.$$

**Example 2 — Horizontal Scale = 1 S/Division**



Example 2 is similar to Example 1 except the UP/DWN CNTR must count up from a Preset value of -32767.

## Block Transfer Services

### Chapter Objectives

In this chapter, you will read about:

- Block transfer descriptions
- Block transfer status word
- Individual block transfer services descriptions

### Block Transfer Descriptions

This chapter contains the message descriptions that you need to set up data files for the block transfer services using an Allen-Bradley PLC or SLC-500 (using a 1747 SN adapter with an SLC 503 or 504). Each block transfer message contains a three word command header and a data buffer:

Header Word 1
Header Word 2
Header Word 3
Data Word 4

The first three words of a message make up the message header, and are common to all RIO messages with the 1336T PLC Communications Adapter Board.

Header word 1 is zero.

Header words 2 and 3 specify which operation you want to perform.

Header and data values vary depending on the operation you are performing. Also included is a description of the status word that is returned from the drive and appears in the Block Transfer Read Header.



**Note:** The sawtooth generator application presented in Chapter 1 is an example of using the block transfer services.

## Block Transfer Status Word

If a block transfer operation is not successful, header word 2 of the drive response contains a negative value (that is, bit 15 is set to 1 when an operation fails). The drive also usually returns a Status Word to indicate the reason for the block transfer failure. The location of the status word varies depending on the message, but it is typically header word 4 in the drive response.

The status word codes that the drive may return are as follows:

Value	Description
0	No error has occurred.
1	The service failed due to an internal reason, and the drive could not perform the request. This may be returned if you tried to write with a read only request or read with a write only request.
2	The requested service is not supported.
3	An invalid value is in block transfer request header word 2.
4	An invalid value is in block transfer request header word 3.
5	An invalid value is in block transfer request header word 2.
6	The data value is out of range.
7	Drive state conflict. The drive is in an incorrect state to perform the function. The drive cannot be running when you perform certain functions.

## Message Data Representation Base

Most services in this chapter use decimal (Base 10) representation, although certain services are more readily understood by using a hexadecimal (Base 16) representation. When working with execution list events and manipulating nodes, hexadecimal representation is used. The download and compile, read event, node value, and link services are also explained using hexadecimal representation.

The following table summarizes the available block transfer services. A complete description of the block transfer write header message is provided on the specified page.

Group	Service	Type	Page
Application Status Services	Event list check sum	RO	5-4
	Read user text	RO	5-6
	Write user text	RW	5-7
	Total events in application	RO	5-8
	Total nodes in application	RO	5-9
	Task status service	RO	5-10
	Fault status service	RO	5-12
Program Limits Information	Library description service	RO	5-15
	Scheduled task interval	RO	5-16
	Maximum number of events allowed	RO	5-17
	Number of fb files in product	RO	5-18
	Maximum number of nodes allowed	RO	5-19
Application Control Commands	BRAM Init	WO	5-20
	BRAM Store	WO	5-20
	BRAM Recall	WO	5-20
	Download & compile service	WO	5-22
	Read individual event	RO	5-27
	Clear all application links	WO	5-29
	Process all fb links	WO	5-29
	Download service Init	WO	5-31
Node Adjustment	Block value read	RO	5-32
	Block value write	WO	5-34
	Block link read	RO	5-35
	Block link write	WO	5-36
	Full node information read	RO	5-38
	Single node value read	RO	5-40
	Single value write	WO	5-41
	Single link read	RO	5-42
	Single link write	WO	5-44

RO = Read Only; RW = Read Write; WO = Write Only

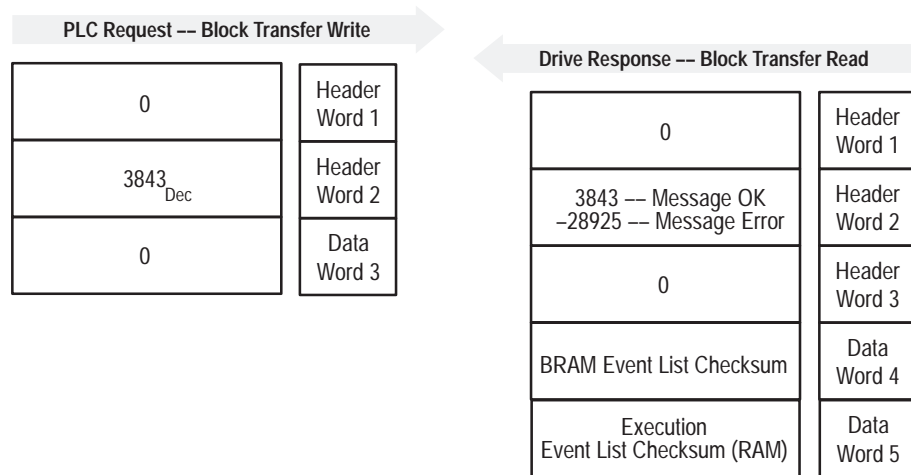
## Application Status Services: Event List Checksum

The **Event List Checksum** message is a simple word addition of the valid events in the current application. This does not include node values or links.

### PLC Block Transfer Instruction Data

BTW Instruction Length: 3 Words  
BTR Instruction Length: 5 Words

#### Message Structure



### Message Operation

The **Event List Checksum** message returns the sum of the valid events in the active application (RAM) and in BRAM. The drive makes a checksum value available so that you can differentiate between the running application and the application stored in BRAM.

When you request an **Event List Checksum**, the drive returns the BRAM event list checksum in word 4 and the execution list checksum in word 5. If the checksum values are the same, then the current application has the same events. However, there may be differences in the links or node values or in the order of events.



## Event List Checksum

(continued)

### Example

In this example, an **Event List Checksum** message has been sent to the drive.

#### Data Format

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N10:10	0	0F03	0							
BTR Data File	N10:90	0	0F03	0	28C3	0A4A					

↑ BRAM Checksum      ↑ RAM Checksum

Notice that the BRAM checksum and the RAM checksum are different. This indicates that the application that is currently executing in the drive is not the same as the application that is stored in BRAM.

If you perform a function block **BRAM Store** operation and then perform another checksum read, the results would look like the following:

#### Data Format

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N10:10	0	0F03	0							
BTR Data File	N10:90	0	0F03	0	0A4A	0A4A					

↑ BRAM Checksum      ↑ RAM Checksum

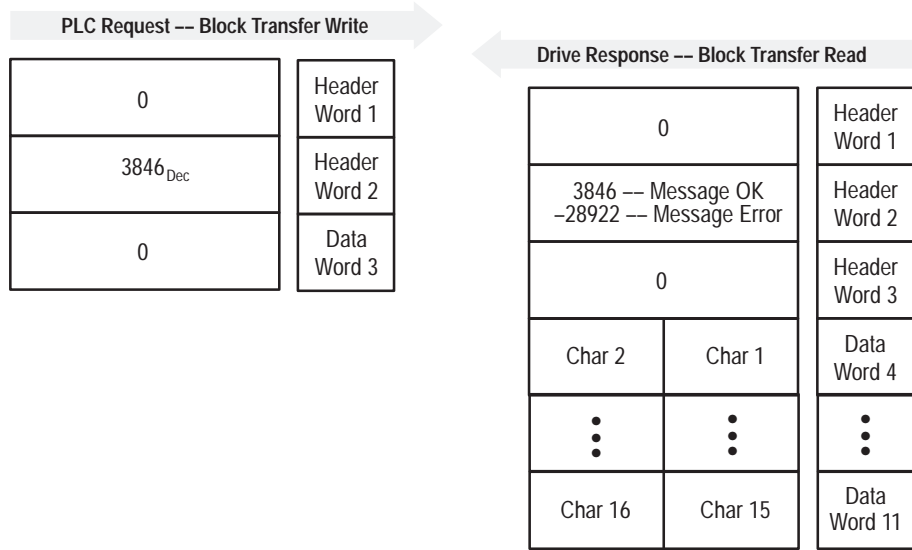
**Application Status Services:**  
**Read Task Name**

The **Read Task Name** message is used to read a text string from a data buffer. The text string is the function block task name, and may contain up to 16 characters.

**PLC Block Transfer Instruction Data**

BTW Instruction Length: 3 Words  
BTR Instruction Length: 11 Words

**Message Structure**



**Message Operation**

The **User ID Text String** allows you to read the task name from a data buffer. The task name can be up to 16 characters.

**Example**

This example shows the **Read Task Name** operation in ASCII representation:

**Data Format**

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N10:10	0	\0F\06	0							
BTR Data File	N10:90	\00\00	\0F\06	\00\00	rd	vi	e	1#			

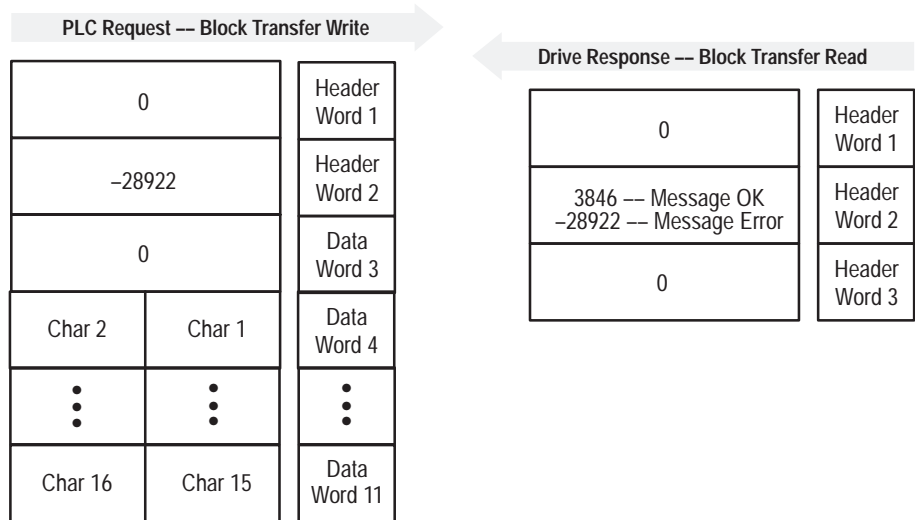
**Application Status Services:**  
**Write Task Name**

The **Write Task Name** message is used to write a task name to a data buffer. The task name may contain up to 16 characters.

**PLC Block Transfer Instruction Data**

BTW Instruction Length: 11 Words  
BTR Instruction Length: 3 Words

**Message Structure**



**Message Operation**

The **User ID Text String** allows you to read or write 16 characters of text from or to a data buffer.

**Example**

This example shows the **Write Task Name** operation in ASCII representation:

**Data Format**

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N10:10	0	\0F\06	\00\00	rd	vi	e	1#			
BTR Data File	N10:90	0	\0F\06	\00\00							

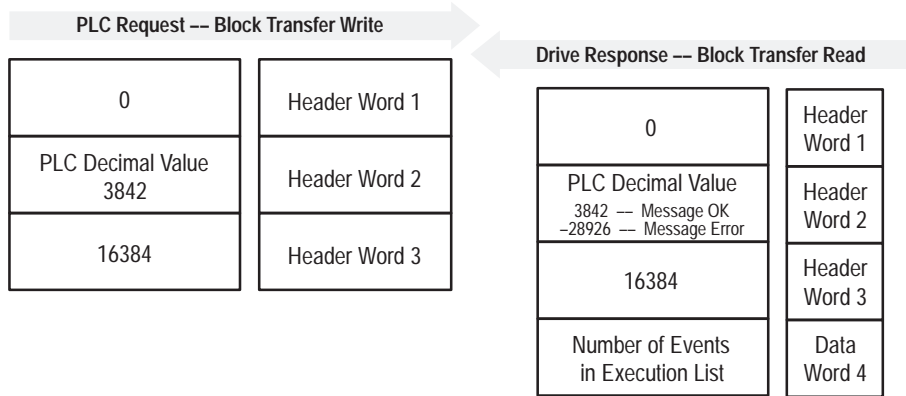
**Application Status Services:  
Total Number of Events in Application**

The **Total Number of Events in Application** message requests the total number of events in the currently active execution list.

**PLC Block Transfer Instruction Data**

BTW Instruction Length: 3 Words  
BTR Instruction Length: 4 Words

**Message Structure**



**Message Operation**

The PLC requests the number of events (function blocks) that are in the current execution list. The drive response indicates the total number of events. An application can have a maximum of 128 events.

