LANGUAGE REFERENCE GUIDE

NETLINX PROGRAMMING LANGUAGE
COPYRIGHT NOTICE
AMX© 2016, all rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of AMX. Copyright protection claimed extends to AMX hardware and software and includes all forms and matters copyrightable material and information now allowed by statutory or judicial law or herein after granted, including without limitation, material generated from the software programs which are displayed on the screen such as icons, screen display looks, etc. Reproduction or disassembly of embodied computer programs or algorithms is expressly prohibited.

LIABILITY NOTICE
No patent liability is assumed with respect to the use of information contained herein. While every precaution has been taken in the preparation of this publication, AMX assumes no responsibility for error or omissions. No liability is assumed for damages resulting from the use of the information contained herein. Further, this publication and features described herein are subject to change without notice.

AMX WARRANTY AND RETURN POLICY
The AMX Warranty and Return Policy and related documents can be viewed/downloaded at www.amx.com.
# Table of Contents

## NetLinx Programming Language

- Overview .................................................................................................................. 17
- Conventions Used in This Document ..................................................................... 17
- NetLinx Programming Overview ............................................................................ 17
- Mainline ................................................................................................................... 17
  - Understanding When DEFINE_PROGRAM Runs .................................................. 18
  - The Four Conditions That Cause the NetLinx Master To Run DEFINE_PROGRAM .. 18
  - Unhandled Events ............................................................................................... 18
  - Writing To a Variable: ....................................................................................... 19
  - The 1/sec Fail-Safe Timer .................................................................................. 19
  - The Empty Event Queue .................................................................................... 20
- Statements and Expressions .................................................................................... 20
  - Statements .......................................................................................................... 20
  - Expressions ......................................................................................................... 20
  - Assignments ....................................................................................................... 20
    - Variables .......................................................................................................... 20
    - Output channels ............................................................................................... 21
  - Comments .......................................................................................................... 21
- Identifiers ................................................................................................................. 22
  - Overview .......................................................................................................... 22
  - Devices .............................................................................................................. 22
  - Device Numbers - Supported Ranges by Device Type ........................................ 22
    - Master Device Number .................................................................................... 22
    - Physical Devices .............................................................................................. 22
    - Dynamically Assigned Devices ..................................................................... 23
    - Virtual Devices ............................................................................................... 23
  - Device Arrays ..................................................................................................... 23
    - Device Array Examples .................................................................................. 23
  - Device-Channels and Device-Channel Arrays .................................................... 24
- Device-Level Arrays ................................................................................................ 24
- Subroutines ............................................................................................................. 26
  - Overview .......................................................................................................... 26
    - DEFINE_CALL Subroutines .......................................................................... 26
    - SYSTEM_CALL Subroutines ......................................................................... 26
    - Function Subroutines ..................................................................................... 26
    - Calling Parameters ......................................................................................... 27
- Subroutine Keywords .............................................................................................. 28
  - CALL .................................................................................................................. 28
  - DEFINE_CALL ................................................................................................. 28
  - SYSTEM_CALL ................................................................................................. 28
# Compiler Directives

- **Overview**
  - `#DEFINE` .......................................................... 29
  - `#DISABLE_WARNING` .............................................. 29
  - `#ELSE` ................................................................. 29
  - `#END_IF` ............................................................... 29
  - `#IF_DEFINED` ....................................................... 29
  - `#IF_NOT_DEFINED` ................................................. 29
  - `#INCLUDE` .......................................................... 29
  - `#WARN` ................................................................. 29

## Array Keywords

- **Overview** .......................................................... 30

- **Multi-Dimensional Arrays** ..................................... 31

- **Array Keywords** ................................................... 32
  - `LENGTH_ARRAY` .................................................... 32
  - `MAX_LENGTH_ARRAY` .............................................. 32
  - `SET_LENGTH_ARRAY` .............................................. 32

## Audit Keywords

- **Overview** .......................................................... 33
  - `AUDIT_NETLINX_GENERIC_EVENT` ......................... 33
  - `AUDIT_NETLINX_SESSION_EVENT` ......................... 33

## Authentication Keywords

- **Overview** .......................................................... 34
  - `VALIDATE_NETLINX_ACCOUNT` ................................. 34
  - `VALIDATE_NETLINX_ACCOUNT_WITH_PERMISSION` ........ 34

## Buffer Keywords

- **Overview** .......................................................... 35
  - `CLEAR_BUFFER` .................................................... 35
  - `CREATE_BUFFER` .................................................... 35
  - `CREATE_MULTI_BUFFER` .......................................... 35
  - `GET_BUFFER_CHAR` ............................................... 36
  - `GET_BUFFER_STRING` ............................................. 36
  - `GET_MULTI_BUFFER_STRING` .................................... 36

## Channel Keywords

- **Overview** .......................................................... 37
  - `OFF` .................................................................. 37
  - `ON` .................................................................. 37
  - `TOTAL_OFF` ......................................................... 37

## Clock Manager Keywords

- **Overview** .......................................................... 38
  - `CLKMGR_SET_DAYLIGHTSAVINGS_OFFSET` ................. 38
  - `CLKMGR_SET_DAYLIGHTSAVINGS_MODE` ................... 39
  - `CLKMGR_SET_DAYLIGHTSAVINGS_RULE` .................... 39
  - `CLKMGR_SET_END_DAYLIGHTSAVINGS_RULE` ............. 39
  - `CLKMGR_SET_NETWORK_SOURCE` ............................... 38
  - `CLKMGR_SET_RESYNC_PERIOD` ............................... 38
  - `CLKMGR_SET_START_DAYLIGHTSAVINGS_RULE` .......... 39
  - `CLKMGR_SET_TIME_ZONE` ....................................... 38
Combine & Uncombine Keywords ................................................................................................. 40

Overview ........................................................................................................................................ 40

Combining and Un-Combining Devices ........................................................................................ 40

Combining Devices ............................................................................................................................ 40

Un-combining Devices ....................................................................................................................... 41

Combining and Un-Combining Levels ........................................................................................ 42

Combining Levels .............................................................................................................................. 42

Un-combining Levels ......................................................................................................................... 42

Combining and Un-Combining Channels ...................................................................................... 43

Combining Channels .......................................................................................................................... 43

Un-combining Channels ..................................................................................................................... 43

COMBINE & UNCOMBINE Keywords .......................................................................................... 46

COMBINE_CHANNELS ...................................................................................................................... 46

COMBINE_DEVICES .......................................................................................................................... 46

COMBINE_LEVELS ............................................................................................................................ 46

DEFINE_COMBINE ........................................................................................................................... 46

DEFINE_CONNECT_LEVEL .................................................................................................................. 46

UNCOMBINE_CHANNELS .................................................................................................................. 46

UNCOMBINE_DEVICES .................................................................................................................... 47

UNCOMBINE_LEVELS ....................................................................................................................... 47

Compiler Keywords ......................................................................................................................... 48

_DATE_ ............................................................................................................................................ 48

_FILE_ .............................................................................................................................................. 48

__LINE__ ........................................................................................................................................... 48

__DATE__ .......................................................................................................................................... 48

__NAME__ ........................................................................................................................................ 48

__TIME__ ......................................................................................................................................... 48

Conditional & Loop Keywords .......................................................................................................... 49

Overview ........................................................................................................................................... 49

Conditionals ....................................................................................................................................... 49

IF...ELSE ......................................................................................................................................... 49

SELECT...ACTIVE ............................................................................................................................ 49

SWITCH...CASE Statements ............................................................................................................ 50

Loops ................................................................................................................................................ 51

FOR Loops ........................................................................................................................................ 51

WHILE Loops .................................................................................................................................. 51

LONG_WHILE statements .................................................................................................................. 51

Conditional and Loop Keywords ..................................................................................................... 52

BREAK ............................................................................................................................................. 52

DEFAULT ........................................................................................................................................ 52

ELSE ............................................................................................................................................... 52

FOR.................................................................................................................................................. 52

IF.................................................................................................................................................... 52

IF...ELSE ....................................................................................................................................... 52

INCLUDE ........................................................................................................................................ 52

SELECT...ACTIVE ........................................................................................................................... 52

SWITCH...CASE ............................................................................................................................... 53

WHILE .......................................................................................................................................... 53

MEDIUM_WHILE .............................................................................................................................. 53

LONG_WHILE ................................................................................................................................. 53

FALSE ............................................................................................................................................. 53
### Table of Contents

**Data Event Keywords** ........................................................................................................... 54
- AWAKE ........................................................................................................................................ 54
- COMMAND ................................................................................................................................. 54
- HOLD ......................................................................................................................................... 54
- ONERROR ................................................................................................................................. 54
- OFFLINE .................................................................................................................................. 54
- ONLINE ..................................................................................................................................... 54
- REPEAT ...................................................................................................................................... 54
- STANDBY ................................................................................................................................. 54

**Data Types and Conversion Keywords** .................................................................................. 55
- Overview ..................................................................................................................................... 55
- Intrinsic Data Types ................................................................................................................ 55
- Intrinsic Data Type Keywords ................................................................................................ 55
  - CHAR ......................................................................................................................................... 55
  - WIDECHAR ............................................................................................................................ 55
  - INTEGER ............................................................................................................................... 55
  - SINTEGER .............................................................................................................................. 55
  - LONG ....................................................................................................................................... 55
  - SLONG ..................................................................................................................................... 55
- Structured Data Types .............................................................................................................. 56
  - DEV ......................................................................................................................................... 56
  - DEVCHAN .............................................................................................................................. 56
  - DEVLEV ................................................................................................................................... 56
- Combining and Uncombining Device/Channel Sets .................................................................. 56
  - FLOAT ....................................................................................................................................... 56
  - DOUBLE .................................................................................................................................. 56
- Type Conversion ...................................................................................................................... 57
- Type Conversion Rules ............................................................................................................. 57
- Conversion Keywords .............................................................................................................. 57
  - ATOI ......................................................................................................................................... 57
  - ATOF ......................................................................................................................................... 57
  - ATOL ......................................................................................................................................... 58
  - CH_TO_WC ............................................................................................................................. 58
  - FTOA ......................................................................................................................................... 58
  - HEXTOI ..................................................................................................................................... 58
  - ITOA ......................................................................................................................................... 58
  - FORMAT .................................................................................................................................... 59
  - ITOHEX ..................................................................................................................................... 59
  - RAW_BE ................................................................................................................................... 59
  - RAW LE .................................................................................................................................... 59

**DEFINE Keywords** .............................................................................................................. 60
- Overview ..................................................................................................................................... 60
  - DEFINE_CALL ....................................................................................................................... 60
  - DEFINE_FUNCTION .............................................................................................................. 60
  - DEFINE_CONSTANT ............................................................................................................... 60
- DEFINE Keywords ................................................................................................................... 61
  - DEFINE_CALL ....................................................................................................................... 61
  - DEFINE_COMBINE .............................................................................................................. 61
  - DEFINE_CONNECT_LEVEL ................................................................................................. 61
  - DEFINE_CONSTANT ............................................................................................................. 61
  - DEFINE DEVICE .................................................................................................................. 61
  - DEFINE EVENT .................................................................................................................... 61
  - DEFINE FUNCTION ............................................................................................................. 62
DEFINE_LATCHING ......................................................................................................................... 62
DEFINE_MODULE ............................................................................................................................... 62
DEFINE_MUTUALLY_EXCLUSIVE ...................................................................................................... 62
DEFINE_PROGRAM ............................................................................................................................ 62
DEFINE_START ................................................................................................................................. 62
DEFINE_TOGGLE ............................................................................................................................... 62
DEFINE_TYPE ..................................................................................................................................... 62
DEFINE_MUTUALLY_EXCLUSIVE and Variables .............................................................................. 63
DEFINE_VARIABLE ............................................................................................................................ 63
PROGRAM_NAME .................................................................................................................................. 63
RETURN .................................................................................................................................................. 63

DEVICE Keywords ............................................................................................................................ 64
DEVICE_ID ................................................................................................................................................ 64
DEVICE_ID_STRING ................................................................................................................................ 64
DEVICE_INFO ......................................................................................................................................... 64
DEVICE_STANDBY ................................................................................................................................... 65
DEVICE_WAKE ......................................................................................................................................... 65
DYNAMIC_APPLICATION_DEVICE ......................................................................................................... 65
MASTER_SLOT .......................................................................................................................................... 65
PUSH_DEVICE ......................................................................................................................................... 65
RELEASE_DEVICE ............................................................................................................................... 65
PUSH_DEVCHAN ..................................................................................................................................... 65
RELEASE_DEVCHAN ........................................................................................................................... 65
REBOOT .................................................................................................................................................. 65
SEND_COMMAND .................................................................................................................................... 65
SYSTEM_NUMBER ..................................................................................................................................... 65

Encode / Decode Keywords ............................................................................................................... 66
Overview - Encoding and Decoding Binary and XML ........................................................................ 66
Encode / Decode Keywords ................................................................................................................. 70
STRING_TO_VARIABLE (VARIABLE DECODE) ..................................................................................... 70
VARIABLE_TO_STRING (VARIABLE ENCODE) .................................................................................... 70
LENGTH_VARIABLE_TO_STRING (VARIABLE Encode) ......................................................................... 70
LENGTH_VARIABLE_TO_XML .................................................................................................................. 71
VARIABLE_TO_XML ............................................................................................................................... 71
XML_TO_VARIABLE ................................................................................................................................ 72

Event Handler Keywords .................................................................................................................... 74
Overview ................................................................................................................................................ 74
Button Events .......................................................................................................................................... 74
Channel Events ....................................................................................................................................... 75
Data Events ............................................................................................................................................. 76
Level Events ........................................................................................................................................... 76
Custom Events ....................................................................................................................................... 79
Event Parameters ................................................................................................................................... 79
Event Handler Keywords .................................................................................................................... 81
BUTTON_EVENT ...................................................................................................................................... 81
CHANNEL_EVENT ................................................................................................................................. 81
DATA_EVENT ........................................................................................................................................... 82
LEVEL_EVENT ......................................................................................................................................... 82
REBUILD_EVENT() ................................................................................................................................... 82

File Operation Keywords .................................................................................................................... 84
FILE_CLOSE ............................................................................................................................................... 84
FILE_COPY ............................................................................................................................................... 84
FILE_CREATEDIR ................................................................................................................................. 84
FILE_DELETE ......................................................................................................................................... 85
FILE_DIR .................................................................................................................................................. 85
Get Keywords

Overview - IP Communication

Client Programming

Initiating a conversation

Terminating a conversation

Sending data

Receiving data

Server Programming

Listening for client requests

Multiple client connections

Closing a local port

Connection-Oriented notifications

Receiving data

Sending data

Receiving Data with UDP

Multicast

Example IP Code

IP Keywords

ADD_URL_ENTRY

DELETE_URL_ENTRY

GET_DNS_LIST

GET_IP_ADDRESS

IP_BOUND_CLIENT_OPEN

IP_CLIENT_CLOSE

IP_CLIENT_OPEN

IP_MC_SERVER_OPEN

IP_SERVER_CLOSE

IP_SERVER_OPEN

IP_SET_OPTION
## Table of Contents

- **Bitwise Operators** ................................................................. 117
- **Assignment Operators** .......................................................... 117
- **Operator Precedence** ............................................................. 118
- **Operator Keywords** ............................................................... 118
  - AND (&&) ..................................................................................... 118
  - BAND (&) .................................................................................... 118
  - BNOT (~) ..................................................................................... 118
  - BOR (||) ....................................................................................... 118
  - BXOR (^) ..................................................................................... 118
  - LSHIFT .......................................................................................... 118
  - MOD (%) ...................................................................................... 118
  - NOT (!) ......................................................................................... 118
  - OR (||) .......................................................................................... 118
  - RSHIFT .......................................................................................... 118
  - XOR (^^) ..................................................................................... 118
- **Port Keywords** ......................................................................... 119
  - DYNAMIC_POLLED_PORT .......................................................... 119
  - FIRST_LOCAL_PORT ..................................................................... 119
  - STATIC_PORT_BINDING .............................................................. 119
- **Push and Release Keywords** .................................................. 120
  - DO_PUSH ..................................................................................... 120
  - DO_PUSH_TIMED ......................................................................... 120
  - DO_RELEASE ............................................................................... 120
  - MIN_TO ........................................................................................ 120
  - PUSH ............................................................................................. 120
  - PUSH_CHANNEL .......................................................................... 120
  - PUSH_DEVCHAN .......................................................................... 120
  - PUSH_DEVICE .............................................................................. 120
  - RELEASE ........................................................................................ 120
  - RELEASE_CHANNEL ..................................................................... 120
  - RELEASE_DEVCHAN .................................................................... 120
  - RELEASE_DEVICE ........................................................................ 120
  - TO ................................................................................................. 120
- **SET Keywords** .......................................................................... 122
  - SET_DNS_LIST ............................................................................. 122
  - SET_IP_ADDRESS .......................................................................... 122
  - SET_LENGTH_ARRAY .................................................................... 122
  - SET_LENGTH_STRING ................................................................. 122
  - SET_OUTDOOR_TEMPERATURE ................................................... 122
  - SET_PULSE_TIME ......................................................................... 122
  - PULSE ........................................................................................... 122
  - SET_SYSTEM_NUMBER .................................................................. 122
  - SET_TIMER ................................................................................... 122
  - SET_VIRTUAL_CHANNEL_COUNT ................................................. 122
  - SET_VIRTUAL_LEVEL_COUNT ....................................................... 122
  - SET_VIRTUAL_PORT_COUNT ......................................................... 122
- **SMTP Keywords** ....................................................................... 123
  - Overview ..................................................................................... 123
    - SMTP_SERVER_CONFIG_SET ....................................................... 123
    - SMTP_SERVER_CONFIG_GET ..................................................... 123
    - SMTP_SEND ............................................................................... 123
- **String Keywords** ...................................................................... 124
  - Overview ..................................................................................... 124
    - String Expressions ..................................................................... 124
    - Wide Strings .............................................................................. 124
### STRING Keywords
- CHARD.................................................. 125
- CHARDM........................................... 125
- COMPARATOR_STRING.......................... 125
- FIND_STRING.................................... 125
- LEFT_STRING.................................... 125
- LENGTH_STRING................................ 126
- LOWER_STRING................................ 126
- MAX_LENGTH_STRING......................... 126
- MID_STRING...................................... 126
- REDIRECT_STRING.............................. 126
- REMOVE_STRING............................... 126
- RIGHT_STRING................................... 126
- SEND_STRING.................................... 127
- SET_LENGTH_STRING......................... 127
- STRING........................................... 127
- STRING_TO_VARIABLE....................... 127
- UPPER_STRING.................................. 127
- VARIABLE_TO_STRING....................... 127

### Structure Keywords
- Overview........................................ 128
  - Example - Using Structures to Define a Database Table....................... 128
- Data Sets........................................ 129
- STRUCTURE Keywords....................... 130
  - DEFINE_TYPE................................. 130
  - STRUCT.......................................... 130
  - STRUCTURE.................................... 130

### Terminal Keywords
- SSH_CLIENT_CLOSE............................ 131
- SSH_CLIENT_OPEN............................. 131

### Time and Date Keywords
- ASTRO_CLOCK.................................. 132
- CLOCK........................................... 132
- DATE............................................. 132
- DATE_TO_DAY................................... 132
- DATE_TO_MONTH............................... 132
- DATE_TO_YEAR................................. 132
- DAY................................................ 132
- DAY_OF_WEEK.................................. 132
- LDATE............................................ 132
- TIME............................................. 133
- TIME_TO_HOUR................................. 133
- TIME_TO_MINUTE............................... 133
- TIME_TO_SECOND............................. 133

### Timeline Keywords
- Overview........................................ 134
  - Creating a Timeline.......................... 134
  - TIMELINE Example........................... 136
- TIMELINE IDs.................................... 138
  - TIMELINE_ACTIVE........................... 138
  - TIMELINE_CREATE......................... 139
  - TIMELINE_EVENT............................ 139
  - TIMELINE_GET............................... 139
  - TIMELINE_KILL............................... 139
  - TIMELINE_PAUSE............................. 139
  - TIMELINE_RELOAD......................... 139
Appendix A - Compiler Warning & Errors ...................................................... 156

Compiler Warnings ...................................................................................................................... 156

(w) Cannot assign unlike types ................................................................................................ 156
(w) DEFINE_CALL is not used ................................................................................................ 156
(w) Integer applies to arrays only .............................................................................................. 156
(w) Long_While within While .................................................................................................... 156
(w) Possibly too many nested levels .......................................................................................... 156
(w) Variable is not used ............................................................................................................. 156

Compiler Errors ......................................................................................................................... 156

A "<symbol>" was expected ......................................................................................................... 156
ACTIVE keyword expected ....................................................................................................... 156
Allowed only in DEFINE_START ............................................................................................... 156
Attempted CALL to undefined subroutine ................................................................................ 156
Comment never ends, EOF encountered ................................................................................. 156
Conditional compile nesting too deep ....................................................................................... 156
Constant type not allowed ........................................................................................................ 156
DEFINE_CALL must have a name ............................................................................................. 156
DEFINE_CALL name already used ........................................................................................... 157

Device values must be equal ................................................................. 157
Duplicate symbol ............................................................................. 157
Evaluation stack overflow ................................................................. 157
Evaluation stack underflow ............................................................. 157
Identifier expected .......................................................................... 157
Identifier is not an array type ......................................................... 157
Include file not found ....................................................................... 157
Invalid include file name ............................................................... 157
Library file not found ....................................................................... 157
Maximum string length exceeded .................................................. 157
Must be char array reference ......................................................... 157
Must be integer reference .............................................................. 157
Out of memory ................................................................................ 157
Parameter mismatch in CALL ......................................................... 157
PROGRAM_NAME must be on line 1 ............................................... 157
Push/Release not allowed within Push/Release ............................. 157
Push/Release not allowed within Wait .......................................... 157
PUSH_CHANNEL not allowed within Wait .................................... 157
RELEASE_CHANNEL not allowed within Wait ............................. 157
PUSH_DEVICE not allowed within Wait ......................................... 157
RELEASE_DEVICE not allowed within Wait ................................. 157
String constant expected ............................................................... 157
String constant never ends, EOF encountered ............................ 157
String literal expected ................................................................... 157
Subroutine may not call itself ....................................................... 157
Syntax error .................................................................................... 157
SYSTEM_CALL name not same as PROGRAM_NAME in <file> .... 157
This variable type not allowed ....................................................... 157

Run-Time Errors ............................................................................. 158
Bad assign 2dim............................................................................ 158
Bad assign Call............................................................................. 158
Bad element assign....................................................................... 158
Bad Off... Bad On... Bad To ......................................................... 158
Bad re-assign Call ....................................................................... 158
Bad run token ............................................................................. 158
Bad Set Length ........................................................................... 158
Bad While .................................................................................... 158
TO statements that occur outside the data flow of PUSH events/statements may not work........ 158
Too few parameters in CALL ....................................................... 158
Too many include files ............................................................... 158
Too many parameters in CALL ................................................... 158
Type mismatch in function CALL ........................................... 158
Undefined identifier ....................................................................... 158
Unmatched #END IF ................................................................ 158
Unrecognized character in input file .......................................... 158
Use SYSTEM_CALL [instance] 'name' ...................................... 158
Variable assignment not allowed here ...................................... 158
Wait not found ........................................................................... 158

Appendix B - Master-To-Master (M2M) ....................................... 159

Overview .................................................................................... 159

Master-to-Master ........................................................................ 159

Master Routing ........................................................................... 159

Route Modes (Normal and Direct) .................................................. 160

Design Considerations, Constraints, and Topologies ................. 161

Design Considerations ................................................................ 161
Constraints ................................................................................ 162

Chain Topology .......................................................................... 162

Star Topology ........................................................................... 162

Cluster Topology ........................................................................ 163
Appendix C - Marshalling Protocol ................................................................. 169

Overview ........................................................................................................ 169

Marshalled Stream Format .............................................................................. 169

Marshalling Protocol (Variables) .................................................................... 170

Encoding Notes .............................................................................................. 171

String Encoding ............................................................................................. 171

Binary Array Encoding .................................................................................. 172

Binary Encoding Result .................................................................................. 172

XML Encoding Result ..................................................................................... 174
# Appendix D - NetLinx vs. Axcess

## Overview

### NetLinx vs. Axcess - Comparison by Structure

- `DEFINE_DEVICE` .................................................................................................................. 175
- `DEFINE_CONSTANT` .............................................................................................................. 175
- `DEFINE_VARIABLES` ........................................................................................................... 176
- `DEFINE_CALL` (Subroutines) ............................................................................................. 176
- `DEFINE_START` .................................................................................................................. 176
- `DEFINE_EVENT` ................................................................................................................... 177
- `DEFINE_PROGRAM` .............................................................................................................. 177

### Axcess/NetLinx Incompatibility

### Combining Devices, Channels and Levels

- Virtual devices, levels and device/channel sets........................................................................ 178
- Combining and Uncombining devices .................................................................................... 178
- Combining and Uncombining levels ....................................................................................... 179
- Combining and Uncombining channels.................................................................................. 179

### String Comparisons

- Axcess code - string comparison .......................................................................................... 179
- NetLinx code - string comparison ....................................................................................... 179

### Modules


Overview

NetLinx® is a superset of the Axcess language with extensions for additional data types, new event handlers, structure support, multi-dimensional arrays, and other features.

This document assumes that you are familiar with Axcess; the focus is on the new language elements and how they extend the functionality of the existing language. For background information on Axcess, refer to the Axcess Programming Language instruction manual.

Conventions Used in This Document

NetLinx contains a number of keywords that define various available operations to perform in a NetLinx command, such as the word CALL in the statement:

```
CALL 'Read Data' (Buffer)
```

Keywords are case insensitive. For example, the PUSH command is the same as push. Keywords are reserved, meaning that identifiers (device names, constants, or variables) must have unique names. These keywords are listed and defined in this document, separated by category.

- Square brackets indicate an optional element in a command.
- Angle brackets indicate substitution.

In the example below, the notation <return type> indicates that a valid data type (such as CHAR, INTEGER, or FLOAT) must be substituted for <return type>.

```
DEFINE_FUNCTION [<return type>] <name> [(Param1, Param2, ...)]
{
    (* body of subroutine *)
}
```

NetLinx Programming Overview

The NetLinx control system was designed to upgrade the processor bus and improve the power of the Axcess programming language. Originally named Axcess2, the NetLinx was designed to be a superset of the Axcess programming language. The relationship between the new language (NetLinx) and Axcess is very similar to the relationship between C++ and C.

Just as C++ brought a whole new level of power to C programming, NetLinx offers a variety of new tools and commands to dynamically increase the speed and power of present and future applications.

**NOTE:** Use the NetLinx Studio software program to create, compile, and transfer Axcess/NetLinx code.

Mainline

Mainline is the program section executed continuously by the NetLinx Central Controller as long as the Controller has power. DEFINE_PROGRAM contains the code known as mainline.

A typical NetLinx program is composed of a number of different sections. Each section defines some aspect of a program such as device definitions, variable declarations, channel characteristics, or event processing. The sections that can comprise a NetLinx program are listed in the following table:

<table>
<thead>
<tr>
<th>Program Sections</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFINE DEVICE</td>
</tr>
<tr>
<td>DEFINE_MUTUALLY_EXCLUSIVE</td>
</tr>
<tr>
<td>DEFINE_COMBINE</td>
</tr>
<tr>
<td>DEFINE_TOGGLING</td>
</tr>
<tr>
<td>DEFINE_CONSTANT</td>
</tr>
<tr>
<td>DEFINE_CALL</td>
</tr>
<tr>
<td>DEFINE_TYPE</td>
</tr>
<tr>
<td>DEFINE_FUNCTION</td>
</tr>
<tr>
<td>DEFINE_VARIABLE</td>
</tr>
<tr>
<td>DEFINE_CONNECT_LEVEL</td>
</tr>
<tr>
<td>DEFINE_LATCHING</td>
</tr>
<tr>
<td>DEFINE_LATCHING</td>
</tr>
<tr>
<td>DEFINE_PROGRAM</td>
</tr>
</tbody>
</table>

Not all of the sections listed above are required to create a complete program. In an Axcess system, only DEFINE_PROGRAM is required. In a NetLinx system, either DEFINE_PROGRAM or DEFINE_EVENT is required. Other sections are required only to support code in one of these two sections, although the compiler might require more.

Axcess communication updates occur only between passes through mainline (or after each iteration through LONG_WHILE loops). This places timing constraints on mainline processing in order for the system to operate properly. NetLinx avoids these constraints by processing network activity through a separate thread of execution. Bus activity is serviced concurrently with event processing and mainline execution. The event processing that previously could occur only through mainline code can now be handled through code in the DEFINE_EVENT section. This provides a more efficient mechanism for processing events; mainline does not have to be traversed to process a single I/O request. A handler can be defined for processing device-specific events, as well as providing feedback for the device initiating the event notification. If a handler is present, mainline will not be called to process the event; the handler is called instead. Once the handler completes its execution, the system is ready to process the next input message. When no more messages are pending, mainline runs.
In effect, mainline in NetLinx is an *idle time* process. With the addition of the DEFINE_EVENT section for processing events, the role of mainline in a NetLinx program becomes greatly diminished if not totally eliminated. Programs can still be written using the traditional technique of processing events and providing feedback through mainline code. However, programs written using the event table structure, provided in the NetLinx system, will run faster and be easier to maintain. FIG. 1 illustrates message and mainline processing as it appears in the NetLinx system. Note that bus servicing is taken care of by a separate process thread (Connection Manager & Message Dispatcher) and, therefore, is not a task that must follow mainline.

**FIG. 1** Message and Mainline Processing in the NetLinx System

**Understanding When DEFINE_PROGRAM Runs**
This section describes the scenarios in which DEFINE_PROGRAM runs (or, why loops in mainline are bad). Understanding this process can explain programs with abnormally high CPU usage and how to fix them.

**Summary:**
- Use STACK_VAR whenever possible.
- Use a short WAIT in DEFINE_PROGRAM when it is not.

**The Four Conditions That Cause the NetLinx Master To Run DEFINE_PROGRAM**

1. An unhandled event occurs
2. A variable is written (this is the CPU usage culprit)
3. The ‘run occasionally anyway’ timer fires (~1/second)
4. The event queue has become empty

**Unhandled Events**
- DEFINE_PROGRAM runs when an unhandled event occurs, which ensures that channel- or level-based feedback is up to date.
- It also aids backwards compatibility by allowing SYSTEM_CALLs and mainline PUSH and RELEASE statements to run.

To understand unhandled events, consider the following code:

```c
BUTTON_EVENT[dvTP,123]
{
  PUSH:
    PULSE[dvRelay,1]
}
```

When someone presses button 123, there are 3 unhandled events and 1 handled event. The button press has been handled by PUSH, but the RELEASE, channel ON, and eventual channel OFF are not handled. The result is that DEFINE_PROGRAM runs far more often than the button push in DEFINE_EVENT. Normally, this is not a large concern. (A user can only poke the system so fast).

However, if want to, you can prevent this by adding empty BUTTON_EVENTS and CHANNEL_EVENTS, like this:

```c
BUTTON_EVENT[dvTP,0]
{
  PUSH:()
  RELEASE:()
}

CHANNEL_EVENT[dvRelay,0]
{
  ON:()
  OFF:()
}
```

Now, all channel-related events will be handled and DEFINE_PROGRAM will not run for these events.
Writing To a Variable:
The second condition (a variable being written to) is the culprit for abnormally high CPU usage. The intent behind this trigger for DEFINE_PROGRAM is to more accurately display feedback in a timely fashion. Since many people use DEFINE_PROGRAM to set button feedback with statements like:

```plaintext
[dvTP,201] = (nCurrentInput == 1)
```

...it makes sense to run DEFINE_PROGRAM if any change is detected in the states of any variable in the program. Normally, this is a very beneficial process. The problem comes with using loops to set feedback.

This code will cause high CPU usage:

```plaintext
DEFINE_VARIABLE
VOLATILE INTEGER INC
DEFINE_PROGRAM
FOR(INC=1;INC<=8;INC++)
{
   [dvTP,200+INC] = (nCurrentInput == INC)
}
```

It is not the loop itself that is the problem. It is the global variable INC being incremented that causes the issue. Since we’ve written to a variable, DEFINE_PROGRAM will want to run again. If there are no other events waiting in the queue, it will do so immediately. Of course, when it runs again, it will set itself up to run yet again.

The net result is that any time the processor would normally spend in an idle state is now consumed by repeatedly running DEFINE_PROGRAM. This does not interfere with processing events that come in, as they are given priority. The only speed penalty that is incurred is that the next incoming event can only be processed when the current pass of DEFINE_PROGRAM is finished. If you have an exceedingly long DEFINE_PROGRAM, it will slow event processing down.

The usual message conveyed with loops in mainline is “DON’T”, but the code above can be fixed quite easily. All we need is a variable that won’t be around when DEFINE_PROGRAM ends.

```plaintext
DEFINE_PROGRAM
{
   INTEGER INC
   FOR(INC=1;INC<=8;INC++)
   {
      [dvTP,200+INC] = (nCurrentInput == INC)
   }
}
```

There are three things to note here:

- We can arbitrarily ‘compound’ statements by placing them in braces
- Compounding allows us to define a local scope variable (they must be the first thing in a compound statement, before any executable code)
- The default local scope variable behavior is STACK_VAR, which is released once you leave that block of code

Since the variable is destroyed upon exiting the code, no variables are left in the dirty state. No dirty variables means no reason to run DEFINE_PROGRAM.

If we were to use a LOCAL_VAR instead, we would be back in a high CPU usage state as a LOCAL_VAR is non-volatile. It keeps its value between uses and is still around and ‘written to’ once DEFINE_PROGRAM is done. If you simply must use a LOCAL_VAR or a global-scope variable, there is still a way to salvage most of the CPU usage. If you employ a WAIT, you can control how often the feedback runs.

```plaintext
DEFINE_PROGRAM
{
   LOCAL_VAR INTEGER INC
   FOR(INC=1;INC<=8;INC++)
   {
      [dvTP,200+INC] = (nCurrentInput == INC)
   }
}
```

Now, no matter how often DEFINE_PROGRAM is compelled to run, the feedback will only run 10 times per second. A particular WAIT in your NetLinx code can only be put in the WAIT list once at any given time. This makes it a great choice for periodic functionality.

The 1/sec Fail-Safe Timer
To make sure that any feedback statements in DEFINE_PROGRAM are enforced eventually, there is a timer that fires every second that compels DEFINE_PROGRAM to run. This is given priority over event processing.

The 1/second mode can be proven easily. Just write this as the only line in the program:

```plaintext
DEFINE_PROGRAM
SEND_STRING 0,'DEFINE_PROGRAM JUST RAN'
```

Turn on NetLinx Internal Diagnostics Messages in NetLinx Studio and you will find that it occurs roughly once per second.

The 1/second fallback can cause one very large problem in one very specific situation. If you manage to write a DEFINE_PROGRAM section that takes more than one whole second to run (&gt; 400,000,000 machine instructions on a current master) then you can actually stop processing any events.
The event queue servicing becomes starved. When the 'run anyway' timer expires, it has the highest priority of any of the triggers. If it fires before finishing the last 'run anyway' DEFINE_PROGRAM run, it will simply run again. If this happens every run of DEFINE_PROGRAM, no events will be processed and the master will appear to be locked up.

In practice, DEFINE_PROGRAM should never run this long. If you are in a situation where you must process this much information, you should consider making one iteration of a loop with each pass of DEFINE_PROGRAM. Instead of this:

```
DEFINE_PROGRAM
FOR(INC=1; INC<=4000000000; INC++)
{
    // DO SOMETHING AWFUL WITH THE UNSIGNED LONG INC
}
```

Do this:

```
DEFINE_PROGRAM
INC = INC MOD 4000000000 // FORCE THE RANGE OF 0-3999999999
INC++                     // THEN ADD ONE
/// DO THE SAME HORRIBLE THING, BUT ONLY 1/PASS OF DEFINE_PROGRAM
```

Of course, if you have reached the point where DEFINE_PROGRAM takes longer than a second to run, you are past the point of needing another master on the job or re-evaluating your approach of the problem.

The Empty Event Queue
The final reason that DEFINE_PROGRAM will be run is when all the events that have come in have been processed. There are two reasons we should not care about this:

1. When it occurs, our system is, by definition, not busy
2. In busier systems, this occurs with decreasing frequency.

Statements and Expressions

Statements
A statement refers to a complete programming instructions such as:

- `Y = X` /* Variable Assignment Statement */
- `X = X + 1` /* Arithmetic Assignment Statement */
- `IF (Y < 10) Y = Y + 1` /* IF Statement */
- `[TP, 5] = [VCR, 1]` /* Feedback Statement */

Each of these statements compile, providing the referenced variables are defined.

Expressions
Expressions are sub-components of statements. The following expressions are used in the above example:

- `X + 1` /* Arithmetic Expression */
- `Y < 10` /* Logical Expression */
- `Y + 1` /* Arithmetic Expression */
- `[TP, 5]` /* I/O Device Expression */
- `[VCR, 1]` /* I/O Device Expression */

Expressions will not compile outside the context of a statement.
- It is strongly recommended that each statement appear on a separate line. The compiler cannot enforce this since full backward compatibility with the previous Axcess language must be maintained.
- It is also strongly recommended that semicolons be used to terminate each statement (as in the C language).

Assignments
Assignment statements include:
- Variables
- Output Channels

Variables
The simplest type of assignment statement is a variable, which assigns the value of an expression to a variable. The expression may be a constant, a variable / mathematical / logical expression, or a return from function. The data type associated with the expression should match the data type of the variable receiving the assignment. If not, the value of the expression is typecast to match the destination variable.

Example:

```
VariableName = <expression>
```

Output channels
This type of statement is typically used for feedback. It sends an output change to the specified channel on the given device.

Example:

```
[Device, Channel] = <expression>
```
The expression is evaluated as follows:
- If it is non-zero, the channel associated with the device is turned on.
- If it is zero, the channel is turned off.

### Comments

Comments are designated with a parentheses-asterisk to begin the comment and asterisk-parentheses to end the comment; for example, (*COMMENT*). These comments can span lines and are not limited in length.

NetLinx supports a second type of comment with a double forward-slash (`//`). All text following the double forward-slash is treated as a comment. This type of comment closely follows the conventions of C++.

Comments are not part of the actual program code; they are not compiled. Comments can appear anywhere except within literal strings, either on the same line as a programming statement or on a separate line. Comments can span multiple lines with a single set of comment delimiters and can be nested. The compiler recognizes nested comments by pairing up sets of comment delimiters. For example:

(* The section to follow contains all variable declarations. *)

Single line comments can be specified using the double forward slash (`//`) notation.

When a pair of forward slash characters is encountered, all text on the same line following the slash pair, except the `*`) end comment sequence, is considered a comment and ignored by the compiler. For example:

(*INTEGER Vol1  // volume for room 1 *)

The `*`) in the line above terminates the open `(*` command even though it appears after a double slash comment command.
Identifiers

Overview

An Identifier is a combination of letters, numbers, or underscores that represents a device, constant, or variable. Identifier types include:

- Devices
- Device Arrays
- Channel Arrays
- Device-Channel Arrays
- Level Arrays
- Device-Level Arrays

Devices

A Device is any hardware component that can be connected to the NetLinx bus. Each device must be assigned a unique number to identify it on the bus.

- NetLinx allows device numbers in the range 0-32767.
- Device 0 refers to the Master; numbers above 32767 are reserved for internal use.

NetLinx requires a Device:Port:System (D:P:S) specification where Access expected only a device number. This D:P:S triplet can be expressed as a series of constants, variables separated by colons, or a DEV structure. For example:

```c
STRUCTURE DEV
{
    INTEGER Number       // Device number
    INTEGER Port         // Port on device
    INTEGER System       // System device belongs to
}
```

A device specification in NetLinx can be expressed in one of two ways:

- Device Number: The compiler replaces the device number with an internally generated DEV structure. This DEV structure contains the specified device Number. If the system and port specifications are omitted (e.g. 128), System 0 (indicating this system - the system executing the code), and Port 1 (indicating the first port), is assumed.
- Device:Port:System (D:P:S): This notation is used to explicitly represent a device number, port, and system. For example, 128:1:0 represents the first port of the device number 128 on this system.

The syntax:

```
NUMBER:PORT:SYSTEM
```

Parameters:

- **Number**: 16-bit integer representing the Device number
  - Physical devices range from 1 to 32,000
  - Virtual devices range from 32,768 to 36,863
- **Port**: 16-bit integer representing the Port number, in the range 1 through the number of ports on the device (1 = this port)
- **System**: 16-bit integer representing the System number (0 = this system).

Device Numbers - Supported Ranges by Device Type

Each device requires a device number within the network, but many devices have range limitations on the device number that may be used. If an incorrect device number outside of that range is assigned to a particular device, the module may not function properly.

Master Device Number

The device number for the Master on a network must always be 0.

Physical Devices

Physical devices may be assigned a device number between 1 and 32000, with the exception of the examples in the table below:

<table>
<thead>
<tr>
<th>Physical Device Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-32000</td>
</tr>
<tr>
<td>1-255</td>
</tr>
<tr>
<td>5001</td>
</tr>
<tr>
<td>5002</td>
</tr>
<tr>
<td>6001-6999</td>
</tr>
<tr>
<td>10001-32000</td>
</tr>
</tbody>
</table>
**Dynamically Assigned Devices**

Device numbers dynamically assigned by the network are limited in range:

<table>
<thead>
<tr>
<th>Dynamically Assigned Device Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>32001-32767</td>
</tr>
</tbody>
</table>

**Virtual Devices**

Virtual devices must be assigned within a range of 32768 to 42000, with specific ranges for virtual device subcategories:

<table>
<thead>
<tr>
<th>Virtual Device Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>32768-42000</td>
</tr>
<tr>
<td>32768-36864</td>
</tr>
<tr>
<td>36865-37864</td>
</tr>
<tr>
<td>37865-40999</td>
</tr>
<tr>
<td>41001-42000</td>
</tr>
<tr>
<td>45001-45999</td>
</tr>
<tr>
<td>46001-46999</td>
</tr>
</tbody>
</table>

**Device Arrays**

In order to specify a group of devices for a command or event handler, NetLinx provides the capability to define an array of DEVs and treat it as a device array. A device array may be used anywhere a device specification is required. The result provides a range of targets for the command or instruction where it is used. Device arrays are declared in the DEFINE_VARIABLE section of the program in one of two ways:

```
DEV DSName[ ]          = {Dev1, Dev2, ..., Devn}
DEV DSName[MaxLen]     = {Dev1, Dev2, ..., Devn}
```

Each device name appearing on the right-hand side of the declaration should be defined as a device in the DEFINE_DEVICE section; however, it can also be defined in the DEFINE_VARIABLE or DEFINE_CONSTANT section.

The first statement above declares a device array whose maximum length is determined by the number of elements in the initialization array on the right-hand side. The second form uses `MaxLen` to specify the maximum length of the device array. In either case, the number of elements in the initialization array determines the effective length of the device array. That value can be determined at run-time by calling LENGTH_ARRAY. The maximum length available for a device array can be determined by calling MAX_LENGTH_ARRAY.

The following program fragment illustrates device array initialization:

```
DEFINE_DEVICE
panel3 = 130
DEFINE_CONSTANT
dev
panel1 = 128:1:0
integer panel2 = 129

DEFINE_VARIABLE
// dvs is an array of three devices:
// 128:1:0
// 129:1:0
// 130:1:0
DEV dvs[ ] = {panel1, panel2, panel3}
```

The individual elements of a device array can be referenced by their defined names (Dev1, Dev2, etc.) or by using array notation with the device array name. For example, the 3rd device in the device array, MyDeviceSet, would be referenced by MyDeviceSet[3].

The index of the last member of the array for which an event notification was received can be determined by calling GET_LAST(MydeviceSet). This is useful for determining which device in an array is referenced in a particular notification message.

**Device Array Examples**

The command below sends 'CHARD10' to all devices in the array, DeviceSetA.

```
DEV DeviceSetA[ ] = {Device1, Device2, Device3}
SEND_COMMAND DeviceSetA, 'CHARD10'
```

The command below sends 'CHARD10' to the third device in the array, DeviceSetA,

```
SEND_COMMAND DeviceSetA[3], 'CHARD10'
```

and is equivalent to:

```
SEND_COMMAND Device3, 'CHARD10'
```

The intent of the feedback statement is to set channel 1 in every device in DeviceSetA to either on or off, depending on the value of the right-hand expression; it is unclear what the right-hand expression evaluates to. The compiler will issue a warning indicating the syntax is unclear and that DeviceSetB[1] is assumed. To avoid this warning, specify a particular device in the array. For example:

```
[DeviceSetA, 1] = [DeviceSetB[1], 2]    (* Correct *)
```
Device-Channels and Device-Channel Arrays

As the name implies, a device-channel (DEVCHAN) is a combination of a device and a channel. It is represented internally as a DEVCHAN structure. This structure combines the fields of a DEV structure representing the device with a field representing the channel number:

```
STRUCTURE DEVCHAN
{
    DEV       //Device
    INTEGER   //Channel
}
```

The first component of a device-channel pair represents the Device Number, Port, and System. It can be specified as either a single device number, a constant DEV structure or as a D:P:S specification. Each device specified in a device-channel pair should be defined in the DEFINE_DEVICE section.

Channels are expressed as integer constants. A DEVCHAN is declared in either the DEFINE_VARIABLE or DEFINE_CONSTANT section. For example, "[128, 1]", "[CONSTANTDPS, 9]" and "[128:1:0, 5]" are all valid representations of device-channel pairs. A DEVCHAN enclosed within square brackets implies an evaluation, whereas a DEVCHAN enclosed within curly braces does not, as illustrated below:

```
DEFINE_VARIABLE
DEVCHAN dcl = {128:1:0, 1}
DEVCHAN dcset[ ] = { {128:1:0, 1}, {128:1:0, 2}, {128:1:0, 3} }

DEFINE_PROGRAM
IF ( [dcl] || [128:1:0, 2] )          // evaluation of 2 devchans
    [dcl] = 1                             // feedback
    dc1 = {129:1:0, 2}                    // assigns a new value to dcl
    [dcl] = {129:1:0, 2}                  // Syntax Error!
```

A DEVCHAN array is declared in the DEFINE_VARIABLE or DEFINE_CONSTANT section in one of two ways:

- Declare a DEVCHAN array whose maximum length is determined by the number of elements in the initialization array on the right-hand side, as shown below:

  ```
  DEVCHAN[ ] DCSName         = {{Dev1,Chan1}, {Dev2,Chan2}, ...}
  ```

- Use MAXLEN to specify the maximum length of the array, as shown below:

  ```
  DEVCHAN[ ] DCSName[MAXLEN] = {{Dev1,Chan1}, {Dev2,Chan2}, ...}
  ```

In either case, the number of elements in the initialization array determines the effective length of the array. That value can be determined at run-time by calling LENGTH_ARRAY. The maximum length available for a DEVCHAN[ ] array can be determined by calling MAX_LENGTH_ARRAY.

The individual elements of a DEVCHAN array can be referenced by their defined names (Dev1, Chan1, Dev2, Chan2, etc.) or by using array notation with the device-channel array name. For example, the third element in the device-channel array, MyDCSet[3], would be referenced by MyDCSet[3]. Furthermore, since a DEVCHAN array is an array of DEVCHAN structures, DEVCHAN members can be referenced using the dot operator notation such as MyDCSet[3].Device or MyDCSet[1].Channel.

A DEVCHAN array can be used anywhere a [Device, Channel] specification is required with the result of providing a range of targets for the command or instruction where it is used. This implies an alternate form for the following commands:

```
Button[(DEVCHAN)] PULSE[(DEVCHAN)]
DO_PUSH[(DEVCHAN)] PUSH[(DEVCHAN)]
DO_RELEASE[(DEVCHAN)] RELEASE[(DEVCHAN)]
OFF[(DEVCHAN)] TO[(DEVCHAN)]
ON[(DEVCHAN)]
```

The index of the last member of the array for which an event notification was received can be determined by calling GET_LAST(MyDCSet). This is useful for determining which device and channel in an array is referenced to in a particular notification message.

Device-Level Arrays

A device-level array (DEVLEV array) is an array of device-level pairs. Each element is represented internally as a DEVLEV structure. This structure combines the fields of a DEV structure representing the device with a field representing the level number.

```
STRUCTURE DEVLEV
{
    DEV       // Device
    INTEGER   // Level
}
```

The first component of a device-level pair (Device) represents the device number, port, and system. It can be specified as either a single device number, a constant DEV structure or as a D:P:S specification. Each device specified in a device-level pair should be defined in the DEFINE_DEVICE section. The second component is the level number on the device. The level number is expressed as an integer constant.
A DEVLEV array is declared in the DEFINE_VARIABLE or DEFINE_CONSTANT section in one of two ways:

- Declare a DEVLEV array whose maximum length is determined by the number of elements in the initialization array on the right-hand side.
  
  ```
  DEVLEV DLName[] = {{Dev1,Level1}, {Dev2,Level2}, ...}
  ```

- Use MAXLEN to specify the maximum length of the array.
  
  ```
  DEVLEV DLName[MAXLEN] = {{Dev1,Level1}, {Dev2,Level2}, ...}
  ```

In either case, the number of elements in the initialization array determines the effective length of the array. That value can be determined at run-time by calling LENGTH_ARRAY. The maximum length available for a DEVLEV array can be determined by calling MAX_LENGTH_ARRAY.

The individual elements of a level array can be referenced by their defined names (Dev1, Level1, Dev2, Level2, etc.) or alternatively, by using array notation with the device-level array name. For example, the 3rd element in the device-level array, MyDLSet, would be referenced by MyDLSet[3]. Furthermore, since a DEVLEV array is an array of DEVLEV structures, DEVLEV members can be referenced using the dot operator notation such as MyDLSet[3].Device or MyDLSet[1].Level.

The index of the last member of the array for which an event notification was received can be determined by calling GET_LAST(MyDLSet). This is useful for determining which device and level in an array is referenced to in a particular notification message.
Subroutines

Overview

A Subroutine is a section of code that stands alone, and can be called from anywhere else in the program.

DEFINE_CALL Subroutines

The DEFINE_CALL is the standard method provided by NetLinx for defining subroutines.

\[
\text{DEFINE_CALL '('<\text{subroutine name}>') [\text{Param1,Param2,...}]} \\
\{ \text{(* statements *)} \}
\]

where (Param1, Param2, ...) refers to a comma-separated list of <datatype><variable> pairs. For example, "INTEGER Size" would be one pair.

DEFINE_CALL names must not conflict with previously defined constants, variables, buffers, or wait names. Unlike identifiers, DEFINE_CALL names are case sensitive.

A subroutine may accept parameters. To do this, each parameter and its type must be listed within the set of parentheses to the right of the subroutine name, as shown below:

\[
\text{DEFINE_CALL 'Read Input' (CHAR Buffer){ ]} \\
\{ } \\
\}
\]

To invoke a user-defined subroutine, use the CALL keyword plus the name of subroutine and any required calling parameters.

\[
\text{CALL 'Read Input' (Buf1)}
\]

In NetLinx, DEFINE_CALL supports the RETURN statement (as shown in the following example), although return values are not supported.

\[
\text{DEFINE_CALL 'Read Input' (CHAR Buffer)} \\
\{ \text{if (nChars = 0)} \}
\{ \text{RETURN // exit subroutine} \}
\{ \text{(* read input *)} \}
\}
\]

SYSTEM_CALL Subroutines

A SYSTEM_CALL subroutine is a special type of DEFINE_CALL subroutine defined in a separate program file called a LIB file with a PROGRAM_NAME entry matching the subroutine name.

\[
\text{PROGRAM_NAME = 'COSX'}
\]

\[
\text{DEFINE_CALL 'COSX' (FLOAT X)} \\
\{ \text{(* body of subroutine *)} \}
\]

To invoke a system call, use the SYSTEM_CALL keyword followed by the name in single quotes and any calling parameters, as shown below:

\[
\text{SYSTEM_CALL 'COSX' (45)}
\]

System calls are resolved automatically at compile time, without requiring an INCLUDE instruction to include the system call source file. For special cases where multiple copies of a system call are needed, an instance number can be specified in the call. The compiler will compile a separate copy of the subroutine for each system call instance number. For example, the following commands force the compiler to include two separate copies of COSX:

\[
\text{SYSTEM_CALL[1] 'COSX' (45)} \\
\text{SYSTEM_CALL[2] 'COSX' (60)}
\]

This technique could be useful in cases where a system call contains a wait instruction that conflicts when multiple calls to the same subroutine were made during a single wait period.

Function Subroutines

A function is similar to a DEFINE_CALL, but is intended for use either standalone or in-line as an expression. Instead of requiring a string literal for its name, it requires a name that follows the rules for naming constants and variables. This eliminates the need for using the CALL keyword to invoke the subroutine. DEFINE_FUNCTION subroutines also differ from DEFINE_CALL by allowing values to be returned using the RETURN statement (see below).

NOTE: The return type may only be one of the 8 intrinsic types. Strings, arrays, structures, classes and other user-defined types may not be returned.

Syntax:

\[
\text{DEFINE_FUNCTION [<return type>] FnName[(Param1,Param2,...)]} \\
\{ \text{(* statements *)} \}
\]

NOTE: You cannot declare and initialize variables in the same line. You must group the declarations first, followed by the initialization.
Example:

```plaintext
define_function integer myFunction (integer var0)
{
    integer nBytes
    stack var result
    nBytes = 0
    result = var0 + nBytes
    return result
}
```

**NOTE:** *When it is a NetLinx function, a syntax where there appears a ([ ]), the () are NOT OPTIONAL but the [ ] are optional.*

The `define_function` subroutine can be called as a single programming statement. For example, the following syntax:

```plaintext
readbuffer (buffer, bufsize)
```

Can be used in an assignment statement such as:

```plaintext
count = readbuffer (buffer, bufsize)
```

or as part of an expression such as:

```plaintext
if (readbuffer (buffer, bufsize) > 0)
{
    (* statements *)
}
```

The rules pertaining to calling parameters are the same for `define_function` as they are for `define_call` subroutines. The parameter list must appear in parentheses to the right of the function name. If the function has no calling parameters a set of parentheses must still be included. For example,

```plaintext
myfunc () // calling a function with no parameters
```

The return type may be omitted, as an alternate way of defining a subroutine. In this case the function cannot be used as part of an expression or in an assignment statement.

`define_function` also allows the use of the return keyword that serves two purposes:

- To return prematurely from a function.
- To return a value from a function.

The format of the return statement is:

```plaintext
return [<return value>]
```

If a return statement is encountered anywhere in the function, execution of the function is terminated immediately and the value (if any) specified as the `<return value>` is returned to the caller. A function that returns a value through the `return` keyword must be declared with a return type. Conversely, a function that is declared without a return type cannot return a value. In the example below, `getBufferSize` returns an unsigned 16-bit integer, `bufsize`. The return type is indicated before the `define_function` keyword.

```plaintext
define_function integer getBufferSize()
local var integer bufsize = 0;
{
    .
    .
    .
    return bufsize;
}
```

To call this function and to retrieve the `return` value, use the following syntax:

```plaintext
bufsize = getBufferSize()
```

where `bufsize` is declared to be of type `integer`.

Even if a function returns a value, it is not necessary to assign the return value to a variable. Both forms of the following call are valid. In the second case, the return value is simply thrown away.

```plaintext
count = readbuffer (buffer, bufsize)
readbuffer (buffer, bufsize) // return value is ignored
```

**NOTE:** *The return type may only be one of the 8 intrinsic types (see Data Types). Strings, arrays, structures, classes and other user-defined types may not be returned.*

### Calling Parameters

Parameters may be passed to any NetLinx function or subroutine. *Calling parameters* are simply variables or constants that originate from the caller and are received by the function or subroutine being invoked. The NetLinx compiler passes all variables by reference. This means that the variable the subroutine operates on is the same variable the caller passed. Any change made to a variable passed as a calling parameter updates the value of the variable from the perspective of the caller. You can take advantage of this pass by reference feature to return an updated value through a calling parameter rather than as the return value.

Constants, on the other hand, are passed by value. When this happens, a copy of the parameter is delivered to the subroutine. Any change made to the variable representing the constant is lost once the function or subroutine finishes.

Function and subroutine declarations must include the type and name of each parameter expected. If the type is omitted, the default type is assumed; arrays are `char` type and non-array parameters are `integer`. To specify an array as a function or subroutine parameter, one set of brackets for each array dimension must follow the variable name, as shown below:

```plaintext
define_call 'process array' (char array[ ] [ ])
{
    (* body of subroutine *)
}
```
The parameter Array is declared to be a 2-dimensional array, by including two sets of brackets after the name. For compatibility with existing programs, the array dimensions may be specified inside the brackets. These dimensions are not required and are ignored by the compiler. The NetLinx interpreter will do bounds checking on the array and generate a run-time error if the array bounds are exceeded.

When calling a subroutine that takes an array as one of its parameters, pass only the name of the array as the calling parameter, as shown below:

```
CHAR Buffer[10][20]
CALL 'Process Array' (Array)
```

If dimensions are specified in the call statement, the compiler will interpret that as specifying a subset of the array. For example, suppose Array were defined as a 3-dimensional array. The third table of that dimensional array could be passed to 'Process Array' as follows:

```
CHAR Buffer[5][5][10]
CALL 'Process Array' (Array [3])
```

### Subroutine Keywords

NetLinx supports the following Subroutine keywords:

| DEFINE Keywords | Use the CALL keyword and the name of the subroutine in single quotes to tell NetLinx to execute a subroutine. For example, to execute the subroutine Lights Off, type the following where you want the CALL to occur:
| CALL          | CALL 'Lights Off' When NetLinx executes the CALL, program execution jumps to the first line inside the braces of the DEFINE_CALL. The subroutine is executed only once, and then NetLinx returns to the statement directly following the CALL statement. |

**DEFINE_CALL**

This keyword defines the implementation of a NetLinx subroutine.

```
DEFINE_CALL '<name>' [(P1,P2,...)]
{
    // body of subroutine
}
```

The subroutine name cannot be a previously defined device name, constant, or variable, or a name assigned to a buffer or a wait statement. DEFINE_CALL names are case sensitive and may contain spaces.

**NOTE:** Subroutines must be defined before they can be used. For this reason, DEFINE_CALLs should appear before the DEFINE_START, DEFINE_EVENT, and DEFINE_PROGRAM sections.

| SYSTEM_CALL | This keyword is similar to CALL except that the subroutine invoked using the SYSTEM_CALL keyword resides in a special file called a library file. When this keyword is used, the compiler generates a call to the subroutine in the library file and automatically includes the library file for compilation. |
## Compiler Directives

### Overview

Compiler Directives are special types of instructions for the compiler. They won’t produce any runtime code. Instead, they allow you to instruct the compiler to conditionally compile parts of the code. See page 48 for a listing of Compiler Keywords.

**NOTE:** Refer to Appendix A - Compiler Warning & Errors section on page 156 for a listing of Compiler Messages.

<table>
<thead>
<tr>
<th>Compiler Directives</th>
<th>Description</th>
</tr>
</thead>
</table>
| **#DEFINE**         | This directive defines a symbol to be used only by #IF_DEFINED and #IF_NOT_DEFINED directives. Syntax: 

```
#DEFINE <symbol>
```

The name of the symbol must be unique among all other identifiers in the program. The symbol can be defined anywhere in the program file but cannot be used in any statement that appears before it is defined.

```
// Specify the INCLUDE_TOGGLE_VIDEO_PROJECTOR_POWER_FUNCTION compiler directive if there is a video projected connected to the controller
#DEFINE INCLUDE_TOGGLE_VIDEO_PROJECTOR_POWER_FUNCTION

#IF_DEFINED INCLUDE_TOGGLE_VIDEO_PROJECTOR_POWER_FUNCTION
DEFINE_FUNCTION toggleVideoProjectorPower()
{
  // code to toggle video projector power goes here
}
#END_IF
```

Notice in the above sample that the #DEFINE compiler directive is specified before the #IF_DEFINED and #END_IF compiler directives. When #IF_DEFINED and #IF_NOT_DEFINED compiler directives are specified in include files, the include file statements (defined by the #INCLUDE compiler directive) need to be declared after the #DEFINE compiler directive statements.

| **#DISABLE_WARNING** | This compiler directive disables a specified warning message from being displayed after the program is compiled. Syntax: 

```
#DISABLE_WARNING warning#
```

For example, to disable the following warning:

```
WARNING: C:\Temp\AMXLoader\AMX home Autopatch Switcher.axi(1191): C10571: Converting type [INTEGER] to [SINTEGER]
```

Add the following to the AXS file to disable the "C10571" warning:

```
#DISABLE_WARNING 10571
```

**NOTE:** Do not include the "C" prefix from the warning message.

<table>
<thead>
<tr>
<th><strong>#ELSE</strong></th>
<th>This directive specifies a counter condition; used optionally in conjunction with #IF_DEFINED and #IF_NOT_DEFINED.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>#ENDIF</strong></td>
<td>This directive marks the end of an #IF_DEFINED or #IF_NOT_DEFINED code block.</td>
</tr>
</tbody>
</table>
| **#IF_DEFINED**    | This directive defines conditional compilation. The code following the #IF_DEFINED and before #ELSE (or before #END_IF, if #ELSE is not present) is compiled only if a symbol is defined (see #DEFINE above). If a symbol is not defined and the #ELSE directive is present, the code following #ELSE and before #ENDIF is compiled instead. Syntax: 

```
#IF_DEFINED symbol
  // code block
#ELSE
  // code block
#ENDIF
```

| **#IF_NOT_DEFINED** | Defines conditional compilation similar to #IF_DEFINED. The code following the #IF_NOT_DEFINED and before #ELSE (or before #END_IF, if #ELSE is not present) is compiled only if symbol is not defined (see #DEFINE above). If a symbol is defined and the #ELSE directive is present, the code following #ELSE and before #ENDIF is compiled instead. Syntax: 

```
#IF_NOT_DEFINED symbol
  // code block
#ELSE
  // code block
#ENDIF
```

| **#INCLUDE**       | To include a file in a program, use the keyword #INCLUDE followed by the filename in single quotes: 

```
#INCLUDE 'TEST.AXI'
```

When the compiler reaches the #INCLUDE statement, it jumps into the specified file and continues compiling. When it has reached the end of that file, it comes back to the line following the #INCLUDE statement and continues compiling. |

| **#WARN**          | Displays a warning message after the program is compiled. Its primary purpose is to remind you of certain conditions related to the program. Syntax: 

```
#WARN 'This code is obsolete'
```

```
Array Keywords

Overview

NetLinx allows arrays of any data type supported by the language as well as arrays of user-defined structures and classes.

- If an initialization statement is included in the variable declaration, the array dimension is not required.
- If the array dimension is omitted, both the maximum and effective length is set to the length needed to hold the data contained in the initialization string.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>STRING[ ] = 'character string'</td>
</tr>
<tr>
<td>WIDECHAR</td>
<td>WideString[ ] = 'wide character string'</td>
</tr>
<tr>
<td>INTEGER</td>
<td>IntegerNum[ ] = {1, 2, 3, 4, 5}</td>
</tr>
<tr>
<td>SINTEGER</td>
<td>SINTEGERNum[ ] = {-1, 5, -6}</td>
</tr>
<tr>
<td>LONG</td>
<td>LONGNum[ ] = {$EFFF, 0, 89000}</td>
</tr>
<tr>
<td>SLONG</td>
<td>SLONGNum[ ] = {-99000, 50, 100, 100000}</td>
</tr>
<tr>
<td>FLOAT</td>
<td>FloatingNum[ ] = {1.0, 20000.0, 17.5, 80.0}</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>DoubleNum[ ] = {1.0e28, 5.12e-6, 128000.0}</td>
</tr>
</tbody>
</table>

String expressions can be used initialization statements only if each byte is separated by a comma:

```plaintext
CHAR sProjOn[] = {$02,'P','O','N',$03}
```

The initialization statement for a single dimension character string is enclosed in single quotes, whereas data for other types is enclosed in braces. In the case of a multidimensional character string, the strings in the initialization statement are separated by commas and enclosed in braces.

Example:

```plaintext
DEFINE_VARIABLE
CHAR StringTable_3[3][5]=
{
  {'STR 1'},
  {'STR 2'},
  {'STR 3'},
}
```

For multidimensional array types, the data pertaining to each dimension is delimited using braces, as shown below:

```plaintext
INTEGER Num2D[ ] [ ] = {{1, 3}, {2, 4}, {7, 8}}
/* This sets the dimensions to Num2D[3][2] */
```

The ”=” operator can be used to assign a one dimensional array to another.

```plaintext
Array1 = Array2
```

The one dimensional arrays must match type. The size of each dimension of the destination array must be greater than or equal to the corresponding array being assigned; otherwise the contents of the array being assigned is truncated to fit into the destination array. If a type mismatch is detected the compiler will issue an appropriate warning.

The lengths of an array are determined by calling LENGTH_ARRAY and MAX_LENGTH_ARRAY.

- LENGTH_ARRAY returns the effective length of a dimension of an array: the length set implicitly through array initialization or explicitly through a call to SET_LENGTH_ARRAY.
- MAX_LENGTH_ARRAY is used to determine the maximum length of a dimension of an array.

Changing an element in array does not change its length. SET_LENGTH_ARRAY is used to change the effective length of an array when necessary, such as when you’ve added elements via a FOR loop.

Example:

```plaintext
DEFINE_VARIABLE
INTEGER Len
INTEGER Len1
INTEGER Len2
INTEGER Array1[] = {3, 4, 5, 6, 7}
INTEGER Array2[10] = {1, 2}
DEFINE_START
Len = MAX_LENGTH_ARRAY(Array1)      // Len = 5
Len = MAX_LENGTH_ARRAY(Array2)      // Len = 10
// LENGTH_ARRAY is called to determine the effective length of Array1
// and Array2.
// This value is set automatically when the arrays are initialized.
Len1 = LENGTH_ARRAY(Array1)         // Len1 = 5
Len2 = LENGTH_ARRAY(Array2)         // Len2 = 2
FOR (Len = 1; Len <= Len1; Len++)   
{
  ARRAY2[Len+Len] = Array1[Len]
}
SET_LENGTH_ARRAY(Array2,Len2 + Len1) // Set Array2 length to new length end
```
Multi-dimension arrays cannot be copied directly to another. Use FOR or WHILE loops to copy them at the lowest dimension.

Example:

```plaintext
DEFINE VARIABLE
CHAR ARRAY1[2][10] = {{'hello '},{'goodbye'}}
CHAR ARRAY2[2][10] = {{'i am the '},{'walrus'}}
INTEGER INDEX
DEFINE_PROGRAM
WAIT 20
{ FOR (INDEX = 1; INDEX <=2; INDEX++)
  ARRAY2[INDEX] = ARRAY1[INDEX]
} SEND_STRING 0,”ARRAY2[1],ARRAY2[2]”
// end
```

Multi-Dimensional Arrays

Any of the single dimension array types listed above can be used to define an array of n-dimensions.

- A 2-dimensional array is simply a collection of 1-dimensional arrays;
- a 3-dimensional array is a collection of 2-dimensional arrays, and so forth.

Here’s an example:

```plaintext
INTEGER Num1D[10] // [Column]
INTEGER Num2D[5][10] // [Row][Column]
INTEGER Num3D[2][5][10] // [Table][Row][Column]
```

One way to view these arrays is to think of Num2D as being a collection of five Num1D’s and Num3D as being a collection of two Num2D’s. When referencing elements of the above arrays:

- `Num1D[1]` refers to the 1st element
- `Num2D[1]` refers to the 1st row
- `Num2D[1][1]` refers to the 1st element of the 1st row
- `Num3D[1]` refers to the 1st table
- `Num3D[1][1]` refers to the 1st row of the 1st table
- `Num3D[1][1][1]` refers to the 1st element of the 1st row of the 1st table

The following operations are legal:

```plaintext
Num2D[2] = Num1D
Num2D[5][5] = Num1D[5]
Num3D[2] = Num2D
Num3D[2][1] = Num1D
Num3D[2][1][1] = Num1D[1]
```

`LENGTH_ARRAY` and `MAX_LENGTH_ARRAY` are used to determine the effective and maximum lengths of multidimensional arrays, as shown in the following examples:

```plaintext
INTEGER Len
INTEGER My3DArray[5][3][4]
Len = MAX_LENGTH_ARRAY(My3DArray) // Len = 5
Len = MAX_LENGTH_ARRAY(My3DArray[1]) // Len = 3
Len = MAX_LENGTH_ARRAY(My3DArray[1][1]) // Len = 4
INTEGER Len
INTEGER My3DArray[5][3][4] =
{ }
{ [1,2,3,4],
  [5,6,7,8],
  [9,10,11]
},
{ [13,14] }
Len = LENGTH_ARRAY(My3DArray) // Len = 2, number of tables *)
Len = LENGTH_ARRAY(My3DArray[2]) // Len = 1, number of rows in table 2 *)
Len = LENGTH_ARRAY(My3DArray[1][3]) // Len = 3, number of columns in table 1, row 3 *)
```
Array Keywords

**LENGTH_ARRAY**

This function returns the effective length of a dimension of an array, implicitly through array initialization or array manipulation operations, or explicitly through a call to the function SET_LENGTH_ARRAY.

_LONG LENGTH_ARRAY (<type> Array[ ])_

Parameters:
- <type>: May be any intrinsic or user-defined data type
- Array: An array of any type.

Result: The effective (or working) length of the array.

```
INTEGER Len
INTEGER Array1[ ] = {3, 4, 5, 6, 7}
INTEGER Array2[ ] = {1, 2}
INTEGER My3DArray[5][3][4] =
  {
    {
      {1,2,3,4},
      {5,6,7,8},
      {9,10,11}
    }
    {
      {13,14}
    }
  }
Len = LENGTH_ARRAY(Array1)     // Len = 5
Len = LENGTH_ARRAY(Array2)     // Len = 2
Len = LENGTH_ARRAY(My3DArray)  (* Len = 2, the number of tables *)
Len = LENGTH_ARRAY(My3DArray[2]) (* Len = 1, the number of rows in table 2 *)
Len = LENGTH_ARRAY(My3DArray[1][3])  (* Len = 3, the number of columns in table 1, row 3 *)
```

See SET_LENGTH_ARRAY, on page 32, for more information.

**MAX_LENGTH_ARRAY**

This function returns the maximum length of a dimension of an array.

_LONG MAX_LENGTH_ARRAY (<type> Array[ ])_

Parameters:
- <type>: May be any intrinsic or user-defined data type.
- Array: An array of any type.

Result: The length of the specified dimension of Array.