► **Note:** Function block types appearing more than once in the execution list may have different ID numbers.

If the drive returns 0 for the number of events in the execution list, no application is currently active in the drive.

► **Note:** The value returned does not reflect what may or may not be stored in BRAM.

**Example**

In this example, the drive's response reports that 58 events are in the drive's execution list. This example uses decimal values.

**Data Format**

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N10:0	0	3842	16384							
BTR Data File	N10:90	0	3842	16384	58						

## Application Status Services:

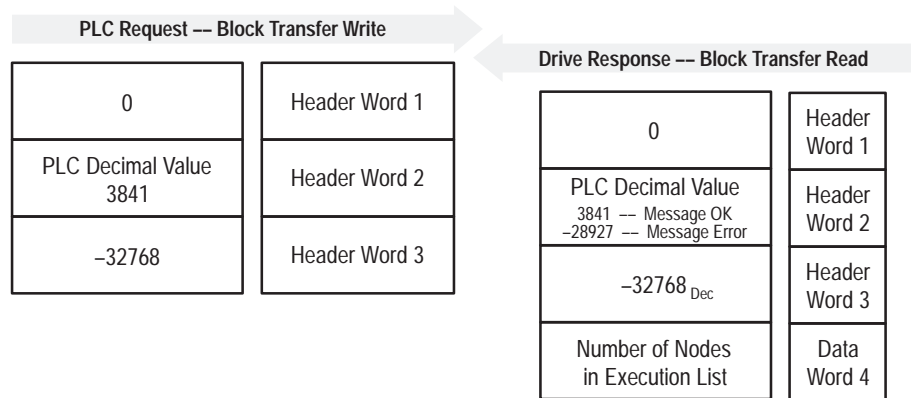
### Total Number of I/O Nodes

The **Total Number of I/O Nodes** message provides the number of nodes used in the function block application currently running in the drive.

### PLC Block Transfer Instruction Data

BTW Instruction Length: 3 Words  
 BTR Instruction Length: 4 Words

#### Message Structure



### Message Operation

The PLC requests the number of nodes that are associated with the active execution list in the drive. The drive response indicates the total number of nodes used by all function blocks in the execution list.

### Example

In this example, the PLC has requested the total number of nodes used. The drive responds that 24 nodes are used in the current application.

#### Data Format

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N10:10	0	3841	-32768							
BTR Data File	N10:90	0	3841	-32768	24						

**Application Status Services:**  
**Read Task Status**

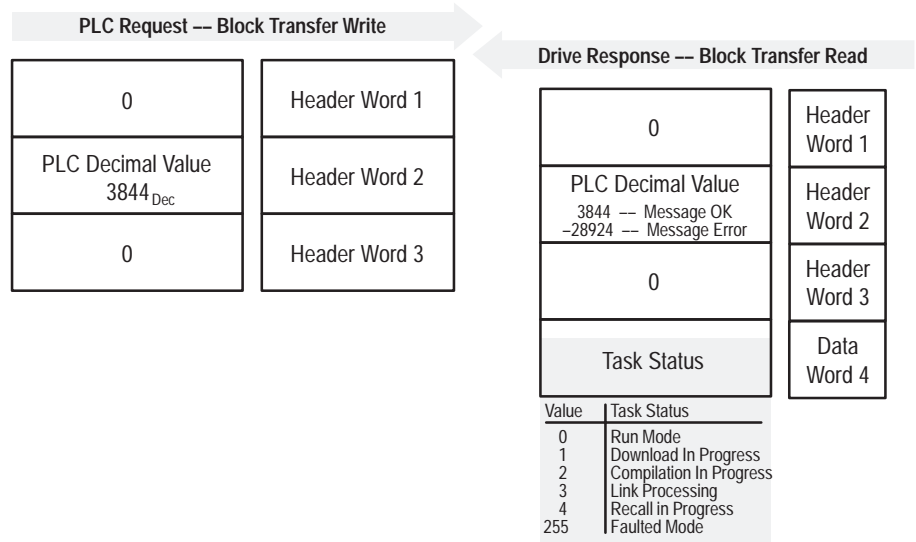
The **Read Task Status** message requests the current status of the function block program in RAM.

**PLC Block Transfer Instruction Data**

BTW Instruction Length: 3 Words

BTR Instruction Length: 4 Words

**Message Structure**



**Message Operation**

The **Read Task Status** message returns a code value that indicates the status of the current function block program.

## Read Task Status

(continued)

Value	Task Status	Description
0	Run Mode	The application is executing within the 20 millisecond task interval. No faults have occurred within the function block portion of functionality.
1	Download in Progress	The previously compiled application is still enabled and executing within the function block task interval. One or more downloaded packets have been received for a new function block program and the function block system is waiting for more data. The currently active application is not interrupted until all packets have been received and the data has been verified for the new function block program before compilation.
2	Compilation in Progress	All packets have been downloaded and the data verified. The service has initiated a compile. Compilation can take seconds when a large application is used.
3	Link Processing	The application is disabled and links between function blocks and drive parameters are being established
4	Recall in Progress	A <b>Recall</b> is in progress.
255	Fault Mode	A function block application has a faulted status. Function block compile time errors create a soft fault condition within the drive. The 1336T system architecture contains a system fault queue that describes the nature of the fault. SCANport provides two fault reporting values should the Task Status word indicate a faulted mode.  The previous application is disabled and will not run until you correct the fault. You cannot clear function block compiler faults with the clear faults command until you correct the function block fault.

### Example

This BTR indicates that the drive is currently in the **Run Mode**.

#### Data Format

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N10:10	0	3844	0							
BTR Data File	N10:90	0	3844	0	0						

## Application Status Services:

### Fault Status Read

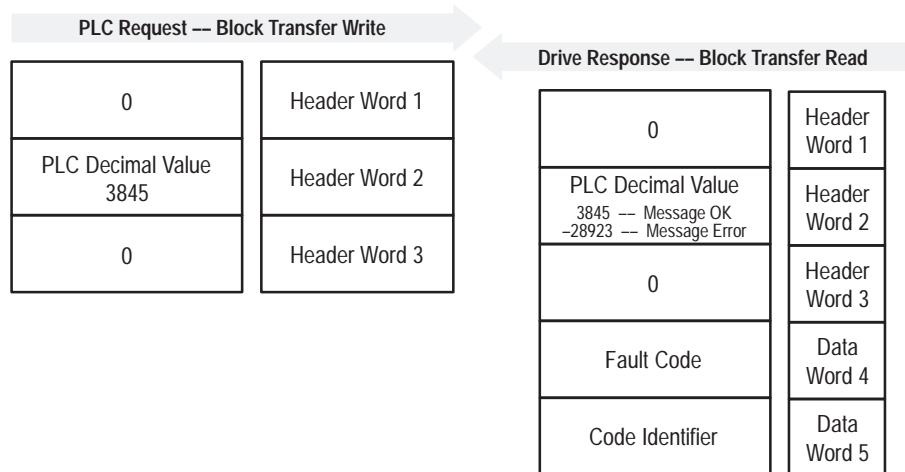
The **Fault Status Read** message reads the fault code information from the drive when a fault is associated with the function block program. A function block fault is indicated by Task Status of FF<sub>Hex</sub>. To get the Task Status, use the **Read Task Status** block transfer routine.

### PLC Block Transfer Instruction Data

BTW Instruction Length: 3 Words

BTR Instruction Length: 5 Words

#### Message Structure



### Message Operation

Word 4 is the actual (bit encoded) value of the function block fault status register. You should use this word mainly as a fault flag during download and compile operations.

The following values are currently valid for Word 4:

Word Value	Bit Number	Definition
2	1	A link processing error occurred.
4	2	An I/O node limit error occurred.
8	3	A memory allocation error occurred.
32	5	A block number limit error occurred.
128	7	A BRAM checksum error occurred.
256	8	A packet number error occurred because a packet number was out of range.
1024	10	An event number error occurred. The same ID number was used with a different block type.



## Fault Status Read (continued)

Word 5 is interpreted differently depending on the type of compiler fault.

- ❑ If Word 4 is 2, Word 5 contains the function block number for the first input parameter or node that has an invalid link. A function block link fault occurs when the compiler is processing links and encounters a function block node with a link to an invalid output node.
- ❑ If Word 4 is clear and a download is incomplete, the first four bits of Word 5 correspond to the packets the system has received. The drive sets the bits as it receives up to four packets.
- ❑ If Word 4 is 0, Word 5 contains the execution number of the first illegal event in the execution list. This warning occurs when the compiler is processing a function block program and encounters a bad event value.



**Note:** The drive fault queue contains a textual description of the current fault.

If you receive an I/O node limit error (indicated when Word 4 is 4), the execution list contains more than the allowable number of nodes.

An invalid node reference can occur if a node does not exist. Chapter 6 provides information about handling invalid node references.

### Example

The following examples show a drive function block fault.

		Data Format									
		0	1	2	3	4	5	6	7	8	9
Example 1	BTW Data File	N10:10	0	0F05	0						
	BTR Data File	N10:90	0	0F05	0	2	810A				
						Specifies a link processing error Link ID of the node that has an invalid link					
		0	1	2	3	4	5	6	7	8	9
Example 2	BTW Data File	N10:10	0	3845	0						
	BTR Data File	N10:90	0	3845	0	1024 <sub>Dec</sub>	6				
						Specifies that block 6 in the execution list is illegal					

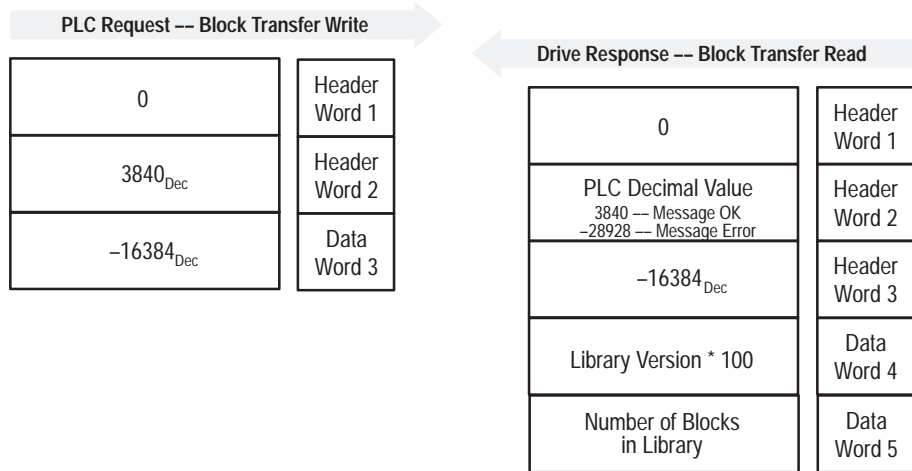
**Program Limits Information:  
Library Description**

The **Library Description** message allows you to read the library's version number and the number of blocks in the library.

**PLC Block Transfer Instruction Data**

BTW Instruction Length: 3 Words  
BTR Instruction Length: 5 Words

**Message Structure**



**Message Operation**

When you request a **Library Description** message, the drive responds by placing the library version number times 100 in Word 4 and the number of blocks the library contains in Word 5.

**Example**

In this example, the **Library Description** requests returns 100<sub>Dec</sub> (version 1.00) for the library version number and 28<sub>Dec</sub> for the number of blocks in the library.

**Data Format**

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N10:10	0	3840	-16384							
BTR Data File	N10:90	0	3840	-16384	100	28					

## Program Limits Information:

### Scheduled Task Interval (mS)

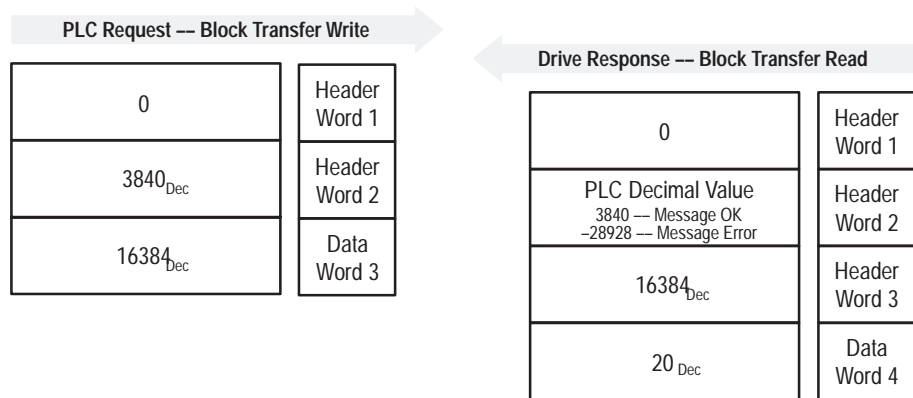
The **Scheduled Task Interval (mS)** message allows you to determine the task interval used when executing your application.