```
FLOAT FPArray[10]
LONG  NumArray[5][3][4]
Len = MAX_LENGTH_ARRAY(FPArray)         // Len = 10
Len = MAX_LENGTH_ARRAY(NumArray)        // Len = 5
Len = MAX_LENGTH_ARRAY(NumArray[1])     // Len = 3
Len = MAX_LENGTH_ARRAY(NumArray[1][1])  // Len = 4
```

**SET_LENGTH_ARRAY**

This function sets the effective length of a dimension of an array.

_SET_LENGTH_ARRAY (<type> Array[ ], LONG Len)_

Parameters:
- <type>: May be any intrinsic or user-defined data type.
- Array: Array of any type
- Len: Value to assign as the length

Arrays are limited by their inability to have multiple data-types within one array. NetLinx supports Structures to remove this limitation. Structures group different data types together as one data unit. Refer to the Structure Keywords on page 128 for more information.
## Audit Keywords

NetLinx supports the following Audit keywords:

<table>
<thead>
<tr>
<th>Audit Keywords</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUDIT_NETLINX_GENERIC_EVENT</td>
<td>This function generates an audit record to the persistent audit trail containing the specified NetLinx Device D:P:S and user name to associate with the audit record and a text message to include in the audit record.</td>
<td><code>sinteger AUDIT_NETLINX_GENERIC_EVENT(DEV device, char username[], char msg[])</code></td>
</tr>
<tr>
<td>AUDIT_NETLINX_SESSION_EVENT</td>
<td>This function generates an audit record in the persistent audit trail containing the specified NetLinx Device D:P:S where the login occurred, the username of the login and the audit type.</td>
<td><code>sinteger AUDIT_NETLINX_SESSION_EVENT(DEV device, char username[], integer audit_type)</code></td>
</tr>
<tr>
<td></td>
<td>Parameters:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• audit_type - Can be one of the following values:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 - Audit Login success</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 - Audit Login fail</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 - Audit Logout</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Returns:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 - Successful audit</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-1 - Audit failed</td>
<td></td>
</tr>
</tbody>
</table>
NetLinx supports the following Authentication keywords:

<table>
<thead>
<tr>
<th>Authentication Keywords</th>
<th>Function Description</th>
<th>Syntax</th>
<th>Parameters</th>
<th>Returns</th>
</tr>
</thead>
</table>
| VALIDATE_NETLINX_ACCOUNT | This function validates the specified user name and password against the NetLinx Master Controller's internal user account database. For the account to be valid the user name must exist with the matching password and the specified user account must have been set up with ICSP Authorization. | sinteger VALIDATE_NETLINX_ACCOUNT(CHAR USERNAME[], CHAR PASSWORD[], LOGIN_INFO_STRUCT INFO) | Parameters:  
  - username - A character array containing the user name to validate.  
  - password - A character array containing the password to validate.  
  - info - A return structure of type LOGIN_INFO_STRUCT which contains the following values:  
    - STRUCTURE LOGIN_INFO_STRUCT  
      - INTEGER FAILED_LOGIN_COUNT;  
      - CHAR LAST_SUCCESSFUL_LOGIN_DATE[46];  
      - CHAR LAST_SUCCESSFUL_LOGIN_LOCATION[46];  
      - CHAR LAST_FAILED_LOGIN_DATE[46];  
      - CHAR LAST_FAILED_LOGIN_LOCATION[46];  
    
Returns:  
  0 - Valid user account.  
-1 - Username parameter is not a valid string  
-2 - Password parameter is not a valid string  
-3 - Invalid user account  
-4 - User account does not have ICSP Authorization  
-5 - Third argument is not a LOGIN_INFO_STRUCT  
-6 - User account matching name is locked out  
-7 - User account matching name has expired |
| VALIDATE_NETLINX_ACCOUNT_WITH_PERMISSION | This function validates the specified user name and password against the NetLinx Master Controller's internal user account database. For the account to be valid the user name must exist with the matching password and the specified user account must have been set up with ICSP Authorization. | sinteger VALIDATE_NETLINX_ACCOUNT(CHAR USERNAME[], CHAR PASSWORD[], CHAR TYPE[], CHAR PERMISSION[], LAST_LOGIN_INFO INFO) | Parameters:  
  - username - A character array containing the user name to validate.  
  - password - A character array containing the password to validate.  
  - type - The authorization type.  
  - permission - The permission type. Valid permission include: Configuration, Console, Diags, EncryptICSP, FTP, HTTP, ICSP, Terminal, AuditLog, User1, User2, User3, and User4  
  - info - A return structure of type LAST_LOGIN_INFO which contains the following values:  
    - STRUCTURE LAST_LOGIN_INFO  
      - INTEGER FAILED_LOGIN_COUNT;  
      - CHAR LAST_SUCCESSFUL_LOGIN_DATE[46];  
      - CHAR LAST_SUCCESSFUL_LOGIN_LOCATION[46];  
      - CHAR LAST_FAILED_LOGIN_DATE[46];  
      - CHAR LAST_FAILED_LOGIN_LOCATION[46];  
    
Returns:  
  0 - Valid user account.  
-1 - Username parameter is not a valid string  
-2 - Password parameter is not a valid string  
-3 - Invalid user account  
-4 - User account does not have ICSP Authorization  
-5 - Third argument is not a LOGIN_INFO_STRUCT  
-6 - User account matching name is locked out  
-7 - User account matching name has expired |
## Buffer Keywords

NetLinx supports the following Buffer keywords:

<table>
<thead>
<tr>
<th>Buffer Keywords</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CLEAR_BUFFER</strong></td>
<td>This command sets the contents of the specified text buffer to zero; therefore, subsequent GET_BUFFER_CHAR calls will not return anything. The CLEAR_BUFFER command does not modify the data in the buffer, just the internal length value.</td>
</tr>
<tr>
<td><strong>CREATE_BUFFER</strong></td>
<td>This keyword creates a buffer and can only appear in the DEFINE_START section of the program.</td>
</tr>
<tr>
<td><strong>CREATE_MULTI_BUFFER</strong></td>
<td>This keyword is the same as CREATE_BUFFER except that it accepts strings from a range of devices. Two forms of this command are supported. The first form of the command is provided for backward-compatibility; it accepts two device numbers as the range of devices. The second form of the command takes a device array rather than the device number pair.</td>
</tr>
</tbody>
</table>

### CLEAR_BUFFER

**Buffer Keywords**

- **Buffer Keywords**
- **CLEAR_BUFFER**
- **CREATE_BUFFER**
- **CREATE_MULTI_BUFFER**

### CREATE_BUFFER

- **CREATE_BUFFER DEV**, **Buffer**
- **CREATE_BUFFER** directs NetLinx to place any strings received from the specified device into the specified buffer (character array). When strings are added to the buffer, the length of the buffer is automatically adjusted. If the buffer is full, all bytes in the buffer are shifted to make room for the new string. A buffer can be manipulated in the same way as a character array.

### CREATE_MULTI_BUFFER

- **CREATE_MULTI_BUFFER**
- **CREATE_MULTI_BUFFER FirstDevice**, **LastDevice**, **Buffer**
- **CREATE_MULTI_BUFFER DeviceSet**, **Buffer**

**Parameters:**
- **FirstDevice**: First number in the range of devices.
- **LastDevice**: Last number in the range of devices.
- **Buffer**: Text buffer to receive the strings.

Each command string placed in the multi-buffer has a three-byte header associated with it:
- **The first header byte ($FF) marks the start of a new command string.**
- **The second header byte is either the number of the device or the index of the DEV[ ] member that received the command string.**
- **The third header byte is the length of the string.**

$FF, device number or DEV[ ] index, length, <string>

**The second form of the command takes a device array rather than the device number pair.**

### CREATE_MULTI_BUFFER

- **CREATE_MULTI_BUFFER DeviceSet**, **Buffer**
- **Parameters:**
  - **DeviceSet**: Set of devices for which the buffer will accept strings.
  - **Buffer**: Text buffer to receive the strings.

Each command string placed in the multi-buffer has a three-byte header associated with it:
- **The first header byte ($FF) marks the start of a new command string.**
- **The second header byte is the index into the DeviceSet of the device that received the string.**
- **The third header byte is the length of the string.**

$FF, device number or DEV[ ] index, length, <string>

This command is not recommended for use in NetLinx due to its limitations. The main limitations to note are:
- For the first form of the command, using FirstDevice and LastDevice, only devices using the same port and system will be allowed. The device in between the First Device and Last Device will be the sequential device numbers using the same port and system (i.e. 1:1:0, 2:1:0, 3:1:0, etc.).
- For the second form of the command, using DeviceSet, only 255 devices will be allowed in the array. This is required since only one byte is used to represent the DeviceSet index in the return string so it has an upper limit of 255.
- Strings from a device longer than 255 bytes will be broken up into multiple "multi" strings within the buffer. For instance, if 300 characters are received from a port, the multi buffer will contain:

"$FF,<index1>, 255,<first 255 characters>,$FF,45,<last 45 characters>"

The recommended replacement for CREATE_MULTI_BUFFER and GET_MULTI_BUFFER_STRING is to use a DeviceSet and a DATA_EVENT to capture strings from multiple devices. See GET_MULTI_BUFFER_STRING for more information.

An example is shown below:

```plaintext
DEFINE_DEVICE
Dev1 = 1:1:0
Dev2 = 1:2:0
Dev3 = 1:3:0
DEFINE_VARIABLE
DEV DeviceSet[] = {Dev1, Dev2, Dev3}
INTEGER DeviceIndex
CHAR DeviceString[1000]
DEFINE_EVENT
DATA_EVENT[DeviceSet]
{
    STRING:
    {
        DeviceIndex = GET_LAST(DeviceSet)
        DeviceString = DATA.TEXT
    }
}
```

See GET_MULTI_BUFFER_STRING for more information.
### Buffer Keywords (Cont.)

<table>
<thead>
<tr>
<th>Buffer Keywords</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GET_BUFFER_CHAR</strong></td>
<td>This keyword removes characters from a buffer.</td>
</tr>
<tr>
<td></td>
<td>Result = GET_BUFFER_CHAR (Array)</td>
</tr>
<tr>
<td></td>
<td>Array may be either a character array or wide character array; the operation is identical in either case.</td>
</tr>
<tr>
<td></td>
<td>The result is a CHAR or WIDECHAR value depending on the variable type of Array.</td>
</tr>
<tr>
<td></td>
<td>GET_BUFFER_CHAR has a two-part operation:</td>
</tr>
<tr>
<td></td>
<td>1. Retrieve the first character in the buffer.</td>
</tr>
<tr>
<td></td>
<td>2. Remove the retrieved character from the buffer and shift the remaining characters by one to fill the gap.</td>
</tr>
</tbody>
</table>

| **GET_BUFFER_STRING**   | This function removes characters from a buffer.                                                                                                 |
|                         | Result = GET_BUFFER_STRING (Array, Length)                                                                                                     |
|                         | Array may be either a character array or wide character array; the operation is identical in either case.                                      |
|                         | Length is the number of characters to remove.                                                                                                   |
|                         | Result is a CHAR or WIDECHAR value depending on the variable type of Array.                                                                    |
|                         | GET_BUFFER_STRING has a two-part operation:                                                                                                     |
|                         | 1. Retrieve <length> number of characters from the buffer.                                                                                     |
|                         | 2. Remove the retrieved character from the buffer and shift the remaining characters up to fill the gap.                                     |

<table>
<thead>
<tr>
<th><strong>GET_MULTI_BUFFER_STRING</strong></th>
<th>To access characters coming into a multi-buffer, you must first use GET_MULTI_BUFFER_STRING to transfer these characters into another array. For example:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Device = GET_MULTI_BUFFER_STRING (Buffer, Array)</td>
</tr>
<tr>
<td></td>
<td>• The next string in the specified buffer is copied to the specified array.</td>
</tr>
<tr>
<td></td>
<td>• All three header bytes are stripped before the string is copied.</td>
</tr>
<tr>
<td></td>
<td>• If CREATE_MULTI_BUFFER was defined using a FirstDevice and LastDevice, the return value Device is the device number (not the Port number) of the card that received the string.</td>
</tr>
<tr>
<td></td>
<td>• If CREATE_MULTI_BUFFER was defined using a DeviceSet, the return value Device is the device index into the DeviceSet array of the card that received the string.</td>
</tr>
</tbody>
</table>
Channel Keywords

NetLinx supports the following CHANNEL keywords:

<table>
<thead>
<tr>
<th>CHANNEL Keywords</th>
<th>See page 46.</th>
</tr>
</thead>
</table>
| OFF                   | This keyword is used to turn off a channel or variable. If used with a variable, OFF sets it to zero.  
|                       | OFF[DEVICE,CHANNEL]                                                        |
|                       | OFF[(DEVCHAN[ ]))]                                                         |
|                       | OFF[Variable]                                                               |
| ON                    | This keyword is used to turn a channel or variable on. If used with a variable, ON sets it to 1.  
|                       | ON[DEVICE,CHANNEL]                                                         |
|                       | ON[(DEVCHAN[ ]))]                                                          |
|                       | ON[Variable]                                                                |
| PUSH_CHANNEL          | See page 120.                                                               |
| RELEASE_CHANNEL       | See page 120.                                                               |
| PUSH_DEVCHAN          | See page 120.                                                               |
| RELEASE_DEVCHAN       | See page 120.                                                               |
| SET_VIRTUAL_CHANNEL_COUNT | See page 122.                                                              |
| TOTAL_OFF             | This keyword turns a channel or variable off. Unlike OFF, TOTAL_OFF turns off the status of a channel or variable that is in a mutually exclusive set. |
### Clock Manager Keywords

NetLinx supports the following Clock Manager keywords:

<table>
<thead>
<tr>
<th>Clock Manager Keywords</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CLKMGR_SET_DAYLIGHT_SAVINGS_OFFSET</strong></td>
<td>Sets the Daylight Savings Offset to the specified value.</td>
</tr>
<tr>
<td><strong>CLKMGR_DELETE_USER_DEFINED_TIMESERVER</strong></td>
<td>Deletes the user-defined entry that has its IP-ADDRESS matching the parameter.</td>
</tr>
<tr>
<td><strong>CLKMGR_GET_ACTIVE_TIMESERVER</strong></td>
<td>Populates the TIMESERVER structure with the currently active time server's data. The function returns a negative SLONG value if it encounters an error.</td>
</tr>
<tr>
<td><strong>CLKMGR_GET_DAYLIGHTSAVINGS_OFFSET</strong></td>
<td>Populates the TIMEOFFSET structure with the current Daylight Savings Offset configured. The function returns a negative SLONG value if it encounters an error.</td>
</tr>
<tr>
<td><strong>CLKMGR_GET_END_DAYLIGHTSAVINGS_RULE</strong></td>
<td>Gets a string representation of when Daylight Savings is supposed to end.</td>
</tr>
<tr>
<td><strong>CLKMGR_GET_RESYNC_PERIOD</strong></td>
<td>Returns the Clock Manager’s re-sync period in minutes. The default setting is one (1) hour. This setting has no effect if the Clock Manager mode is set to STANDALONE.</td>
</tr>
<tr>
<td><strong>CLKMGR_GET_START_DAYLIGHTSAVINGS_RULE</strong></td>
<td>Gets a string representation of when Daylight Savings is supposed to START.</td>
</tr>
<tr>
<td><strong>CLKMGR_GET_TIMESERVERS</strong></td>
<td>Populates the currently configured time server entries from the Clock Manager into the specified TIMESERVER array. The function returns a negative SLONG value if it encounters an error, otherwise the return value is set to the number of records populated into the CLKMGR_TIMESERVER_STRUCT array.</td>
</tr>
<tr>
<td><strong>CLKMGR_GET_TIMEZONE</strong></td>
<td>Returns Timezone as a string in the format: (UTC[+</td>
</tr>
<tr>
<td><strong>CLKMGR_IS_DAYLIGHTSAVINGS_ON</strong></td>
<td>Returns FALSE/0 or TRUE/1. The default setting is FALSE/0.</td>
</tr>
<tr>
<td><strong>CLKMGR_IS_NETWORK_SOURCED</strong></td>
<td>Returns FALSE/0 or TRUE/1. The default setting is FALSE/0.</td>
</tr>
<tr>
<td><strong>CLKMGR_SET_ACTIVE_TIMESERVER</strong></td>
<td>Sets the time server entry that has the matching IP-ADDRESS to the IP parameter as the active time server entry.</td>
</tr>
<tr>
<td><strong>CLKMGR_SET_CLK_SOURCE</strong></td>
<td>Sets the source for the Clock Manager. Can be set to: CLKMGR_MODE_NETWORK or CLKMGR_MODE_STANDALONE</td>
</tr>
</tbody>
</table>
## Clock Manager Keywords (Cont.)

| **CLKMGR_SET_DAYLIGHTSAVINGS_MODE** | Sets Daylight Savings mode to Off or On.  
  | **CLKMGR_SET_DAYLIGHTSAVINGS_MODE** (CONSTANT INTEGER ONOFF)  
  Can be set to:  
  ON/TRUE  
  or  
  OFF/FALSE  
  |  |
| **CLKMGR_SET_DAYLIGHTSAVINGS_OFFSET** | Sets the Daylight Savings Offset to the specified value.  
  | **CLKMGR_SET_DAYLIGHTSAVINGS_OFFSET** (CONSTANT CLKMGR_TIMEOFFSET_STRUCT T)  
  |  |
| **CLKMGR_SET_END_DAYLIGHTSAVINGS_RULE** | Sets the END Daylight Savings rule to the specified string which must be in either the Fixed-Date format or the Occurrence-Of-Day format.  
  | **CLKMGR_SET_END_DAYLIGHTSAVINGS_RULE** (CONSTANT CHAR RECORD[])  
  The function returns a negative SLONG value if it encounters an error.  
  The Fixed-Date rules have the form:  
  "fixed:DAY,MONTH,HH:MM:SS"  
  with all fields as numeric except for the word "fixed"  
  (e.g.: "fixed:21,3,02:00:00" ===> March 21 @ 02:00:00AM).  
  The Occurrence-Of-Day rules have the form:  
  "occurrence:OCCURRENCE,DAY-OF-WEEK,MONTH,HH:MM:SS"  
  with all fields as numeric except for the word "occurrence"  
  DAY-OF-WEEK translates as:  
  • 1=Sunday  
  • 2=Monday  
  • 3=Tuesday  
  • 4=Wednesday  
  • 5=Thursday  
  • 6=Friday  
  • 7=Saturday  
  (e.g.: "occurrence:3,1,10,02:00:00" ===> 3rd Sunday in October @ 02:00:00AM).  
  |  |
| **CLKMGR_SET_RESYNC_PERIOD** | Sets the re-sync period to the specified minute value.  
  | **CLKMGR_SET_RESYNC_PERIOD** (CONSTANT INTEGER PERIOD)  
  The upper bound is 480 minutes (i.e., 8 hours).  
  |  |
| **CLKMGR_SET_START_DAYLIGHTSAVINGS_RULE** | Sets the START Daylight Savings rule to the specified string which must be in either the Fixed-Date format or the Occurrence-Of-Day format.  
  | **CLKMGR_SET_START_DAYLIGHTSAVINGS_RULE** (CONSTANT CHAR RECORD[])  
  The function returns a negative SLONG value if it encounters an error.  
  The Fixed-Date rules have the form:  
  "fixed:DAY,MONTH,HH:MM:SS"  
  with all fields as numeric except for the word "fixed"  
  (e.g.: "fixed:21,3,02:00:00" ===> March 21 @ 02:00:00AM).  
  The Occurrence-Of-Day rules have the form:  
  "occurrence:OCCURRENCE,DAY-OF-WEEK,MONTH,HH:MM:SS"  
  with all fields as numeric except for the word "occurrence"  
  DAY-OF-WEEK translates as:  
  • 1=Sunday  
  • 2=Monday  
  • 3=Tuesday  
  • 4=Wednesday  
  • 5=Thursday  
  • 6=Friday  
  • 7=Saturday  
  (e.g.: "occurrence:3,1,10,02:00:00" ===> 3rd Sunday in October @ 02:00:00AM).  
  |  |
| **CLKMGR_SET_TIMEZONE** | Sets the Timezone.  
  | **CLKMGR_SET_TIMEZONE** (CONSTANT CHAR TIMEZONE[])  
  Input string must have the correct format:  
  UTC[±]-HH:MM |  |
Combine & Uncombine Keywords

Overview

The Axcess language supports the concept of combining several panels to make them behave as if they were one panel, in order to simplify code. This feature allows the combination of functionally identical devices, such as identically programmed Touch Panels and Softwire Panels. When the program references one of these devices, all other combined device arrays are also referenced. In Axcess, device combine operations are done in the DEFINE_COMBINE section of the code, and can produce mixed results (any time one or more panels are dropped off-line).

NetLinx further addresses the issues surrounding combining panels (and their associated channels and levels), and allows you to combine and un-combine panels on the fly. The primary difference between the way that the Axcess and NetLinx languages handle combine operations is that NetLinx utilizes the concept of the virtual device. A virtual device is a device that does not physically exist but merely represents one or more devices.

NOTE: If you have combined Devices, Levels and/or Channels, they must be un-combined before they can be added as part of a new COMBINE function.

Combining and Un-Combining Devices

To approach setting up combine and un-combine operations in NetLinx, let’s first look at the way that combine operations are done in the Axcess language.

Combining Devices

The example below illustrates how an Axcess program combines three touch panels to act as one.

```
DEFINE_DEVICE
TP1 = 128
TP2 = 129
TP3 = 130

DEFINE_COMBINE
(TP1, TP2, TP3)

DEFINE_PROGRAM
RELEASE[TP1,1]
{
  (*Do Something*)
}
```

NOTE: The code shown in the Axcess example will not work in NetLinx, due to incompatibilities between the languages (i.e. Axcess does not allow virtual devices, which are required for Combine/Uncombine operations in NetLinx).

This combines a common level to each of three devices TP1, TP2, and TP3. If an input change occurs on any of the three devices, Axcess sees the input as coming only from the first device in the list (TP1). If button [TP2,12] is pressed, Axcess will see the input coming from [TP1,12] due to the combination. Likewise, any output change sent to any device in the list will automatically be sent to all devices in the list. This includes level changes. For example, the statement ON [TP1,50] will turn on channel 50 for all three devices in the list.

Now let’s see how the code example shown above would translate into NetLinx:

```
DEFINE_COMBINE
DEFINE_DEVICE

VIRTUAL1 = 33000
TP1 = 128
TP2 = 129
TP3 = 130

DEFINE_COMBINE
(VIRTUAL1, TP1, TP2, TP3)

DEFINE_PROGRAM
RELEASE[VIRTUAL1,1]
{
  (*Do Something*)
}
```

Note the use of the virtual device (VIRTUAL1) in the above example. Combine operations in NetLinx require that the first device in the list (the primary device) must be a virtual device. By specifying a virtual device as the primary device in a DEFINE_COMBINE statement, NetLinx code can be written targeting the virtual device, but effectively operating on each physical device. Furthermore, since a virtual device is not an actual physical device, the primary device cannot be taken off-line or removed from the system (which avoids the potential problems that occurred in Axcess). The virtual device's address number must be in the range of 32768 to 36863.
The example above combines the three touch panel devices: TP1, TP2 and TP3. Whenever an input change occurs on any of the three devices, NetLinx detects the input as coming only from VIRTUAL1. For example, if button [TP3, 5] is pressed, NetLinx sees input coming from [VIRTUAL1, 5] as a result of the combination.

Output changes (including level changes) sent to any device in the list will automatically be sent to all devices in the list. For instance, the statement: ON [VIRTUAL1, 50] turns on channel 50 on all three panels and OFF [VIRTUAL1, 10] turns off channel 10 on all three panels.

The example below illustrates the use of a device array (Dev[ ]), instead of specifying the individual devices (TP1, TP2, and TP3). Device arrays can further simplify your code and allow you to dynamically combine/un-combine devices. Any input events for any device in the array will appear to the program as coming from the virtual device. Output changes, directed to the virtual device or any device in the array, are sent to all devices in the array. Here's a syntax example:

```
COMBINE_DEVICES (VIRTUAL1, TP1, TP2, TP3)
```

In addition to virtual devices and device arrays, NetLinx contains several new keywords for combine and un-combine operations:

- COMBINE_DEVICES, UNCOMBINE_DEVICES
- COMBINE_LEVELS, UNCOMBINE_LEVELS
- COMBINE_CHANNELS, UNCOMBINE_CHANNELS

**NOTE:** Refer to the Combining and Un-Combining Levels section on page 42 for more information.

**Un-combining Devices**

UNCOMBINE_DEVICES reverses the effect of COMBINE_DEVICES. All combines related to the specified virtual device are disabled. A syntax example is:

```
UNCOMBINE_DEVICES (VDC)
```

**Parameters:**

- **VDC** The virtual device-channel passed to COMBINE_DEVICES.

The following NetLinx code example illustrates combining and un-combining the panels from the previous example:

**NOTE:** Input and output changes occurring on non-combined panels will not affect combined panels, and vice versa.

```
DEFINE_DEVICE
VIRTUAL1 = 33000
TP1 = 128
TP2 = 129
TP3 = 130
TP4 = 131

DEFINE_PROGRAM
(* Activate dynamic device combine*)
RELEASE[TP4,1]
{
   COMBINE_DEVICES(VIRTUAL1, TP1, TP2, TP3)
}
(*Remove dynamic device combine*)
RELEASE[TP4,1]
{
   UNCOMBINE_DEVICES(VIRTUAL1)
}
(*Pushes come here when a combine is active*)

RELEASE[VIRTUAL1,1]
{
   (*Do Something*)
}
(*This will only see pushes when combine is NOT active*)
RELEASE[TP1,1]
{
   (*Do Something*)
}
```
Combining and Un-Combining Levels

To approach setting up level combine and un-combine operations in NetLinx, let's first look at the way that level combine operations are done in the Axcess language. The example below illustrates how an Axcess program would combine three Touch Panel levels to act as one.

**NOTE:** The code shown in the Axcess example will not work in NetLinx, due to incompatibilities between the languages (i.e. Axcess does not allow virtual devices, which are required for Combine/Uncombine operations in NetLinx).

```
DEFINE_DEVICE
TP1 = 128
TP2 = 129
TP3 = 130

DEFINE_CONNECT_LEVEL
(TP1,1, TP2,1, TP3,1)
```

TP1, TP2, and TP3 are devices; this example combines Level 1 on each device. If a level change occurs on any of the three devices, Axcess sees the level coming only from the first device in the list (TP1). Likewise, any level change sent to any device in the list will automatically be sent to all devices in the list. Now let's see how the code example shown above would translate into NetLinx. This is code that would function correctly within a NetLinx system, but still uses the Axcess-based.

```
DEFINE_CONNECT_LEVEL
DEFINE_DEVICE
VIRTUAL1 = 33000
TP1 = 128
TP2 = 129
TP3 = 130

DEFINE_CONNECT_LEVEL
(VIRTUAL1, 1, TP1,1, TP2,1, TP3,1)
```

The example above combines the levels for the three touch panels: TP1, TP2, and TP3. Whenever a level change occurs on any of the three devices, NetLinx detects the level as coming only from VIRTUAL1.

The example below illustrates the use of a device array (`Dev[ ]`), instead of specifying the individual devices (`TP1`, `TP2`, and `TP3`). Device arrays further simplify code and allow you to dynamically combine/un-combine levels. Any input events for any device in the array will appear to the program as coming from the virtual device. Output changes, directed to the virtual device or any device in the array, are sent to all devices in the array. The syntax must follow one of these two forms:

```
DEFINE_CONNECT_LEVEL
(Vdevice1, 1, DEVLEV [ ])
- or -
DEFINE_CONNECT_LEVEL
(VDEVLEV, DEVLEV [ ])
```

**Combining Levels**

`COMBINE_LEVELS` connects a single device-level array (`DEVLEV[ ]`) to a DEVLEV array. Any element in a DEVLEV array appears to come from the virtual device-level representing the group, and output to any element in a DEVLEV array is directed to all elements in the group. Here’s a syntax example:

```
COMBINE_LEVELS (DEVLEV VDLSET, DEVLEV[ ] DLSETS)
```

Parameters:
- **VDLSET** Virtual device-level. Each element will represent one device-level combine group.
- **DLSETS** Device-level sets containing the device-level pairs to combine. Corresponding elements in each set are combined with the corresponding element in the virtual device-level array.

**Un-combining Levels**

`UNCOMBINE_LEVELS` undoes the effect of `COMBINE_LEVELS`. All combines related to the specified virtual device-level are disabled.

**NOTE:** Input and output changes occurring on non-combined panels will not affect combined panels, and vice versa.

```
UNCOMBINE_LEVELS (DEVLEV)
```

Parameters:
- **VDL** The virtual device-level passed to `COMBINE_LEVELS`.
- **DEVLEV** The device-level passed to `COMBINE_LEVELS`.

The NetLinx code example below illustrates how to dynamically combine and un-combine levels.

```
DEFINE_DEVICE
VIRTUAL1 = 33000
TP1 = 128
TP2 = 129
TP3 = 130
TP4 = 131
```
DEFINE_PROGRAM
  (*Activate dynamic level combine*)
RELEASE[TP4,1]
  {
    COMBINE_LEVELS(VIRTUAL1,1,TP1,1,TP2,1,TP3,1)
  }
  (*Remove dynamic level combine*)
RELEASE[TP4,1]
  {
    UNCOMBINE_LEVELS(VIRTUAL1,1)
  }

Combining and Un-Combining Channels

Combining Channels

COMBINE_CHANNELS connects a single virtual device-channel to one or more channels on another device (or devices). Stated another way, COMBINE_CHANNELS combines a single virtual DEVCHAN or [DEV,CHAN] pair to one or more DEVCHANS or [DEV,CHAN] pairs. Any element in a DEVCHAN[ ] set combined appears to come from the virtual device-channel representing the group, and output to the virtual device-channel is directed to all elements in the DEVCHAN[] set.

COMBINE_CHANNELS (DEVCHAN VDC, DEVCHAN[ | DCSets)

Parameters:

- **VDC**: Virtual device-channel that represents one device-channel combine group.
- **DCSets**: Device-channel array containing the device-channel pairs to combine. The VDC is combined with each element in the device-channel array.

Un-combining Channels

UNCOMBINE_CHANNELS reverses the effect of COMBINE_CHANNELS. All combines related to the specified virtual device-channel are disabled.

UNCOMBINE_CHANNELS (DEVCHAN VDC)

Parameters:

- **VDC**: The virtual device-channel passed to COMBINE_CHANNELS.

NOTE: When using COMBINE_XXXX and UNCOMBINE_XXXX functions dynamically based upon a button event, the combining and combining must be done on the release of the button (the active event must be complete before a COMBINE_XXXX or UNCOMBINE_XXXX function is invoked).

The examples in the program below demonstrate the use of COMBINE_CHANNELS and UNCOMBINE_CHANNELS:

PROGRAM_NAME='CombineChannelsExample'
DEFINE_DEVICE               // common devices for all examples below
dvTP = 128:1:0
dvREL10 = 301:1:0
dvIO10 = 310:1:0
dvControl = 33000:1:0
// example of combining a DEVCHAN set to a virtual [DEV,CHAN] pair
DEFINE_VARIABLE
DEVCHAN dc1[] = {{dvIO10,1},{dvREL10,1},{dvTP,1}}
DEFINE_EVENT
BUTTON_EVENT[dvTP,11]      // COMBINE_CHANNELS 1
  {
    RELEASE:
      {
        COMBINE_CHANNELS (dvControl,1,dc1)
      }
  }

BUTTON_EVENT[dvTP,12]      // UNCOMBINE_CHANNELS 1
  {
    RELEASE:
      {
        UNCOMBINE_CHANNELS (dvControl,1)
      }
  }

BUTTON_EVENT[dvControl,1] // this will work when the COMBINE_CHANNELS above is invoked
  {
    PUSH:
      {
        TO[BUTTON.INPUT]
      }
  }
// example of combining individual DEVCHANs to a virtual [DEV,CHAN] pair
DEFINE_VARIABLE
DEVCHAN dc2[] = {{dvIO10,2},{dvREL10,2},{dvTP,2}}
DEFINE_EVENT
BUTTON_EVENT[dvTP,13] // COMBINE_CHANNELS 2
{
    RELEASE:
    {
        COMBINE_CHANNELS (vdvControl,2,dc2[1],dc2[2],dc2[3])
    }
}
BUTTON_EVENT[dvTP,14] // UNCOMBINE_CHANNELS 2
{
    RELEASE:
    {
        UNCOMBINE_CHANNELS (vdvControl,2)
    }
}
BUTTON_EVENT[vdvControl,2] // this will work when the COMBINE_CHANNELS above is invoked
{
    PUSH:
    {
        TO[BUTTON.INPUT]
    }
}
// example of combining individual [DEV,CHAN] pairs to a virtual [DEV,CHAN] pair
DEFINE_VARIABLE
DEVCHAN dc3[] = {{dvIO10,3},{dvREL10,3},{dvTP,3}}
DEFINE_EVENT
BUTTON_EVENT[dvTP,15] // COMBINE_CHANNELS 3
{
    RELEASE:
    {
        COMBINE_CHANNELS (vdvControl,3,
                        dc3[1].DEVICE,
                        dc3[1].CHANNEL,
                        dc3[2].DEVICE,
                        dc3[2].CHANNEL,
                        dc3[3].DEVICE,
                        dc3[3].CHANNEL)
    }
}
BUTTON_EVENT[dvTP,16] // UNCOMBINE_CHANNELS 3
{
    RELEASE:
    {
        UNCOMBINE_CHANNELS (vdvControl,3)
    }
}
BUTTON_EVENT[vdvControl,3] // this will work when the COMBINE_CHANNELS above is invoked
{
    PUSH:
    {
        TO[BUTTON.INPUT]
    }
}
// example of combining a DEVCHAN set to a virtual DEVCHAN
DEFINE_VARIABLE
DEVCHAN vdc4 = {vdvControl,4}
DEVCHAN dc4[] = {{dvIO10,4},{dvREL10,4},{dvTP,4}}
DEFINE_EVENT
BUTTON_EVENT[dvTP,17] // COMBINE_CHANNELS 4
{
    RELEASE:
    {
        COMBINE_CHANNELS (vdc4,dc4)
    }
}
BUTTON_EVENT[dvTP,18] // UNCOMBINE_CHANNELS 4
{
    RELEASE:
    {
        UNCOMBINE_CHANNELS (vdc4)
    }
}
BUTTON_EVENT[dvc4]  // this will work when the COMBINE_CHANNELS above is invoked
{
    PUSH:
    {
        TO[BUTTON.INPUT]
    }
}
// example of combining individual DEVCHANs to a virtual DEVCHAN
DEFINE_VARIABLE
DEVCHAN vdc5 = {vdvControl,5}
DEVCHAN dc5[] = {{dvIO10,5},{dvREL10,5},{dvTP,5}}
DEFINE_EVENT
BUTTON_EVENT[dvTP,19]  // COMBINE_CHANNELS 5
{
    RELEASE:
    {
        COMBINE_CHANNELS (vdc5,dc5[1],dc5[2],dc5[3])
    }
}
BUTTON_EVENT[dvTP,20]  // UNCOMBINE_CHANNELS 5
{
    RELEASE:
    {
        UNCOMBINE_CHANNELS (vdc5)
    }
}

BUTTON_EVENT[dvc5]  // this will work when the COMBINE_CHANNELS above is invoked
{
    PUSH:
    {
        TO[BUTTON.INPUT]
    }
}
// example of combining individual [DEV,CHAN] pairs to a virtual DEVCHAN
DEFINE_VARIABLE
DEVCHAN vdc6 = {vdvControl,6}
DEVCHAN dc6[] = {{dvIO10,6},{dvREL10,6},{dvTP,6}}
DEFINE_EVENT
BUTTON_EVENT[dvTP,21]  // COMBINE_CHANNELS 6
{
    RELEASE:
    {
        COMBINE_CHANNELS (vdc6,
        dc6[1].DEVICE,
        dc6[1].CHANNEL,
        dc6[2].DEVICE,
        dc6[2].CHANNEL,
        dc6[3].DEVICE,
        dc6[3].CHANNEL)
    }
}
BUTTON_EVENT[dvTP,16]  // UNCOMBINE_CHANNELS 6
{
    RELEASE:
    {
        UNCOMBINE_CHANNELS (vdc6)
    }
}
BUTTON_EVENT[dvc6]  // this will work when the COMBINE_CHANNELS above is invoked
{
    PUSH:
    {
        TO[BUTTON.INPUT]
    }
}
// end
COMBINE & UNCOMBINE Keywords

NetLinx supports the following COMBINE and UNCOMBINE keywords:

<table>
<thead>
<tr>
<th>COMBINE &amp; UNCOMBINE Keywords</th>
<th>Description</th>
</tr>
</thead>
</table>
| **COMBINE_CHANNELS** | This command connects a single virtual device-channel to one or more channels on another device (or devices). Any element in a DEVCHAN[] set appears to come from the virtual device-channel representing the group, and output to the virtual device-channel is directed to all elements in the DEVCHAN[] set.  
  **Example:**  
  `COMBINE_CHANNELS (DEVCHAN VDC, DEVCHAN[ ] DCSets)`  
  **Parameters:**  
  - VDC: Virtual device-channel that represents one device-channel combine group.  
  - DCSets: Device-channel array containing the device-channel pairs to combine. Each element in each set is combined with the virtual device-channel. |
| **COMBINE_DEVICES** | This keyword defines the combination of functionally identical devices, such as identically programmed touch panels. When the program references one of these devices, all other combined devices in the array are also referenced. The devices in a given array must be enclosed in parentheses. A virtual device is one that does not actually exist but merely represents one or more physical devices.  
  The first device in the list (the primary device) must be a virtual device. By specifying a virtual device as the primary device, NetLinx code can target the virtual device but have the effect of operating on each physical device. Furthermore, since a virtual device is not an actual physical device, the primary device cannot be taken off-line or removed from the system. An example of virtual devices is shown below:  
  **Example:**  
  `COMBINE_DEVICES (VIRTUAL1, TP1, TP2, TP3)`  
  **The example above combines the three touch panel devices: TP1, TP2 and TP3. Whenever an input change occurs on any of the three devices, NetLinx detects the input as coming only from VIRTUAL1. For example, if button [TP3, 5] is pressed, NetLinx sees input coming from [VIRTUAL1, 5] as a result of the combination. Output changes (including level changes) sent to any device in the list will automatically be sent to all devices in the list. For instance, ON(VIRTUAL1, 50) will turn on channel 50 on all three panels and OFF(VIRTUAL1, 10) will turn off channel 10 on all three panels.**  
  **The example below is equivalent to the first except that it uses a device array (Dev[ ]) instead of specifying the individual devices (TP1, TP2, and TP3). Any input events for any device in the array will appear to the program as coming from the virtual device. Output changes directed to the virtual device or any device in the array are sent to all devices in the array.**  
  **Example:**  
  `COMBINE_DEVICES (VIRTUAL1, Dev[ ])`  
  When using a device array, the array can be manipulated at run-time to add or remove devices. A device that is added to the array is combined with the others and a device that is removed is uncombined.  
  The process of adding or removing devices does not require the system to be powered down and restarted. |
| **COMBINE_LEVELS** | This keyword connects a single device-level array (DEVLEV[ ]) to a DEVLEV array. Any element in a DEVLEV array appears to come from the virtual device-level representing the group, and output to any element in a DEVLEV array is directed to all elements in the group.  
  **Example:**  
  `COMBINE_LEVELS (DEVLEV VDLSET, DEVLEV[ ] DLSETS)`  
  **Parameters:**  
  - VDLSET: Virtual device-level sets; each element represents one device-level combine group.  
  - DLSETS: Device-level sets containing the device-level pairs to combine. Corresponding elements in each set are combined with the corresponding element in the virtual device-level array. |
| **DEFINE_COMBINE** | This keyword defines the combination of functionally identical devices, such as identically programmed touch panels. When the program references one of these devices, all other combined devices are also referenced. The devices in a given combine must be enclosed in parentheses. The first device in the list (the primary device) must be a virtual device.  
  **Example:**  
  `DEFINE_COMBINE(VDevice, Panel1, Panel2, Panel3)`  
  The example below uses a device array (Dev[ ]) instead of specifying the individual devices (Panel1, Panel2, and Panel3). Any input events for any device in the array will appear to the program as coming from the virtual device. Output changes directed to the virtual device or any device in the array is sent to all devices in the array.  
  **Example:**  
  `DEFINE_COMBINE(VDevice, Dev[ ])`  
  See the *Combine & Uncombine Keywords* on page 40 for more information on virtual devices and device arrays. |
| **DEFINE_CONNECT_LEVEL** | This keyword defines level connections. A single connection is defined by listing the device-level pairs inside parentheses. The first level in the list (the primary level) must be a virtual level (a level on a virtual device). A virtual level does not actually exist but merely represents one or more levels on physical devices.  
  The example above combines the levels [Device1, Level1] and [Device2, Level2].  
  `(Device1, Level1, Device1, Level1, Device2, Level1)`  
  The next example combines all levels in the device-level array. Changes to any level listed in the connection will automatically be reflected in the other levels so that all level values are the same.  
  **Example:**  
  `DEFINE_CONNECT_LEVEL(VDevLev, MyDL[ ])`  
  By specifying a virtual level as the primary level, NetLinx code targets the virtual level but operates on each physical level. Since the primary level is virtual, the primary device (a virtual device) cannot be taken off-line or removed from the system. |
| **UNCOMBINE_CHANNELS** | This keyword reverses the effect of COMBINE_CHANNELS. All combines related to the specified virtual device-channel are disabled.  
  **Example:**  
  `UNCOMBINE_CHANNELS VDC`  
  **Parameters:**  
  - VDC: The virtual device-channel passed to COMBINE_CHANNELS. |
<table>
<thead>
<tr>
<th><strong>COMBINE &amp; UNCOMBINE Keywords (Cont.)</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UNCOMBINE_DEVICES</strong></td>
<td>This keyword reverses the effect of COMBINE_DEVICES. All combines related to the specified virtual device are disabled.</td>
</tr>
<tr>
<td></td>
<td><strong>UNCOMBINE_DEVICES VD</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Parameters:</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>VD</strong>: The virtual device passed to COMBINE_DEVICES.</td>
</tr>
<tr>
<td><strong>UNCOMBINE_LEVELS</strong></td>
<td>This keyword reverses the effect of COMBINE_LEVELS. All combines related to the specified virtual device-level are disabled.</td>
</tr>
<tr>
<td></td>
<td><strong>UNCOMBINE_LEVELS VDL</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Parameters:</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>VDL</strong>: The virtual device-channel passed to COMBINE_LEVELS.</td>
</tr>
</tbody>
</table>
## Compiler Keywords

NetLinx supports the following Compiler keywords:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DATE</strong></td>
<td>At compile time, this keyword is replaced by a string (mm/dd/yy) containing the date of compilation. The example below sends the date of compilation to a variable text button on a touch panel.</td>
<td>SEND_COMMAND TP, &quot;'!T',1,<strong>DATE</strong>&quot;</td>
</tr>
<tr>
<td><strong>FILE</strong></td>
<td>At compile time, this keyword is replaced with a string that contains the filename of the currently executing program file.</td>
<td></td>
</tr>
<tr>
<td><strong>LDATE</strong></td>
<td>At compile time, this keyword is replaced by a string (mm/dd/yyyy), containing the date of compilation. The example below sends the date of compilation to a variable text button on a touch panel.</td>
<td>SEND_COMMAND TP, &quot;'!T',1,<strong>LDATE</strong>&quot;</td>
</tr>
<tr>
<td><strong>LINE</strong></td>
<td>At compile time, this keyword is replaced by a constant that contains the line number the keyword is on.</td>
<td>SEND_STRING 0,&quot;ITOA(<strong>LINE</strong>)&quot;</td>
</tr>
<tr>
<td><strong>NAME</strong></td>
<td>At compile time, this keyword is replaced by a string that contains the PROGRAM_NAME description found on the first line of the program.</td>
<td></td>
</tr>
<tr>
<td><strong>TIME</strong></td>
<td>At compile time, this keyword is replaced by a string (hh:mm:ss) representing the time of compilation. The example below sends the time of compilation to a variable text button on a touch panel.</td>
<td>SEND_COMMAND TP, &quot;'!T',1,<strong>TIME</strong>&quot;</td>
</tr>
</tbody>
</table>

See page 48 for a listing of Compiler Keywords. Refer to the Appendix A - Compiler Warning & Errors section on page 156 for a listing of Compiler Messages.
Conditional & Loop Keywords

Overview

NetLinx supports the following types of conditional statements and loops:

- Conditional statements:
  - IF...ELSE statements
  - SELECT...ACTIVE statements
  - SWITCH...CASE statements
- Loops:
  - FOR statements
  - WHILE statements
  - LONG_WHILE statements

MEDIUM_WHILE statements are obsolete in NetLinx due to eliminating the timeout of WHILE loops. LONG_WHILE loops now differ from WHILE loops in the way input change notifications are processed during the programming loop. WHILE, MEDIUM_WHILE and LONG_WHILE statements are all accepted syntax to provide compatibility with existing Axcess programs.

Conditionals

IF...ELSE

The IF...ELSE statement provides a structure for conditional branching of program execution. If a condition evaluates to true, the statement(s) associated with it are executed; otherwise, statements are not executed. An example is:

```
IF (<conditional expression 1>)
{
(* statements for condition 1 *)
}
ELSE IF (<conditional expression 2>)
{
(* statements for condition 2 *)
}
ELSE
{
(* statements for all other conditions *)
}
```

Regarding IF statements:

- ELSE IF is optional.
- Braces are generally recommended in all cases but are only required if multiple statements are assigned to a given condition.
- IF statements may be nested to any number of levels.

SELECT...ACTIVE

The SELECT...ACTIVE statement provides a programming structure for selective execution of code blocks based on the evaluation of a series of conditions. The first block whose ACTIVE condition evaluates to true is executed; the remaining blocks are ignored. If no ACTIVE condition evaluates to true, no statements are executed. An example is:

```
SELECT
{
ACTIVE (<condition 1>) :
{
(* statements for condition 1*)
}
ACTIVE (<condition 2>) :
{
(* statements for condition 2*)
}
ACTIVE (<condition n>) :
{
ACTIVE (1)
(* statements for condition n*)
}
}
```

Regarding SELECT...ACTIVE statements:

- Only the statements associated with the first condition evaluated to true are executed.
- If no condition evaluates to true, no statements are executed.
- Braces underneath individual ACTIVE statements are required only if multiple statements are assigned to a given condition.
### SWITCH...CASE Statements

SWITCH...CASE statements provide selective execution of code blocks evaluated by a single condition. The value of the SWITCH expression is tested against each CASE value (which must be a numeric constant or a string literal). If a match is found, the statements associated with the CASE are executed. All other CASE statements are ignored. If no match is found, the DEFAULT case statements (if any) are executed. The SWITCH expression is evaluated only once. The following is the structure for the SWITCH...CASE statement:

```plaintext
SWITCH (<expression>)
{
    CASE <numeric constant or string literal>:
    {
        (* statements for CASE 1 *)
    }
    CASE <numeric constant or string literal>:
    {
        (* statements for CASE 2 *)
    }
    CASE <numeric constant or string literal>:
    {
        (* statements for CASE n; there can be as many cases as necessary *)
    }
    DEFAULT <numeric constant or string literal>:
    {
        (* statements for DEFAULT case *)
    }
}
```

The following rules apply to SWITCH...CASE statements:

- Only the statements associated with the first case that matches the value of the expression are executed.
- Multiple CASE statements can be stacked within the SWITCH...CASE statement.
- If the value matches one of the CASE statements, the statements associated with the stack will be executed.
- If no CASE matches the SWITCH expression, then the statements under the default case (if available) are executed. The default statement must be the last case within the SWITCH...CASE, otherwise the remaining case statements will not execute.
- All cases must be unique.
- Braces should be used to bracket the statements in a case. They are required only if variables are declared within the case.
- The BREAK statement applies to the SWITCH and takes execution to the end of the SWITCH. Unlike C and C++, cases do not fall through to the next case if a break is not used. Because of this, BREAK statements are not required between cases.

For example:

```plaintext
SWITCH (var)
{
    CASE 1:
    {
        (*statements go here*)
    }
    BREAK
    CASE 3:
    {
        (*statements go here*)
    }
    BREAK
    CASE 5:
    {
        (*statements go here*)
    }
    BREAK
    DEFAULT:
    {
        (*statements go here*)
    }
    BREAK
}
```

### Loops

#### FOR Loops

The FOR loop structure allows you to define initialization statements; statements to execute after each pass through the loop and a condition to test after each pass. If the condition evaluates to true, another pass is made. Otherwise, the loop is terminated. The syntax of the FOR loop is as follows:

```plaintext
FOR (<INITIAL>; <condition>; <after pass>)
{
    (* loop statements *)
}
```
Parameters:

```plaintext
<INITIAL>      One or more statements that are executed one time before any FOR loop statements are executed. Each statement must be separated with a comma; this is typically a FOR loop index initialization statement.

<condition>    A condition whose value is computed before each pass. If the condition evaluates to TRUE, the FOR loop statements are executed. If the condition evaluates to FALSE, the loop is terminated.

<after pass>   One or more statements that are executed after each pass through the statements. Each statement must be separated with a comma. This is typically a statement that increments the FOR loop index.
```

The number of loop executions is usually stated at the beginning of the loop, unlike WHILE and LONG_WHILE loops:

```plaintext
FOR (COUNT=0 ; COUNT<10 ; COUNT++)
{
 (* loop statements *)
}
```

By defining the loop like this, you clearly see how it is initialized and incremented. No errors appear if you forget to initialize the WHILE loop or counter. The FOR loop helps to insure proper structure.

### WHILE Loops

A WHILE statement executes its statement block as long as its associated condition evaluates to true. The condition is evaluated before the first pass through the statements. Therefore, if the conditional expression is never true, the conditional statements are never executed.

The WHILE Loop structure:

```plaintext
WHILE (<conditional expression>)
{
 (* conditional statements *)
}
```

Regarding WHILE statements:

- Statements are executed repeatedly while the conditional expression evaluates to true.
- The condition is tested before each pass through the conditional statements.
- There is no timeout period - the NetLinx Controller handles bus updates through a separate execution thread, thereby eliminating this potential problem.

### LONG_WHILE statements

A LONG_WHILE differs from a WHILE statement in the way input change notifications are processed during the programming loop. The system checks the input queue for a change notification message before execution of each loop, beginning with the second loop. The message is retrieved if one exists. This message must be processed before another one is retrieved, either at the start of the next loop or the beginning of the next mainline iteration. Otherwise, the message is lost.

The LONG_WHILE Loop structure:

```plaintext
LONG_WHILE (<conditional expression>)
{
 (* conditional statements *)
}
```

- DEFINE_EVENT events are still processed even if mainline is in a LONG_WHILE.
- Special care should be taken to avoid spawning concurrent LONG_WHILEs via DEFINE_EVENT code. This can cause excessive drag on system resources.

### Conditional and Loop Keywords

NetLinx supports the following Conditional and Loop keywords:

<table>
<thead>
<tr>
<th>Conditional and Loop Keywords</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BREAK</strong></td>
<td>The BREAK command terminates execution of the current WHILE, LONG_WHILE, or FOR loop and resumes program execution at the first instruction following that loop. BREAK also jumps to the end of a SWITCH statement.</td>
</tr>
<tr>
<td><strong>DEFAULT</strong></td>
<td>Specifies the default case in a SWITCH...CASE statement. See SWITCH...CASE on page 52.</td>
</tr>
<tr>
<td><strong>ELSE</strong></td>
<td>If the corresponding IF statement is false, the program will jump to the ELSE section of the IF...ELSE set of statements.</td>
</tr>
</tbody>
</table>

Conditional and Loop Keywords

The BREAK command terminates execution of the current WHILE, LONG_WHILE, or FOR loop and resumes program execution at the first instruction following that loop. BREAK also jumps to the end of a SWITCH statement.

```plaintext
WHILE (<condition>)
{
  // statements
  IF (<condition>)
  {
    BREAK // Go to statement:  X = X + 1
  }
}
// Execution continues here after BREAK or
// after normal completion of the WHILE loop.
X = X + 1
```
Conditional and Loop Keywords (Cont.)

FOR

This keyword defines a FOR loop. The looping structure allows you to define initialization statements, statements to execute after each pass through the loop and a condition to test after each pass. If the condition evaluates to true, another pass is made; otherwise the loop is terminated.

FOR (<initial>; <condition>; <after pass>)
{
    (* for loop statements *)
}

IF

This keyword defines an IF statement; the IF statement provides conditional branching of program execution.

IF (<expression>)
{
    // statements
}
ELSE IF (<expression>)
{
    // statements
}
ELSE
{
    // statements
}

The ELSE IF and ELSE statements are optional. The braces delimiting the statements, associated with each condition, are required only if there is more than one statement. For example, the following syntax is correct:

IF (X > 0)
X = X - 1

Example:

IF (<conditional expression 1>)
{
    (* statements for condition 1 *)
}
ELSE IF (<conditional expression 2>)
{
    (* statements for condition 2 *)
}
ELSE
{
    (* statements for all other conditions *)
}

Regarding IF statements:
- ELSE IF and ELSE are optional
- Braces are only required if multiple statements are assigned to a given condition but are generally recommended in all cases
- IF statements may be nested to any number of levels

INCLUDE

This keyword allows you to include programming instructions from an external file and have those instructions inserted at any point in the program.

INCLUDE '<filename>'

The parameter filename can be any valid (long) filename. If the file extension is omitted, "AXI" is assumed. An INCLUDE statement can appear anywhere in a program.

NOTE: There is no difference in functionality between the INCLUDE reserved identifier and the #INCLUDE compiler directive. INCLUDE is supported for backward-compatibility to Axcess (see #INCLUDE on page 29).

SELECT...ACTIVE

This keyword statement provides a programming construct for selective execution of code blocks based on the evaluation of a series of conditions.

SWITCH...CASE

This keyword statement provides a programming construct for selective execution of code blocks based on the evaluation of a single condition.

SWITCH (var)
{
    CASE 1:
        IF(Var2=1)
            BREAK // IF Var2=1 STOP EXECUTION
        (statements go here if Var2 < > 1)
    CASE 3:
        (statements go here)
    CASE 5:
        (statements go here)
    DEFAULT:
        (statements go here)
}

<table>
<thead>
<tr>
<th>Conditional and Loop Keywords (Cont.)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>WHILE</strong></td>
</tr>
<tr>
<td>This keyword executes its statement block as long as its associated condition evaluates to true. The condition is evaluated before the first pass through the statements. Therefore, if the conditional expression is never true the conditional statements will never be executed.</td>
</tr>
<tr>
<td><strong>WHILE</strong> (&lt;conditional expression&gt;)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>/* conditional statements */</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td><strong>MEDIUM_WHILE</strong></td>
</tr>
<tr>
<td>This keyword is obsolete in the new NetLinx system. The compiler will treat it as a WHILE keyword.</td>
</tr>
<tr>
<td><strong>LONG_WHILE</strong></td>
</tr>
<tr>
<td>This keyword is the same as a WHILE statement except that input messages are retrieved after each pass to allow the LONG_WHILE statements to process the input.</td>
</tr>
<tr>
<td><strong>LONG_WHILE</strong> (&lt;conditional expression&gt;)</td>
</tr>
<tr>
<td>{ /* conditional statements */ }</td>
</tr>
<tr>
<td>See the <strong>LONG_WHILE statements</strong> section on page 51 for more information.</td>
</tr>
<tr>
<td><strong>FALSE</strong></td>
</tr>
<tr>
<td>This keyword is a <em>CHAR</em> constant containing the value 0. While NetLinx does not support a BOOLEAN data type, zero is considered <em>false</em> conditional expressions.</td>
</tr>
<tr>
<td><strong>TRUE</strong></td>
</tr>
<tr>
<td>This keyword is a <em>CHAR</em> constant and contains the value 1. While NetLinx does not support a BOOLEAN data type, an non-zero value is considered <em>true</em> for conditional expressions.</td>
</tr>
</tbody>
</table>
Data Event Keywords

NetLinx supports the following Data Event keywords:

<table>
<thead>
<tr>
<th>Data Event Keywords</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AWAKE</strong></td>
<td>This keyword defines a section in a DATA event handler for processing AWAKE notifications. This event is triggered when the master recognizes that a device on the bus has exited STANDBY state. Once the device is AWAKE, communication to the device from the master can resume.</td>
</tr>
<tr>
<td><strong>COMMAND</strong></td>
<td>This keyword defines a section in a DATA event handler for processing SEND_COMMAND instructions.</td>
</tr>
<tr>
<td><strong>HOLD</strong></td>
<td>This keyword defines a section in a BUTTON event handler for processing HOLD events.</td>
</tr>
<tr>
<td><strong>ONERROR</strong></td>
<td>This keyword defines a section in a DATA event handler for processing ONERROR notifications. Any error triggers an ONERROR event.</td>
</tr>
<tr>
<td><strong>OFFLINE</strong></td>
<td>This keyword defines a section in a DATA event handler for processing OFFLINE notifications. This is one of the important aspects of the DATA_EVENT that is triggered when the master recognizes that a device has been dropped off the bus.</td>
</tr>
<tr>
<td><strong>ONLINE</strong></td>
<td>This keyword defines a section in a DATA event handler for processing ONLINE notifications. This is one aspect of DATA_EVENT that is triggered when the master recognizes that a device has been added to the bus. In NetLinx, every device triggers an ONLINE event when the master is reset. This ensures that the device is initialized on startup and that the device is initialized any time the device comes online.</td>
</tr>
<tr>
<td><strong>REPEAT</strong></td>
<td>This keyword may be used with the HOLD keyword to specify that the (hold button) event should be allowed to repeat. See the Event Handler Keywords on page 74 for more information.</td>
</tr>
<tr>
<td><strong>STANDBY</strong></td>
<td>This keyword defines a section in a DATA event handler for processing STANDBY notifications. This event is triggered when the master recognizes that a device on the bus has gone into a STANDBY state. While in standby state, all communication to the device from the master is dropped.</td>
</tr>
</tbody>
</table>
Data Types and Conversion Keywords

Overview
NetLinx supports integers up to 32-bits and signed values to allow positive and negative values.

Intrinsic Data Types
The following Data Types are intrinsic to the NetLinx language:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Data Type</th>
<th>Sign</th>
<th>Size</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>Byte</td>
<td>Unsigned</td>
<td>8-bit</td>
<td>0 to 255</td>
</tr>
<tr>
<td>WIDECHAR</td>
<td>Integer</td>
<td>Unsigned</td>
<td>16-bit</td>
<td>0 to 65,535</td>
</tr>
<tr>
<td>INTEGER</td>
<td>Integer</td>
<td>Unsigned</td>
<td>16-bit</td>
<td>0 to 65,535</td>
</tr>
<tr>
<td>SINTEGER</td>
<td>Integer</td>
<td>Signed</td>
<td>16-bit</td>
<td>-32,768 to 32,767</td>
</tr>
<tr>
<td>LONG</td>
<td>Long Integer</td>
<td>Unsigned</td>
<td>32-bit</td>
<td>0 to 4,294,967,295</td>
</tr>
<tr>
<td>SLONG</td>
<td>Long Integer</td>
<td>Signed</td>
<td>32-bit</td>
<td>-2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>FLOAT</td>
<td>Floating Point</td>
<td>Signed</td>
<td>32-bit</td>
<td>-3.40282346e+38 to 3.40282346e+38</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>Double Precision Floating Point</td>
<td>Signed</td>
<td>64-bit</td>
<td>-1.79769313e+308 to 1.79769313e+308</td>
</tr>
</tbody>
</table>

Intrinsic Data Type Keywords

CHAR
CHAR is used to store single-byte values and character strings. This data type is used with ANSI character strings.
- Data Type: Byte
- Sign: Unsigned
- Size: 8-bit
- Range: 0 - 255
- Sample of Stored Values: 'a', 145, $FE, 'The quick gray fox'

WIDECHAR
WIDECHAR is an intrinsic data type representing a 16-bit unsigned integer. This data type is intended for use with Unicode fonts that use 16-bit character codes (and most Far-eastern fonts).
- Data Type: Integer
- Sign: Unsigned
- Size: 16-bit
- Range: 0 to 65,535
- Sample of Stored Values: "OFF", 500

INTEGER
INTEGER is the default variable value to store values up to and including 65535. This is the default data type if a non-array variable is declared without a data type specified.
- Data Type: Integer
- Sign: Unsigned
- Size: 16-bit
- Range: 0 to 65,535
- Sample of Stored Values: 512, 32468, 12

SINTEGER
SINTEGER is used to store signed integer values both greater than and less than zero.
- Data Type: Integer
- Sign: Signed
- Size: 16-bit
- Range: -32,768 to 32,767
- Sample of Stored Values: 24, -24, 568, -568

LONG
LONG defines an intrinsic data type representing a 32-bit unsigned value. It is used to store large integer values greater than 65535.
- Data Type: Long Integer
- Sign: Unsigned
- Size: 32-bit
- Range: 0 - 4,294,967,295
- Sample of Stored Values: 1000000, 2000046

SLONG
SLONG defines an intrinsic data type representing a 32-bit signed integer. It is used to store signed large integer values less than -32767 and greater than 32767.
- Data Type: Long Integer
- Sign: Signed
- Size: 32-bit
- Range: -2,147,483,648 to 2,147,483,647
- Sample of Stored Values: -1000000, 1000000, -2000000, 2000000
### Structured Data Types

#### Structured Data Type Keywords

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Definition</th>
<th>Example</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEV</td>
<td>A data type (structure) used to represent a specific device, port, and system. In NetLinx, the DEV structure is the actual (internal) representation of a NetLinx device.</td>
<td><code>{ INTEGER Number; INTEGER Port; INTEGER System; }</code></td>
<td>See the Device Arrays section on page 23 for more information.</td>
</tr>
<tr>
<td>DEVCHAN</td>
<td>DEVCHAN defines a data type (structure) containing fields used to represent a specific device number, port, system, and channel.</td>
<td><code>{ DEV Device; INTEGER Channel; }</code></td>
<td>See the Device-Channels and Device-Channel Arrays section on page 24 for more information.</td>
</tr>
<tr>
<td>DEVLEV</td>
<td>A data type (structure) containing fields used to represent a specific device number, port, system and level. This structure is used to implement an array DEVLEV[].</td>
<td><code>{ DEV Device; INTEGER Level; }</code></td>
<td>See the Device-Level Arrays section on page 24 for more information.</td>
</tr>
</tbody>
</table>

#### Combining and Uncombining Device/Channel Sets

Combining DEVCHAN sets is unique to NetLinx. The format for `COMBINE_CHANNELS` and `UNCOMBINE_CHANNELS` is:

```plaintext
SLONG COMBINE_CHANNELS (<virtual DEVCHAN[]>, <DEVCHAN1[]>, <DEVCHAN2[]>...)  
SLONG UNCOMBINE_CHANNELS (<virtual DEVCHAN[]>)  
```

To explain the concept of combining DEVCHAN sets, it is necessary to understand how the DEVCHAN sets are arranged. Rather than the DEVCHAN set being a set of like functions, such as a set of volume mute buttons across different devices, the DEVCHAN set should be a group of different functions on the same device, such as 5 lighting presets on an AXU-MSP16. For example:

```
DEVCHAN dcMSP1 = {{MSP1,PRESET1},{MSP1,PRESET2},{MSP1,PRESET3},{MSP1,PRESET4},{MSP1,PRESET5}}  
DEVCHAN dcMSP2 = {{MSP2,PRESET1},{MSP2,PRESET2},{MSP2,PRESET3},{MSP2,PRESET4},{MSP2,PRESET5}}  
DEVCHAN dcMSP3 = {{MSP3,PRESET1},{MSP3,PRESET2},{MSP3,PRESET3},{MSP3,PRESET4},{MSP3,PRESET5}}  
```

Similar to `COMBINE_DEVICES` and `COMBINE_LEVELS`, the first DEVCHAN set in a `COMBINE_CHANNEL` function needs to be referenced to a Virtual Device, as shown below:

```
DEVCHAN dcVDEV = {{VDEV,PRESET1},{VDEV,PRESET2},{VDEV,PRESET3},{VDEV,PRESET4},{VDEV,PRESET5}}  
```

All of the DEVCHAN sets in the `COMBINE_CHANNELS` function must have the same number of array elements. The actual `COMBINE_CHANNELS` statement is:

```plaintext
COMBINE_CHANNELS (dcVDEV, dcMSP1, dcMSP2, dcMSP3)  
```

The actual element positions of the DEVCHAN arrays are combined within the program. In essence, all of the PRESET1 channels are handled through `[VDEV,PRESET1]` defined as `dcVDEV[1]`.

---

### Intrinsic Data Type Keywords (Cont.)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Definition</th>
<th>Example</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOAT</td>
<td>FLOAT defines an intrinsic data type representing a 64-bit signed floating-point value. It is used to store small real numbers with 5 digits of precision.</td>
<td>-3.40282346e+38 to 3.40282346e+38</td>
<td>Sample of Stored Values: 1.2345, 123.4512345e5, -16.3231415</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>DOUBLE defines an intrinsic data type representing a 64-bit (double precision) signed floating-point value. It is used to store large real numbers with 15 digits of precision.</td>
<td>-1.79769313e+308 to 1.79769313e+308</td>
<td>Sample of Stored Values: 1.23456789012345, 12,345.678.9012545, 3.14159265358979, -0.048512934</td>
</tr>
</tbody>
</table>
NOTE: When using COMBINE_XXXX and UNCOMBINE_XXXX functions dynamically based upon a button event, the combining and combining must be done on the release of the button (the active event must be complete before a COMBINE_XXXX or UNCOMBINE_XXXX function is invoked).

Type Conversion

Although explicit type casting is not supported in NetLinx, the compiler is forced to do type conversion in situations where an arithmetic assignment or other operation is defined with constants and/or variables having mixed data types. Type conversions will occur under the following circumstances:

- A value of one type is assigned to a variable of another type.
- A value passed as a parameter to a subroutine does not match the declared parameter type.
- The value returned by a subroutine does not match the declared return type.

Type Conversion Rules

- If the expression contains a 32 or 64-bit floating-point variable or constant, all variables and constants in the expression are converted to 64-bit floating point before resolving.
- If the expression contains only whole number value variables and constants, all variables and constants in the expression are converted to 32-bit integers before resolving.
- If type conversion is required for an assignment or as a result of a parameter or return type mismatch, the value is converted to fit the type of the target variable. This may involve truncating the high order bytes(s) when converting to a smaller size variable, or sign conversion when converting signed values to unsigned or vice versa.

Conversion Keywords

NetLinx supports the following Conversion keywords:

<table>
<thead>
<tr>
<th>Conversion Keywords</th>
<th>Description</th>
</tr>
</thead>
</table>
| ATOI                | Converts a character representation of a number to a signed 32-bit integer. It recognizes a character representation of a value that would be within the ranges of the data types: INTEGER, SINTEGER, and SLONG. Syntax: \[\text{SLONG ATOI (CHAR STRING[ ])} \] Parameters: 
  - **STRING**: string containing the character representation of the integer. Valid input characters are "0" through "9" and the sign designators ("+" and "-"). If no valid characters are found, zero is returned as a result. Result: 
  - A 32-bit integer representing the converted string.
  - Any non-numeric characters in the string are ignored.
  - ATOI returns the value representing the first complete set of characters that represents an integer. **NOTE:** While you can pass in larger values, ATOI will truncate any value outside the range -2147483648 to 2147483647 to the value -2147483648 (if negative) or 2147483647 (if positive). Example: 
  
Vol = ATOI('Volume=100%') // Vol = 100
Num = ATOI('-3758')       // Num = -3758 |
| ATOF                | Converts a character representation of a number to a 64-bit floating-point value. ATOF recognizes a character representation of a value that would be within the ranges of all intrinsic data types, with the exception of CHAR. Syntax: \[\text{FLOAT ATOF (CHAR STRING[ ])} \] Parameters: 
  - **STRING**: An input string containing the character representation of the floating-point number. Valid input characters are "0" through "9", ".", the sign designators ("+" and "-"), and the exponent ("e" or "E"). If no valid characters are found, zero is returned as a result. Result: 
  - 64-bit floating-point number representing the converted string.
  - Any non-numeric characters in the string are ignored.
  - ATOF returns the value representing the first complete set of characters that represents a floating-point value. **NOTE:** When assigning the result to a DOUBLE the effective range is \[±2.22507E-1\] to \[±1.79769E+308\]. Example: 
  
Num = ATOF('-1.25e-3')// Num = -0.00125 |
## Conversion Keywords (Cont.)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Syntax</th>
<th>Parameters</th>
<th>Result</th>
</tr>
</thead>
</table>
| **ATOL** | ATOL converts a character representation of a number to a signed 32-bit integer. ATOL recognizes a character representation of a value that would be within the ranges of the data types: INTEGER, SINTEGER, and SLONG. | \[
\text{SLONG ATOL (CHAR STRING[ ])}
\] | • STRING - string containing the character representation of the integer. Valid input characters are "0" through "9" and the sign designators ("+" and "-"). If no valid characters are found, zero is returned as a result. | • A 32-bit integer representing the converted string. • Any non-numeric characters in the string are ignored. • ATOL returns the value representing the first complete set of characters that represents an integer. |

**NOTE:** While you can pass in larger values, ATOL will truncate any value outside the range -2147483648 to 2147483647 to the value -2147483648 (if negative) or 2147483647 (if positive). |

Example: |

Vol = ATOL('Volume=100%') // Vol = 100
Num = ATOL('-3758')       // Num = -3758

| **CH_TO_WC** | This keyword converts a CHAR array to a WIDECHAR array. | \[
\text{WIDECHAR[] CH_TO_WC(CHAR STRING[])}
\] | • STRING - a character string to be converted. | • A WIDECHAR array containing the values from the CHAR array. |

Example: |

WIDECHAR wcData[] = CH_TO_WC('ASCII')

| **FTOA** | This function converts a 64-bit floating-point value to an ASCII string containing the decimal representation of the number. | \[
\text{CHAR[] FTOA (DOUBLE Num)}
\] | • Num: 64-bit Floating-point number to convert to a decimal string. | • Character string that contains the decimal representation of the specified floating point number, rounded to 6 digits of precision. • The character representation uses exponents as necessary, according to the following rule: For 0.0001 <= \(|n|\) < 1000000, FTOA returns the result in non-exponential form; otherwise, it returns the result in exponential form. Examples: |

<table>
<thead>
<tr>
<th>n</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000</td>
<td>'1E+06'</td>
</tr>
<tr>
<td>1234567</td>
<td>'1.23457E+06'</td>
</tr>
<tr>
<td>-0.001</td>
<td>'-0.001'</td>
</tr>
<tr>
<td>0.00045</td>
<td>'0.00045'</td>
</tr>
<tr>
<td>0.000045</td>
<td>'4.5E-05'</td>
</tr>
<tr>
<td>123.45678</td>
<td>'123.457'</td>
</tr>
</tbody>
</table>

| **HEXTOI** | This function converts an ASCII string containing the hexadecimal representation of a number to an unsigned 32-bit integer. | \[
\text{LONG HEXTOI (CHAR STRING[ ])}
\] | • STRING: Hexadecimal formatted string to be converted to an integer. | • 32-bit unsigned integer representing the converted string. • Any non-hexadecimal characters in the string are ignored. • HEXTOI returns a value representing the first complete set of characters that represents an integer. • Valid characters are "0" through "9", "A" through "F" and "a" through "f". • If no valid characters are found, zero is returned as a result. |

Example: |

Num = HEXTOI('126EC')     // Num = 75500

| **ITOA** | This function converts a 32-bit signed integer to a decimal ASCII string. | \[
\text{CHAR[] ITOA (LONG Num)}
\] | • Num: The 32-bit unsigned integer to convert to a decimal string. | A character string that contains the decimal representation of the specified integer. |

Example: |

STRING = ITOA(501)       // STRING = '501'
### Conversion Keywords (Cont.)

**FORMAT**

Provides a mechanism similar to C's `printf` statement for formatting the display of numbers. This function is similar to `ITOA`, but is infinitely more powerful.

```c
CHAR[] FORMAT(CHAR FormatLine[], CHAR Value) FORMAT(CHAR FormatLine[], WIDECHAR Value) CHAR[] FORMAT(CHAR FormatLine[], INTEGER Value) CHAR[] FORMAT(CHAR FormatLine[], SINTEGER Value) CHAR[] FORMAT(CHAR FormatLine[], LONG Value) CHAR[] FORMAT(CHAR FormatLine[], SLONG Value) CHAR[] FORMAT(CHAR FormatLine[], FLOAT Value) CHAR[] FORMAT(CHAR FormatLine[], DOUBLE Value)
```

**Parameters:**

- **FormatLine:** A formatted string of text that defines how the (return) string should be formatted. The format string contains plain characters and a conversion specification. Plain characters are copied, as is, directly. Conversion characters conform to the following format:
  
  %[flags][width].[prec]type

  - **flags:** Output justification, numeric signs, decimal points, trailing zeros, octal and hex prefixes. By default, output is right justified. Use a '-' to left justify as in %5d.
  - **-:** Causes left justification, padding with blanks
  - **0:** Zeros are used to pad instead of spaces if a field length is given.
  - **+:** Output always begins with + or -. Blank: Positive values begin with a blank.
  - **Blank:** Positive values begin with a blank.
  - **width:** Minimum number of characters to print. If the output would be less than this width, it is padded with spaces to be width characters wide. If the output is larger than width the entire output is provided (i.e. it is not truncated).
  - **.prec:** Maximum number of characters to print or number of digits to the right of the decimal point for a float or double type.
  - **type:** Conversion type:
    - **c:** Value is treated as an integer, and presented as the character with that ASCII value.
    - **d:** Value is treated as a signed integer, and presented as a decimal number.
    - **f:** Value is treated as a double, and presented as a floating-point number.
    - **o:** Value is treated as a signed integer, and presented as an octal number.
    - **u:** Unsigned integer.
    - **x:** Value is treated as an integer and presented as a hexadecimal number (with lowercase letters).
    - **X:** Value is treated as an integer and presented as a hexadecimal number (with uppercase letters).
    - %: A literal percent character.

- **Value:** The value to be converted to a string. The result is a formatted text string.

```c
fTemperature = 98.652
STR = FORMAT('The current temperature is %3.2f', fTemperature)
// Displays "The current temperature is 98.65"
```

The table below shows some examples of the output of `FORMAT` for several different format lines and values:

<table>
<thead>
<tr>
<th>FORMAT Statement</th>
<th>Result of FORMAT function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FORMAT('%-5.2f', 123.234)</code></td>
<td>'123.23'</td>
</tr>
<tr>
<td><code>FORMAT('%5.2f', 3.234)</code></td>
<td>'3.23'</td>
</tr>
<tr>
<td><code>FORMAT('%+4d', 6)</code></td>
<td>'+6'</td>
</tr>
</tbody>
</table>

The result is a formatted text string.

```c
fTemperature = 98.652
STR = FORMAT('The current temperature is %3.2f', fTemperature)
// Displays "The current temperature is 98.65"
```

**ITOHEX**

This function converts a 32-bit unsigned integer to an ASCII string containing the hexadecimal representation of the number.

```c
CHAR[] ITOHEX(LONG Num)
```

**Parameters:**

- **Num:** The 32-bit unsigned integer to convert to a hexadecimal string.

```c
STRING = ITOHEX(1000)    // STRING = '3E8'
```

**RAW_BE**

This routine takes an intrinsic variable and converts it into a Character Array in Big Endian Format representing the variable.

```c
CHAR[] RAW_BE(IntrinsicVariable)
```

**RAW_LE**

This routine takes an intrinsic variable and converts it into a Character Array in Little Endian Format representing the variable.

```c
CHAR[] RAW_LE(IntrinsicVariable)
```

**NOTE:** NetLinx also provides a set of Encode & Decode keywords. See Encode / Decode Keywords on page 70 for details.
DEFINE Keywords

Overview

NetLinx has two methods for creating subroutines: DEFINE_CALL and DEFINE_FUNCTION.

DEFINE_CALL

DEFINE_CALL is intended to run segments of code that are repeated throughout the program, but don't require a return value. For example, this DEFINE_CALL creates a macro to lower a screen, turn on the projector, and set the lights to Preset 1. The subroutine executes three commands and no values are returned to the program.

```plaintext
DEFINE_CALL 'PRESENTATION MACRO'
{
  SYSTEM_CALL [1] 'SCREEN1' (0, 0, 1, 0, SCREEN, 1, 2, 3, 0)
  SEND_STRING VPROJ, "'PON', $0D,$0A"
  SEND_STRING RADIA, "'1B', $0D"
}
```

The NetLinx compiler passes all variables by reference. This means that the variable the subroutine operates on is the same variable the caller passed. Any change made to the variable, passed as a calling parameter, updates the variable's value from the caller's perspective. You can take advantage of this pass by reference feature by returning an updated value through a calling parameter rather than as the return value. Constants, on the other hand, are passed by value. When this happens, a copy of the parameter is delivered to the subroutine. Any change made to the variable representing the constant is lost once the function or subroutine is lost. To specify an array as a function or subroutine parameter, one set of brackets for each array dimension must follow the variable name, as shown in the following example:

```plaintext
DEFINE_CALL 'READ INPUT' (CHAR BUFFER[][]){
  /* body of the subroutine */
}
```

The parameter BUFFER is declared to be a two-dimensional array by including two sets of brackets after the name. For compatibility with existing programs, the array dimensions may be specified inside the brackets. These dimensions, however, are not required and are ignored by the compiler. The NetLinx Interpreter will do bounds checking on the array and generate a run-time error if the array bounds are exceeded.

**NOTE:** Subroutines must be defined before they can be used. For this reason, DEFINE_CALLs should appear before the DEFINE_START, DEFINE_EVENT, and DEFINE_PROGRAM sections.

DEFINE_FUNCTION

DEFINE_FUNCTION provides a way to return a value to a statement. It has the same functionality as a DEFINE_CALL. The DEFINE_FUNCTION is used in-line in a statement, where a DEFINE_CALL must be used as a standalone statement. The basic structure is:

```plaintext
DEFINE_FUNCTION [<return type>]<name>[(<param1>,<param2>, _ <parameN>)]
{
  (* statements *)
}
```

The following DEFINE_FUNCTION creates a subroutine to cube a number and returns a LONG integer value:

```plaintext
DEFINE_FUNCTION LONG CUBEIT (LONG VALUE)
{
  STACK_VAR RESULT
  RESULT = VALUE * VALUE * VALUE
  RETURN RESULT
}
```

```plaintext
DEFINE_PROGRAM
PUSH[TP1, 1]
{
  CUBED_VAL = CUBEIT ( 3 )
  /* CUBED_VAL = 27 */
}
```
**DEFINE_CONSTANT**
The standard format for DEFINE_CONSTANT is:

```
<constant name> = <constant expression>
```

NetLinx allows variables to be defined as constants in the DEFINE_VARIABLE section of the program or module, and in the LOCAL_VAR section of a DEFINE_CALL or a DEFINE_FUNCTION. The scope of the constant extends throughout the module in which it is defined. If the DEFINE_CONSTANT section appears in the main program or in an include file, the constant’s scope extends globally throughout the program. DEFINE_CONSTANT accepts data in the following formats:

<table>
<thead>
<tr>
<th>Type</th>
<th>Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal Integer</td>
<td>0000</td>
<td>1500</td>
</tr>
<tr>
<td>Hexadecimal Integer</td>
<td>$000</td>
<td>$DE60</td>
</tr>
<tr>
<td>Floating Point</td>
<td>000.0</td>
<td>924.5</td>
</tr>
<tr>
<td>Exponential Notation</td>
<td>0.0e0</td>
<td>.5e-12</td>
</tr>
<tr>
<td>Character</td>
<td>'c' or &lt;char code&gt;</td>
<td>'R' or 255</td>
</tr>
<tr>
<td>String Literal</td>
<td>&quot;ssss&quot;</td>
<td>'Reverse'</td>
</tr>
</tbody>
</table>

**DEFINE Keywords**

**DEFINE_CALL**
This keyword defines the implementation of a NetLinx subroutine.

```plaintext
DEFINE_CALL '<name>' [(P1,P2,...)]
{
// body of subroutine
}
```

The subroutine name cannot be a previously defined device name, constant, or variable, or a name assigned to a buffer or a wait statement. DEFINE_CALL names are case sensitive and may contain spaces. Subroutines must be defined before they can be used. For this reason, DEFINE_CALLs should appear before the DEFINE_START, DEFINE_EVENT, and DEFINE_PROGRAM sections.

**DEFINE_COMBINE**
This keyword defines the combination of functionally identical devices, such as identically programmed touch panels. When the program references one of these devices, all other combined devices are also referenced. The devices in a given combine must be enclosed in parentheses. The first device in the list (the primary device) must be a virtual device.

```plaintext
DEFINE_COMBINE(VDevice, Panel1, Panel2, Panel3)
```

The example below uses a device array (DEV[ ]) instead of specifying the individual devices (Panel1, Panel2, and Panel3).

Any input events for any device in the array will appear to the program as coming from the virtual device. Output changes directed to the virtual device or any device in the set is sent to all devices in the array.

```plaintext
DEFINE_COMBINE(VDevice, DEV[ ])  
```

See the Combine & Uncombine Keywords on page 40 for more information on virtual devices and device arrays.

**DEFINE_CONNECT_LEVEL**
This keyword defines level connections. A single connection is defined by listing the device-level pairs inside parentheses. The first level in the list (the primary level) must be a virtual level (a level on a virtual device). A virtual level does not actually exist but merely represents one or more levels on physical devices. The example below combines the levels [Device1, Level1] and [Device2, Level2].

```plaintext
(VDevice, Level1, Device1, Level1, Device2, Level1)
```

The next example combines all levels in the device-level array.

Changes to any level listed in the connection will automatically be reflected in the other levels so that all level values are the same.

```plaintext
DEFINE_CONNECT_LEVEL(VDevLev, MyDL[ ])  
```

By specifying a virtual level as the primary level, NetLinx code targets the virtual level but operates on each physical level. Since the primary level is virtual, the primary device (a virtual device) cannot be taken off-line or removed from the system.

**DEFINE_CONSTANT**
This keyword defines program constants; the value of a constant cannot be changed within the program.

```plaintext
DEFINE_CONSTANT
PLAY = 1
STOP = 2
STRING='HELLO'
```

Refer to the DEFINE_CONSTANT section on page 61 for more information.

**DEFINE_DEVICE**
This keyword defines the devices referenced in the program.

```plaintext
DEFINE DEVICE
TP1 = T28:1:0  // device number = 128, port = 1, system = 0
TP2 = T29:1:0  // device number = 129, port = 1, system = 0
TP3 = T30:1:0  // device number = 130, port = 1, system = 0
VCR1 = T10:1:0 // device number = 10, port = 1, system = 0
VCR2 = T11:1:0 // device number = 11, port = 1, system = 0
```

Devices can be specified by a single device number such as "TP = 128" or as a fully-qualified device specification such as "TP = T28:1:0"
<table>
<thead>
<tr>
<th>DEFINE Keywords (Cont.)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFINE_EVENT</td>
<td>This keyword provides the basis for the construction of the event table, which is where event-handling code is placed. When NetLinx receives an incoming event, the event table is searched for a handler for that event. A handler is a block of code that performs the necessary processing for an event notification received from a given device (and possibly associated with a particular channel). See the Event Handler Keywords on page 74 for more information.</td>
</tr>
<tr>
<td>DEFINE_FUNCTION</td>
<td>This keyword defines the implementation of a NetLinx function. The return type is optional and can be any intrinsic data type or array of intrinsic types that NetLinx supports except a structure or an array of structures. The function name must not be a previously defined constant or variable or a name assigned to a buffer, a wait, DEFINE_CALL, or Function. Function names are not case sensitive. See the DEFINE_FUNCTION section on page 60 for more information.</td>
</tr>
<tr>
<td>DEFINE_LATCHING</td>
<td>This keyword section is where latching channels and variables are defined. A latching channel is one that changes its state once per activation. If a latching channel is activated by a TO keyword, it changes its state. When the TO is stopped by releasing the button that started it, the channel does not go back to its previous state. The status of a latching channel (that is not part of a mutually exclusive group) will always reflect the on/off state of the channel. In the following example, the device-channel [RELAY, SYSTEM_POWER] is defined as latching. The next statement uses the double periods (...) to define a range of VCR channels as latching. In the last statement, the variable VAR1 is defined as latching.</td>
</tr>
</tbody>
</table>
DEFINE Keywords (Cont.)

**DEFINE Type**
This keyword section defines custom data types such as structures and arrays. An example DEFINE_TYPE section is shown below.

```plaintext
DEFINE_TYPE
STRUCTURE MyStruct
{
    LONG Num
    CHAR Name[30]
}
```

See the Structure Keywords on page 128 for a discussion of structures.

**DEFINE Variable**
This keyword declares global variables. Any variable defined in this section is static (its value is maintained throughout the duration of program execution) and module scope (it is accessible from any instruction in the current module).

```plaintext
DEFINE_VARIABLE
INTEGER INT1
FLOAT FP1
VOLATILE INTEGER BIGARRAY[1000][1000]
```

**NOTE:** 1000 marks the limit of the string.

See the Variables - Overview on page 9 for more information.

**Program Name**
This keyword declares the program name. It must appear on the first line of the program and cannot appear more than once in any single program or include file.

```plaintext
PROGRAM_NAME = '<program name>'
```

**Return**
This keyword is used in a DEFINE_FUNCTION or DEFINE_CALL subroutine to prematurely terminate execution and/or to return a value to the caller. Only DEFINE_FUNCTION functions can return values using the RETURN statement.

The syntax of the RETURN statement is either:

```plaintext
RETURN    // DEFINE_CALL or function with no return value
or
RETURN Value    // function with return value
```

Upon execution of the RETURN statement, program control is immediately returned to the caller. If the function containing the RETURN statement has a declared return type, the parameter Value must be included and match the specified type. If the function has no declared return type, the parameter Value must be omitted.

**DEFINE mutually exclusive and Variables**

**Symptom:** If you have a set of variables that are mutually exclusive and you set one of the variables to a non-zero value by assignment, e.g. Var1 = 1 or the Studio Debug window, then the other variables in the set stay “on” i.e. non-zero.

```plaintext
DEFINE_VARIABLE
INTEGER var[4]
INTEGER x
DEFINE_MUTUALLY_EXCLUSIVE(var[1],var[2],var[3],var[4])
DEFINE_PROGRAM
WAIT 20
{
    x++; IF (x > 4) x = 1;
    var[x] = x // This will not invoke the mutually exclusive magic
}
```

In the NetLinx code example above, all elements of var will eventually be non-zero.

**Cause:** This has always worked this way, even in Axcess.

**Resolution:** Use ON to set variables if they are members of a mutually exclusive set:

```plaintext
DEFINE_VARIABLE
INTEGER var[4]
INTEGER x
DEFINE_MUTUALLY_EXCLUSIVE(var[1],var[2],var[3],var[4])
DEFINE_PROGRAM
WAIT 20
{
    x++; IF (x > 4) x = 1;
    ON[var[x]]    // This will work as expected - only one element of var will have a value of 1 at any time
}
```

This issue does not occur with DEVCHAN's. Using ON or assigning them a non-zero value will work as expected:

```plaintext
DEFINE_DEVICE
dvRelay = 305:1:0
DEFINE_VARIABLE
INTEGER x
DEFINE_MUTUALLY_EXCLUSIVE([dvRelay,1]..[dvRelay,4])
([dvRelay,5]..[dvRelay,8])
DEFINE_PROGRAM
WAIT 20
{
    x++; IF (x > 4) x = 1;
    ON[dvRelay,x]    // This works as expected: only 1 relay of relays 1 to 4 will be on at a time
    [dvRelay,x + 4] = x // This works as expected: only 1 relay of relays 5 to 8 will be on at a time
}
DEVICE Keywords

NetLinx supports the following DEVICE keywords:

**DEVICE ID**

Every device in the NetLinx system has a unique ID number identifying its device type, such as an infrared/serial card or touch panel. The DEVICE_ID keyword returns the ID number pertaining to the specified device. If the device does not exist in the system, zero is returned. This keyword is usually used to determine whether or not a device is present in the system.

```
DeviceID = DEVICE_ID(Device)
```

For example:
```
IF (DEVICE_ID(55:1:0) <> 0) 
{ // device 55 exists in the system }
```

**DEVICE ID_STRING**

This keyword returns a string description/model number for the specified device.
```
DeviceString = DEVICE_ID_STRING(55:1:0)
```

**DEVICE_INFO**

NetLinx stores information, such as manufacturer, device name and device ID, for each device in the system. The DEVICE_INFO keyword allows a programmer to access all available information for any device. If the device does not exist in the system, a Device ID of zero is returned. This keyword is usually used to determine the firmware version of a device in the system.

```
DEVICE_INFO(Device, DEV_INFO_STRUCT Info)
```

**Parameters:**
- **Device:** The device to query.
- **Info:** A DEV_INFO_STRUCT variable to populate with the device information.

**Result:** DEVICE_INFO does not return a result. However, if the DEVICE_INFO call is successful, the DEVICE_ID element of the structure will be non-zero. If DEVICE_ID is zero, the structure contains no useful information.

The DEV_INFO_STRUCT contains the following information:
- **Info. MANUFACTURER_STRING** - A string identifying the manufacturer of the device.
- **Info. MANUFACTURER** - A integer identifying the manufacturer.
- **Info. DEVICE_ID_STRING** - A string description/model number for the specified device. This is the same information returned by the DEVICE_ID_STRING keyword.
- **Info. DEVICE_ID** - A unique ID number identifying its device type, such as an infrared/serial card or touch panel. This is the same information returned by the DEVICE_ID keyword.
- **Info. VERSION** - A string identifying the firmware version of the device. This is not available for AxLink devices.
- **Info. FIRMWARE_ID** - A unique ID number identifying the firmware for this device. This is not available for AxLink devices.
- **Info.SERIAL_NUMBER** - A 16-character serial number of the specified device. The serial number of every device is established when manufactured. This is the same information returned by GET_SERIAL_NUMBER keyword. This is not available for AxLink devices.
- **Info. SOURCE_TYPE** - An integer identifying how the device is connected to the master. This value can be any of the following:
  - $00 (SOURCE_TYPE_NO_ADDRESS) - There is no source address.
  - $01 (SOURCE_TYPE_NEURON_ID) - The device is connected via ICSNet.
  - $02 (SOURCE_TYPE_IP_ADDRESS) - The device is connected via IP.
  - $03 (SOURCE_TYPE_AXLINK) - The device is connected via ICSNet.
  - $06 (SOURCE_TYPE_IPV4_PORT_MAC_ADDRESS) - The device is connected via IP.
  - $10 (SOURCE_TYPE_NEURON_SUBNODE_ICSP) - The device is connected via ICSNet.
  - $11 (SOURCE_TYPE_NEURON_SUBNODE_PL) - The device is connected via ICSNet.
  - $12 (SOURCE_TYPE_IP_SOCKET_ADDRESS) - This device is a NetLinx socket.
  - $13 (SOURCE_TYPE_RS232) - This device is connected via RS232.
  - $18 (SOURCE_TYPE_IPV4_PORT_MAC_IPV6) - The device is connected via IP.
  - $20 (SOURCE_TYPE_INTERNAL) - This device is internal to the NetLinx controlled.
- **Info. SOURCE_STRING** - A string identifying the source address. Normally, this contains only useful information when Info.SOURCE_TYPE is $02 (IP), in which case this contains the IP address of the device.

Example:
```
DEFINE_DEVICE
dvNL = 0:1:0
DEFINE_VARIABLE
DEV_INFO_STRUCT sDeviceInfo
DEFINE_EVENT
DATA_EVENT[dvNL]
{ ONLINE:
  { DEVICE_INFO(dvNL, sDeviceInfo)
    SEND_STRING 0,"'MANUFACTURER_STRING=', sDeviceInfo.MANUFACTURER_STRING"
    SEND_STRING 0,"'MANUFACTURER=', ITOA(sDeviceInfo.MANUFACTURER)"
    SEND_STRING 0,"'DEVICE_ID_STRING=', sDeviceInfo.DEVICE_ID_STRING"
    SEND_STRING 0,"'DEVICE_ID=', ITOA(sDeviceInfo.DEVICE_ID)"
    SEND_STRING 0,"'VERSION=', sDeviceInfo.VERSION"
    SEND_STRING 0,"'FIRMWARE_ID=', ITOA(sDeviceInfo.FIRMWARE_ID)"
    SEND_STRING 0,"'SERIAL_NUMBER=', sDeviceInfo.SERIAL_NUMBER"
    SEND_STRING 0,"'SOURCE_TYPE=', ITOA(sDeviceInfo.SOURCE_TYPE)"
    SEND_STRING 0,"'SOURCE_STRING=', sDeviceInfo.SOURCE_STRING"
  }
}
## DEVICE Keywords (Cont.)

### DEVICE_INFO (Cont.)
Telnet displays this information:

- **MANUFACTURER_STRING**=AMX Corp.
- **MANUFACTURER**=1
- **DEVICE_ID_STRING**=NXC-ME260
- **DEVICE_ID**=256
- **VERSION**=v2.30.128
- **FIRMWARE_ID**=256
- **SERIAL_NUMBER**=2010-00372
- **SOURCE_TYPE**=1
- **SOURCE_STRING**=00A066452001

### DEVICE_STANDBY
This command requests that a device goes to standby state. If the device supports the standby state, the device will transition to standby, generating an asynchronous STANDBY DATA_EVENT.

Example:

```plaintext
DEVICE_STANDBY(DEVICE, NORMAL_STANDBY)
```

### DEVICE_WAKE
This command requests that a device in standby state wake up. If the device both supports and is in standby state, it will transition to awake, generating an asynchronous AWAKE DATA_EVENT (see page 54).

Example:

```plaintext
DEVICE_WAKE (DEVICE, NORMAL_WAKE)
```

Due to the nature of STANDBY state, a device in standby syncs with the master at regular intervals. The request to wake will not be processed until one of these sync events. So the AWAKE state will appear delayed.

### DYNAMIC_APPLICATION_DEVICE
Specifies a Duet device that is completely dynamic. A dynamically discovered device matching the specified deviceType could be bound to the duetDevice from anywhere in the system.

```plaintext
DYNAMIC_APPLICATION_DEVICE (DEV duetDevice, char[] deviceType, char[] friendlyName)
```

### MASTER_SLOT
This keyword represents the slot number the master card is plugged into. "0" is the primary master; "1" is the secondary master. This keyword is primarily associated with Axcess systems. NetLinx systems have only one master, so MASTER_SLOT in NetLinx is always "0".

### PUSH_DEVICE
See page 120.

### RELEASE_DEVICE
See page 120.

### PUSH_DEVCHAN
See page 120.

### RELEASE_DEVCHAN
See page 120.

### REBOOT
This keyword causes the device to reset and is equivalent to doing a power down and up on the master.

```plaintext
REBOOT (DEVICE)
```

Parameters:

- **DEVICE** = ICSP device number to reboot.

**NOTE:** Not all ICSP devices implement the reboot command.

**DEVICE** refers to:
- Device – a single device number.
- Dps – a DEV structure.
- D:P:S – a device specification such as 128:1:0.
- DEV[] – a device array.

Examples:

```plaintext
REBOOT (0:0:0)  
REBOOT (0:1:0) or REBOOT (0)
```

Any of these examples will cause the master to reboot.

### SEND_COMMAND
This keyword sends device-specific commands to a NetLinx device.

**Syntax:**

```plaintext
SEND_COMMAND DEV, '<command string>'
```

- or -

```plaintext
SEND_COMMAND DEV[], '<command string>'
```

### SYSTEM_NUMBER
This keyword defines an unsigned 16-bit integer system constant that contains the system number.
Encode / Decode Keywords

Overview - Encoding and Decoding Binary and XML

There are six functions used to encode and decode variables in NetLinx. This encoding process takes a NetLinx variable, no matter how complex, and converts it into a string. The decode process will take this string and copy the contents back into a variable. These functions can be used to take the contents of NetLinx variables and convert them to string. Once the variable exists in string form, it can then be sent across an RS-232 connection, sent over and IP socket or saved to the NetLinx master’s file system (disc on chip). Once the string is retrieved, either from a data event or by reading the information from the NetLinx master’s file system, the data can be converted back to a variable.

There are two version of this encoding and decoding: Binary and XML.

- The binary conversion routines are: STRING_TO_VARIABLE, VARIABLE_TO_STRING and LENGTH_VARIABLE_TO_STRING.
- The XML routines are XML_TO_VARIABLE, VARIABLE_TO_XML and LENGTH_VARIABLE_TO_XML.

Both sets of routines accomplish the same function but the encoded string differs in protocol. The binary conversion routines uses a compact binary representation of the variable while the XML represents the variable as a ASCII text only XML document. The binary routines are ideal when sending data from one NetLinx system to another NetLinx system over RS-232 or IP since the variable will be as compact as possible. It is also ideal for saving a file to the NetLinx master’s file system if you do not intend to edit the file later. The binary routines encode and decode a variable sequentially meaning that the order and type of the variables must match on both the encoding and decoding side.

The XML routines are ideal when sending data from one NetLinx system to another type of system over RS232 or IP, since XML is more universally accepted by other types of computer systems. XML is also ideal for saving a file to the NetLinx master’s file system if you intend to edit the file later since it is entirely ASCII text. It should be noted that while the XML is more universal, is not very compact. The XML routines encode and decode a variable non-sequentially, meaning that the order and type of variables do not need to match on both the encoding and decoding side.

Below are some examples of how to use these encoding routines:

```
PROGRAM_NAME='ConversionExample'

(*|{PS_SOURCE_INFO(PROGRAM STATS) |*)
***********************************************************************
(* FILE CREATED ON: 05/22/2001 AT: 11:09:27 |*)
***********************************************************************
(* FILE_LAST_MODIFIED_ON: 05/22/2001 AT: 11:26:44 |*)
***********************************************************************
(* ORPHAN_FILE_PLATFORM: 1 |*)
***********************************************************************
(*!!FILE REVISION: | *)
(* REVISION DATE: 05/22/2001 | *)
(* | *)
(* COMMENTS: | *)
(* | *)
***********************************************************************
(*|PS_SOURCE_INFO |*)
***********************************************************************
(* System Type : NetLinx |*)
***********************************************************************
(* REV HISTORY: | *)
***********************************************************************

***********************************************************************
(* DEVICE NUMBER DEFINITIONS GO BELOW |)
***********************************************************************
DEFINDE_DEVICE
dvTP        = 128:1:0

***********************************************************************
(* CONSTANT DEFINITIONS GO BELOW |)
***********************************************************************
DEFINDE_CONSTANT
nFileRead   = 1
nFileWrite  = 2
```
DEFINE_TYPE  
STRUCTURE _AlbumStruct  
{  
LONG lTitleID  
CHAR sArtist[100]  
CHAR sTitle[100]  
CHAR sCopyright[100]  
CHAR sLabel[100]  
CHAR sReleaseDate[100]  
INTEGER nNumTracks  
CHAR sCode[100]  
INTEGER nDiscNumber  
}  
STRUCTURE _AlbumStruct2  
{  
CHAR sArtist[100]  
CHAR sTitle[100]  
INTEGER nNumTracks  
}

DEFINE_VARIABLE 
VOLATILE _AlbumStruct   AlbumStruct[3]  
VOLATILE _AlbumStruct2  
AlbumStruct2[3]  
VOLATILE CHAR   sBinaryString[10000]  
VOLATILE CHAR   sXMLString[50000]  
VOLATILE LONG   lPos  
VOLATILE SLONG  slFile  
VOLATILE SLONG  slReturn

DEFINE_START  
(* assign some values *)  
AlbumStruct[1].lTitleID = 11101000  
AlbumStruct[1].sArtist = 'Buffet, Jimmy'  
AlbumStruct[1].sTitle = 'Living & Dying in 3/4 Time'  
AlbumStruct[1].sCopyright = 'MCA'  
AlbumStruct[1].sLabel = 'MCA'  
AlbumStruct[1].sReleaseDate = '1974'  
AlbumStruct[1].nNumTracks = 11  
AlbumStruct[1].sCode = '3132333435'  
AlbumStruct[1].nDiscNumber = 91  
AlbumStruct[2].lTitleID = 17248229  
AlbumStruct[2].sArtist = 'Buffet, Jimmy'  
AlbumStruct[2].sTitle = 'Off to See the Lizard'  
AlbumStruct[2].sCopyright = 'MCA'  
AlbumStruct[2].sLabel = 'MCA'  
AlbumStruct[2].sReleaseDate = '1989'  
AlbumStruct[2].nNumTracks = 11  
AlbumStruct[2].sCode = '3132333436'  
AlbumStruct[2].nDiscNumber = 105  
AlbumStruct[3].lTitleID = 12328612  
AlbumStruct[3].sArtist = 'Buffet, Jimmy'  
AlbumStruct[3].sTitle = 'A-1-A'  
AlbumStruct[3].sCopyright = 'MCA'  
AlbumStruct[3].sLabel = 'MCA'  
AlbumStruct[3].sReleaseDate = '1974'  
AlbumStruct[3].nNumTracks = 11  
AlbumStruct[3].sCode = '3132333437'  
AlbumStruct[3].nDiscNumber = 189
DEFINE_EVENT
(* CONVERT AND SAVE *)
BUTTON_EVENT[dvTP,1]
{
  PUSH:
  {
    (* CONVERT TO BINARY *)
    lPos = 1
    slReturn = VARIABLE_TO_STRING (AlbumStruct,sBinaryString,lPos)
    SEND_STRING 0, "'POSITION=',ITOA(lPos), '; RETURN=',ITOA(slReturn)"
    (* CONVERT TO XML *)
    lPos = 1
    slReturn = VARIABLE_TO_XML (AlbumStruct,sXMLString,lPos,0)
    SEND_STRING 0, "'POSITION=',ITOA(lPos), '; RETURN=',ITOA(slReturn)"
    (* NOW WE CAN SAVE THESE BOTH TO DISCS *)
    slFile = FILE_OPEN('BinaryEncode.xml',nFileWrite)
    IF (slFile > 0)
    {
      slReturn = FILE_WRITE(slFile,sBinaryString,LENGTH_STRING(sBinaryString))
      IF (slReturn < 0) SEND_STRING 0, "'FILE WRITE FAIL RETURN=',ITOA(slReturn)"
      slReturn = FILE_CLOSE(slFile)
      IF (slReturn < 0) SEND_STRING 0, "'FILE CLOSE FAIL RETURN=',ITOA(slReturn)"
    }
    slFile = FILE_OPEN('XMLEncode.xml',nFileWrite)
    IF (slFile > 0)
    {
      slReturn = FILE_WRITE(slFile,sXMLString,LENGTH_STRING(sXMLString))
      IF (slReturn < 0) SEND_STRING 0, "'FILE WRITE FAIL RETURN=',ITOA(slReturn)"
      slReturn = FILE_CLOSE(slFile)
      IF (slReturn < 0) SEND_STRING 0, "'FILE CLOSE FAIL RETURN=',ITOA(slReturn)"
    }
    (* Clear string *)
    sBinaryString = ""
    sXMLString = ""
  }
  (* READ AND DECODE *)
  BUTTON_EVENT[dvTP,2]
  {
    PUSH:
    {
      (* NOW WE CAN SAVE THESE BOTH TO DISCS *)
      slFile = FILE_OPEN('BinaryEncode.xml',nFileRead)
      IF (slFile > 0)
      {
        slReturn = FILE_READ(slFile,sBinaryString,MAX_LENGTH_STRING(sBinaryString))
        IF (slReturn < 0) SEND_STRING 0, "'FILE WRITE FAIL RETURN=',ITOA(slReturn)"
        slReturn = FILE_CLOSE(slFile)
        IF (slReturn < 0) SEND_STRING 0, "'FILE CLOSE FAIL RETURN=',ITOA(slReturn)"
      }
      slFile = FILE_OPEN('XMLEncode.xml',nFileRead)
      IF (slFile > 0)
      {
        slReturn = FILE_READ(slFile,sXMLString,MAX_LENGTH_STRING(sXMLString))
        IF (slReturn < 0) SEND_STRING 0, "'FILE WRITE FAIL RETURN=',ITOA(slReturn)"
        slReturn = FILE_CLOSE(slFile)
        IF (slReturn < 0) SEND_STRING 0, "'FILE CLOSE FAIL RETURN=',ITOA(slReturn)"
      }
      (* CONVERT TO BINARY *)
      lPos = 1
      slReturn = STRING_TO_VARIABLE (AlbumStruct,sBinaryString,lPos)
      SEND_STRING 0, "'POSITION=',ITOA(lPos), '; RETURN=',ITOA(slReturn)"
      (* CONVERT TO XML *)
      lPos = 1
      slReturn = XML_TO_VARIABLE (AlbumStruct,sXMLString,lPos,0)
      SEND_STRING 0, "'POSITION=',ITOA(lPos), '; RETURN=',ITOA(slReturn)"
    }
  }
}
Decode / Encode Keywords

(* READ AND DECODE *)
(* THE BINARY WILL FAIL SINCE THE DECODE TYPE DOES NOT MATCH THE ENCODE TYPE *)
(* THE XML WILL NOT FAIL SINCE IT DOES NOT REQUIRE DATA TO BE THE SEQUENTIAL *)

BUTTON_EVENT([dvTP,3])
{
    PUSH:
    {
        (* NOW WE CAN SAVE THESE BOTH TO DISCS *)
        slFile = FILE_OPEN('BinaryEncode.xml',nFileRead)
        IF (slFile > 0)
        {
            slReturn = FILE_READ(slFile,sBinaryString,MAX_LENGTH_STRING(sBinaryString))
            IF (slReturn < 0) SEND_STRING 0,'"FILE WRITE FAIL RETURN=',ITOA(slReturn)"
            slReturn = FILE_CLOSE(slFile)
            IF (slReturn < 0) SEND_STRING 0,'"FILE CLOSE FAIL RETURN=',ITOA(slReturn)"
        }
        slFile = FILE_OPEN('XMLEncode.xml',nFileRead)
        IF (slFile > 0)
        {
            slReturn = FILE_READ(slFile,sXMLString,MAX_LENGTH_STRING(sXMLString))
            IF (slReturn < 0) SEND_STRING 0,'"FILE WRITE FAIL RETURN=',ITOA(slReturn)"
            slReturn = FILE_CLOSE(slFile)
            IF (slReturn < 0) SEND_STRING 0,'"FILE CLOSE FAIL RETURN=',ITOA(slReturn)"
        }
        (* CONVERT TO BINARY *)
        lPos = 1
        slReturn = STRING_TO_VARIABLE (AlbumStruct2,sBinaryString,lPos)
        SEND_STRING 0,'"POSITION=',ITOA(lPos),'; RETURN=',ITOA(slReturn)"
        (* CONVERT TO XML *)
        lPos = 1
        slReturn = XML_TO_VARIABLE (AlbumStruct2,sXMLString,lPos,0)
        SEND_STRING 0,'"POSITION=',ITOA(lPos),'; RETURN=',ITOA(slReturn)"
    }
}

***********************************************************
(*            THE ACTUAL PROGRAM GOES BELOW                *)
***********************************************************

DEFINE_PROGRAM

***********************************************************
(*                     END OF PROGRAM                      *)
***********************************************************

(* DO NOT PUT ANY CODE BELOW THIS COMMENT *)
***********************************************************
## Encode / Decode Keywords

The NetLinx programming language supports the following Encode & Decode keywords:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>STRING_TO_VARIABLE</strong> (VARIABLE DECODE)</td>
<td>This routine takes the Encode data from buffer and loads the values into the DECODE variable. The DECODE variable must match the type of the variables in the encoded string. When the ENCODE variable is a structure, the decode variable members must match in type and order. If the number of members of the structures doesn't match then the routine will fill all it can or skip any unused data members.</td>
</tr>
<tr>
<td>Parameters:</td>
<td>• DECODE: Any type of variable. This is the variable to be decoded into.</td>
</tr>
<tr>
<td></td>
<td>• BUFFER: Must be of char array type. This is where the encoded data is found.</td>
</tr>
<tr>
<td></td>
<td>• POSITION: This is where the first byte of the decode data. It is also modified to point to the next location after the last decoded byte. That means that successive calls to this function can be made without modifying position. The position should be set to one on the first call.</td>
</tr>
<tr>
<td>Result:</td>
<td>• 2: Decode data too small, more members in structure</td>
</tr>
<tr>
<td></td>
<td>• 1: Structure too small, more members in decode string</td>
</tr>
<tr>
<td></td>
<td>• 0: Decoded OK</td>
</tr>
<tr>
<td></td>
<td>• -1: Decode variable type mismatch</td>
</tr>
<tr>
<td></td>
<td>• -2: Decode data too small, decoder ran out of data</td>
</tr>
<tr>
<td></td>
<td><strong>VARIABLE_TO_STRING</strong> (VARIABLE ENCODE)</td>
</tr>
<tr>
<td>Parameters:</td>
<td>• ENCODE: Any type of variable. This is the variable to be encoded.</td>
</tr>
<tr>
<td></td>
<td>• BUFFER: This is where the encode data is placed.</td>
</tr>
<tr>
<td></td>
<td>• POSITION: This is where the first byte of the encoding is placed. Is it also modified to point to the next location after the last encoded byte. That means that successive calls to this function can be made without modifying position. Position should be set to one on the first call.</td>
</tr>
<tr>
<td>Result:</td>
<td>0: Encoded OK</td>
</tr>
<tr>
<td></td>
<td>-1: Encoded variable unrecognized type</td>
</tr>
<tr>
<td></td>
<td>-2: Encoded data would not fit into buffer; the buffer is too small.</td>
</tr>
<tr>
<td><strong>LENGTH_VARIABLE_TO_STRING</strong> (VARIABLE Encode)</td>
<td>This routine calculates how many bytes it takes to encode a variable.</td>
</tr>
<tr>
<td>Parameters:</td>
<td>• Encode: The variable (any type) to be encoded.</td>
</tr>
<tr>
<td>Result:</td>
<td>&gt;0: Number of bytes required to encode variable.</td>
</tr>
<tr>
<td></td>
<td>0: Encoded variable error, unrecognized type</td>
</tr>
</tbody>
</table>

### Example

```c
DEFINE_TYPE

STRUCTURE _AlbumStruct
{
    LONG lTitleID
    CHAR sArtist[100]
    CHAR sTitle[100]
}
```
Encode / Decode Keywords (Cont.)

### LENGTH_VARIABLE_TO_XML

**Syntax:**
```c
CHAR LENGTH_VARIABLE_TO_XML(CONSTANT VARIANTARRAY A, LONG B)
```

**Where:**
- `A` is the variable (any type) to be encoded.
- `B` is the encoding flag. These can be used together.
  - Value `$01` is "Encode with Types". If the bit is set, types will be included for every variable being encoded. The default is to not include types.
  - Value `$10` is "Encoded CHAR arrays as using data list". See Binary Array Encoding on page 223.
  - Value `$20` is "Array Encoding is Little-Ending".

The return is the length needed to encode the variable.

### VARIABLE_TO_XML

**Syntax:**
```c
SINTEGER VARIABLE_TO_XML(CONSTANT VARIANTARRAY A, CHAR B[], LONG C, LONG D)
```

**Where:**
- `A` is the variable (any type) to be encoded:
- `B` is the CHAR array to hold the resulting XML.
- `C` is the beginning encoding position. Encoding will start as `B[C]`.
- `D` is the encoding flag. These can be used together.
  - Value `$01` is "Encode with Types". If the bit is set, types will be included for every variable being encoded. The default is to not include types. The constant `XML_ENCODE_TYPES` can be used to specify this flag.
  - Value `$10` is "Encoded CHAR arrays as using data list". The constant `XML_ENCODE_CHAR_AS_LIST` can be used to specify this flag. See the Encoding and Decoding: Binary and XML section on page 175.
  - Value `$20` is "Array Encoding is Little-Ending". The constant `XML_ENCODE_LE` can be used to specify this flag.

**Return:**
- `3` = XML decode data type mismatch
- `2` = XML decode data too small, more members in structure
- `1` = structure too small, more members in XML decode string
- `0` = decoded OK
- `-1` = decode variable type mismatch
- `-2` = decode data too small, decoder ran out of data. Most likely poorly formed XML.
- `-3` = output character buffer was too small

**Example:**
```c
DEFINE_TYPE

STRUCTURE _AlbumStruct
{
    LONG lTitleID
    CHAR sArtist[100]
    CHAR sTitle[100]
}

DEFINE_VARIABLE

_AlbumStruct MyAlbumStruct[3]
LONG lPos
SLONG slReturn
SLONG slFile
SLONG slResult
CHAR sBinaryString[10000]
CHAR sXMLString[50000]

DEFINE_START

MyAlbumStruct[1].lTitleID = 11101000
MyAlbumStruct[1].sArtist = 'Buffet, Jimmy'
MyAlbumStruct[1].sTitle = 'Living & Dying in ¾ Time'
MyAlbumStruct[2].lTitleID = 11101012
MyAlbumStruct[2].sArtist = 'Sinatra, Frank'
MyAlbumStruct[2].sTitle = 'Come Fly With Me'
MyAlbumStruct[3].lTitleID = 33101000
MyAlbumStruct[3].sArtist = 'Holiday, Billie'
MyAlbumStruct[3].sTitle = 'Lady in satin'

DEFINE_EVENT

BUTTON_EVENT[TP,1] / /Convert And Save
{
    PUSH:
    {
        // Convert To Binary
        lPos = 1
        slReturn = VARIABLE_TO_STRING(MyAlbumStruct, sBinaryString, lPos)
        SEND_STRING 0,"'POSITION=',ITOA(lPos),', Result = ',ITOA(slReturn)"
        // Convert To XML
        lPos = 1
        slReturn = VARIABLE_TO_XML(MyAlbumStruct, sXMLString, lPos, 0)
        SEND_STRING 0,"'POSITION=',ITOA(lPos),', Result = ',ITOA(slReturn)"
    }
```
Encode / Decode Keywords (Cont.)

VARIABLE_TO_XML (Cont.)

// Save Structure to Disk - Binary
slFile = FILE_OPEN('BinaryEncode.xml', 2)
slReturn = FILE_WRITE(slFile, sBinaryString, LENGTH_STRING(sBinaryString))
slReturn = FILE_CLOSE(slFile)

// Save Structure To Disk – XML
slFile = FILE_OPEN('xmlEncode.xml', 2)
slReturn = FILE_WRITE(slFile, sXMLString, LENGTH_STRING(sXMLString))
slReturn = FILE_CLOSE(slFile)

RELEASE:
{
}

BUTTON_EVENT[TP,2] // Read and Decode
{
PUSH:
{
  // Read Binary File
  slFile = FILE_OPEN('BinaryEncode.xml',1)
  slResult = FILE_READ(slFile, sBinaryString, MAX_LENGTH_STRING(sBinaryString))
  slResult = FILE_CLOSE (slFile)
  // Read XML File
  slFile = FILE_OPEN('XMLEncode.xml',1)
  slResult = FILE_READ(slFile, sXMLString, MAX_LENGTH_STRING(sXMLString))
  slResult = FILE_CLOSE (slFile)
}

RELEASE:
{
}

// Convert To Binary
lPos = 1
slReturn = STRING_TO_VARIABLE(MyAlbumStruct, sBinaryString, lPos)

// OR Convert To XML
slPos = 1
slReturn = XML_TO_VARIABLE (MyAlbumStruct, sXMLString, slPos, 0)

XML_TO_VARIABLE

Syntax:
SINTEGER XML_TO_VARIABLE(VARIANTARRAY A,CONSTANT CHAR B[],LONG C, LONG D)

Where:
• A is the variable (any type) to be encoded.
• B is the CHAR array holding the source XML.
• C is the next beginning encoding position. Encoding ended at B[C-1].
• D are the decoding flags. They can be used together.

Value $01 is "Force Types When Decoding". If the type in the XML does not match the variable typed being decoded to, the variable will not be written and the variable will be skipped in the XML. The constant XML_DECODE_TYPES can be used to specify this flag.

Value $10 is "Do Not preserve current value of A". If set, A will be cleared if not explicitly set. The constant XML_DECODE_NO_PRESERVE can be used to specify this flag.

The return value is:
3 = XML decode data type mismatch
2 = XML decode data too small, more members in structure
1 = Structure too small, more members in XML decode string
0 = Decoded OK
-1 = Decode variable type mismatch
-2 = Decode data too small, decoder ran out of data. Most likely poorly formed XML.
-3 = Output character buffer was too small.

Example:
DEFINE_TYPE

  STRUCTURE _AlbumStruct
  {
    LONG lTitleID
    CHAR sArtist[100]
    CHAR sTitle[100]
  }
XML_TO_VARIABLE (Cont.)