### PLC Block Transfer Instruction Data

BTW Instruction Length: 3 Words

BTR Instruction Length: 4 Words

#### Message Structure



### Message Operation

The **Scheduled Task Interval (mS)** message returns the task interval used when executing your application. The task interval is given in milliseconds. Currently, this value is always 20 milliseconds.

### Example

In this example, the drive returns 20<sub>Dec</sub> for the task interval.

#### Data Format

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N10:10	0	3840	16384							
BTR Data File	N10:90	0	3840	16384	20						

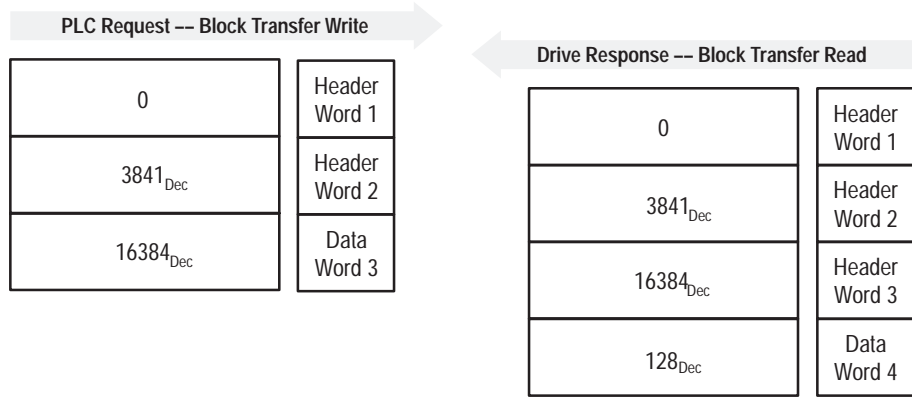
**Program Limits Information:  
Maximum Number of Events per Application**

The **Maximum Number of Events per Application** message allows you to determine the maximum number of events that you can have in each application.

**PLC Block Transfer Instruction Data**

BTW Instruction Length: 3 Words  
 BTR Instruction Length: 4 Words

**Message Structure**



**Message Operation**

This message returns the maximum number of events that you can have in an application. Currently, this number is always 128.

**Example**

In this example, the drive returns 128 for the maximum number of events per application.

**Data Format**

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N10:10	0	3841	16384							
BTR Data File	N10:90	0	3841	16384	128						

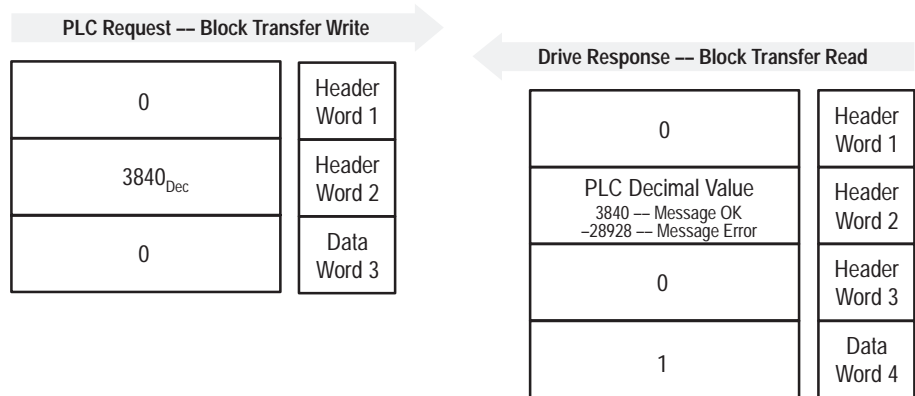
## Program Limits Information: Number of Function Block Task Files in Product

The **Number of Function Block Task Files in Product** message allows you to determine the number of function block task files in the product.

### PLC Block Transfer Instruction Data

BTW Instruction Length: 3 Words  
BTR Instruction Length: 4 Words

#### Message Structure



### Message Operation

This message returns the number of function block task files that are in the product. Currently, only one file is available.

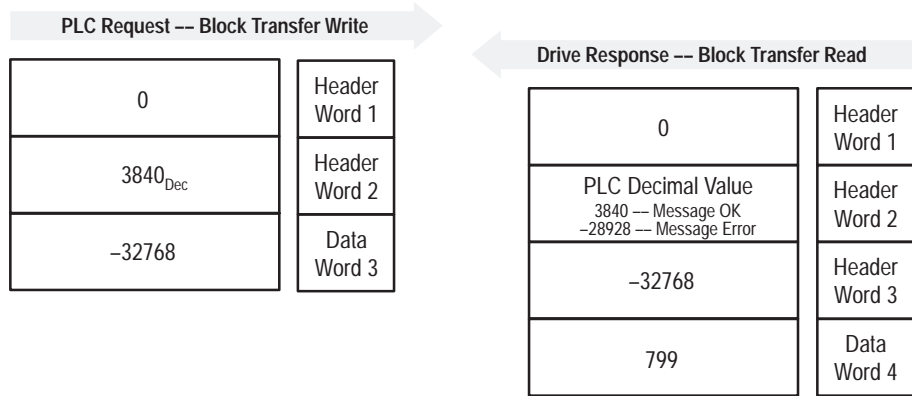
**Program Limits Information:  
Maximum Number of I/O Nodes Allowed per Application**

The **Maximum Number of I/O Nodes Allowed per Application** message allows you to determine the maximum number of I/O nodes that you can have in each application.

**PLC Block Transfer Instruction Data**

BTW Instruction Length: 3 Words  
BTR Instruction Length: 4 Words

**Message Structure**



**Message Operation**

This message returns the maximum number of I/O nodes that you can have in each application. Currently, this request always returns 799.

**Example**

In this example, the drive returns 799 for the maximum number of available I/O nodes per application.

**Data Format**

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N10:10	0	3840	-32768							
BTR Data File	N10:90	0	3840	-32768	799						

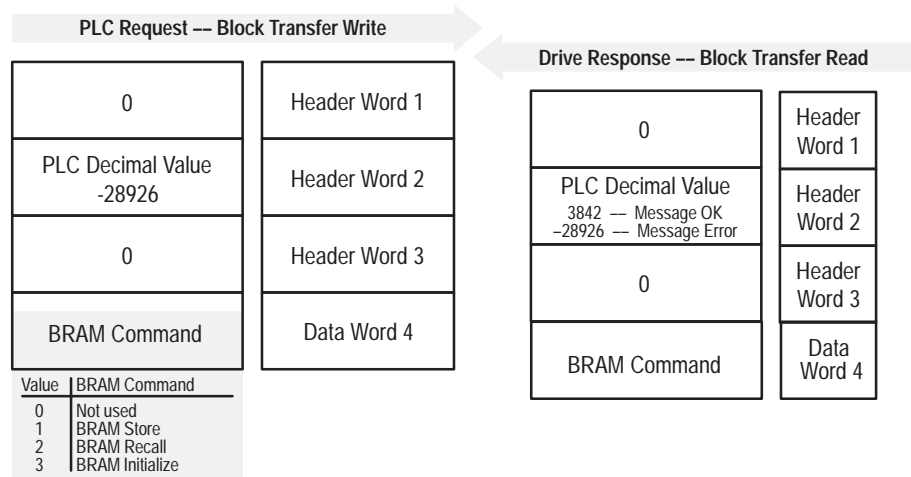
## Application Control Commands: BRAM Functions Store, Recall, and Initialize

The PLC Communication Board sends this message to activate the function block BRAM function that is detailed in the message request.

### PLC Block Transfer Instruction Data

BTW Instruction Length: 4 Words  
BTR Instruction Length: 4 Words

#### Message Structure



### Message Operation

A **BRAM Store** saves the function block program that is currently in RAM to the function block BRAM/EEPROM.

A **BRAM Recall** copies the function block program that is stored in BRAM to RAM. The drive executes the program stored in RAM.

A **BRAM Initialize** erases the function block program stored in RAM and all associated function block links including links to linear drive parameters. A BRAM Initialize function does not actually clear out the BRAM itself; it only clears the function block application out of the working RAM area.



Refer to Chapter 3 for a more detailed description of BRAM functionality.

## BRAM Functions

(continued)

### Examples

This example shows a successful BRAM **Store** operation.

Data Format		0	1	2	3	4	5	6	7	8	9
BTW Data File	N10:10	0	-28926	0	1						
BTR Data File	N10:90	0	3842	0							

This example shows a successful BRAM **Recall** operation.

Data Format		0	1	2	3	4	5	6	7	8	9
BTW Data File	N10:10	0	-28926	0	2						
BTR Data File	N10:90	0	3842	0							

This example shows a successful BRAM **Init** operation.

Data Format		0	1	2	3	4	5	6	7	8	9
BTW Data File	N10:10	0	-28926	0	3						
BTR Data File	N10:90	0	3842	0							



## Application Control Commands: Download and Compile

The **Download and Compile** message does the following:

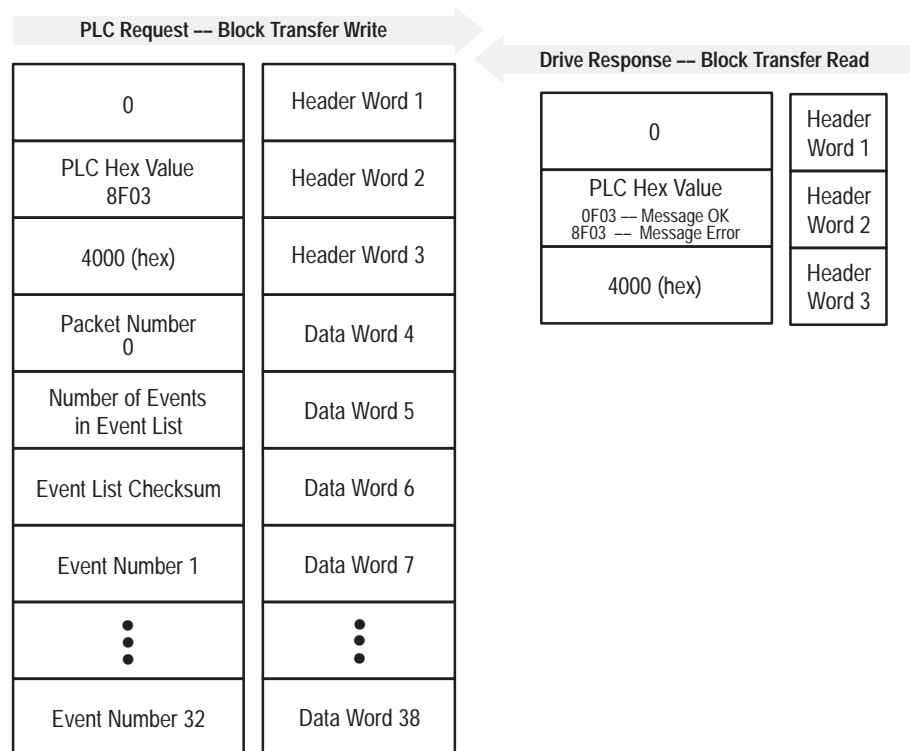
- Downloads up to 32 valid events.
- Performs service checks.
- Compiles these events into a function block application.

Because the download and compile service downloads a maximum of 32 events in one message, your application may need to be downloaded in multiple messages, or packets. The number of events in your application determines the number of packets. Currently, the maximum number of packets you can have is 4 (packet 0 through packet 3) because of the 128 event limitation.