```c
DEFINE_VARIABLE
  _AlbumStruct MyAlbumStruct[3]
  LONG lPos
  SLONG slReturn
  SLONG slFile
  SLONG slResult
  CHAR sBinaryString[10000]
  CHAR sXMLString[50000]

DEFINE_START
MyAlbumStruct[1].lTitleID = 11101000
MyAlbumStruct[1].sArtist = 'Buffet, Jimmy'
MyAlbumStruct[1].sTitle = 'Living & Dying in ¾ Time'
MyAlbumStruct[2].lTitleID = 11101012
MyAlbumStruct[2].sArtist = 'Sinatra, Frank'
MyAlbumStruct[2].sTitle = 'Come Fly With Me'
MyAlbumStruct[3].lTitleID = 33101000
MyAlbumStruct[3].sArtist = 'Holiday, Billie'
MyAlbumStruct[3].sTitle = 'Lady in satin'

DEFINE_EVENT
  BUTTON_EVENT[TP,1] //Convert And Save
  {
    PUSH:
    {
      // Convert To Binary
      lPos = 1
      slReturn = VARIABLE_TO_STRING(MyAlbumStruct, sBinaryString, lPos)
      SEND_STRING 0, "'POSITION=',ITOA(lPos),', Result = ',ITOA(slReturn)"
    }
    // Convert To XML
    lPos = 1
    slReturn = VARIABLE_TO_XML(MyAlbumStruct, sXMLString, lPos, 0)
    SEND_STRING 0, "'POSITION=',ITOA(lPos),', Result = ',ITOA(slReturn)"
    // Save Structure To Disk - Binary
    slFile = FILE_OPEN('BinaryEncode.xml', 2)
    slReturn = FILE_WRITE(slFile, sBinaryString, LENGTH_STRING(sBinaryString))
    slReturn = FILE_CLOSE(slFile)
    // Save Structure To Disk – XML
    slFile = FILE_OPEN('XMLEncode.xml', 2)
    slReturn = FILE_WRITE(slFile, sXMLString, LENGTH_STRING(sXMLString))
    slReturn = FILE_CLOSE(slFile)
  }
  RELEASE:
  {
  }

  BUTTON_EVENT[TP,2] // Read and Decode
  {
    PUSH:
    {
      // Read Binary File
      slFile = FILE_OPEN('BinaryEncode.xml',1)
      slResult = FILE_READ(slFile, sBinaryString, MAX_LENGTH_STRING(sBinaryString))
      slResult = FILE_CLOSE (slFile)
      // Read XML File
      slFile = FILE_OPEN('XMLEncode.xml',1)
      slResult = FILE_READ(slFile, sXMLString, MAX_LENGTH_STRING(sXMLString))
      slResult = FILE_CLOSE (slFile)
      // Convert To Binary
      lPos = 1
      slReturn = STRING_TO_VARIABLE(MyAlbumStruct, sBinaryString, slPos)
      // OR Convert To XML
      slPos = 1
      slReturn = XML_TO_VARIABLE (MyAlbumStruct, sXMLString, slPos, 0)
    }
    RELEASE:
    {
    }
  }
```
Event Handler Keywords

Overview

NetLinx provides a special program section called DEFINE EVENT to define handlers for incoming events/notifications. These handlers are stored in an event table providing quick access to code that must be executed when an event is received. There are handlers to support five types of events:

- **Button events** include pushes, releases, and holds, which are associated with a push or release on a particular device-channel.
- **Channel events** occur when an output change (On/Off) is detected on a device-channel.
- **Data events** include commands, strings, status, and error messages.
- **Level events** are received as a result of a level change on a particular device.
- **Timeline events** trigger events based on a sequence of times.

**NOTE:** The processing of an event associated with a given member of a device, channel, device-channel, level, or device-level array must be completed before processing can begin on another event associated with the same array.

All incoming events are stored in a queue pending processing. Messages are processed in the order they are received. The steps to processing an event are:

1. Check all events for a handler matching the specified event. If a handler is found, run it.
2. If there is no event handler, run MAINLINE.

![Diagram of event processing](image)

**FIG. 2** Processing an Event

**NOTE:** More than one handler can be defined for the same event. In this case, the handlers are executed in the order in which they are defined in the program.

The event handler descriptions are:

- **DEVICE** refers to a device specification:
  - `DEVICE` A single device number constant
  - `D:P:S` A constant device specification such as 128:1:0
  - `DEV[ ]` A device array

- **CHANNEL** refers to:
  - `CHANNEL` A single channel number constant
  - `CHAN[ ]` An integer array of channel numbers
  - `DEVCHAN[ ]` A device-channel array

- **LEVEL** refers to:
  - `LEVEL` A single level number constant
  - `LEV[ ]` An integer array of level numbers
  - `DEVLEV[ ]` A device-level array

**NOTE:** The processing of an event associated with a given member of a device, channel, device-channel, level, or device-array must be completed before processing can begin on another event associated with the same array.

**Button Events**

Events associated with a button on a touch panel or keypad are referred to as **Button Events**. Button events include pushes, releases, and holds. These events are associated with a push or release on a particular device-channel.

The format for a Button Event is shown below:
**Event Handler Keywords**

```plaintext
BUTTON_EVENT[DEVICE,CHANNEL] or BUTTON_EVENT[(DEVCHAN[ ])]
{
    PUSH:
    {
        // PUSH event handler
    }
    RELEASE:
    {
        // RELEASE event handler
    }
    HOLD[TIME]: or HOLD[TIME, REPEAT]:
    {
        // HOLD event handler
    }
}
```

- The `[<DEVICE>, <CHANNEL>]` declaration can contain a DEV device set, or a DEVCHAN device-channel set in addition to individual device and channel declarations.
- A HOLD event handler specifies the actions that should be performed when a button is pressed and held for a minimum length of time indicated by the TIME parameter (TIME is specified in .10 second increments).
- The REPEAT keyword specifies that the event notification should be repeated in TIME increments as long as the button is held.
- The BUTTON object is available to the button event handler as a local variable. The following table lists the information contained in Button Objects.

### Button Objects

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Button.Input</td>
<td>DEVCHAN</td>
<td>The device, channel combination that caused the BUTTON_EVENT to happen.</td>
</tr>
<tr>
<td>Button.Input.Channel</td>
<td>INTEGER</td>
<td>The channel number that caused the BUTTON_EVENT to happen.</td>
</tr>
<tr>
<td>Button.Input.Device</td>
<td>DEV</td>
<td>The device that caused the BUTTON_EVENT to happen.</td>
</tr>
<tr>
<td>Button.Input.Device.Number</td>
<td>INTEGER</td>
<td>The device number of the device that caused the BUTTON_EVENT to happen.</td>
</tr>
<tr>
<td>Button.Input.Device.Port</td>
<td>INTEGER</td>
<td>The port of the device that caused the BUTTON_EVENT to happen.</td>
</tr>
<tr>
<td>Button.Input.Device.System</td>
<td>INTEGER</td>
<td>The system number of the device that caused the BUTTON_EVENT to happen.</td>
</tr>
<tr>
<td>Button.HoldTime</td>
<td>LONG</td>
<td>Current hold time in one-millisecond (1 ms) increments</td>
</tr>
<tr>
<td>Button.SourceDev</td>
<td>DEV</td>
<td>Source device of button event. (This property is no longer used.)</td>
</tr>
<tr>
<td>Button.SourceDev.Number</td>
<td>INTEGER</td>
<td>Source device number. (This property is no longer used.)</td>
</tr>
<tr>
<td>Button.SourceDev.Port</td>
<td>INTEGER</td>
<td>Source device port. (This property is no longer used.)</td>
</tr>
<tr>
<td>Button.SourceDev.System</td>
<td>INTEGER</td>
<td>Source device system. (This property is no longer used.)</td>
</tr>
</tbody>
</table>

If the event handler is specified using an array for DEV,CHANNEL, or a DEVCHAN array, GET_LAST can determine which index in the array caused the event to run.

### Channel Events

Channel Events are similar to Button Events. Channel Events are generated by ON, OFF, PULSE, TO, or MIN_TO. The format for a Channel Event:

```plaintext
CHANNEL_EVENT[DEVICE,CHANNEL] or CHANNEL_EVENT[(DEVCHAN[ ])]
{
    ON:
    {
        // CHANNEL ON event handler
    }
    OFF:
    {
        // CHANNEL OFF event handler
    }
}
```

**NOTE:** The Channel object is available to the Channel Event handler as a local variable.

Like Button Events, the `[<device>, <channel>]` declaration can contain a DEV device set, or a DEVCHAN device-channel set in addition to individual device and channel declarations. In the following example, a Channel Event is defined for the 'Projector Lift Up' relay, telling the system to turn off the projector every time this relay is turned on:

```plaintext
DEFINE_EVENT
.
BUTTON_EVENT[TP1,21] (* LIFT UP BUTTON *)
{
    PUSH:
    {
        PULSE[RELAY,LIFT_UP]
    }
}
```
Event Handler Keywords

BUTTON_EVENT[TP1,22] (* SYSTEM OFF BUTTON *)
{
    PUSH:
    
    PULSE[RELAY,RACK_OFF]
    PULSE[RELAY,LIFT_UP]
}
}

CHANNEL_EVENT[RELAY,LIFT_UP] (* LIFT UP RELAY EVENT *)
{
    ON:
    
    PULSE[VPROJ,VP_POWER_OFF]
}

**NOTE:** Since turning on or pulsing the relay does not produce a push, a Button Event is not generated.

The following table lists the information contained in Channel events:

<table>
<thead>
<tr>
<th>Channel Objects</th>
<th>Property Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Channel.Device</td>
<td>DEV</td>
<td>The device that caused the CHANNEL_EVENT to happen.</td>
</tr>
<tr>
<td></td>
<td>Channel.Device.Number</td>
<td>INTEGER</td>
<td>The device number of the device that caused the CHANNEL_EVENT to happen.</td>
</tr>
<tr>
<td></td>
<td>Channel.Device.Port</td>
<td>INTEGER</td>
<td>The port of the device that caused the CHANNEL_EVENT to happen.</td>
</tr>
<tr>
<td></td>
<td>Channel.Device.System</td>
<td>INTEGER</td>
<td>The system number of the device that caused the CHANNEL_EVENT to happen.</td>
</tr>
<tr>
<td></td>
<td>Channel.Channel</td>
<td>INTEGER</td>
<td>The channel number that caused the CHANNEL_EVENT to happen.</td>
</tr>
<tr>
<td></td>
<td>Channel.SourceDev</td>
<td>DEV</td>
<td>Source Device of Channel Event</td>
</tr>
<tr>
<td></td>
<td>Channel.SourceDev.Number</td>
<td>INTEGER</td>
<td>Source Device Number</td>
</tr>
<tr>
<td></td>
<td>Channel.SourceDev.Port</td>
<td>INTEGER</td>
<td>Source Device Port</td>
</tr>
<tr>
<td></td>
<td>Channel.SourceDev.System</td>
<td>INTEGER</td>
<td>Source Device System</td>
</tr>
</tbody>
</table>

If the event handler is specified using an array for DEV, CHANNEL, or a DEVCHAN array, GET_LAST can be used to determine which index in the array caused the event to run.

**Data Events**

Data Events provide some interesting capabilities in a NetLinx system. At first glance, it seems to be concerned with receiving strings of data either from a serial data device (such as an NXC-COM2 card) or an interface device (such as a touch panel or keypad). While this is a valid function, DATA_EVENT has many more capabilities and works with many devices.

The format for Data Events:

DATA_EVENT[DEVICE]
{
    COMMAND:
    
    // Command processing goes here

    STRING:
    
    // String processing goes here

    ONLINE:
    
    // OnLine processing goes here

    OFFLINE:
    
    // OffLine processing goes here

    ONERROR:
    
    // OnError processing goes here

    STANDBY:
    
    // Standby processing goes here

    AWAKE:
    
    // Awake processing goes here
}

**NOTE:** The data object is available to the Data Event handler as a local variable.

The following table lists the information contained in data objects:
The Event Number is a number associated with a command, error condition or the device ID associated with an online/offline event. The numeric value is stored as either a floating-point number or integer, as appropriate; the value can be assigned to a variable of any numeric type. This field could be a value associated with a command event or error condition.

Text Associated with The Event is associated with a command, string, or error notification. It can also be the device ID string in the case of an online/offline event.

A device triggers an ONLINE event when the master recognizes a device has come on the bus or has dropped off the bus.

NetLinx has the ability to process data received by a DATA EVENT in real time. When data is received, it enters the message queue and triggers a data event. If a buffer has been created for the device, the data is placed within the buffer and can be used by either the DATA_EVENT or mainline. The data can be evaluated in two ways. The actual string that is received by the message queue can be evaluated using the DATA.TEXT object within the event. The string in DATA.TEXT is also added to the end of the device's buffer. This becomes a factor when receiving large strings, or when receiving strings with an embedded string length or start and end characters. DATA_EVENT then evaluates the buffer to see if the entire string has been received before processing it; however, the evaluation is done immediately upon receipt of another chunk of data, and is only done when data is received.

For example, DATA.TEXT may equal ‘over the lazy brown dog’.ETX and the DATA_BUFFER[500] might equal (STX,’The quick gray fox jumps over the lazy brown dog’.ETX). By evaluating the buffer value, you can evaluate the entire string at once.

Two other important aspects of the DATA_EVENT are the ONLINE and OFFLINE event handlers. ONLINE and OFFLINE events are triggered when the master recognizes a device has come on the bus or has dropped off the bus.
DEFINE_START
CREATE_BUFFER TP1, TP1_BUFFER 
.
DEFINE_EVENT
-- DATA_EVENT[TP1] (* EVALUATE TP1 DATA *)
{
  STRING:
  {
    SELECT
    {
      ACTIVE (FIND_STRING (DATA.TEXT,'PAGE-',1)):
      {
        JUNK = REMOVE_STRING (DATA.TEXT,'PAGE-',1)
        CUR_PAGE = DATA.TEXT
      }
      ACTIVE (FIND_STRING (DATA.TEXT,'KEYP-',1)):
      {
        (* keypad code *)
      }
      ACTIVE (FIND_STRING (DATA.TEXT,'KEYB-',1)):
      {
        (* keyboard code *)
      }
      ACTIVE (1):
      {
        (* default code *)
      }
    }
    CLEAR_BUFFER TP1_BUFFER
  }
  ONLINE:
  {
    SEND_COMMAND TP1, 'TPAGEON'
  }
}
.
.
Each event handler contains several embedded data objects that pass data values into the event handler code.

Level Events

Level Events are triggered by a level change on a particular device. This eliminates constantly evaluating a level against a previous value. The format for LEVEL_EVENTS:

LEVEL_EVENT[DEVICE,LEVEL] or LEVEL_EVENT[(DEVLEV[])]
{
    // level event handler
}

**NOTE:** The level object is available to the Level Event handler as a local variable.

It contains the information shown in the table below:

<table>
<thead>
<tr>
<th>Level Objects</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level.Input</td>
<td>DEVLEV</td>
<td>Device + Level that caused the event to occur</td>
</tr>
<tr>
<td>Level.Input.Device</td>
<td>DEV</td>
<td>The device that caused the LEVEL_EVENT to happen</td>
</tr>
<tr>
<td>Level.Input.Device.Number</td>
<td>INTEGER</td>
<td>The device number of the device that caused the LEVEL_EVENT to happen.</td>
</tr>
<tr>
<td>Level.Input.Device.Port</td>
<td>INTEGER</td>
<td>The level port of the bargraph that caused the LEVEL_EVENT to happen.</td>
</tr>
<tr>
<td>Level.Input.Device.System</td>
<td>INTEGER</td>
<td>The system number of the device that caused the LEVEL_EVENT to happen.</td>
</tr>
<tr>
<td>Level.Input.Level</td>
<td>INTEGER</td>
<td>The level code of the bargraph that caused the LEVEL_EVENT to happen.</td>
</tr>
<tr>
<td>Level.SourceDev</td>
<td>DEV</td>
<td>Source Device of Level Event</td>
</tr>
<tr>
<td>Level.SourceDev.Number</td>
<td>INTEGER</td>
<td>Source Device Number</td>
</tr>
<tr>
<td>Level.SourceDev.Port</td>
<td>INTEGER</td>
<td>Source Device Port</td>
</tr>
<tr>
<td>Level.SourceDev.System</td>
<td>INTEGER</td>
<td>Source Device System</td>
</tr>
<tr>
<td>Level.Value</td>
<td>Numeric</td>
<td>The value of the bargraph when the LEVEL_EVENT occurred.</td>
</tr>
</tbody>
</table>

LEVEL_VALUE is an embedded object value in the LEVEL_EVENT statement. If the event handler is specified using an array for DEV, CHANNEL, or a DEVCHAN array, GET_LAST can be used to determine which index in the array caused the event to run. The numeric value is stored either as a floating-point number or integer, as appropriate; but the value can be assigned to a variable of any numeric type.

Example Level Event:
LEVEL_EVENT [ TEMP, 1 ]
{
  IF (LEVEL.VALUE >= COOL_POINT)
  {
    ON[RELAY,FAN]
  }
  ELSE IF (LEVEL.VALUE <= HEAT_POINT)
  {
    OFF[RELAY,FAN]
  }
}

**Custom Events**

A Custom Event is generated by certain devices in response to query commands or unique device events. For instance, G4 touch panels generate custom events in response to button query commands or mouse clicks. An example channel event is shown below:

```
CUSTOM_EVENT[DEVICE,ID,TYPE] or CUSTOM_EVENT[DEVCHAN,EVENTID]
{
}
```

The EVENTID is specific to each device. For instance, the EVENTID sent in response to a button text query command for G4 touch panels is 1001.

**NOTE:** For more information on EVENTID values and the values of the custom event for each EVENTID, see the programming section of the device manual with which you are working.

The following table lists the information contained in Custom events:

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom.Device</td>
<td>DEV</td>
<td>Device</td>
</tr>
<tr>
<td>Custom.Device.Number</td>
<td>INTEGER</td>
<td>Device number</td>
</tr>
<tr>
<td>Custom.Device.Port</td>
<td>INTEGER</td>
<td>Device port</td>
</tr>
<tr>
<td>Custom.Device.System</td>
<td>INTEGER</td>
<td>System number</td>
</tr>
<tr>
<td>Custom.ID</td>
<td>INTEGER</td>
<td>The ID of the custom event as defined by the device</td>
</tr>
<tr>
<td>Custom.Type</td>
<td>INTEGER</td>
<td>The TYPE of the custom event as defined by the device</td>
</tr>
<tr>
<td>Custom.Flag</td>
<td>INTEGER</td>
<td>A flag associated with the event</td>
</tr>
<tr>
<td>Custom.Value1</td>
<td>SLONG</td>
<td>The first value associated with the event</td>
</tr>
<tr>
<td>Custom.Value2</td>
<td>SLONG</td>
<td>The second value associated with the event</td>
</tr>
<tr>
<td>Custom.Value3</td>
<td>SLONG</td>
<td>The third value associated with the event</td>
</tr>
<tr>
<td>Custom.Text</td>
<td>CHAR[]</td>
<td>Text associated with the event</td>
</tr>
<tr>
<td>Custom.Encode</td>
<td>CHAR[]</td>
<td>A string encoded with VARIABLE_TO_STRING encoding for complex data types.</td>
</tr>
<tr>
<td>Custom.SourceDev</td>
<td>DEV</td>
<td>Source device of custom event</td>
</tr>
<tr>
<td>Custom.SourceDev.Number</td>
<td>INTEGER</td>
<td>Source device number</td>
</tr>
<tr>
<td>Custom.SourceDev.Port</td>
<td>INTEGER</td>
<td>Source device port</td>
</tr>
<tr>
<td>Custom.SourceDev.System</td>
<td>INTEGER</td>
<td>Source device system</td>
</tr>
</tbody>
</table>

If the event handler is specified using an array for DEV, INTEGER, or a DEVCHAN array, GET_LAST can determine which index in the array caused the event to run.

**Event Parameters**

It has already been stated that DEFINE_EVENT handlers are stored in an event table providing quick access to code that must be executed when an event is received. The event table keeps a list of all events in a sorted order to more quickly determine which code needs to be accessed for a given incoming event. The event table is built before DEFINE_START runs and it not changed anytime after that. As a result, there are certain rules that must be applied to the parameters used in DEFINE_EVENTs.

Since the event table is built before DEFINE_START, all event parameters must contain the correct information prior to DEFINE_START. This requires that all EVENT parameters must be defined at compile time. In addition, many parameter "shortcuts" to help fulfill this requirement. Using BUTTON_EVENT as an example, the simplest version of event parameters is a device and channel reference. In the following example:

```
DEFINE_DEVICE
dvTp = 128:1:0

DEFINE_EVENT

BUTTON_EVENT[dvTp,1]
{
  PUSH:
    SEND_STRING 0,'Button 1 of dvTp was pushed'
}
```

The device, dvTp, was defined in the DEFINE_DEVICE section, which has the effect of making it an initialized variable of type DEV, and the channel number was a hard-coded value of 1. Since both of these value were defined at compile time, the event is entered into the event table correctly. Let's take another example:
DEFINE_DEVICE
dvTp = 128:1:0
DEFINE_VARIABLE
Integer nMyChannel
DEFINE_START
nMyChannel = 1
DEFINE_EVENT
BUTTON_EVENT[dvTp,nMyChannel]
{
   PUSH:
   Send_String 0,"'Button ',ITOA(nMyChannel),' of dvTp was pushed'"
}

In this example, the event will not perform as the previous one did. When the code is compiled, the event parameters are dvTp, which is already assigned, and nMyChannel, which has a value of 0. nMyChannel does not get assigned a value of 1 until DEFINE_START, at which time the event has already been added to the event table. If you were to run this code, you would discover that it did in fact run when button 1 was pushed, leading us to one of the "shortcuts":

**NOTE:** A value of 0 for a Channel or Level Number in a BUTTON_EVENT, CHANNEL_EVENT or LEVEL_EVENT will be interpreted as an event handler for all events of that type from the given device number(s).

So, the reason the above example runs when button 1 was pushed is that the above example runs when any button on dvTp is pushed. This "shortcut" was added so you could define an event handler for all buttons, channel or levels of a device without having to define a DEVCHAN of DEVLEV containing every value you may want to handle. To make the example 2 behave like the example 1, we simply need to make sure the value of nMyChannel contains a value of 1 at compile time. This is simply done by initializing nMyChannel a value of 1 in the DEFINE_VARIABLE section. The new example reads:

**Example 3:**
DEFINE_DEVICE
dvTp = 128:1:0
DEFINE_VARIABLE
Integer nMyChannel = 1
DEFINE_EVENT
BUTTON_EVENT[dvTp,nMyChannel]
{
   PUSH:
   Send_String 0,"'Button ',ITOA(nMyChannel),' of dvTp was pushed'"
}

You may be tempted to use a more traditional variable as the channel number, mainly PUSH_CHANNEL or RELEASE_CHANNEL. It is important to realize that the identifiers are nothing more than global (system) variable. At compile time, the values are defined and contain a value of 0. So the following code will have the effect you expect button probably for a different reason than you expect.

**DEFINE_EVENT**
BUTTON_EVENT[dvTp,PUSH_CHANNEL]
{
   PUSH:
   Send_String 0,"'Button ',ITOA(BUTTON.INPUT.CHANNEL),' of dvTp was pushed'"
   RELEASE:
   Send_String 0,"'Button ',ITOA(BUTTON.INPUT.CHANNEL),' of dvTp was released'"
}

Although the event will run for both the push and release of all buttons for dvTp, you may also be tempted to think that you need to make sure the event runs for RELEASE_CHANNEL by adding the following:

**DEFINE_EVENT**
BUTTON_EVENT[dvTp,PUSH_CHANNEL]
BUTTON_EVENT[dvTp,RELEASE_CHANNEL]
{
   PUSH:
   Send_String 0,"'Button ',ITOA(BUTTON.INPUT.CHANNEL),' of dvTp was pushed'"
   RELEASE:
   Send_String 0,"'Button ',ITOA(BUTTON.INPUT.CHANNEL),' of dvTp was released'"
}

However, since both PUSH_CHANNEL and RELEASE_CHANNEL have a value of 0 at compile time, you are in effect stacking two events that are interpreted as running for any button pushed on the panel and as a result, the event is run twice every time a button is pushed or released. This may not seem like a big problem until you try to toggle a variable in the event: since the event runs twice for every button push, the variable toggles on then toggles off again.

There are some additional parameter "shortcuts" available. In all cases, the following rules apply:

- When a DEV can be used, a DEV array can also be used.
- When a DEVCHAN can be used, a DEVCHAN array can be used.
- When a DEVLEV can be used, a DEVLEV array can be used.
When a Char, Integer or Long can be used, a Char, Integer or Long array can also be used.

You can apply more than 1 of the above rules at a time in a given event handler.

GET_LAST() can be used to determine which index of an array (any type) caused the event to fire.

The above rules can let you write some interesting event handler. Let’s say you wanted to handle 4 buttons from 6 panels all with one button event. You could write:

```python
DEFINE_DEVICE
dvPanel1 = 128:1:0
dvPanel2 = 129:1:0
dvPanel3 = 130:1:0
dvPanel4 = 131:1:0
dvPanel5 = 132:1:0
dvPanel6 = 133:1:0
DEFINE_VARIABLE
DEV dvMyPanels[] = {dvPanel1, dvPanel2, dvPanel3, dvPanel4, dvPanel5, dvPanel6}
INTEGER nMyButtons[] = {4, 3, 2, 1}
INTEGER nPanelIndex
INTEGER nButtonIndex
DEFINE_EVENT
BUTTON_EVENT[dvMyPanels, nMyButtons]
{
  PUSH:
  {
    nPanelIndex = GET_LAST(dvMyPanels)
    nButtonIndex = GET_LAST(nMyButtons)
    Send_String 0, "'Button Index=', ITOA(nButtonIndex), ' was pushed on Panel Index=', ITOA(nPanelIndex)"
  }
}
```

This event will be run for all combinations of `dvMyPanel` and `nMyButtons`, 24 buttons in all. The `GET_LAST()` function is very useful when running event using array as parameters. `GET_LAST()` returns an index value, starting at 1, for the element that triggered the event. In the case of `nButtonIndex`, it will contain a value of 1 when button 4 was pressed, a value of 2 when button 3 was pressed... This can be very useful in the case of transmitters and wired panels where the channel number may not reflect a numerical sequence you would like, such as with Numeric Keypads.

**Event Handler Keywords**

The NetLinx programming language supports the following Event Handler keywords:

<table>
<thead>
<tr>
<th>Event Handler Keywords</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BUTTON_EVENT</strong></td>
<td>Defines a button event handler and can only be used in the DEFINE_EVENT section of the program. This type of handler processes PUSH, RELEASE, and HOLD events.</td>
</tr>
</tbody>
</table>
| ```python
  BUTTON_EVENT[DEVICE, CHANNEL] or
  BUTTON_EVENT[DEVCHAN[]]
  {
    PUSH:
    {
      // Push statements go here
    }
    RELEASE:
    {
      // Release statements go here
    }
    HOLD[TIME, [REPEAT]]:
    {
      // Hold statements go here
    }
  }
``` |
| **CHANNEL_EVENT** | Defines a channel event handler. This type of handler is invoked when an output change occurs on the specified device-channel and can only be used in the DEFINE_EVENT section of the program. |
| ```python
  CHANNEL[DEVICE, CHANNEL] or
  CHANNEL[DEVCHAN[]]
  {
    ON:
    {
      // Channel ON event handling
    }
    OFF:
    {
      // Channel OFF event handling
    }
  }
``` |

DEVICE refers to:
- Device – a single device number constant.
- D:P:S – a constant device specification such as TP:1:0.

CHANNEL refers to:
- Channel – a single channel number constant.
- DEVCHAN[] refers to a device-channel array.
### Event Handler Keywords (Cont.)

#### DATA_EVENT

Defines a data event handler. This type of handler processes COMMAND, STRING, ONLINE, OFFLINE and ONERROR events. It can only be used in the DEFINE_EVENT section of the program.

```plaintext
DATA_EVENT[DEVICE]
{
    COMMAND:
    {
        // Command processing goes here
    }
    STRING:
    {
        // String processing goes here
    }
    ONLINE:
    {
        // OnLine processing goes here
    }
    OFFLINE:
    {
        // OffLine processing goes here
    }
    ONERROR:
    {
        // OnError processing goes here
    }
    STANDBY:
    {
        // Standby processing goes here
    }
    AWAKE:
    {
        // Awake processing goes here
    }
}
```

See the Data Events on page 76 for more information.

#### LEVEL_EVENT

Defines a level event handler and can only be used in the DEFINE_EVENT section of the program. This type of handler is invoked when a level change occurs on the specified device-channel. The level object is available to the level event handler as a local variable.

```plaintext
LEVEL_EVENT[DEVICE,LEVEL] or LEVEL_EVENT[DEVLEV[]]
{
    // level event handler
}
```

See the Level Events on page 78 for more information.

#### REBUILD_EVENT()

This function rebuilds the NetLinx event table for level, channel, button, timeline, and data events.

- Modifications to variables used in event declarations affect NetLinx event handling when REBUILD_EVENT() is called after the variables are modified.
- REBUILD_EVENT() works on a module-by-module basis (i.e. calling the function in one module does not affect the event table of another module).
- REBUILD_EVENT() rebuilds the event table for variables modified in the same block of code in which it resides.
- With no braces, a REBUILD_EVENT() in DEFINE_START rebuilds event tables that use any variable modified in DEFINE_START, above the REBUILD_EVENT() statement.

You can reduce the scope of the REBUILD_EVENT() by delineating a block with braces as shown at the bottom of the following example:

```plaintext
// end
// REBUILD_EVENT() rebuilds the event table for variables modified in the same block of code in which it resides. With no braces, a REBUILD_EVENT() in DEFINE_START should rebuild the event tables that use any variable modified in DEFINE_START, above the REBUILD_EVENT() statement.
// You can reduce the scope of the REBUILD_EVENT() by delineating a block with braces:
DEFINE_DEVICE
dvTP = 10001:1:0
DEFINE_VARIABLE
INTEGER X // loop counter
INTEGER nBTNS[4000]
DEFINE_START
FOR (X = 1; X <= 4000; X++)
{
    nBtns[X] = X
}
// the braces below enclose a variable update and REBUILD_EVENT statement in a single block
SET_LENGTH_ARRAY(nBtns,4000)
REBUILD_EVENT()
```
Event Handler Keywords (Cont.)

<table>
<thead>
<tr>
<th>REBUILD_EVENT() (Cont.)</th>
<th>BUTTON_EVENT[dvTP,nBtns]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>PUSH:</td>
</tr>
<tr>
<td></td>
<td>// ...</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>// end</td>
</tr>
</tbody>
</table>

The code below demonstrates how to use the NetLinx REBUILD_EVENT(). function:

```plaintext
DEFINE_DEVICE
dvApoc1 = 128:1:0
dvApoc2 = 1505:1:0
dvApoc3 = 1303:1:0
(* CONSTANT DEFINITIONS GO BELOW *)
(*--------------------------------------------------------------------------*)
DEFINE_CONSTANT
DEV panel[] = {dvApoc1,dvApoc2}
(*--------------------------------------------------------------------------*)
(* DEFINE TYPE DEFINITIONS GO BELOW *)
(*--------------------------------------------------------------------------*)
DEFINE_TYPE
(*--------------------------------------------------------------------------*)
(* VARIABLE DEFINITIONS GO BELOW *)
(*--------------------------------------------------------------------------*)
DEFINE_VARIABLE
DEV curModApoc
(*--------------------------------------------------------------------------*)
(* EVENT DEFINITIONS GO BELOW *)
(*--------------------------------------------------------------------------*)
DEFINE_EVENT
BUTTON_EVENT[panel,1]
{      
  PUSH:     
  ON[panel,1]
  curModApoc = dvApoc2
  // updates program event table to handle BUTTON_EVENT[1505:1:0,5]
  REBUILD_EVENT()
  }  
  RELEASE: OFF[panel,1]
}  
BUTTON_EVENT[panel,2]
{      
  PUSH:     
  ON[panel,2]
  curModApoc = dvApoc3
  // updates program event table to handle BUTTON_EVENT[1303:1:0, 5]
  REBUILD_EVENT()
  // the following assignment has no affect on the program event table     
  curModApoc = dvApoc1
  }  
  RELEASE: OFF[panel,2]
}  
BUTTON_EVENT[curModApoc,5]
{      
  PUSH: ON[dvApoc3,5]
  RELEASE: OFF[dvApoc3,5]
}  
```
# File Operation Keywords

NetLinx supports the following File Operation keywords:

<table>
<thead>
<tr>
<th><strong>File Operation Keywords</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FILE_CLOSE</strong></td>
<td>This function closes a file opened with <code>FILE_OPEN</code>. This function should be called when all reading or writing to the file is completed.</td>
</tr>
<tr>
<td><code>SLONG File_Close (LONG hFile)</code></td>
<td>Parameters:</td>
</tr>
<tr>
<td>• hFile: Handle to the file returned by <code>FILE_OPEN</code>.</td>
<td>Result:</td>
</tr>
<tr>
<td>0: Operation was successful</td>
<td>- Operation was successful</td>
</tr>
<tr>
<td>-1: Invalid file handle</td>
<td>-1: Invalid file handle</td>
</tr>
<tr>
<td>-5: Disk I/O error</td>
<td>-5: Disk I/O error</td>
</tr>
<tr>
<td>-7: File already closed</td>
<td>-7: File already closed</td>
</tr>
<tr>
<td>There is a limit to the number of file handles available from the system. If files are not closed, it may not be possible to open a file.</td>
<td>Result = File_Close(hFile)</td>
</tr>
</tbody>
</table>

| **FILE_COPY** | This function copies the specified file. |
| `SLONG File_Copy (CHAR SrcFilePath[ ], CHAR DstFilePath[ ] )` | Parameters:  |
| • SrcFilePath: Path name of the file to copy (source). | Result:  |
| • DstFilePath: Path name of the copied file (destination). | 0: Operation was successful  |
| | -2: Invalid file name  |
| | -5: Disk I/O error  |
| | -11: Disk full  |
| If either path name fails to specify a directory, the current directory is assumed. The current directory is either the top-level directory or the subdirectory specified in the last call to `FILE_SETDIR`. | Result = File_Copy('OLDFILE.TXT', 'NEWFILE.TXT')  |
| Example:  |
| `CHAR Buffer[1024]` | `SLONG NumFiles = 1` |
| `LONG Entry = 1` | `WHILE (NumFiles > 0)` |
| `{` | `NumFiles = FILE_DIR ('AAA:', Buffer, Entry)` |
| `Entry = Entry + 1` | `// add code to display contents of Buffer` |
| `}` |

| **FILE_CREATEDIR** | Creates a specified directory path. |
| `SLONG File_CreateDir (CHAR DirPath[ ] )` | Syntax:  |
| This function will not create the number of subdirectories needed to complete the directory path if they do not exist. The subdirectories must be created one level at a time. | Parameters:  |
| **NOTE:** The `LONG` command cannot pass negative numbers, so if you have errors these will never be recognized. `SLONG` must be assigned or errors will be typecast to positive numbers. | • DirPath - string containing the directory path to create. | Result:  |
| 0 = operation was successful | -4 = invalid directory path  |
| -5 = disk I/O error | -13 = directory name exists  |
| Example:  |
| `File_CreateDir ('\CDLIST\')` | `File_CreateDir ('\CDLIST\TEMP\')` |
| Creates both \CDLIST and \CDLIST\TEMP subdirectories. |  |
### File Operation Keywords (Cont.)

**FILE_DELETE**

This function deletes a specified file.

```c
SLONG FILE_DELETE (CHAR FilePath[ ]) 
```

**Parameters:**
- FilePath: Path name of the file to delete.

**NOTE:** Wildcard characters (* and ?) are NOT permitted in the path name. You must use actual filenames to avoid a disk I/O error.

**NOTE:** The **LONG** command cannot pass negative numbers, so if you have errors these will never be recognized. **SLONG** must be assigned or errors will be typecast to positive numbers.

**Result:**
- 0: Operation was successful
- -2: Invalid file path or name
- -5: Disk I/O error

```c
// delete 'myFile.txt' in the directory \CDLIST\TEMP\ 
Result = FILE_DELETE('\CDLIST\TEMP\myFile.txt')
```

**FILE_DIR**

This function returns a list of files located at the specified path. The syntax:

```c
SLONG FILE_DIR (CHAR DirPath[ ], CHAR Buffer[ ], LONG Entry) 
```

**Parameters:**
- DirPath: String containing the path to the requested directory.
- Buffer: Buffer to hold the directory list.
- Entry: Requested directory entry.

This function returns the number of remaining files in the directory, or:
- -4: Invalid directory path
- -5: Disk I/O error
- -6: Invalid parameter (i.e. Entry points beyond the end of the directory, or is 0)
- -10: Buffer too small
- -12: Directory not loaded

**NOTE:** Each directory entry will have a <CR><LF> character pair appended to the end.

**NOTE:** The **LONG** command cannot pass negative numbers, so if you have errors these will never be recognized. **SLONG** must be assigned or errors will be typecast to positive numbers.

**Example:**

```c
CHAR Buffer[1024] 
LONG NumFiles = 1 
LONG Entry = 1 
WHILE (NumFiles > 0) 
{ 
    NumFiles = FILE_DIR ('\CDLIST', Buffer, Entry) 
    Entry = Entry + 1 
    // add code to display contents of Buffer 
}
```

**FILE_GETDIR**

This function returns the current working directory.

```c
SLONG FILE_GETDIR (CHAR DirPath[ ]) 
```

**Parameters:**
- DirPath: Buffer to receive the current working directory.

**Result:**
- 0: Operation was successful
- -10: Size of **DirPath** buffer insufficient to hold directory path name

```c
CHAR Buffer[256]Result = FILE_GETDIR (Buffer)
```

**FILE_OPEN**

This function opens a file for reading or writing.

```c
SLONG FILE_OPEN (CHAR FilePath[ ], LONG IODflag) 
```

**Parameters:**
- FilePath: String containing the path to the file to be opened
- IODflag:
  1. **Read**: The file is opened with READ ONLY status. The constant **FILE_READ_ONLY** is defined as a value of 1 for specifying this flag.
  2. **R/W New**: The file is opened with READ WRITE status. If the file currently exists, its contents are erased. The constant **FILE_RW_NEW** is defined as a value of 2 for specifying this flag.
  3. **R/W Append**: The file is opened with READ WRITE status. The current contents of the file are preserved and the file pointer is set to point to the end of the file. The constant **FILE_RW_APPEND** is defined as a value of 3 for specifying this flag.

If the open operation is successful, this function returns a non-zero integer value representing the handle to the file. This handle must be used in subsequent read, write, and close operations.

- >0: Handle to file (open was successful)
- -2: Invalid file path or name
- -3: Invalid value supplied for IODflag
- -5: Disk I/O error
- -14: Maximum number of files are already open (max is 10)
- -15: Invalid file format

If the file is opened successfully, it must be closed after all reading or writing is completed, by calling **FILE_CLOSE**. If files are not closed, subsequent file open operations may fail due to the limited number of file handles available.

```c
// Open MYFILE.TXT for reading hFile = FILE_OPEN('MYFILE.TXT', FILE_READ_ONLY)
```
<table>
<thead>
<tr>
<th>File Operation Keywords (Cont.)</th>
<th>FILE_OPERATIONS</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FILE_READ</strong></td>
<td>This function reads a block of data from the specified file.</td>
<td>SLONG FILE_READ (LONG hFile, CHAR Buffer[ ], LONG BufLen)</td>
</tr>
<tr>
<td>Parameters:</td>
<td>hFile: Handle to the file returned by FILE_OPEN</td>
<td></td>
</tr>
<tr>
<td>• Buffer: Buffer to hold the data to be read</td>
<td>BufLen: Maximum number of bytes to read</td>
<td></td>
</tr>
<tr>
<td>Result:</td>
<td>&gt;0: The number of bytes actually read</td>
<td></td>
</tr>
<tr>
<td>• -1: Invalid file handle</td>
<td>-5: Disk I/O error</td>
<td></td>
</tr>
<tr>
<td>• -6: Invalid parameter</td>
<td>-9: End-of-file reached</td>
<td></td>
</tr>
<tr>
<td>This function reads (from the current location of the file pointer) the number of bytes specified by BufLen (or fewer bytes if the end of file is reached). The bytes are read from the file identified by hFile and are stored in Buffer. The file pointer will automatically be advanced the correct number of bytes so the next read operation continues where the last operation left off.</td>
<td>CHAR Buffer[1024]nBytes = FILE_READ (hFile, Buffer, 1024)</td>
<td></td>
</tr>
<tr>
<td><strong>FILE_READ_LINE</strong></td>
<td>This function reads a line of data from the specified file.</td>
<td>SLONG FILE_READ_LINE (LONG hFile, CHAR Buffer[ ], LONG BufLen)</td>
</tr>
<tr>
<td>Parameters:</td>
<td>hFile: Handle to the file returned by FILE_OPEN</td>
<td></td>
</tr>
<tr>
<td>• Buffer: Buffer to hold the data to be read</td>
<td>BufLen: Maximum number of bytes to read</td>
<td></td>
</tr>
<tr>
<td>Result:</td>
<td>=0: The number of bytes actually read</td>
<td></td>
</tr>
<tr>
<td>• -1: Invalid file handle</td>
<td>-5: Disk I/O error</td>
<td></td>
</tr>
<tr>
<td>• -6: Invalid parameter (buffer length must be greater than zero)</td>
<td>-9: End-of-file reached</td>
<td></td>
</tr>
<tr>
<td>This function reads from the current location of the file pointer up to the next carriage return or to the end-of-file (EOF), whichever comes first. A complete line will not be read if the buffer length is exceeded before a carriage return (or EOF) is encountered. The bytes are read from the file identified by hFile and are stored in Buffer. The &lt;CR&gt; or &lt;CR&gt;&lt;LF&gt; pair will not be stored in Buffer. If a complete line is read, the file pointer is advanced to the next character in the file after the &lt;CR&gt; or &lt;CR&gt;&lt;LF&gt; pair or to the EOF if the last line was read.</td>
<td>CHAR Buffer[80]nBytes = FILE_READ_LINE (hFile, Buffer, 80)</td>
<td></td>
</tr>
<tr>
<td><strong>FILE_REMOVEDIR</strong></td>
<td>This function removes the specified directory path (including subdirectories), only if it is empty. If any files are present in the directory path, the function will not work. If any file(s) exist in the directory that you are trying to remove, the operation will return a -5 (disk IO error).</td>
<td>SLONG FILE_REMOVEDIR (CHAR DirPath[ ])</td>
</tr>
<tr>
<td>Parameters:</td>
<td>DirPath: String containing the directory path to remove.</td>
<td></td>
</tr>
<tr>
<td>Result:</td>
<td>0: Operation was successful</td>
<td></td>
</tr>
<tr>
<td>• -4: Invalid directory path</td>
<td>-5: Disk I/O error</td>
<td></td>
</tr>
<tr>
<td>Example:</td>
<td>The following code will delete the \CDLIST\TEMP directory, assuming that there are no files or subdirectories present:</td>
<td>FILE_REMOVEDIR('\CDLIST\TEMP')</td>
</tr>
<tr>
<td><strong>FILE_RENAME</strong></td>
<td>This function renames the specified file.</td>
<td>SLONG FILE_RENAME (CHAR FilePath[ ], CHAR NewFileName[ ])</td>
</tr>
<tr>
<td>Parameters:</td>
<td>FilePath: Path name of the file to rename.</td>
<td></td>
</tr>
<tr>
<td>• NewFileName: New file name. This name must not contain a directory path.</td>
<td>Result:</td>
<td></td>
</tr>
<tr>
<td>Result:</td>
<td>0: Operation was successful</td>
<td></td>
</tr>
<tr>
<td>• -2: Invalid file name</td>
<td>-5: Disk I/O error</td>
<td></td>
</tr>
<tr>
<td>• -8: File name exists</td>
<td>// renames \CDLIST\OLDFILE.TXT to \CDLIST\NEWFILE.TXT</td>
<td></td>
</tr>
<tr>
<td>Result = FILE_RENAME ('\CDLIST\OLDFILE.TXT', 'NEWFILE.TXT')</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>FILE_SEEK</td>
<td>This function sets the file pointer to the specified position.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>SLONG FILE_SEEK (LONG hFile, LONG Pos)</strong></td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td>• hFile: Handle to the file returned by FILE_OPEN.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Pos: The byte position to set the file pointer (0 = beginning of file, -1 = end of file).</td>
<td></td>
</tr>
<tr>
<td>Result</td>
<td>&gt;=0: Operation was successful and the result is the current file pointer value</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-1: Invalid file handle</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-5: Disk I/O error</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-6: Invalid parameter; pos points beyond the end-of-file (position is set to the end-of-file)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>After FILE_SEEK is successfully called, subsequent read or write operations begin at the byte number specified by Pos.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>// Sets the file pointer to byte number 1000. Subsequent</td>
<td></td>
</tr>
<tr>
<td></td>
<td>// read or write operations will begin at byte number 1000.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Result = FILE_SEEK (hFile, 1000)</td>
<td></td>
</tr>
<tr>
<td>FILE_SETDIR</td>
<td>This function sets the current working directory to the specified path.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>SLONG FILE_SETDIR (CHAR DirPath[])</strong></td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td>• DirPath: String containing the directory path.</td>
<td></td>
</tr>
<tr>
<td>Result</td>
<td>0: Operation successful</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-4: Invalid directory path</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-5: Disk I/O error</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Result = FILE_SETDIR (&quot;\C\D\L\I\S\T\E\M&quot;)</td>
<td></td>
</tr>
<tr>
<td>FILE_WRITE</td>
<td>This function writes a block of data to the specified file.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>SLONG FILE_WRITE (LONG hFile, CHAR Buffer[], LONG BufLen)</strong></td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td>• hFile: Handle to the file returned by FILE_OPEN.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Buffer: Buffer containing the data to write.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• BufLen: Number of bytes to write.</td>
<td></td>
</tr>
<tr>
<td>Result</td>
<td>&gt;0: The number of bytes actually written</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-1: Invalid file handle</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-5: Disk I/O error</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-6: Invalid parameter (buffer length must be greater than zero)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-11: Disk full</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The data will overwrite or append to the current contents of the file depending on the current position of the file pointer.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CHAR Buffer[1024]Result = FILE_WRITE (hFile, Buffer, 1024)</td>
<td></td>
</tr>
<tr>
<td>FILE_WRITE_LINE</td>
<td>This function writes a line of data to the specified file.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>SLONG FILE_WRITE_LINE (LONG hFile, CHAR Line[], LONG LineLen)</strong></td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td>• hFile: Handle to the file returned by FILE_OPEN.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Line: Buffer containing the line of data to write.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• LineLen: Number of bytes to write.</td>
<td></td>
</tr>
<tr>
<td>Result</td>
<td>&gt;0: The number of bytes actually written</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-1: Invalid file handle</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-5: Disk I/O error</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-6: Invalid parameter (LineLen must be greater than zero)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-11: Disk full</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>NOTE:</strong> A <code>&lt;CR&gt;</code>&lt;LF&gt; character pair is automatically appended to the end of the line.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CHAR Line[80]Result = FILE_WRITE_LINE (hFile, Line, 80)</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>NOTE:</strong> The LONG command cannot pass negative numbers, so if you have errors these will never be recognized. SLONG must be assigned or errors will be typecast to positive numbers.</td>
<td></td>
</tr>
</tbody>
</table>
Get Keywords

NetLinx supports the following GET keywords:

<table>
<thead>
<tr>
<th>GET Keywords</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET_AVAILABLE_FLASH_DISK_SPACE</td>
<td>This function returns the number of bytes of flash disk space available.</td>
</tr>
<tr>
<td>Syntax:</td>
<td>long GET_AVAILABLE_FLASH_DISK_SPACE()</td>
</tr>
<tr>
<td>GET_DNS_LIST</td>
<td>See page 96.</td>
</tr>
<tr>
<td>GET_IP_ADDRESS</td>
<td>See page 97.</td>
</tr>
<tr>
<td>GET_LAST</td>
<td>This function returns the index of the array element that most recently caused an event handler to be triggered.</td>
</tr>
<tr>
<td>DEFINE_VARIABLE</td>
<td>DEVCHAN dcMyDCSet[] = { {TP, 5}, {TP, 4}, {TP, 3}, {TP, 2}, {TP, 1}}</td>
</tr>
<tr>
<td>INTEGER Index</td>
<td>BUTTON_EVENT[dcMyDCSet]</td>
</tr>
<tr>
<td>{</td>
<td>PUSH:</td>
</tr>
<tr>
<td>}</td>
<td>Index = GET_LAST(dcMyDCSet)</td>
</tr>
<tr>
<td>Switch (Index)</td>
<td>Case 1: {} (Button 5 was pressed)</td>
</tr>
<tr>
<td></td>
<td>Case 2: {} (Button 4 was pressed)</td>
</tr>
<tr>
<td></td>
<td>Case 3: {} (Button 3 was pressed)</td>
</tr>
<tr>
<td></td>
<td>Case 4: {} (Button 2 was pressed)</td>
</tr>
<tr>
<td></td>
<td>Case 5: {} (Button 1 was pressed)</td>
</tr>
<tr>
<td>}</td>
<td>Result:</td>
</tr>
<tr>
<td></td>
<td>• 0: No Event was triggered using this array.</td>
</tr>
<tr>
<td></td>
<td>• &gt;0: The index that causes an event to be triggered.</td>
</tr>
<tr>
<td></td>
<td>Since the PUSH and RELEASE keywords can be written using DEVCHAN arrays, this function can also be used to determine which element causes a push or release to be triggered. The function can be called anywhere in code but is usually called from within an event handler. A classic application of this function is to determine the keypad number pressed when the channel codes for the keypad are out of order, which they typically are for a wireless transmitter.</td>
</tr>
<tr>
<td>GET_MAX_FLASH_DISK_SPACE</td>
<td>This function returns the maximum number of bytes of flash disk space available.</td>
</tr>
<tr>
<td>Syntax:</td>
<td>long GET_MAX_FLASH_DISK_SPACE()</td>
</tr>
<tr>
<td>GET_PULSE_TIME</td>
<td>This keyword returns the current duration of PULSE and MINTO commands as set by SET_PULSE_TIME.</td>
</tr>
<tr>
<td>Time is measured in tenths of a second; the default is 5 (0.5 seconds).</td>
<td>PulseTime = GET_PULSE_TIME</td>
</tr>
<tr>
<td>GET_SERIAL_NUMBER</td>
<td>This function returns the 16-character serial number of the specified device. The serial number of every device is established when manufactured.</td>
</tr>
<tr>
<td>NOTE: GET_SERIAL_NUMBER only returns the serial number of the local master, not other masters or devices.</td>
<td>NOTE: The LONG command cannot pass negative numbers, so if you have errors these will never be recognized. SLONG must be assigned or errors will be typecast to positive numbers.</td>
</tr>
<tr>
<td>Parameters:</td>
<td>SLONG GET_SERIAL_NUMBER(DEV Device, CHAR SerialNumber[ ] )</td>
</tr>
<tr>
<td></td>
<td>• Device: Device from which the serial number will be retrieved.</td>
</tr>
<tr>
<td></td>
<td>• SerialNumber: String that will receive the device's serial number.</td>
</tr>
<tr>
<td>Result:</td>
<td>• 0: Operation was successful</td>
</tr>
<tr>
<td></td>
<td>• -1: Specified device is invalid or is not online</td>
</tr>
<tr>
<td></td>
<td>Result = GET_SERIAL_NUMBER(128:11:0, serialNum)</td>
</tr>
<tr>
<td>GET_SYSTEM_NUMBER</td>
<td>This function returns the system number of the NetLinx Master.</td>
</tr>
<tr>
<td>INTEGER GET_SYSTEM_NUMBER( )</td>
<td>The result is an integer representing the system number of the NetLinx Master.</td>
</tr>
<tr>
<td>SystemNum = GET_SYSTEM_NUMBER()</td>
<td>// get local system num</td>
</tr>
<tr>
<td>NOTE: When it is a NetLinx function the () are NOT OPTIONAL even if there are no parameters.</td>
<td></td>
</tr>
<tr>
<td>GET_TIMER</td>
<td>This keyword returns an unsigned long integer representing the value currently held by the system timer.</td>
</tr>
<tr>
<td></td>
<td>• Time is measured in tenths of a second.</td>
</tr>
<tr>
<td></td>
<td>• The system timer is set to zero on power-up.</td>
</tr>
<tr>
<td></td>
<td>SystemTime = GET_TIMER</td>
</tr>
</tbody>
</table>
GET Keywords (Cont.)

GET_UNIQUE_ID

This function returns a 48-bit hardware constant guaranteed to be unique in the domain of NetLinx Masters. Possible uses for GET_UNIQUE_ID include identification of a particular system for the purpose of providing system specific capability or limiting the functionality of a NetLinx program to operate on a specific master.

CHAR[6] GET_UNIQUE_ID()

The result is a 48-bit constant returned as a 6-element character array.

SYSID = GET_UNIQUE_ID() // get the master's h/w ID
IF(SysID = "$00,$01,$09,$73,$25,$01")
{ // allow system to operate normally

GET_URL_LIST

This function returns a list of URLs that the specified device is programmed to actively attempt to connect to. The function requires an array of URL_STRUCT Structures that will get filled in with the device's URL list.

SLONG Get_URL(List(DEV Device,URLSTRUCT UrlList[],INTEGER Type))

Parameters:
- Device: Device number of the device from which the URLs will be retrieved. Typically, they are stored on the local master (0:1:0), but if you are currently connected to another master you can use <0:1:system number of remote master>.
- UrlList: Array of URL_STRUCTs that will receive the device's URLs
- Type: Indicates the type(s) of URLs desired—NetLinx language programmed, IDE programmed, or both
1: All URLs
2: NetLinx programmed URLs
3: IDE programmed URLs

The function returns the number of URLs updated in the supplied array of URL_STRUCTs.
-1: Specified device is invalid or is not online
-2: Request timed out
-3: Busy

URLS may be programmed by either the Integrated Development environment or via the ADD_URL_ENTRY function. The Type parameter filters the list of URLs so that only the desired URLs are returned in the URL_STRUCT(s). The function requires an array of URL_STRUCTs.

The URL_STRUCT is predefined as follows:

```
STRUCTURE URL_STRUCT
{
  CHAR  Flags;           // Connection Type (normally 1)
  INTEGER Port;            // TCP port (normally 1319)
  CHAR  URL[128];        // string: URL or IP address
  CHAR  User[20];        // optional account info for ICSPS
  CHAR  Password[20];    // optional account info for ICSPS
}
```

The following definitions exist for the Flags member of the URL_STRUCT structure.

```
CONSTANT CHAR URL_Flg_TCP = 1               // TCP connection
CONSTANT CHAR URL_Flg_TEMP = $10
CONSTANT CHAR URL_Flg_Stat_PrgNetLinx = $20 // URL set by NetLinx
                                               // ADD_URL_ENTRY
CONSTANT CHAR URL_Flg_Stat_Mask = $C0       // status mask
CONSTANT CHAR URL_Flg_Stat_Lookup = $00     // Looking up IP
CONSTANT CHAR URL_Flg_Stat_Connecting = $40 // connecting
CONSTANT CHAR URL_Flg_Stat_Waiting = $80    // waiting
CONSTANT CHAR URL_Flg_Stat_Connected = $C0  // connected
```

See GET_URL_LIST Flags Member Bit Fields below.

Example:
```
URL_STRUCT UrlList [10]
Result = GET_URL_LIST(0:0:0,UrlList,0)       (* Get ALL URLs *)
-OR-
Result = GET_URL_LIST(0:0:0,UrlList,1)       (* Get NetLinx-programmed URLs *)
-OR-
Result = GET_URL_LIST(0:0:0,UrlList,2)       (* Get IDE-programmed URLs *)
```

NOTE: There is a known issue with this function: If you have only 1 URL entry, it will return nothing. If you have 2 entries, it will return the second entry.
GET_URL_LIST Flags Member Bit Fields
The Flags member is a bit field that is used for several different purposes. Each bit is defined in the table below:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Mathematical Value</th>
<th>Normal value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0</td>
<td>1 (0x01)</td>
<td>1</td>
<td>0 = Establishes a UDP connection. 1 = Establishes a TCP connection.</td>
</tr>
<tr>
<td>Bit 1</td>
<td>2 (0x02)</td>
<td>0</td>
<td>Unused</td>
</tr>
<tr>
<td>Bit 2</td>
<td>4 (0x04)</td>
<td>0</td>
<td>Unused</td>
</tr>
<tr>
<td>Bit 3</td>
<td>8 (0x08)</td>
<td>0</td>
<td>Unused</td>
</tr>
<tr>
<td>Bit 4</td>
<td>16 (0x10)</td>
<td>0</td>
<td>Establishes a Temp Connection. A Temp Connection is one that is set, but not stored in flash, and therefore is not restored when the master reboots. If the NetLinx code is adding URL entries, it is recommended to make them temporary so that the flash is not constantly being written, especially since the code handles all the connections anyway.</td>
</tr>
<tr>
<td>Bit 5</td>
<td>32 (0x20)</td>
<td>0</td>
<td>Source of URL. 0 = Programmed by the IDE. 1 = Programmed by NetLinx ADD_URL_ENTRY.</td>
</tr>
<tr>
<td>Bit 6</td>
<td>64 (0x40)</td>
<td>0</td>
<td>Encoded status indication (Read only). These 2 bits together form one of 4 possible codes indicating the status of the connection. • 0x00 - Looking up IP address or URL. • 0x40 - Connecting to URL. • 0x80 - Waiting for connection to establish. • 0xC0 - Connected.</td>
</tr>
<tr>
<td>Bit 7</td>
<td>128 (0x80)</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Overview - IP Communication

Clients and servers communicate via Internet Protocol (IP) using either a connection-oriented or connection-less protocol. Connection-oriented input/output (I/O) channels require a connection or virtual circuit to be established between the client and server before data can be transmitted or received. Transmission Control Protocol (TCP) is the transport protocol typically used for connection-oriented I/O. With TCP, delivery of the data is guaranteed.

With connection-less I/O, a connection is not established between the client and server before data is exchanged. Instead, the identity of the client and server is established each time data is sent or received. This type of communication is usually recommended for applications that transfer only small amounts of data. User Datagram Protocol (UDP) is the transport protocol used for connection-less I/O. With UDP, delivery of the data is not guaranteed. Both the client and server must be able to identify incoming and outgoing data for a particular conversation. To achieve this, each application assigns a unique number to the conversation. This number is the local port number. A local port is not a physical port but rather a virtual port that identifies the source or destination for data exchanged during the conversation. Local ports are specific to either the client or the server; they need not match across applications.

The application assigns the number for the local port - as opposed to letting the system assign it (for instance, as the return value for IP_CLIENT_OPEN or IP_SERVER_OPEN) - to satisfy the static nature of DEFINE_EVENT handlers. All event handlers must specify a device, port, and system to identify the events' source. This device information must be constant; that is, it cannot change at run-time. A constant IP device specification can be defined using a local port number.

For example:

Device Number = 0     The master
Port = LocalPort      The local port number
System = 0            This system (where the application is running)

A range of numbers is reserved for local port numbers to make sure that this IP device-naming convention does not interfere with future naming schemes. The program can only assign local port numbers at or above the value of the keyword, FIRST_LOCAL_PORT. All port numbers below FIRST_LOCAL_PORT are reserved for future use.

For example:

DEFINE_CONSTANT
PORT_REMOTE_MASTER1 = FIRST_LOCAL_PORT
PORT_REMOTE_MASTER2 = FIRST_LOCAL_PORT + 1
PORT_REMOTE_MASTER3 = FIRST_LOCAL_PORT + 2

Client Programming

Initiating a conversation

To initiate a conversation with a server, the client must use the IP_CLIENT_OPEN command and supply either the IP address or domain name of the server and a port number for the requested service. The client must also specify a local port number to use for sending and receiving data. This number represents a virtual port on the client machine; it is not the actual port number used to create the client-end socket. A local port number may not be used in another call to IP_CLIENT_OPEN until IP_CLIENT_CLOSE is called for that port number.

The syntax is shown below:

IP_Client_Open(LocalPort, ServerAddress, ServerPort, Protocol)

Parameters:
- LocalPort: A user-defined, non-zero integer value representing the virtual port on the client machine that will be used for this conversation. This port number must be passed to IP_CLIENT_CLOSE to close the conversation.
- ServerAddress: A string containing either the IP address (in dotted-quad-notation) or the domain name of the server to connect to.
- ServerPort: The port number on the server that identifies the program or service the client is requesting.
- Protocol: The transport protocol to use (1 = TCP, 2 = UDP). If this parameter is not specified, TCP (1) is assumed. The constants IP_TCP and IP_UDP can be used to specify this parameter.

Terminating a conversation

To terminate a conversation, you must use the IP_CLIENT_CLOSE command and pass the number of the local port used for the conversation.

The syntax:

IP_Client_Close(LocalPort)

Parameters:
- LocalPort: A user-defined, non-zero integer value representing the virtual port on the client machine that will be used for this conversation.
**Sending data**

To send data to the server, use the `SEND_STRING` command.

```
SEND_STRING 0:LocalPort:0, 'string'
```

The device specification (0:LocalPort:0) is interpreted as follows:

- Device Number: 0: The master
- Port: LocalPort: The local port number
- System: 0: This system (the client)

**Receiving data**

To receive data from the server, use a DATA event handler or a buffer created with `CREATE_BUFFER` or `CREATE_MULTI_BUFFER`. If an event handler is used, the data is located in the Text field of the DATA object. The syntax is shown below:

```
Data_Event [Device]
{
  STRING:
    // process incoming string (Data.Text)
}
```

**Parameters:**

- `Device` is (or contains as part of an array) the device representing the conversation (0:LocalPort:0)

When using IP sockets in NetLinx, it is not uncommon to create a buffer using a `CREATE_BUFFER` keyword and processing the buffer in the `DATA_EVENT...OFFLINE` event. NetLinx has an important behavior than can affect the performance of IP socket code. This is not a bug but a feature. If you are aware of it, you can write your code to take maximum advantage of the speed NetLinx offers. When processing string data from a device, whether it is a regular device or an IP socket, the master will attempt to copy this data to a buffer, if one has been created using the `CREATE_BUFFER` keyword, and then try to run a `DATA_EVENT...STRING` handler for this device.

If a `DATA_EVENT...STRING` handler does not exist, NetLinx will run mainline to allow for any buffer processing that might occur in mainline. At the end of a conversation with an IP device, there will usually be an incoming string event followed by an offline event. The NetLinx master will copy the string to a buffer, if it exists, check for a string event handler, run mainline if one does not exist, then process the offline event.

If you are processing that data in an offline event for an IP device, you will see a time delay between the IP device or server closing the connection and the processing of the offline event. This delay will vary with the size and complexity of mainline. To eliminate this delay, simply include and empty string event handler in the `DATA_EVENT` section. This will keep NetLinx from running mainline between the last incoming string and the offline event. See this example:

```
DATA_EVENT [dvIP]
{
  OFFLINE:
    (* PROCESS THE DATA HERE*)
  }

  STRING:
    (* DO NOT REMOVE ME! *)
  }
}
Server Programming

Listening for client requests
A client gains access to a service by sending a request to the server specifying the port assigned to the service. For the request to be acknowledged, the server must be listening on that port. To do this, the server calls IP_SERVER_OPEN. This opens the port and allows the server to listen for requests from client applications. IP_SERVER_OPEN requires the caller to supply a local port number. This local port number is a virtual port, as opposed to an actual physical port on the server.

- When TCP is the transport protocol, the local port represents a single client connection on the server’s physical port.
- When UDP is the transport protocol, it represents a single point where all client requests on the associated port are routed.

The local port number is the key to identifying data sent to or received from a client application. A local port number may not be used in another call to IP_SERVER_OPEN, until IP_SERVER_CLOSE is called for that port number.

The syntax:
IP_SERVER_OPEN(LocalPort, ServerPort, Protocol)

Parameters:
- LocalPort: The local port number to open. This port number must be passed to IP_CLIENT_CLOSE to close the conversation.
- ServerPort: The port number on the server identifies the program or service the client is requesting.
- Protocol: The transport protocol to use (1 = TCP, 2 = UDP). If this parameter is not specified, TCP (1) is assumed. The constants IP_TCP and IP_UDP can be used to specify this parameter.

Multiple client connections
With connection-oriented I/O (TCP), more than one client could request a connection with the server at the same time. Support for multiple client connections applies only to connection-oriented I/O, that is, TCP protocol. Opening multiple ports using UDP as the protocol serves no purpose. In that case, any additional open commands will fail.

To support concurrent requests, the server must call IP_SERVER_OPEN once for each simultaneous connection allowed. For example:
IP_SERVER_OPEN (First_Local_Port, 10510, IP_TCP)
IP_SERVER_OPEN (First_Local_Port, 10510, IP_TCP)
IP_SERVER_OPEN (First_Local_Port, 10510, IP_TCP)

This allows three simultaneous connections on port 10510. Note that each call to IP_SERVER_OPEN uses a different local port number.

Closing a local port
To close a local port, the server application must call IP_SERVER_CLOSE. Once that is called, no I/O can be handled using the specified local port. The syntax:
IP_SERVER_CLOSE(LocalPort)

Parameters:
- LocalPort: The local port number to close.

Connection-Oriented notifications
The server receives the following notifications when a client connects or disconnects.
The protocol in this case must be TCP.
DATA[0:LocalPort:0]
{
    ONLINE:
    {
        // client has connected
    }
    OFFLINE:
    {
        // client has disconnected
    }
}

Parameters:
- Device is (or contains as part of an array) the device representing the conversation (0:LocalPort:0).

Receiving data
To receive data from a client, use a DATA event handler or a buffer created with CREATE_BUFFER or CREATE_MULTI_BUFFER. If an event handler is used, the data is located in the Text field of the DATA object.
The syntax:
Data_Event[Device]
{
    STRING:
    {
        // process incoming string (Data.Text)
    }
}
Parameters:
- Device is (or contains as part of an array) the device representing the conversation (0:LocalPort:0).

Sending data
To send data to the client, use the SEND_STRING command.
```
SEND_STRING 0:LocalPort:0, '<string>'
```
The device specification (0:LocalPort:0) is interpreted as follows:
- Device Number: 0: The master
- Port: LocalPort: The local port number
- System: 0: This system (the client)

Receiving Data with UDP
Since UDP is connection-less, no formal agreement has been made between the client and server to exchange data. The client simply sends a UDP message and hopes the server is listening. In many protocols that use UDP for communication, there is an implied agreement for the client to receive data from the server. When a UDP client socket is created, the socket is assigned a UDP/IP port number, not to be confused with local port. This UDP/IP port will be used to send UDP messages. The server, if listening, will receive this message along with the IP address and UDP/IP of the client who sent the message.

Some UDP protocols have an implied agreement that the server will be able to respond to the client by sending a response back to the IP address and UDP/IP from where the message originated. Although the UDP protocol does not specify that the client must expect to receive messages in this way, many UDP/IP require the client to listen for responses after sending a message.

NetLinx has two UDP client implementations. These are UDP (2) and UDP With Receive (3).
- UDP only sends messages and cannot receive messages.
- UDP with Receive will send and receive messages on a single UDP/IP port.

It may seem like UDP (2) is not needed; however, it still serves and important purpose. Imagine you wanted to send a UDP message and expect a response. The proper way to open this type of socket, assuming you want to send a UDP message to 192.168.0.1 on UDP port 6000, is:
```
IP_CLIENT_OPEN(dvUDPClient, '192.168.0.1', 6000, IP_UDP_2WAY)
```
Now, if you were also writing the code for 192.168.0.1, you would need to have opened a UDP server using the following:
```
IP_SERVER_OPEN(dvUDPServer, 6000, IP_UDP)
```
When the message is received at 192.168.0.1, the message will be delivered to the DATA_EVENT for dvUDPServer and the IP address UDP/IP port of the sender of the message will be available in the DATA.SOURCEIP and DATA.SOURCEPORT variables. A UDP (2) socket would be used in this case to send a response to the client. Since we will no longer need to listen after sending the response, since there would be no response to the response, we would open the socket using the following:
```
IP_CLIENT_OPEN(dvUDPClient, DATA.SOURCEIP, DATA.SOURCEPORT, IP_UDP)
```
Note that UDP with Receive (3) is only available when calling IP_CLIENT_OPEN.

Multicast
NetLinx can send and receive multi-cast UDP messages. To send a multi-cast UDP message, all you need to do is specify a multi-cast address and port in the IP_CLIENT_OPEN function such as the following:
```
IP_CLIENT_OPEN (dvIPClient.Port, '239.255.255.250', 1900, IP_UDP)
```
To receive multi-cast UDP messages, you must call the IP_MC_SERVER_OPEN function:
```
IP_MC_SERVER_OPEN (dvIPServer, '239.255.255.250', 1900)
```
The NetLinx master will join the multi-cast session and allow you to receive and transmit UDP multi-cast messages.