### PLC Block Transfer Instruction Data

BTW Instruction Length: 38 Words  
BTR Instruction Length: 3 Words

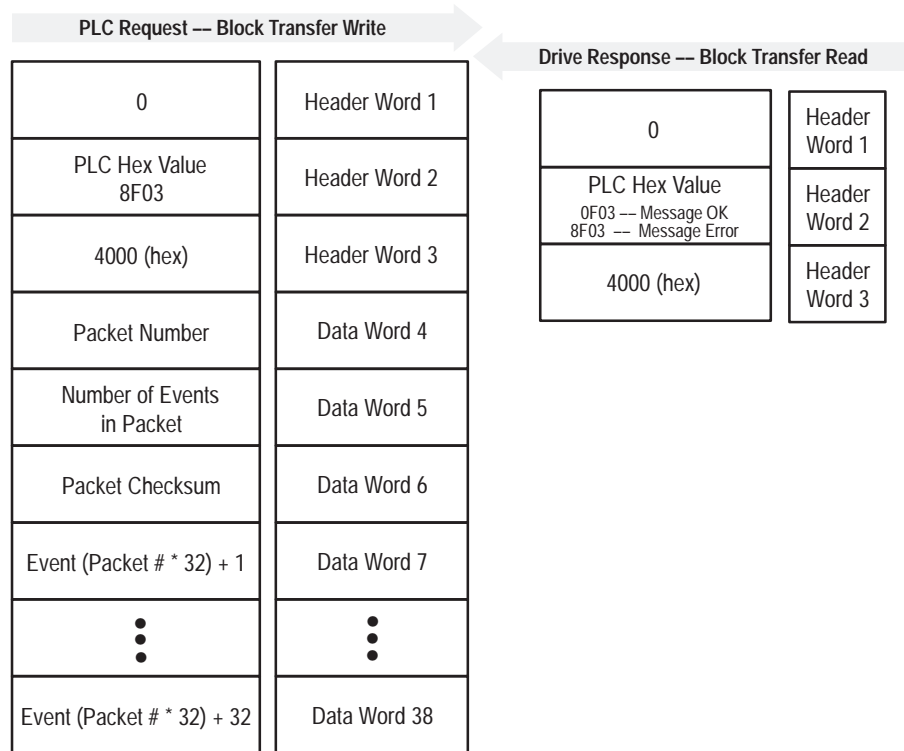
### Packet 0 Message Structure



## Download and Compile

(continued)

### Subsequent Packet Message Structure (Packets 1 – 3)



### Message Operation for Packet 0

When Word 4 is 0, it defines packet 0, which is the command packet. Word 5 of packet 0 defines the total number of events in the entire application. The total number of events in the application tells the service how many packets are required to complete a download. The number of packets required is determined by the number of events divided by 32, rounded up to the next highest integer.

When Word 4 is 0, Word 6 contains the execution list checksum. The execution list checksum is the summation of all the values that represent each event. For example, if you have an execution list with two function blocks, 0115 and 021A, the execution list checksum would be 032F. If the sum of the execution list is more than four digits (hexadecimal), then you should use the right-most (least significant) digits. For example, if the sum of the events is 10ABCD, the checksum would be ABCD.

## Download and Compile (continued)

### Message Operation for Subsequent Packets

The packet number determines where the packet's event data is written in the drive's event list. The drive uses the information contained in Word 4 of packet 0 to re-assemble the packets in the correct order. When Word 4 is not 0, Word 4 contains the packet number, Word 5 contains the number of valid events within the packet, and Word 6 contains the checksum for the events in that packet.

The Task Status service indicates when a download is in progress and the service is waiting for event packets. If a download is in progress and there are no errors, the Fault Read service's code identifier (word 5) indicates which packets have been received. The packets do not need to be sent in order.

After the drive receives all packets required for the application, the drive performs a number of checks on the data. After all the service checks have passed, the function block compiler is activated and the service is acknowledged. However, compilation is not complete when the last packet is acknowledged. Instead, compilation is performed as a background task.

Compile time errors will soft fault the drive. If no compiler errors are found, all function blocks are processed. If no errors occur after this, the function block task is enabled for execution.



**Note:** If the data fails the tests, the drive will **Not AcKnowledge (NAK)** the service but will not fault the drive, and a warning will be logged in the drive's warning queue. These service failures reset the service to accept another download and compile attempt. The currently active application is not interrupted.

**Important:** You must disable the drive to perform a **Download and Compile** operation. The drive will reject the download and compile operation if the drive is running.

The download operation should be performed as a one-shot procedure.

For more information about downloading and compiling, refer to the following sections:

For Information About	Refer to Chapter
Events	2
Examples	2
Compiler Process	2
Compile Modes	3
Task Status	5
Fault Status	5

## Download and Compile (continued)

### Example

In this example, a **Download and Compile** message was sent to the drive. The application contains four events and is downloaded in one packet.

Data Format

	0	1	2	3	4	5	6	7	8	9	
BTW Data File	N10:10	0	8F03	4000	0000	0004	0A4A	010C	0216	0315	0413
BTR Data File	N10:90	0	0F03	4000							

Labels above the table: Packet Number (2), Events in List (3), Event List Checksum (4), Event 1 (6), Event 2 (7), Event 3 (8), Event 4 (9). Arrows point from these labels to the corresponding columns in the BTW Data File row.

In this next example, an application that has 60<sub>Hex</sub> (98<sub>Dec</sub>) events is downloaded in three separate packets. To download the three packets, you can either send three separate block transfer write (BTW) and block transfer read (BTR) pairs to the drive, or you can set up one BTW and BTR pair and move the data for each of the packets into the BTW statement after the preceding packet has been sent and received.



**Note:** If less than 32 events are in the packet, the rest of the event data is padded with zeros.

## Download and Compile (continued)

Regardless of the method that you use, the data (in hexadecimal) for the three packets is as follows:

### Packet 0 Data

	0	1	2	3	4	5	6	7	8	9	
BTW Data File	N67:0	0	8F03	4000	0	60	663A	705	815	0C14	0
	N67:10	1015	141B	1814	1C1A	0	2001	2414	2801	0	2C14
	N67:20	3015	3208	3413	3515	0	3715	380D	3C14	401B	4415
	N67:30	2D0C	6216	5915	4A13	0	0	0	0	0	0

### Packet 1 Data

	0	1	2	3	4	5	6	7	8	9	
BTW Data File	N67:100	0	8F03	4000	1	20	58F1	501B	541A	5814	5C14
	N67:110	0	6014	6414	6814	6C1A	0	7014	7414	7814	7C08
	N67:120	0	801A	840D	8817	8C08	0	9007	911C	9204	9304
	N67:130	0	950C	9619	9715	9914	9A1B	9C0C	A014	0	0

### Packet 2 Data

	0	1	2	3	4	5	6	7	8	9	
BTW Data File	N67:200	0	8F03	4000	2	20	D49E	B014	B20D	B41C	B614
	N67:210	0	BA14	BC14	BE14	0	C41B	C614	C808	CA07	CB1C
	N67:220	CC0C	CD04	CE04	0	D019	D115	D408	D515	D60A	D714
	N67:230	D81A	DA14	DC0C	0	0	0	0	0	0	0



**Note:** Zeros indicate NULL events.

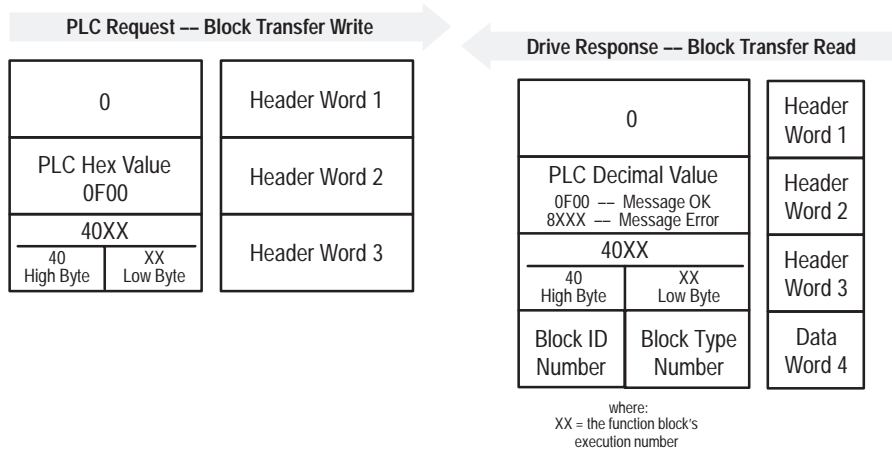
**Application Control  
Commands:  
Read Single Event**

The **Read Single Event** message reads the block type number and block ID number for the requested execution number within the drive's execution list.

**PLC Block Transfer Instruction Data**

BTW Instruction Length: 3 Words  
BTR Instruction Length: 4 Words

**Message Structure**



**Message Operation**

**PLC Request** — The low byte of Word 3 (indicated by XX in the message structure) represents the function block's execution number in the execution list.

**Drive Response** — The drive response contains the block ID number in the high byte of Word 4 and the block type number in the low byte of Word 4 as shown in the execution list.

## Read Single Event

(continued)

### Example

In this example, the information corresponding to the seventh function block in the event list is requested. The drive response indicates the seventh block in the event list is a Scale function block (block type  $20_{\text{Dec}}$ ,  $14_{\text{Hex}}$ ) with a block ID number of  $12_{\text{Dec}}$  ( $0C_{\text{Hex}}$ ).

#### Data Format

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N10:10	0	0F00	4007							
BTR Data File	N10:90	0	0F00	4007	0C14						

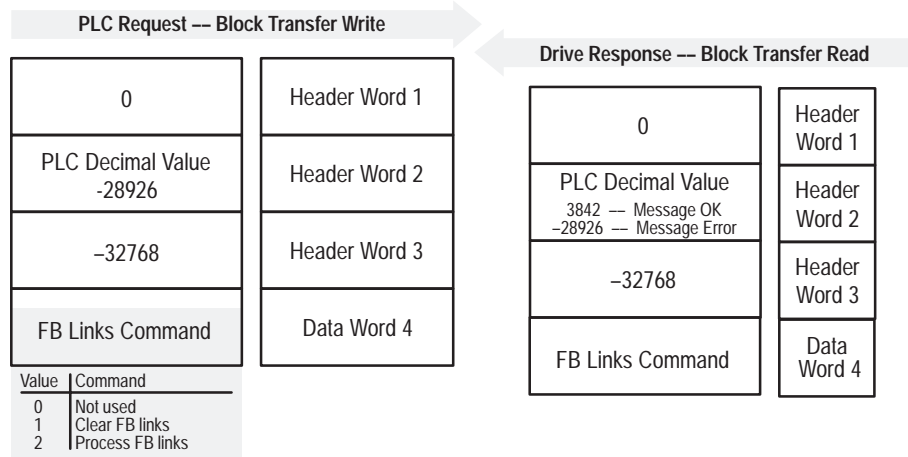
## Application Control Commands: Clear/Process Links

The **Clear/Process Links** message is used to clear or process all function block node links in the event list.

### PLC Block Transfer Instruction Data

BTW Instruction Length: 4 Words  
BTR Instruction Length: 4 Words

#### Message Structure



### Message Operation

When word 4 is 1, this request clears all function block links in the drive. The **Clear Links** operation also clears links for linear drive parameters that receive information from function block nodes. To clear a single link, you need to:

1. Perform a node link.
2. Link the destination node to 0.



**Note:** Clearing links using the **Clear Links** operation clears the links from the working RAM area. This operation does not affect data stored in BRAM.

When word 4 is 2, the compiler processes and re-establishes all function block links in the drive.

The **Process Links** operation re-processes all node links stored in RAM tables.



## Clear/Process Links

(continued)

### Example

This example shows the data format of a **Clear All Function Block Links** operation.

#### Data Format

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N10:0	0	-28926	-32768	1						
BTR Data File	N10:90	0	3842	-32768	1						

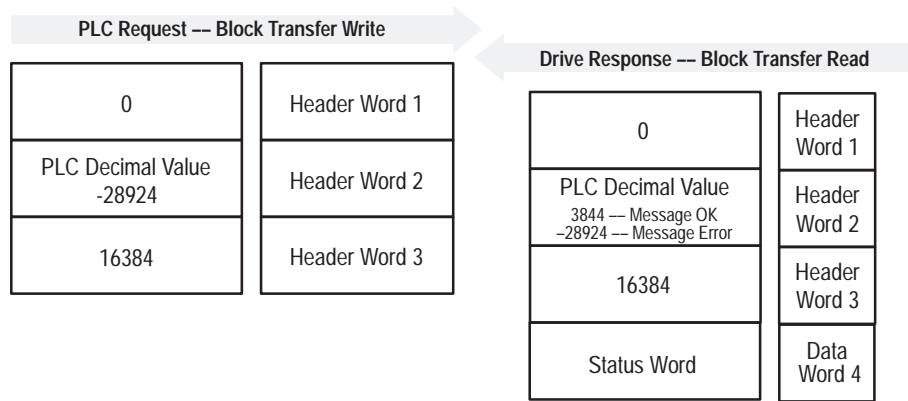
**Application Control  
Commands:  
Download Service Init**

The **Download Service Init** message initializes the download service.

**PLC Block Transfer Instruction Data**

BTW Instruction Length: 3 Words  
BTR Instruction Length: 4 Words

**Message Structure**



**Message Operation**

The **Download Service Init** message is available for you to use when you need to initialize the download service.

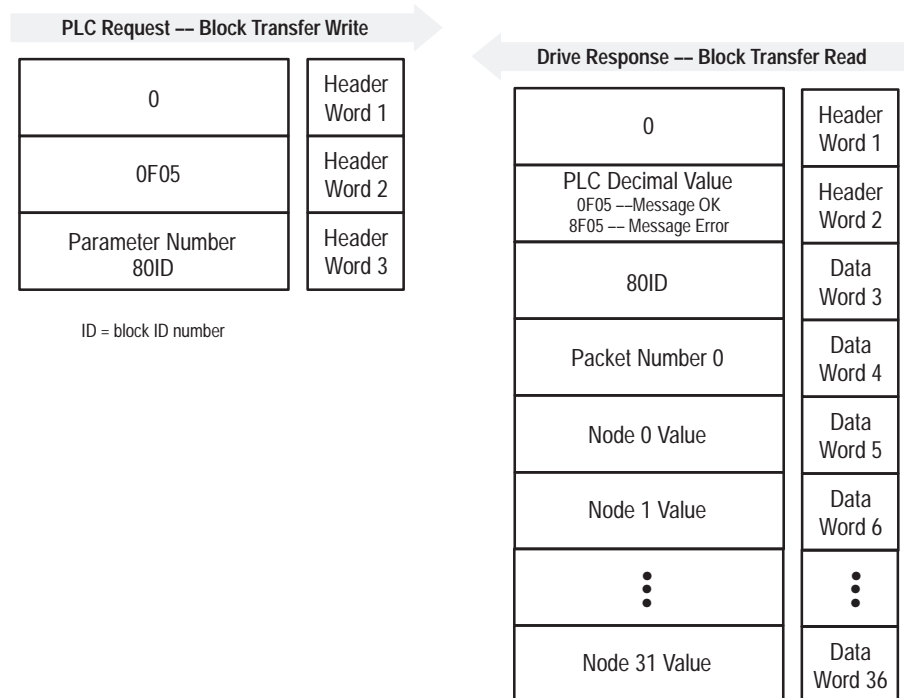
## Node Adjustment: Read Block Value

The **Read Block Value** message reads the 16-bit parameter data value for the specified function block. This returns all node values for one block.

### PLC Block Transfer Instruction Data

BTW Instruction Length: 3 Words  
BTR Instruction Length: 36 Words

#### Message Structure



### Message Operation

The **Read Block Value** function specified in the BTW reads the specified function block values from the drive and places the values (or an error code) in words 5 through 36 of the BTR data file. If an error has occurred, word 2 of the BTR is 8F05<sub>Hex</sub> and word 4 contains the status code.

## Read Block Value

(continued)

### Example

In this example, a **Read Block Value** message was sent to the drive:

Data Format		0	1	2	3	4	5	6	7	8	9
BTW Data File	N10:10	0	0F05	8004							
BTR Data File	N10:90	0	0F05	8004	7FFF	0	7FFF	FFFF	584A	0	0

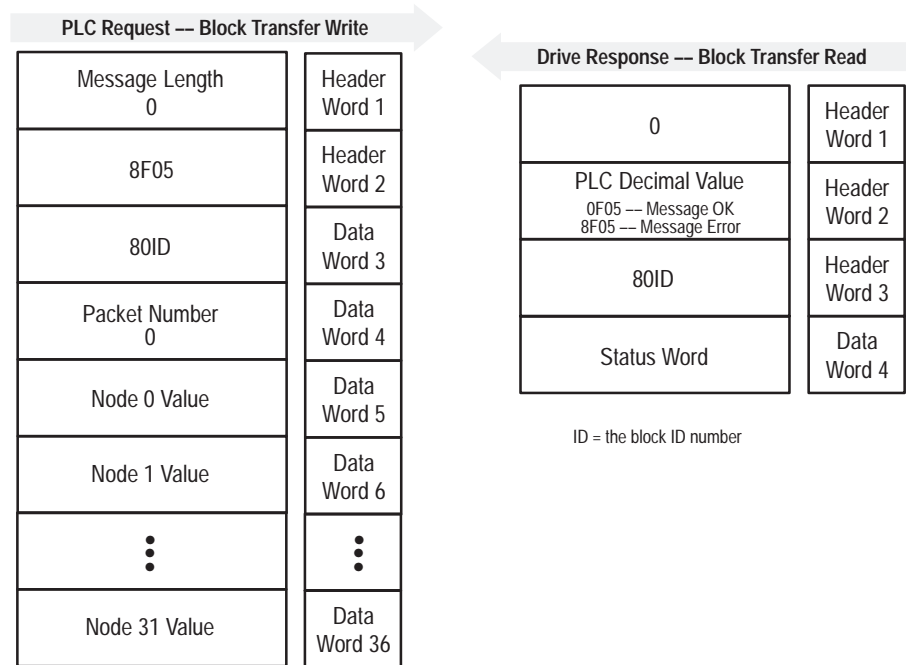
## Node Adjustment: Write Block Value

The **Write Block Value** message writes the 16 bit data values to the specified function block. All the values for the entire function block are sent.

### PLC Block Transfer Instruction Data

BTW Instruction Length: 36 Words  
BTR Instruction Length: 4 Words

#### Message Structure



### Message Operation

The data buffer contains the packet number and up to 32 valid node values. Therefore, you must pad the length of the buffer with zeros to reach the 32 value length if the function block has less than 32 nodes.

Currently, the packet number must be zero.

If any of the values are out of range for any given node, the service will fail.

Values beyond the valid number of inputs for the given block ID are ignored.

## Write Block Value

(continued)

### Example

In this example, a **Write Block Value** message was sent to the drive:

Data Format		0	1	2	3	4	5	6	7	8	9
BTW Data File	N10:10	0	8F05	8004	7FFF	0	7FFF	FFFF	584A	0	0
BTR Data File	N10:90	0	0F05	8004							

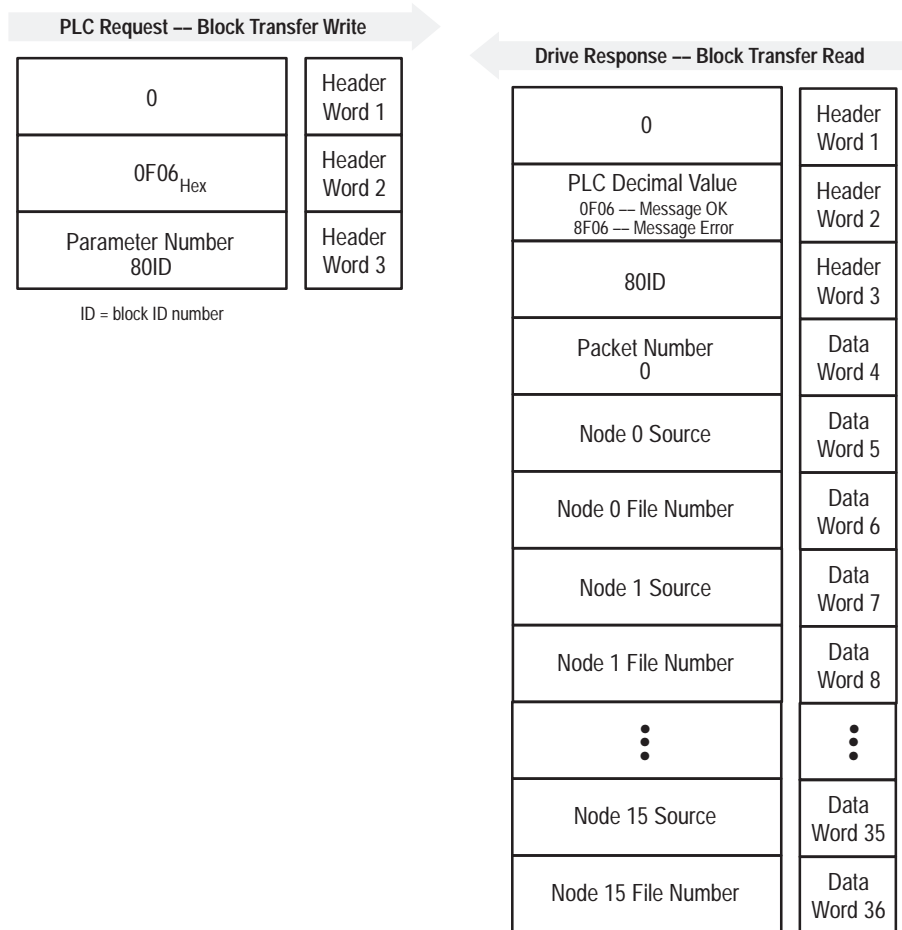
## Node Adjustment: Read Block Link

The **Read Block Link** message reads the link information for an entire block.

### PLC Block Transfer Instruction Data

BTW Instruction Length: 3 Words  
BTR Instruction Length: 36 Words

#### Message Structure



### Message Operation

The **Read Block Link** message reads the source nodes that are linked to the specified block. The returned message packet contains 33 data buffer words, and is always padded with zeros to length.

Currently, the packet number must be zero.

Currently, you can disregard the file number.

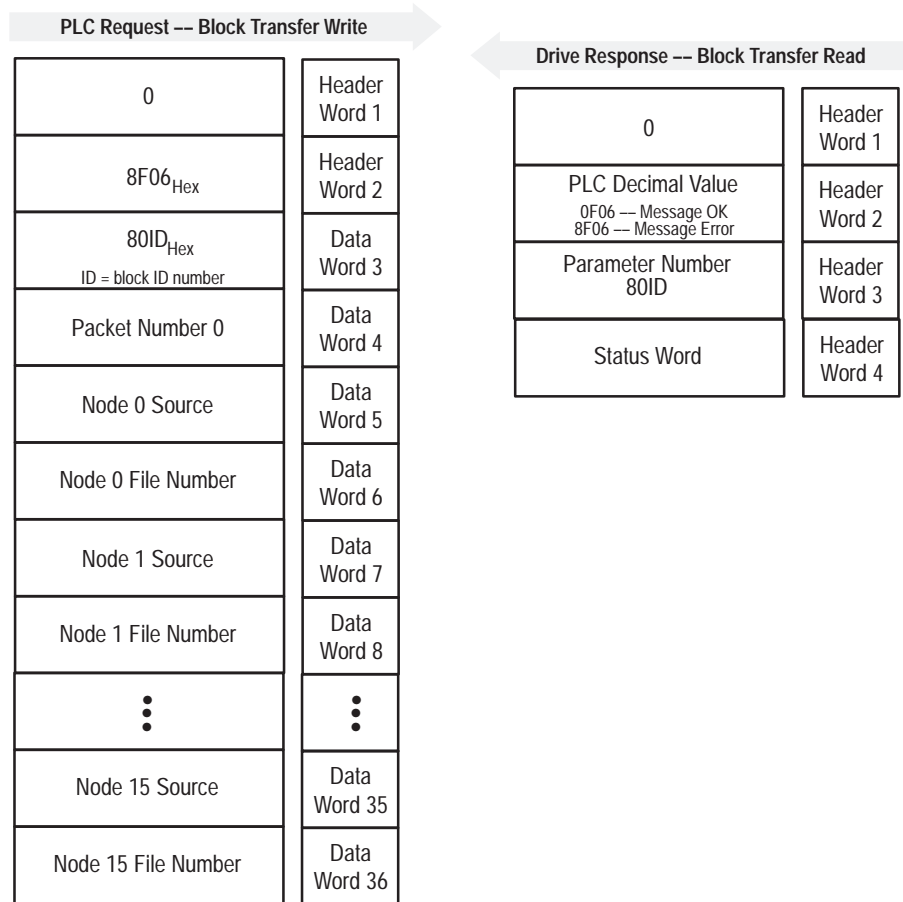
**Node Adjustment:  
Write Block Link**

The **Write Block Link** message writes the link information for an entire block.

**PLC Block Transfer Instruction Data**

BTW Instruction Length: 36 Words  
 BTR Instruction Length: 4 Words

**Message Structure**



**Message Operation**

The **Write Block Link** message writes the output parameter to the input parameter. The data words contain the packet number and up to 15 source nodes. The buffer must always be padded with zeros if 32 nodes are not being written.

If the service encounters any invalid output references, the drive stops processing the service and returns an error.



## Write Block Link

(continued)

Output values that are beyond the number of valid linkable input nodes are ignored. Therefore, if a block type has only one linkable input and five links are written, only one is established.

Currently, the packet number must be zero.

Currently, you can disregard the file number. The file numbers are for future use. For now, you should set the file number to 0 for a linear parameter output and 1 for a function block node.

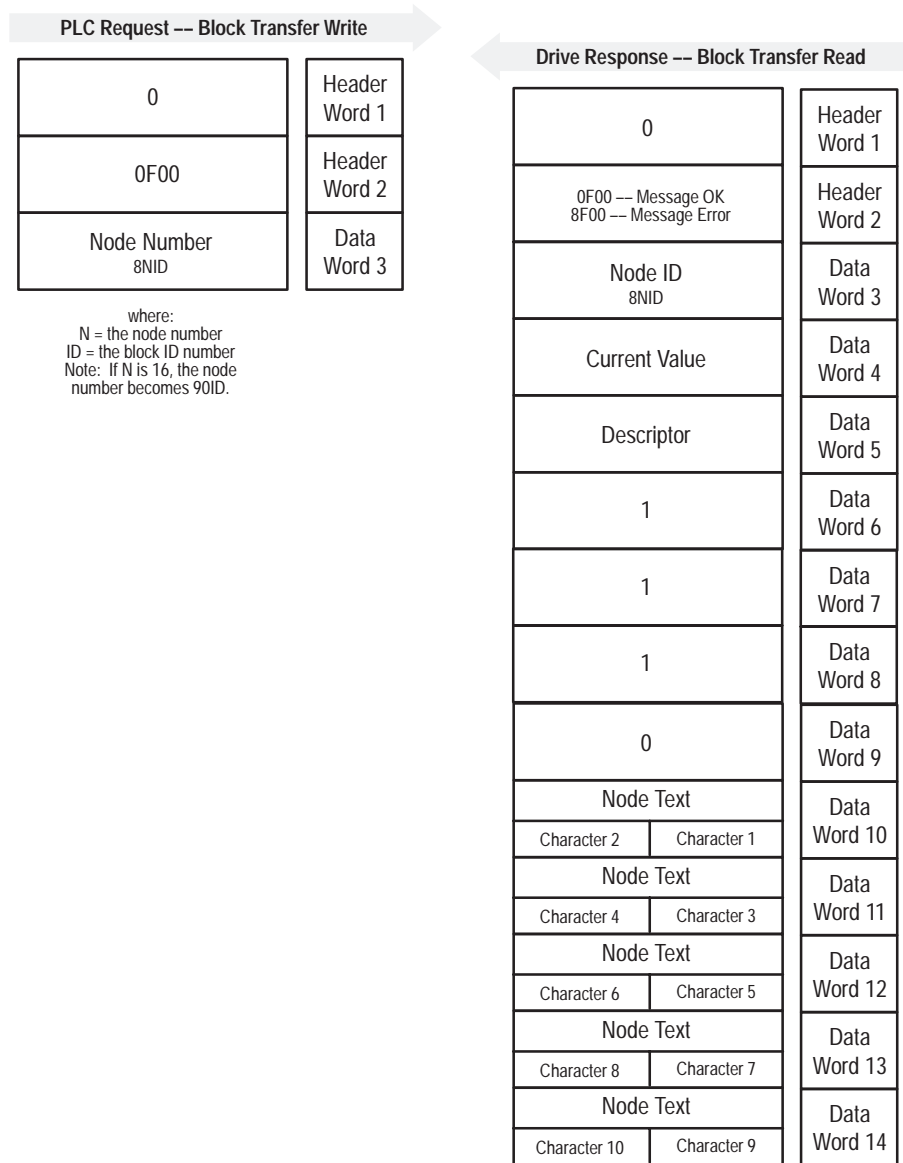
**Node Adjustment:  
Read Full Node Information**

The **Read Full Node Information** message provides all known attributes for any node in the current application. This information includes the node's current value, descriptor, multiply and divide value, base value, offset value, text string, file group and element reference, minimum value, maximum value, and default value.

**PLC Block Transfer Instruction Data**

BTW Instruction Length: 3 Words  
 BTR Instruction Length: 22 Words

**Message Structure**



## Read Full Node Information

(continued)

### Block Transfer Read (Continued)

Parameter Text		Data Word 15
Character 12	Character 11	
Parameter Text		Data Word 16
Character 14	Character 13	
Parameter Text		Data Word 17
Character 16	Character 15	
Minimum Value		Data Word 18
Maximum Value		Data Word 19
Default Value		Data Word 20
		Data Word 21
		Data Word 22

### Message Operation

**Read Full Node Information** provides all known attributes for any node in the current application.

## Read Full Node Information

(continued)

### Example

In this example, a **Read Full Node Information** message was sent to the drive. Word 4 shows the present value in drive units. Word 5 through word 8 provide scaling information, used to convert drive units to engineering units. Word 9 through word 12 provide the parameter name.

This example shows the response message N10:90 through N10:111 in both binary and ASCII. The parameter name characters return in reverse order for each word. N10:99 has the ASCII value of **aR**. To read this, invert the word to read **Ra**. The next word, **et**, inverted gives you **te**. These words along with the following two words form the words **Rate Lim**.

#### Data Format

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N10:10	0000	0F00	8504							
BTR Data File	N10:90	0000	0F00	8504	CCCE	002B	0001	0001	0001	0000	6152
	N10:100	6574	4620	6D69	2320	2034	754F	2074	8001	7FFF	0000
	N10:110	2020	2020								

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N10:10	\00\00	\0F\00	à \04							
BTR Data File	N10:90	\00\00	\0F\00	à \04	\CC\CE	\00 +	\00\01	\00\01	\00\01	\00\00	a R
	N10:100	e t	L	m i	#	4	u 0	T	ç \01	\7F\FF	0000
	N10:110										

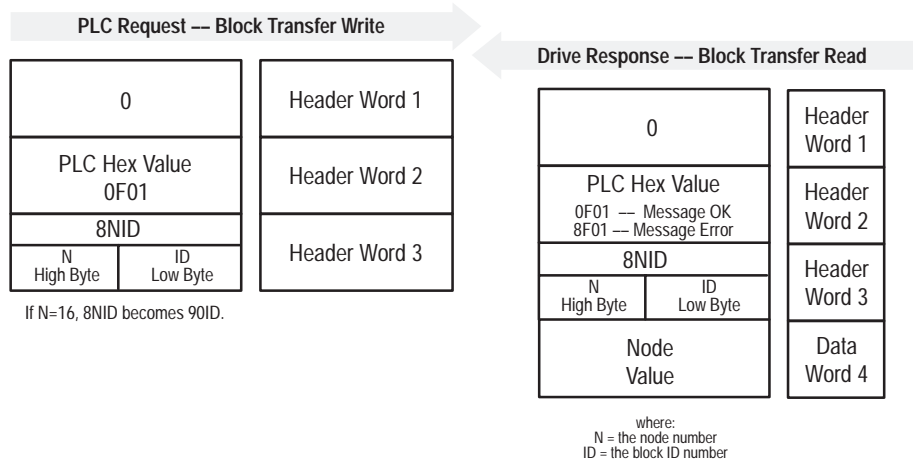
## Node Adjustment: Read Node Value

The **Read Node Value** message reads the value of the specified function block node.

### PLC Block Transfer Instruction Data

BTW Instruction Length: 3 Words  
 BTR Instruction Length: 4 Words

#### Message Structure



### Message Operation

Word 3 specifies the function block node.

- ID indicates the block ID number is specified in the low byte.
- N indicates the node number is specified in the high byte (bits 8 through 14).

### Example

In this example, a **Read Node Value** message was sent to the drive to read the value of node 1 of the Scale function block (block ID 11<sub>Dec</sub>, 0B<sub>Hex</sub>). The drive response indicates the value of node 1 is 4.

#### Data Format

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N10:10	0	0F01	810B							
BTR Data File	N10:90	0	0F01	810B	4						



**Note:** The output of a Bin2Dec block is node 16<sub>Dec</sub>, and would be specified as 900B if it were assigned a block ID number of 11<sub>Dec</sub>.

**Node Adjustment:  
Write Node Value**

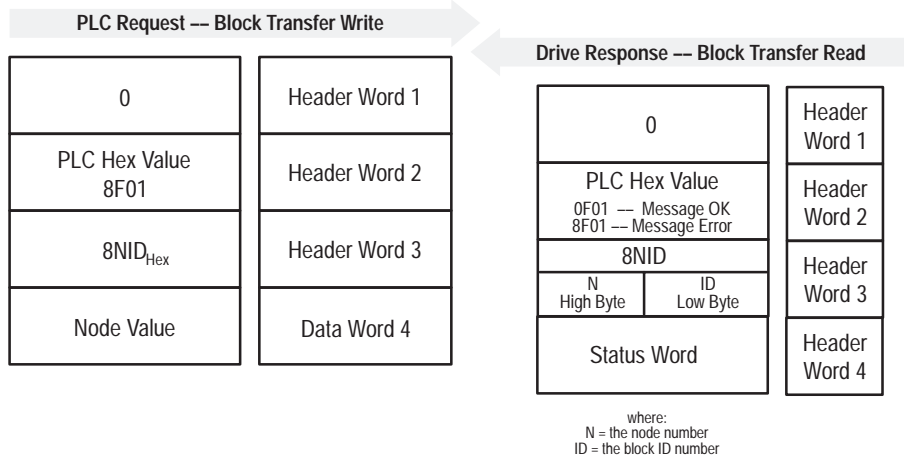
The **Write Node Value** operation writes a value to a specified function block node.

**PLC Block Transfer Instruction Data**

BTW Instruction Length: 4 Words

BTR Instruction Length: 4 Words

**Message Structure**



**Message Operation**

Word 3 specifies the function block node.

- ID, specified in the low byte, indicates the block ID number.
- N, specified in the high byte, indicates the node number.

The value in word 4 is written to this node. If this node is linked to another function block node or drive parameter, the parameter output value overwrites the node value. If the most significant bit of word 2 is set, word 4 contains an error message.

**Example**

In this example, a value of 4 is written to node 1 of the function block with a block ID number of 11<sub>Dec</sub> (0B<sub>Hex</sub>).

**Data Format**

		0	1	2	3	4	5	6	7	8	9
BTW Data File	N10:0	0	8F01	810B	4						
BTR Data File	N10:90	0	0F01	810B							

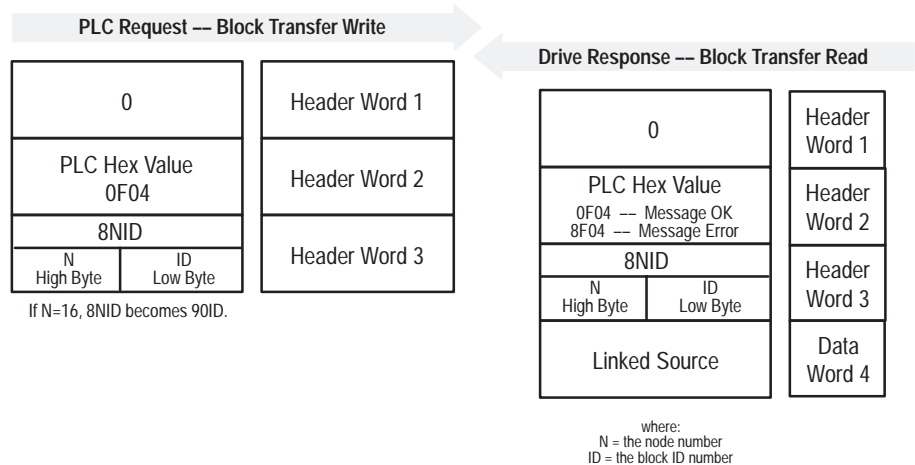
## Node Adjustment: Read Node Link

The **Read Node Link** operation reads the drive parameter or the node numbers of the source that is linked to the specified function block node.

### PLC Block Transfer Instruction Data

BTW Instruction Length: 3 Words  
BTR Instruction Length: 4 Words

#### Message Structure



### Message Operation

The **Read Node Link** operation reads the output parameter that is linked to the function block node specified in word 3.

- N, specified in the high byte, indicates the node number.
- ID, specified in the low byte, indicates the block ID number.

The data returned from the drive identifies the source parameter. If the node is linked to a drive parameter, word 4 contains the parameter number. If the node is linked to another function block node, word 4 contains the node number and block ID number.

Chapter 2 contains additional information that is helpful for understanding node reference numbers.

## Read Node Link (continued)

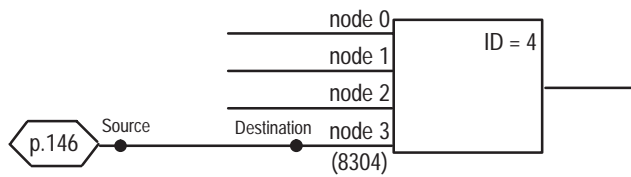
### Example

In both examples, the PLC has requested to read node number 3<sub>Dec</sub> (3<sub>Hex</sub>), block ID number 4<sub>Dec</sub> (4<sub>Hex</sub>).

In Example 1, the drive response indicates that the node is linked to parameter 146<sub>Dec</sub> (92<sub>Hex</sub>), the Motor Overload Limit parameter.

Data Format		0	1	2	3	4	5	6	7	8	9
Example 1	BTW Data File	N10:10	0	0F04	8304						
	BTR Data File	N10:90	0	0F04	8304	92					

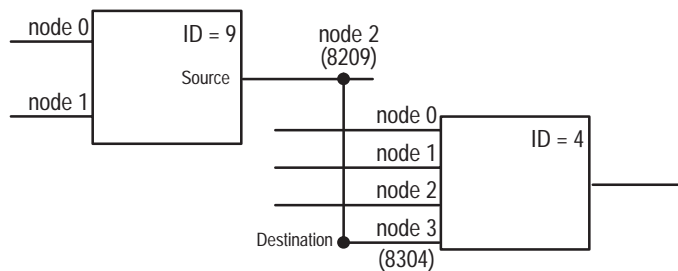
This example can be visually shown as follows:



In Example 2, the drive response indicates that the node is linked to another function block node: node number 2<sub>Dec</sub> (2<sub>Hex</sub>), block ID number 9<sub>Dec</sub> (9<sub>Hex</sub>).

Data Format		0	1	2	3	4	5	6	7	8	9
Example 2	BTW Data File	N10:10	0	0F04	8304						
	BTR Data File	N10:90	0	0F04	8304	8209					

The second example can be visually shown as follows:





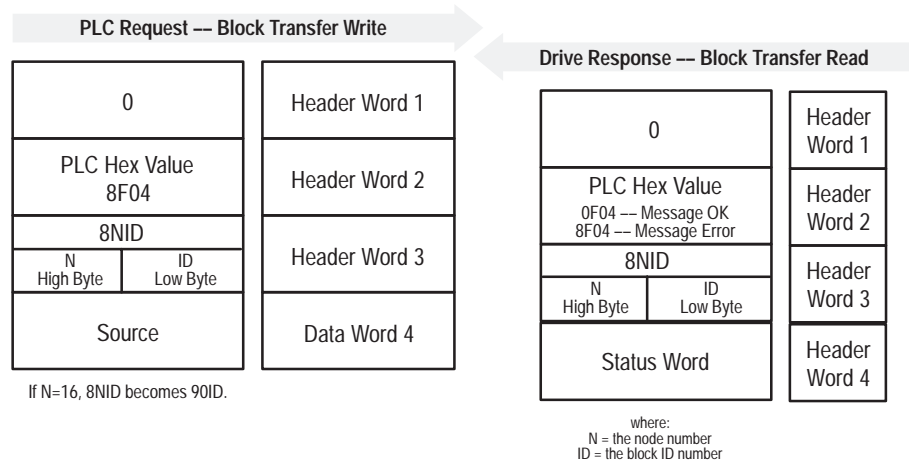
## Node Adjustment: Write Node Link

The **Write Node Link** operation creates a single node link between a specified drive linear parameter number or function block node and the destination node.

### PLC Block Transfer Instruction Data

BTW Instruction Length: 4 Words  
BTR Instruction Length: 3 Words

#### Message Structure



### Message Operation

Word 3 specifies the function block node that is the input or destination.

- N, specified in the high byte, indicates the node number.
- ID, specified in the low byte, indicates the block ID number.

The value in word 4 is the linked destination parameter or node number.



**Note:** You must make sure that the drive is disabled when you perform a **Write Node Link** operation. If the drive is running, it will not accept the link and it will reject the message.

To clear an individual link, link the destination to 0.

The source can be either a linear parameter or a function block node.

## Write Node Link (continued)

### Example

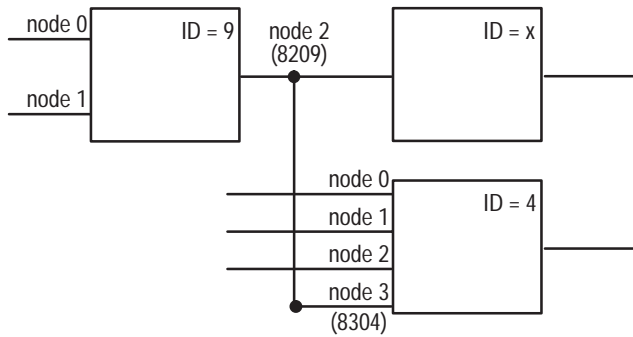
In both examples, the PLC has requested to write to node number  $3_{Dec}$  ( $3_{Hex}$ ), block ID number  $4_{Dec}$  ( $4_{Hex}$ ).

In Example 1, the PLC request indicates that the linked source is parameter  $146_{Dec}$  ( $92_{Hex}$ ), the Motor Overload Limit parameter.

In Example 2, the PLC request indicates that the linked source is a function block node: node number  $2_{Dec}$  ( $2_{Hex}$ ), block ID number  $9_{Dec}$  ( $9_{Hex}$ ).

		Data Format									
		0	1	2	3	4	5	6	7	8	9
Example 1	BTW Data File	N10:10	0	8F04	8304	92					
	BTR Data File	N10:90	0	0F04	8304						
Example 2	BTW Data File	N10:10	0	8F04	8304	8209					
	BTR Data File	N10:90	0	0F04	8304						

The second example can be visually represented as the following:



## Handling Exceptions — Faults and Warnings

### Chapter Objectives

This chapter provides the following information:

- Handling function block faults and warnings
- Accessing the system fault and warning queues
- Handling download service errors
- Handling compiler faults
- Using the Task Status service
- Using the Fault Status service
- Fault codes

### Handling Function Block Faults and Warnings

If there is a problem with your function block application, either a fault or a warning will occur.

#### Function Block Faults

All function block faults are soft faults. Soft faults indicate that an error has been detected that could damage drive components or the motor. Soft faults may also indicate potential undesirable operation.

You cannot enable the drive if you have a soft fault condition.

You need to correct the problem that caused the fault before you can clear the function block system fault with a clear faults command. If you use a clear faults command before you correct the function block problem, the function block fault will remain in the queue but the other faults in the system queue will be cleared.

When the drive encounters a function block fault, the following occurs:

- The drive disables the 20 millisecond task interval.
- The function block task status returns a faulted status value of  $0xFF_{Hex}$  when read.
- Word 4 of the function block fault read service is non-zero.
- An entry is logged in the system fault queue. You can view the system fault queue from DriveManager.

Most function block system faults are designated for compile time errors. The download service performs a number of tests before starting the compile process for the purpose of avoiding compile faults.

The most common function block system fault is likely to be an invalid function block link error. This fault can occur during power up, reset, a linear parameter BRAM recall, or a download and compile operation. Handling of this particular error is covered later in this chapter.

### Function Block Warnings

Function block warnings announce conditions that could cause a soft fault and stop the drive. Warnings are less severe faults. When a warning occurs, a warning code is entered in the warning queue and status parameters reflect the condition. A drive stop does not occur, and drive operation is not affected. You can clear a warning by issuing a clear warning command.

### General Information about Faults and Warnings

The three mechanisms for identifying specific function block system faults and warnings are:

- the function block fault read service (covered in chapter 5)
- the system fault queue
- the system warning queue

When a fault or warning occurs, an entry is logged into the respective queue. Each entry contains a code and some descriptive text. The fault and warning queues keep a history of 32 fault or warning events until you clear the fault/warning queue.

To handle the faults and warnings, you need to access the system fault and warning queues that are associated with each drive. Each entry in the queue shows the type of fault and the time and date when the fault occurred.

## Accessing the System Fault and Warning Queues

Once a fault or warning occurs, the information about the fault is maintained in BRAM until you clear the fault queue using a Clear Fault Queue command. To clear the function block fault once you have corrected the problem within the function block application, perform one of the following tasks:

- Issue a Clear Fault command.
- Issue a drive reset command.
- Cycle drive power.

The fault queue contains information for up to 32 faults. The following information is also maintained for each fault:

- a fault queue entry number to indicate the position of the fault in the fault queue
- a trip point (TP) to indicate which entry in the fault queue caused the drive to trip (all faults displayed before the TP fault occurred after the TP was logged)
- a five character decimal fault code, which is covered later in this chapter
- the time and date that the fault occurred
- descriptive fault text
- all clear fault commands and the time when they were executed

You need to use the Fault Queue Entry Read service to access the information in the fault queue. Refer to the PLC Communications User Manual for more information about the Fault Entry Read service.

## Handling Download Service Errors

Because the download operation is more complicated than most of the block transfer services, it is more common to see faults/warnings at this time. The download operation may require that multiple messages be sent before initiating a compile operation. When an application requires more than one message packet, the service ensures that a compile operation does not begin until the last required packet is received and acknowledged.

After receiving the last packet, the drive performs a series of checks to verify the new execution list:

- The drive goes over the execution list and verifies the checksum. If the checksum fails, the service fails.
- The drive checks that any new events with block numbers that match block numbers found within the current application have the same block type. If the block number verification fails, the service fails.
- The service verifies that the block type numbers are within the valid range.
- The service verifies that events with a zero for a block ID have a zero value for a block type number and that events with a zero for a block type number have a zero for a block ID.

When the download service fails, the following steps are taken:

1. Download failures will **NAK (Not AcKnowledge)** the service, but will not fault the drive nor interrupt the currently running function block application.
2. Download failures will reset the service to allow it to accept another download and compile attempt.
3. Values from the currently valid function block application overwrite the execution list.
4. One of the following warnings will be logged:

Warning	Description
FB DNLD Bad Evnt	An attempt was made to download a bad execution list.
FB BAD Pkt Num	The service received a bad packet number.
FB DNLD Bk# Wrn	An invalid block number was received.
FB DNLD Cksm Wrn	The execution list did not pass the checksum test.

## Handling Compile Faults

When a new execution list has been downloaded and all the service checks have been passed:

- The currently active application is taken off line.
- The compiler is initiated as a background process.
- The function block task status indicates that a compile is in process.

Compile time errors will soft fault the drive.

Even though it is unlikely that you will receive a fault once the event list has passed the service checks, several faults are possible.

**Important:** With the exception of function block link processing faults, you should perform a function block **BRAM Init** operation after any function block faults occur. You then need to perform a **Recall** or download the program again.

### Link Processing Fault

If a link processing fault occurs, the execution list has been compiled and all the blocks are valid. Link processing faults are actually generated after the compilation is complete. If no other faults occurred, you can:

1. Adjust the invalid links.
2. Clear the fault queue.

The function block application is then enabled for execution without being re-compiled.

**Important:** You cannot clear function block faults with a clear fault operation without first addressing the problem. The function block system will not make an assumption about what to do with an illegal link. You must either clear the link to this node or reconnect the node to a valid node. You can then use the clear faults mechanism to clear the faults and allow the drive to run.

### I/O Node Limit Fault

If you receive an I/O node limit fault, you have created more than 799 function block nodes due to the number and type of blocks you entered into your execution list. You need to:

1. Perform a function block **BRAM Init**.
2. Remove extra event blocks to reduce the number of nodes.

### Memory Limit Fault

If you receive a memory limit fault, the system is out of dynamic RAM. You need to:

1. Perform a function block **Init**.
2. Remove extra function blocks.

### **BRAM Checksum Fault**

If you receive a BRAM Checksum fault, the data in the BRAM has been corrupted. You need to:

- 1.** Perform a function block **BRAM Init**.
- 2.** Perform a function block **BRAM Store**.
- 3.** Download the execution list.



## Using the Task Status Service

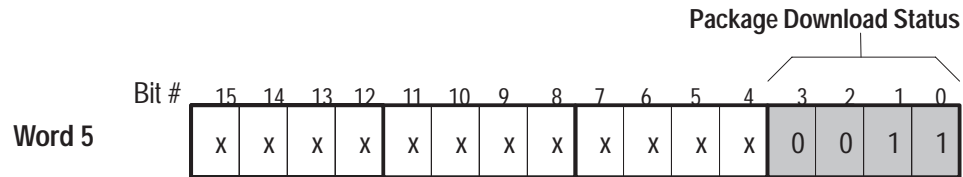
To check to see if a fault occurred, you can use the **Read Task Status** service. The Task Status is returned in word 4 of the drive's response. The Task Status returns a code value to indicate the status of the current function block program. The following are the valid code values.

Value	Task Status	Description
0	Run Mode	The application is executing within the 20 millisecond task interval. No faults have occurred within the function block portion of functionality.
1	Download in Progress	The previously compiled application is still enabled and executing within the function block task interval. One or more downloaded packets have been received for a new function block program and the function block system is waiting for more data. The currently active application is not interrupted until all packets have been received and the data has been verified for the new function block program before compilation.
2	Compilation in Progress	All packets have been downloaded and the data verified. The service has initiated a compile. Compilation can take seconds when a large application is used.
3	Link Processing	The application is disabled and links between function blocks and drive parameters are being established
4	Recall in Progress	A <b>Recall</b> is in progress.
0x00FF	Fault Mode	A function block application has a faulted status. Function block compile time errors create a soft fault condition within the drive. The 1336T system architecture contains a system fault queue that describes the nature of the fault. SCANport provides two fault reporting values should the Task Status word indicate a faulted mode.  The previous application is disabled and will not run until you correct the fault. You cannot clear function block compiler faults with the clear faults command until you correct the function block fault.

## Using the Fault Status Service

The function block fault status service actually does more than identify function block system faults. It also provides information on other parts of the function block system when not faulted. This helps when you are troubleshooting the system when first setting up, particularly when you are downloading with a PLC.

During a download operation while the function block Task Status operation has a value of 1, word 5 of the Fault Status indicates which packets have been received successfully.



In this case, word 4 of the Fault Status is zero because the system has not encountered a fault.

### Download Errors

If an attempted download fails the service checks, a warning is created and word 4 of the Fault Status is zero. Word 5 of the Fault Status service indicates the problem with the last download:

- If bit 8 is set ( $0x01xx_{Hex}$ ), an event value error/warning occurred.
- If bit 9 is set ( $0x02xx_{Hex}$ ), a checksum error/warning occurred.
- If bit 10 is set ( $0x04xx_{Hex}$ ), a block ID number error/warning occurred.

The rest of the download service is reset as if nothing occurred to allow another download attempt.

### Invalid Link Fault Condition

If the drive encounters an invalid link during a linear parameter **Recall**, a reset, or power up sequence:

- Word 4 of the Fault Status will have a value of  $200_{Hex}$  or  $512_{Dec}$ .
- The drive will soft fault invalid function block links in the system fault queue.
- The function block Task Status will be  $0xFF_{Hex}$ .
- You cannot clear the system fault with the Clear Fault command until you adjust the invalid link.

Word 5 of the function block fault status returns the destination parameter or node holding the invalid link.

For example, if you issue a **Fault Status Read** request and the drive response indicates that word 4 is 200<sub>Hex</sub> and word 5 is 183<sub>Hex</sub> (387<sub>Dec</sub>), an invalid link occurred at parameter 387<sub>Dec</sub>. You need to adjust the link to parameter 387 before you can clear the fault queue.

To adjust the link, you can:

- Clear the link with a **Clear All FB Links** command.
- Clear the link by writing a source value of 0.
- Reconnect the parameter to a valid parameter or node.

Once you adjust the link, you need to issue a **Process All FB Links** command to re-check the the function block links.

### Clear Faults Command

The system **Clear Faults** command initiates the **Process All FB Links** service. If no other illegal links are found, the system fault is cleared and drive enable is allowed.

## Fault Codes

The following table provides a description of the possible faults and warnings and the action that you need to take to clear the problem.

Fault Text	Code	Description	Action
FB Internal Err	24027	An internal function block error occurred.	Restart your system.
Invalid FB Link	24028	Your application contains an invalid function block link.	<ol style="list-style-type: none"> <li>1. Determine which node has the incorrect link reference.</li> <li>2. Clear the link or link the node to a valid source.</li> <li>3. Clear the fault queue.</li> </ol>
FB I/O Limit Err	24029	The system is out of dynamic RAM.	<ol style="list-style-type: none"> <li>1. Perform a function block <b>BRAM Init</b>.</li> <li>2. Remove extra event blocks to reduce the number of nodes.</li> </ol>
FB Mem Alloc Err	24030	A function block memory allocation error occurred.	Perform a function block <b>BRAM Init</b> .
FB BRAM Chsm Err	24034	The data in the BRAM has been corrupted and the checksums do not match.	<ol style="list-style-type: none"> <li>1. Perform a function block <b>BRAM Init</b>.</li> <li>2. Perform a function block <b>BRAM Store</b>.</li> <li>3. Download a new execution list.</li> </ol>
Init FB BRAM Flt	24037	The data in the BRAM has been corrupted.	Clear the fault queue.
FB Near Mem Lim	24044	The system is almost out of dynamic RAM.	You can either ignore this warning until the system runs out of dynamic RAM and you receive a FB I/O Limit Err, or perform a function block <b>BRAM Init</b> and remove extra event blocks.
FB DNLD Bad Evnt	24045	An attempt was made to download an execution list that contains a bad event.	Verify that all events have a valid ID number and that no numbers are duplicated.
FB Bad Pkt Num	24046	The service received a bad packet number. This occurs when the drive receives a packet that is outside of the range specified by packet 0.	Restart the download process.
FB Dnld Blk# Wrn	24049	An invalid block number was received.	<ol style="list-style-type: none"> <li>1. Check the data from the previous download process.</li> <li>2. Restart the download process.</li> </ol>
FB Dnld Cksm Wrn	24050	The execution list did not pass the checksum test.	<ul style="list-style-type: none"> <li>• Verify the execution list data.</li> <li>• Verify that the checksum was calculated correctly (if using a PLC).</li> </ul>
FB Near Exec Lim	24052	The application is close to the 20 millisecond task interval.	Check the execution time of the 20 millisecond function block task.













# We Want Our Manuals to be the Best!

You can help! Our manuals must meet the needs of you, the user. This is your opportunity to make sure they do just that. By filling out this form you can help us provide the most useful, thorough, and accurate manuals available. Please take a few minutes to tell us what you think - then mail this form or FAX it.

FAX: your local Allen-Bradley Sales Office or 414/512-8579

**PUBLICATION NAME** \_\_\_\_\_

**PUBLICATION NUMBER, DATE AND PART NUMBER (IF PRESENT)** \_\_\_\_\_

**✓ CHECK THE FUNCTION THAT MOST CLEARLY DESCRIBES YOUR JOB.**

- SUGGEST/RESPONSIBLE FOR THE PURCHASE OF EQUIPMENT
- DESIGN/IMPLEMENT ELECTRICAL SYSTEMS
- SUPERVISE FLOOR OPERATIONS
- MAINTAIN/OPERATE PROGRAMMABLE MACHINERY
- TRAIN/EDUCATE MACHINE USERS

**✓ WHAT LEVEL OF EXPERIENCE DO YOU HAVE WITH EACH OF THE FOLLOWING PRODUCTS?**

	NONE	LITTLE	MODERATE	EXTENSIVE
PROGRAMMABLE CONTROL	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
AC/DC DRIVES	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
PERSONAL COMPUTERS	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
NC/CNC CONTROLS	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
DATA COMMUNICATIONS/LAN	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

**✓ RATE THE OVERALL QUALITY OF THIS MANUAL BY CIRCLING YOUR RESPONSE BELOW. (1) = POOR (5) = EXCELLENT**

HELPFULNESS OF INDEX/TABLE OF CONTENTS	1	2	3	4	5
CLARITY	1	2	3	4	5
EASE OF USE	1	2	3	4	5
ACCURACY AND COMPLETENESS	1	2	3	4	5
QUALITY COMPARED TO OTHER COMPANIES' MANUALS	1	2	3	4	5
QUALITY COMPARED TO OTHER ALLEN-BRADLEY MANUALS	1	2	3	4	5

**✓ WHAT DID YOU LIKE MOST ABOUT THIS MANUAL?**

**✓ WHAT DID YOU LIKE LEAST ABOUT THIS MANUAL?**

**✓ PLEASE LIST ANY ERRORS YOU FOUND IN THIS MANUAL (REFERENCE PAGE, TABLE, OR FIGURE NUMBERS).**

**✓ DO YOU HAVE ANY ADDITIONAL COMMENTS?**

**✓ COMPLETE THE FOLLOWING.**

NAME \_\_\_\_\_ COMPANY \_\_\_\_\_

TITLE \_\_\_\_\_ DEPARTMENT \_\_\_\_\_

STREET \_\_\_\_\_ CITY \_\_\_\_\_ STATE \_\_\_\_\_ ZIP \_\_\_\_\_

TELEPHONE \_\_\_\_\_ DATE \_\_\_\_\_

CUT ALONG DOTTED LINE 

FOLD HERE

FOLD HERE

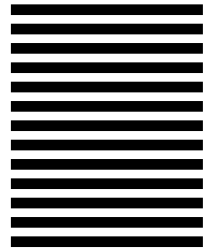


NO POSTAGE  
NECESSARY  
IF MAILED  
IN THE  
UNITED STATES

**BUSINESS REPLY MAIL**  
FIRST CLASS      PERMIT NO. 413      MEQUON, WI

POSTAGE WILL BE PAID BY ADDRESSEE

**ALLEN-BRADLEY**  
Attn: Marketing Communications  
P.O. Box 760  
Mequon, WI 53092-9907



PLC is a registered trademark of Allen-Bradley Company, Inc.

SCANport is a trademark of Allen-Bradley Company, Inc.

1336 FORCE and 1336 PLUS are trademarks of Allen-Bradley Company, Inc.



Allen-Bradley, a Rockwell Automation Business, has been helping its customers improve productivity and quality for more than 90 years. We design, manufacture and support a broad range of automation products worldwide. They include logic processors, power and motion control devices, operator interfaces, sensors and a variety of software. Rockwell is one of the world's leading technology companies.

### Worldwide representation.



Argentina • Australia • Austria • Bahrain • Belgium • Brazil • Bulgaria • Canada • Chile • China, PRC • Colombia • Costa Rica • Croatia • Cyprus • Czech Republic • Denmark • Ecuador • Egypt • El Salvador • Finland • France • Germany • Greece • Guatemala • Honduras • Hong Kong • Hungary • Iceland • India • Indonesia • Ireland • Israel • Italy • Jamaica • Japan • Jordan • Korea • Kuwait • Lebanon • Malaysia • Mexico • Netherlands • New Zealand • Norway • Pakistan • Peru • Philippines • Poland • Portugal • Puerto Rico • Qatar • Romania • Russia-CIS • Saudi Arabia • Singapore • Slovakia • Slovenia • South Africa, Republic • Spain • Sweden • Switzerland • Taiwan • Thailand • Turkey • United Arab Emirates • United Kingdom • United States • Uruguay • Venezuela • Yugoslavia

Allen-Bradley Headquarters, 1201 South Second Street, Milwaukee, WI 53204 USA, Tel: (1) 414 382-2000 Fax: (1) 414 382-4444