Example IP Code
```
PROGRAM_NAME='IPExample'
{"***********************************************************}
(*          DEVICE NUMBER DEFINITIONS GO BELOW             *)
{*****************************************************}
DEFINE_DEVICE
 dvIPServer = 0:2:0
 dvIPClient = 0:3:0
{**********************************************************}
(*          CONSTANT DEFINITIONS GO BELOW                 *)
{*****************************************************}
DEFINE_CONSTANT
 nIPPort     = 8000
{**********************************************************}
(*          VARIABLE DEFINITIONS GO BELOW                 *)
{*****************************************************}
DEFINE_VARIABLE
 IP_ADDRESS_STRUCT MyIPAddress   (* .Flags                  *)
 (* .HostName               *)
 (* .IPAddress              *)
 (* .SubnetMask             *)
 (* .Gateway                *)
IP Keywords

[*******************************************************************]
(*                STARTUP CODE GOES BELOW                  *)
[*******************************************************************]
DEFINE_START
(* Get My IP Address *)
GET_IP_ADDRESS(0:0:0,MyIPAddress)
(* Open The Server *)
IP_SERVER_OPEN(dvIPServer.Port,nIPPort,IP_TCP)
(* Open The Client *)
IP_CLIENT_OPEN(dvIPClient.Port,MyIPAddress.IPAddress,nIPPort,IP_TCP)

[*******************************************************************]
(*                THE EVENTS GO BELOW                      *)
[*******************************************************************]
DEFINE_EVENT
(* Server Data Handler *)
DATA_EVENT[dvIPServer]
{
  ONERROR:
  {
    SEND_STRING 0,"'error: server=',ITOA(Data.Number)"
  }
  ONLINE:
  {
    SEND_STRING 0,"'online: server'"
  }
  OFFLINE:
  {
    SEND_STRING 0,"'offline: server'"
  }
  STRING:
  {
    SEND_STRING 0,"'string: client=',Data.Text"
    IF (FIND_STRING(Data.Text,'ping',1))
    SEND_STRING 0:2:0,"'pong',13"
  }
}

(* Client Data Handler *)
DATA_EVENT[dvIPClient]
{
  ONERROR:
  {
    SEND_STRING 0,"'error: client=',ITOA(Data.Number)"
  }
  ONLINE:
  {
    SEND_STRING 0,"'online: client'"
  }
  OFFLINE:
  {
    SEND_STRING 0,"'offline: client'"
  }
  STRING:
  {
    SEND_STRING 0,"'string: client=',Data.Text"
  }
}

[*******************************************************************]
(*            THE ACTUAL PROGRAM GOES BELOW                *)
[*******************************************************************]
DEFINE_PROGRAM
(* Send Ping To Server *)
WAIT 50
SEND_STRING dvIPClient,"'ping',13"

[*******************************************************************]
(*                     END OF PROGRAM                      *)
[*******************************************************************]
## IP Keywords

NetLinx supports the following IP keywords:

<table>
<thead>
<tr>
<th>IP Keywords</th>
<th>Description</th>
<th>Parameters</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ADD_URL_ENTRY</strong></td>
<td>This function adds a URL entry to the specified device. This function requires a pre-initialized URL_STRUCT that will be sent to the specified device.</td>
<td>Device: Device number of the device that stores the URL. Typically, it is stored on the local master (0:1:0); if you are currently connected to another master, you can use &lt;0:1:system number of remote master&gt;. URL: URL_STRUCT that will be programmed into the device.</td>
<td>• 0: Success&lt;br&gt;• -1: Specified device is invalid or is not online&lt;br&gt;• -2: Time out occurred&lt;br&gt;• -3: Function is already actively adding a URL entry (i.e. busy)&lt;br&gt;• -4: Add failed&lt;br&gt;Note that NetLinx will automatically set bit 5 of the Flags member of the URL_STRUCT structure. See the <strong>ADD_URL_ENTRY Flags Member Bit Fields</strong> section on page 100 for details.</td>
</tr>
<tr>
<td><strong>DELETE_URL_ENTRY</strong></td>
<td>This function deletes a URL entry to the specified device. This function requires a pre-initialized URL_STRUCT that will be sent to the specified device.</td>
<td>Device: Device to which the URL will be sent. URL: URL_STRUCT that will be programmed into the device.</td>
<td>• 0: Success&lt;br&gt;• -1: Specified device is invalid or is not online&lt;br&gt;• -2: Time out occurred&lt;br&gt;• -3: Function is already actively deleting a URL entry (i.e. busy)&lt;br&gt;• -4: Delete failed&lt;br&gt;See the <strong>ADD_URL_ENTRY Flags Member Bit Fields</strong> section on page 100 for details.</td>
</tr>
<tr>
<td><strong>GET_DNS_LIST</strong></td>
<td>This function returns the domain name and list of DNS server IP addresses that the specified device is programmed to utilize. The order of the returned list is the preferred server order.</td>
<td>Device: Device from which the DNS servers will be retrieved. DnsList: A DNS_STRUCT that will receive the device's DNS server list.</td>
<td>• 0: Operation was not successful&lt;br&gt;• -1: Specified device is invalid or is not online&lt;br&gt;• -2: Request timed out&lt;br&gt;• -3: Busy&lt;br&gt;The function requires a DNS_STRUCT. The DNS_STRUCT is predefined as follows:</td>
</tr>
</tbody>
</table>
IP Keywords (Cont.)

**GET_IP_ADDRESS**

This function returns the TCP/IP configuration of the specified device. The configuration information includes DHCP/Static configuration, IP address, subnet mask, gateway, and host name.

```c
SLONG GET_IP_ADDRESS(DEV Device, IP_ADDRESS_STRUCT IPAddress)
```

**Parameters:**

- **Device:** Device from which the TCP/IP configuration will be retrieved.
- **IPAddress:** An IP_ADDRESS_STRUCT that will receive the device's TCP/IP configuration.

**Result:**

- 0: Operation was successful
- -1: Specified device is invalid or is not online
- -2: Request timed out
- -3: Busy

The function requires an IP_ADDRESS_STRUCT. The IP_ADDRESS_STRUCT is predefined as follows:

```c
STRUCTURE IP_ADDRESS_STRUCT
{
    CHAR Flags          // Configuration flags
    CHAR HostName[128]  // Host name
    CHAR IPAddress[15]  // IP address unit
    CHAR SubnetMask[15] // subnet mask
    CHAR Gateway[15]    // IP address of gateway
}
```

The following definitions exist for the Flags member of the IP_ADDRESS_STRUCT structure.

```c
CONSTANT CHAR IP_Addr_Flg_DHCP = 1 // Use DHCP
```

The Flags member is a bit field that may be used for several different purposes. See the GET_IP_ADDRESS Flags Member Bit Fields section on page 101 for details. Differing configuration parameters may be obtained, depending upon the configuration of the network DHCP server. It is possible that the DHCP server will provide the host name, IP address, subnet mask, gateway, and even DNS information. In a minimal configuration, the DHCP server will only supply the IP address and subnet mask.

```c
IP_ADDRESS_STRUCT IPAddressResult = GET_IP_ADDRESS(0:0:0, IPAddress)
```

**IP_BOUND_CLIENT_OPEN**

Opens a port for IP communication with a server using a specific local IP port number. Similar to IP_CLIENT_OPEN, but where IP_CLIENT_OPEN uses the first available local IP Port number, IP_BOUND_CLIENT_OPEN allows the user to specify the local IP port number.

**Parameters:**

- **LocalPort:** A user-defined (non-zero) integer value representing the local port number on the client machine to use for this conversation. This local port number must be passed to IP_CLIENT_CLOSE to close the conversation.
- **LocalIPPort:** A user-defined (non-zero) integer value representing the local IP port number the IP client socket must be bound to.
- **ServerAddress:** A string containing either the IP address (in dotted-quad-notation) or the domain name of the server to connect to.
- **ServerPort:** The port number on the server that identifies the program or service that the client is requesting.
- **Protocol:** The transport protocol to use:
  - 1 = TCP
  - 2 = UDP
  - 3 = UDP with Receive

If this parameter is not specified, TCP (1) is assumed. The constants IP_TCP, IP_UDP and IP_UDP_2WAY can be used to specify this parameter.

**Result:**

- This function always returns 0.
- Errors are returned via the DATA_EVENT ONERROR method.

The following errors may be returned:

- 2: General failure (out of memory)
- 4: Unknown host
- 6: Connection refused
- 7: Connection timed out
- 8: Unknown connection error
- 14: Local port already used
- 16: Too many open sockets
- 17: Local Port Not Open

**Example:**

```
IP_BOUND_CLIENT_OPEN(PORT1, 3000, SvAddr, SvPort, IP_TCP)
```

**IP_CLIENT_CLOSE**

This function closes a port opened with IP_CLIENT_OPEN.

**Parameters:**

- **LocalPort:** A non-zero integer value representing the local port on the client machine to close.

**Result:**

- This function always returns 0.
- Errors are returned via the DATA_EVENT ONERROR method.

The following error may be returned:

- 9: Already closed
<table>
<thead>
<tr>
<th>IP Keywords (Cont.)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IP_CLIENT_OPEN</strong></td>
<td>This function opens a port for IP communication with a server.</td>
<td></td>
</tr>
<tr>
<td>SLONG IP_CLIENT_OPEN</td>
<td>(INTEGER LocalPort, CHAR ServerAddress[], LONG ServerPort, INTEGER Protocol)</td>
<td></td>
</tr>
<tr>
<td><strong>Parameters:</strong></td>
<td>• LocalPort: A user-defined (non-zero) integer value representing the local port on the client machine to use for this conversation. This local port number must be passed to IP_CLIENT_CLOSE to close the conversation.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• ServerAddress: A string containing either the IP address (in dotted quad-notation) or the domain name of the server to connect to.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• ServerPort: The port number on the server that identifies the program or service the client is requesting.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Protocol: The transport protocol to use:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 = TCP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 = UDP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 = UDP with Receive</td>
<td></td>
</tr>
<tr>
<td></td>
<td>If this parameter is not specified, TCP (1) is assumed. The constants IP_TCP, IP_UDP and IP_UDP_2WAY can be used to specify this parameter.</td>
<td></td>
</tr>
<tr>
<td><strong>Result:</strong></td>
<td>• This function always returns 0.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Errors are returned via the DATA_EVENT ONERROR method.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The following errors may be returned:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2: General failure (out of memory) - Underlying OS socket call failed, reasons undefined.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4: Unknown Host (IP_CLIENT_OPEN) - The specified host name (ex. 'www.amx.com') or IP address (ex. '192.168.200.79') is not resolvable to a physical host.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6: Connection Refused (IP_CLIENT_OPEN) - The specified host does not have a server socket listening on the requested port and therefore refused the connection.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7: Connection Timed Out (IP_CLIENT_OPEN) - The specified host has not replied to the request to connect within a reasonable time.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8: Unknown connection Error (IP_CLIENT_OPEN) - Some other undefined error has occurred with the connection request.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9: Already Closed (IP_CLIENT_CLOSE/IP_SERVER_CLOSE) - The specified connection has already been closed.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10: Binding Error (IP_SERVER_OPEN) - An error has occurred during the underlying OS &quot;bind&quot; function of a socket to a server port number. Possibly the server port is already being listened on.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11: Listening Error (IP_SERVER_OPEN) - An underlying error has occurred; checking for possible client connects to a server socket.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12: Socket not connected - Tried to send data (string or command) on a TCP socket that is not connected either because the open failed, or the connection has been closed.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>13: Send to Socket Unknown - Tried to send data (string or command) on a UDP socket that has failed to open.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>14: Local Port already used (IP_CLIENT_OPEN/IP_SERVER_OPEN) - The local TCP client or serve port (D:P:S) is already open for use by an earlier IP_CLIENT_OPEN or IP_SERVER_OPEN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15: UDP socket already listening (IP_SERVER_OPEN) - The local UDP port (D:P:S) is already being listened on.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16: Too many open sockets (IP_CLIENT_OPEN/IP_SERVER_OPEN) - NetLinx enforces a limit on the number of allowed open sockets. The current limit is 200. All requests to open a socket beyond this limit will fail.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>17: Local port not open - The specified local port (D:P:S) has never been opened by a IP_CLIENT_OPEN or IP_SERVER_OPEN call.</td>
<td></td>
</tr>
<tr>
<td><strong>Example:</strong></td>
<td>IP_CLIENT_OPEN(PORT1, SvAddr, SvPort, IP_TCP)</td>
<td></td>
</tr>
</tbody>
</table>

| **IP_MC_SERVER_OPEN** | This function opens a server port to listen for UDP multicast messages. | |
| INTEGER IP_MC_SERVER_OPEN(INTEGER LocalPort, CHAR MultiCastIP[], LONG ServerPort) | | |
| **Parameters:** | • LocalPort: The local port number to open. This number must be passed to IP_SERVER_CLOSE to close the port. | |
| | • MultiCastIP: A character string representing the multicast address to receive on in the form of: | |
| | '239.255.255.250'. | |
| | • ServerPort: The UDP multicast port number to listen on. | |
| **Result:** | • This function always returns 0. | |
| | • Errors are returned via the DATA_EVENT ONERROR method. | |
| | The following errors may be returned: | |
| | 2: General failure (out of memory) - Underlying OS socket call failed, reasons undefined | |
| | 10: Binding error - An error has occurred during the underlying OS "bind" function of a socket to a server port number. Possibly the server port is already being listened on. | |
| | 11: Listening error - An underlying error has occurred checking for possible client connects to a server socket. | |
| | 14: Local port already used - The local TCP client or serve port (D:P:S) is already open for use by an earlier IP_CLIENT_OPEN or IP_SERVER_OPEN. | |
| | 15: UDP socket already listening - The local UDP port (D:P:S) is already being listened on. | |
| | 16: Too many open sockets - NetLinx enforces a limit on the number of allowed open sockets. The current limit is 200. All requests to open a socket beyond this limit will fail. | |
| **Example:** | IP_MC_SERVER_OPEN (PORT1,'239.255.255.250',1900) | |
### IP Keywords (Cont.)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Syntax</th>
<th>Parameters</th>
<th>Result</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IP_SERVER_CLOSE</strong></td>
<td>This function closes a port opened with IP_SERVER_OPEN or IP_MC_SERVER_OPEN.</td>
<td><code>IP_SERVER_CLOSE (INTEGER LocalPort)</code></td>
<td>• <strong>LocalPort</strong>: The number of the local port to close.</td>
<td>This function always returns 0.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Errors are returned via the DATA_EVENT ONERROR method.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The following error may be returned:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>9: Already closed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Example:</td>
<td>IP_Server_Close(PORT1)</td>
<td></td>
</tr>
<tr>
<td><strong>IP_SERVER_OPEN</strong></td>
<td>This function opens a server port to listen for client requests.</td>
<td><code>IP_SERVER_OPEN (INTEGER LocalPort, LONG ServerPort, INTEGER Protocol)</code></td>
<td>• <strong>LocalPort</strong>: The local port number to open. This number must be passed to IP_SERVER_CLOSE to close the port.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>ServerPort</strong>: The number of the server port to listen on.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>Protocol</strong>: The transport protocol to use:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 = TCP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 = UDP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If this parameter is not specified, TCP (1) is assumed. The constants IP_TCP and IP_UDP can be used to specify this parameter.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Result (via ONERROR event):</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2: General Failure - Underlying OS socket call failed, reasons undefined</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10: Binding error - An error has occurred during the underlying OS &quot;bind&quot; function of a socket to a server port number. Possibly the server port is already being listened on.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11: Listening error - An underlying error has occurred checking for possible client connects to a server socket.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>14: Local port already used - The local TCP client or serve port (D:P:S) is already open for use by an earlier IP_CLIENT_OPEN or IP_SERVER_OPEN.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15: UDP socket already listening - The local UDP port (D:P:S) is already being listened on.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>16: Too many open sockets - NetLinx enforces a limit on the number of allowed open sockets. The current limit is 200. All requests to open a socket beyond this limit will fail.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Example:</td>
<td>IP_SERVER_OPEN (PORT1, SvPort, IP_TCP)</td>
<td></td>
</tr>
<tr>
<td><strong>IP_SET_OPTION</strong></td>
<td>Allows for specific option settings on IP client or server connections.</td>
<td><code>IP_SET_OPTION (INTEGER LocalPort, INTEGER OptionID, INTEGER OptionValue)</code></td>
<td>• <strong>LocalPort</strong> - a user-defined (non-zero) integer value representing the local port on the client machine to use for this conversation. This local port number was previously specified in an IP_CLIENT_OPEN (page 100) or IP_SERVER_OPEN (page 101) call.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>OptionID</strong> - Identifier value for the option to be set. Current valid option IDs are:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IP_MULTICAST_TTL_OPTION - Set the time-to-live value for all outbound UDP Multicast packet transmissions on the specified port. Predefined constant option values are:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IP_TTL_SUBNET = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IP_TTL_SITE = 32</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IP_TTL_REGION = 64</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IP_TTL_CONTINENT = 128</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IP_TCP_NODELAY_OPTION - Outgoing TCP data is transmitted immediately. (default = OFF):</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IP_NODelay_ON - When the NODelay option is ON, all data is transmitted immediately upon send. This ensures that no data will be left in transmit buffers upon closure of the connection.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IP_NODelay_OFF - By default, the NODelay option is disabled (OFF). Data will be buffered, and transmission is determined by the operating system.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>OptionValue</strong> - Integer containing the option value.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Example:</td>
<td>IP_SET_OPTION(PORT1, IP_MULTICAST_TTL_OPTION, IP_TTL_REGION)</td>
<td></td>
</tr>
</tbody>
</table>
IP Keywords (Cont.)

**SET_IP_ADDRESS**

This function programs the TCP/IP configuration of the specified device. This function requires a pre-initialized IP_ADDRESS_STRUCT structure that will be sent to the specified device.

```c
SLONG SET_IP_ADDRESS(DEV Device,IP_ADDRESS_STRUCT IPAdress )
```

**NOTE:** **SET_IP_ADDRESS** takes effect after the system is rebooted.

**Parameters:**

- **Device:** Device to which the IPAddress list will be sent.
- ** IPAddress:** An IP_ADDRESS_STRUCT containing the desired TCP/IP configuration for the specified device.

**Result:**

- 0: Operation was successful.
- -1: Specified device is invalid or is not online.
- -2: Time out occurred.
- -3: Function is already actively attempting to set an IP Address (i.e. busy).

See GET_IP_ADDRESS, on page 97, for a description of the IP_ADDRESS_STRUCT structure.

```c
IPAddress.Flags = 0                      // use static IP address
IPAddress.HostName = 'NetLinx1'          // host name
IPAddress.IPAddress = '19.00.100.00'
IPAddress.SubnetMask = '255.255.255.0'
IPAddress.Gateway = ' 19.00.100.01'
Result = SET_IP_ADDRESS(0:0:0,IPAddress) // config master
```

**SET_DNS_LIST**

This function programs a domain name and the list of DNS servers that the specified device will use to lookup domain names. It requires a pre-initialized DNS_STRUCT structure that will be sent to the specified device.

```c
SLONG SET_DNS_LIST(DEV Device,DNS_STRUCT DnsList)
```

**Parameters:**

- **Device:** Device to which the DNS list will be sent
- **DnsList:** A DNS_STRUCT that contains the list of DNS server IP addresses that will be programmed in to the device

**Result:**

0: Operation was successful
-1: Specified device is invalid or is not online
-2: Time out occurred
-3: Function is already actively attempting to set a DNS list (i.e. busy)
-4: Set DNS failed
-5: A portion of the DNS structure contains invalid information

```c
DnsList.DomainName = 'amx.com'
DnsList.DNS1 = '19.00.100.00'
DnsList.DNS2 = ''
DnsList.DNS3 = ''
Result = SET_DNS_LIST(0:0:0,DnsList) // set master's list
```

See GET_DNS_LIST, on page 96, for a description of the DNS_STRUCT structure.

**ADD_URL_ENTRY Flags Member Bit Fields**

The Flags member is a bit field that is used for several different purposes, as defined below:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Mathematic Value</th>
<th>Normal Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 (0x01)</td>
<td>1</td>
<td>0 = Establishes a UDP connection. 1 = Establishes a TCP connection.</td>
</tr>
<tr>
<td>1</td>
<td>2 (0x02)</td>
<td>0</td>
<td>0 = Account information not included 1 = Include account information (User Name / Password) required to connect to remote URL</td>
</tr>
<tr>
<td>2</td>
<td>4 (0x04)</td>
<td>0</td>
<td>Unused</td>
</tr>
<tr>
<td>3</td>
<td>8 (0x08)</td>
<td>0</td>
<td>Unused</td>
</tr>
<tr>
<td>4</td>
<td>16 (0x10)</td>
<td>0</td>
<td>Establishes a Temp Connection. A Temp Connection is one that is set, but is not stored in flash, and therefore is not restored when the master reboots. If the NetLinx code is adding URL entries, it is recommended to make them temporary so that the flash is not constantly being written, especially since the code handles all the connections anyway.</td>
</tr>
<tr>
<td>5</td>
<td>32 (0x20)</td>
<td>0</td>
<td>Source of URL. 0 = Programmed by the IDE. 1 = Programmed by NetLinx ADD_URL_ENTRY.</td>
</tr>
<tr>
<td>6</td>
<td>64 (0x40)</td>
<td>0</td>
<td>Encoded status indication (Read only). These 2 bits together form one of 4 possible codes indicating the status of the connection. 0x00 - Looking up IP address or URL. 0x40 - Connecting to URL. 0x80 - Waiting for connection to establish. 0xC0 - Connected.</td>
</tr>
</tbody>
</table>
| 7   | 128 (0x80)       | 0x00         | }
**GET_IP_ADDRESS Flags Member Bit Fields**

The Flags member is a bit field that is used for several different purposes, as defined below:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Mathematic Value</th>
<th>Normal Value</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| 0   | 1 (0x01)         | 1            | 0 = Use the provided static IP address  
|     |                  |              | 1 = Use DHCP to obtain an IP address   |
| 1   | 2 (0x02)         | 0            | Unused  |
| 2   | 4 (0x04)         | 0            | Unused  |
| 3   | 8 (0x08)         | 0            | Unused  |
| 4   | 16 (0x10)        | 0            | Unused  |
| 5   | 32 (0x20)        | 0            | Unused  |
| 6   | 64 (0x40)        | 0            | Unused  |
| 7   | 128 (0x80)       | 0x00         | Unused  |
# Level Keywords

NetLinx supports the following LEVEL keywords:

<table>
<thead>
<tr>
<th>Level Keywords</th>
<th>Description</th>
</tr>
</thead>
</table>
| ~LEVSYNCON              | Enables a feature that helps synchronize level values. By default, this feature is disabled for compatibility reasons. The synchronization algorithm works by setting the level value of a level five seconds after receiving a level value from a level. While it may not be apparent, this makes sure that level values remain in sync with each other if they ever get out of sync. The only way levels could ever get out of sync is when the situation of “dueling levels” arises. A typical example of “dueling levels” is when two touch panels with active sliders are combined with a volume control. If one slider attempts to raise the volume level while the other is attempting to lower the volume level the level value bounces back and forth somewhere between the desired levels. If both sliders are released at the exact same time, it is possible that one of level values displayed on the touch panel's slider is inaccurate. The level synchronization algorithm corrects the incorrect level five seconds after activity ceases. The commands are ~LEVSYNCON and ~LEVSYNCOFF are sent to the level that should have the synchronization algorithm enabled or disabled. The command itself is never sent to the device because the master intercepts the command and processes it internally. Both commands accept a single parameter that specifies the level number. Using the “dueling levels” example above, the following send commands will turn on the synchronization algorithm for level #1 of Touch Panel 1, level #4 of touch panel #2, and level #2 of the volume control.  
SEND_COMMAND dvTouchPanel1,'~LEVSYNCON 1'
SEND_COMMAND dvTouchPanel2,'~LEVSYNCON 4'
SEND_COMMAND dvVolume,'~LEVSYNCON 2'
Note that for some devices, turning the level synchronization algorithm on can cause undesired results. The undesired results will vary from device to device so it is difficult to indicate any specific failure mode. Keep in mind that the algorithm should only be turned on when necessary. Also note that the LEVSYNCON and LEVSYNCOFF SEND_COMMANDs may not be sent to remote devices (devices that belong to other systems) and only the device's master may issue these commands. |
| ~LEVSYNCOFF             | Disables a feature that helps synchronize level values. By default, this feature is disabled for compatibility reasons. |
| COMBINE_LEVELS          | See page 46.                                                                                     |
| CREATE_LEVEL            | This keyword creates an association between a specified level of a device and a variable that will contain the value of the level. This can only appear in the DEFINE_START section of the program.  
CREATE_LEVEL DEV, Level, Value  
Parameters:  
• DEV: The device from which to read the level.  
• Level: The level of the device to read.  
• Value: Variable in which to store the level value.  
• DL[]: A DevLev structure.  
• Value: Variable in which to store the level value CREATE_LEVEL DevLev, Value.  
During execution of the program, NetLinx continuously updates the variable to match the level it represents. |
| DEFINE_CONNECT_         | See page 46.                                                                                     |
| LEVEL                   |                                                                                                  |
| SEND_LEVEL              | This keyword sends a value to a specific level on a NetLinx device/port. The syntax follows any one of the four following examples:  
SEND_LEVEL DEV, Level, Value  
SEND_LEVEL DEV[], Level, Value  
SEND_LEVEL DEV Levl, Value  
SEND_LEVEL DEV Levl[], Value  
Parameters:  
• DEV: Device containing the specified level.  
• Level: Number of the level to receive the new value.  
• Value: New level value.  
• DEV[]: Device array (each device contains the specified level).  
• DEV Levl: Device-level to receive the new value.  
• DL[]: Device-level array (each will receive the new value). |
| SET_VIRTUAL_LEVEL_      |                                                                                                  |
| COUNT                   | See page 122.                                                                                     |
Listview Keywords

The NetLinx programming language supports the following Listview keywords. These keywords support the Listview and Dynamic Data functionality in Modero G5 touch panels and TPDesign5:

**LISTVIEW_ON_ROW_SELECT_EVENT**

LISTVIEW_ON_ROW_SELECT_EVENT is a Custom Event that is raised in response to a user selection of an item in a Listview button. When the user selects an item on the Listview button, a LISTVIEW_ON_ROW_SELECT_EVENT is raised and the entire data feed record for that selection is sent to the master. The user can then use DATA_GET_EVENT_RECORD to retrieve the specific values of interest.

*NOTE:* "payloadId" specifies the data access identifier to be retrieved from the custom event object, and is predefined as "custom.value1". "payloadType" specifies the data type to be retrieved from the custom event object and is predefined as "custom.value2".

The following code example illustrates how the LISTVIEW_ON_ROW_SELECT_EVENT Custom Event is used to retrieve two data fields ("name" and "number") when a listview item is selected.

```c
// The custom event that is raised whenever a listview item is // selected on the panel
CUSTOM_EVENT(dvTP,btnListview,LISTVIEW_ON_ROW_SELECT_EVENT)
{
    SLONG payloadId
    SLONG payloadType
    // just a char array to hold the data we want to use in the // custom event.
    CHAR fields[2][16]
    // char variables to hold our data for "name" & "number"
    CHAR name[DATA_MAX_VALUE_LENGTH]
    CHAR number[DATA_MAX_VALUE_LENGTH]
    // variable record, of type DATA_RECORD, to hold the record // we retrieve from the custom event
    DATA_RECORD record
    // Get the data access ID from the custom event // variable is payloadId - custom.value1 is predefined
    payloadId = custom.value1
    // Get the data type from the custom event // variable is payloadType - custom.value2 is predefined
    payloadType = custom.value2
    if (payloadId > 0 && payloadType == DATA_STRUCTURE_DATARECORD)
    {
        // Specify which fields we want to retrieve from the payload // (these are the IDs we defined earlier)
        fields[1] = 'name'
        fields[2] = 'number'
        // Retrieve the record and get our requested fields
        if (DATA_GET_EVENT_RECORD(dvTP, payloadId, fields, record) > 0)
        {
            // The record existed and contained our fields
            // let's retrieve the values that we are interested in
            name = record.content[1].value
            number = record.content[2].value
            // Send the name & number that was retrieved to the // appropriate buttons & show the popup
            SEND_COMMAND dvTP,"'^TXT-50,0,\',name"
            SEND_COMMAND dvTP,"'^TXT-51,0,\',number"
            SEND_COMMAND dvTP,"'^PPN-Calling'"
        }
    }
}
```

**DATA_FEED**

The DATA_FEED structure contains information describing a Listview Dynamic Data feed. DATA_FEED structure fields:

- **NAME** - A string identifying the data feed name
- **DESCRIPTION** - A string containing a short description of the data feed
- **SOURCE** - A string describing the source of the data feed
- **LASTUPDATE** - A LONG value to indicate the time of the last update

*NOTE:* See the "Listview Buttons & Dynamic Data" section of the TPDesign5 Instruction Manual for a detailed description of format and usage of the DATA_FEED structure.
### Listview Keywords (Cont.)

<table>
<thead>
<tr>
<th>Structure</th>
<th>Description</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DATA_FIELD</strong></td>
<td>The DATA_FIELD structure contains information describing an individual data field within a DATA_RECORD metadata or content.</td>
<td>• ID - A string containing a unique identifier for the field&lt;br&gt;• TYPE - A string containing the field type. Valid data field types are:&lt;br&gt;  - DATA_TYPE_UNKNOWN&lt;br&gt;  - DATA_TYPE_STRING&lt;br&gt;  - DATA_TYPE_DATETIME&lt;br&gt;  - DATA_TYPE_DATE&lt;br&gt;  - DATA_TYPE_TIME&lt;br&gt;  - DATA_TYPE_IMAGE&lt;br&gt;• FORMAT - A string containing the field format. Valid data field formats are:&lt;br&gt;  - DATA_FORMAT_URL&lt;br&gt;  - DATA_FORMAT_PHONE&lt;br&gt;  - DATA_FORMAT_EMAIL&lt;br&gt;  - DATA_FORMAT_ISO8601&lt;br&gt;• LABEL - A string containing the field label&lt;br&gt;• VALUE - A string containing the field value&lt;br&gt;<strong>NOTE:</strong> See the “Listview Buttons &amp; Dynamic Data” section of the TPD5 Instruction Manual for a detailed description of format and usage of the DATA_FEED structure.</td>
</tr>
<tr>
<td><strong>DATA_RECORD</strong></td>
<td>The DATA_RECORD structure contains information describing a data record within a data feed.</td>
<td>• METADATA - DATA_FIELD array containing a list of metadata values associated with the record&lt;br&gt;• CONTENT - DATA_FIELD array containing a list of data fields for the record&lt;br&gt;<strong>NOTE:</strong> See the “Listview Buttons &amp; Dynamic Data” section of the TPD5 Instruction Manual for a detailed description of format and usage of the DATA_FEED structure.</td>
</tr>
<tr>
<td><strong>WC_DATA_FEED</strong></td>
<td>The WC_DATA_FEED structure contains information describing a WIDECHAR Listview Dynamic Data feed.</td>
<td>• NAME - A string identifying the data feed name&lt;br&gt;• DESCRIPTION - A WIDECHAR string containing a short description of the data feed&lt;br&gt;• SOURCE - A WIDECHAR string describing the source of the data feed&lt;br&gt;• LASTUPDATE - A LONG value to indicate the time of the last update&lt;br&gt;<strong>NOTE:</strong> See the “Listview Buttons &amp; Dynamic Data” section of the TPD5 Instruction Manual for a detailed description of format and usage of the DATA_FEED structure.</td>
</tr>
<tr>
<td><strong>WC_DATA_FIELD</strong></td>
<td>The WC_DATA_FIELD structure contains information describing an individual data field within a WC_DATA_RECORD metadata or content.</td>
<td>• ID - A WIDECHAR string containing a unique identifier for the field&lt;br&gt;• TYPE - A string containing the field type. Valid data field types are:&lt;br&gt;  - DATA_TYPE_UNKNOWN&lt;br&gt;  - DATA_TYPE_STRING&lt;br&gt;  - DATA_TYPE_DATETIME&lt;br&gt;  - DATA_TYPE_DATE&lt;br&gt;  - DATA_TYPE_TIME&lt;br&gt;  - DATA_TYPE_IMAGE&lt;br&gt;• FORMAT - A string containing the field format. Valid data field formats are:&lt;br&gt;  - DATA_FORMAT_URL&lt;br&gt;  - DATA_FORMAT_PHONE&lt;br&gt;  - DATA_FORMAT_EMAIL&lt;br&gt;  - DATA_FORMAT_ISO8601&lt;br&gt;• LABEL - A WIDECHAR string containing the field label&lt;br&gt;• VALUE - A WIDECHAR string containing the field value&lt;br&gt;<strong>NOTE:</strong> See the “Listview Buttons &amp; Dynamic Data” section of the TPD5 Instruction Manual for a detailed description of format and usage of the DATA_FEED structure.</td>
</tr>
<tr>
<td><strong>WC_DATA_RECORD</strong></td>
<td>The WC_DATA_RECORD structure contains information describing a WIDECHAR data record within a WIDECHAR data feed.</td>
<td>• METADATA - WC_DATA_FIELD array containing a list of metadata values associated with the record&lt;br&gt;• CONTENT - WC_DATA_FIELD array containing a list of data fields for the record&lt;br&gt;<strong>NOTE:</strong> See the “Listview Buttons &amp; Dynamic Data” section of the TPD5 Instruction Manual for a detailed description of format and usage of the WC_DATA_RECORD structure.</td>
</tr>
</tbody>
</table>
### Listview Keywords (Cont.)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Syntax</th>
<th>Parameters</th>
<th>Result</th>
<th>Example</th>
<th>Note</th>
</tr>
</thead>
</table>
| **DATA_CREATE_FEED**   | The `DATA_CREATE_FEED` function creates a NetLinx data feed with the supplied values.                                                                                                                    | `SINTEGER DATA_CREATE_FEED (DATA_FEED FEED)`                          | • FEED: A `DATA_FEED` structure populated with the desired data feed identification values | 1 = Data feed created  
-1 = Data feed create failed due to invalid parameter | `STACK_VAR DATA_FEED datafeed`  
// CREATE A NEW DATA FEED  
// datafeed.name = 'phonelist'  
datafeed.description = 'Employees'  
datafeed.source = 'netlinx Listview Example code'  
`DATA_CREATE_FEED(datafeed)`                                                         | See the "Listview Buttons & Dynamic Data" section of the TPD5 Instruction Manual for a detailed description of format and usage of the `WC_DATA_RECORD` structure. |
| **DATA_DELETE_FEED**   | The `DATA_DELETE_FEED` function deletes the NetLinx data feed with the specified name.                                                                                                                   | `SINTEGER DATA_CREATE_FEED (CHAR FEED[])`                             | • FEED: A string containing the name of the data feed to delete              | 1 = Data feed deleted  
-1 = Data feed create failed due to invalid parameter | `DATA_DELETE_FEED('phonelist')`                                              |                                                                                                                                 |
| **DATA_PUBLISH_FEED**  | The `DATA_PUBLISH_FEED` function publishes the specified data feed to a file and returns a string containing the URL reference to the file.                                                              | `CHAR[] DATA_PUBLISH_FEED (CHAR FEED[])`                             | • FEED: A string containing the name of the data feed to publish           | A string containing the URL of the published data feed file OR a textual error message indicating a publish failure | `DEFINE_VARIABLE`  
`STACKVAR CHAR publishedURL[DATA_MAX_VALUE_LENGTH]`  
publishedURL = `DATA_PUBLISH_FEED('phonelist')`                                         | See the "Listview Buttons & Dynamic Data" section of the TPD5 Instruction Manual for a detailed description of format and usage of the `WC_DATA_RECORD` structure. |
| **DATA_GET_PUBLISHED_FEED** | The `DATA_GET_PUBLISHED_FEED` function returns the URL of an already published feed matching the specified data feed name.                                                                                  | `CHAR[] DATA_GET_PUBLISHED_FEED (CHAR FEED[])`                       | • FEED: A string containing the name of the published data feed            | A string containing the URL of the published data feed file OR a textual error message indicating an error | `DEFINE_VARIABLE`  
`STACKVAR CHAR publishedURL[DATA_MAX_VALUE_LENGTH]`  
publishedURL = `DATA_GET_PUBLISHED_FEED('phonelist')`                                         | See the "Listview Buttons & Dynamic Data" section of the TPD5 Instruction Manual for a detailed description of format and usage of the `WC_DATA_RECORD` structure. |
**DATA_ADD_RECORD**

The `DATA_ADD_RECORD` function adds a new record to a data feed.

**Syntax:**
```
SINTEGER DATA_ADD_RECORD (CHAR FEED[], CHAR [] RECORDSET_ID, DATA_RECORD REC)
```

**Parameters:**
- **FEED:** A string containing the name of the data feed to add the record to
- **RECORDSET_ID:** A string containing the name of the record set the record belongs to
- **REC:** A DATA_RECORD containing the record values to add to the data feed

**Result:**
- 1 = Record added
- -1 = Record failed to add due to invalid parameter

**Example:**
```
STACK_VAR DATA_RECORD record

// Records can have metadata fields and content fields. In this example we won't use any metadata
SET_LENGTH_ARRAY(record.metadata, 0)

// We will have 3 content fields per record: photo, name and phone number
SET_LENGTH_ARRAY(record.content, 3)

// Initialize the field attributes that will be the same for every record
// the first field in a record will be the image
record.content[1].id = 'photo';
record.content[1].type = DATA_TYPE_IMAGE;
record.content[1].format = DATA_FORMAT_URL;

// The label can be something different from the id but in our case we'll keep them the same
record.content[1].label = 'photo';

// The second field in a record will be the name
record.content[2].id = 'name';
record.content[2].type = DATA_TYPE_STRING;
record.content[2].format = '';
record.content[2].label = 'name';

// The third field will be the phone number
record.content[3].id = 'number';
record.content[3].type = DATA_TYPE_STRING;
record.content[3].format = DATA_FORMAT_PHONE;
record.content[3].label = 'number';

// The next step is to put in the actual values for the 3 fields
// Do this for the first record
record.content[1].value = 'http://192.168.222.333/ftp/listview/hunter.jpg'
record.content[2].value = 'Hunter Pence'
record.content[3].value = '888-555-1111'

// Add the record to the feed phonelist data feed
DATA_ADD_RECORD('phonelist', 'phonelist', record)
```

**NOTE:** See the "Listview Buttons & Dynamic Data" section of the TPD5 Instruction Manual for a detailed description of format and usage of the WC_DATA_RECORD structure.

---

**DATA_GET_EVENT_RECORD**

The `DATA_GET_EVENT_RECORD` function retrieves data feed event record values from a data feed custom event.

**Syntax:**
```
SINTEGER DATA_GET_EVENT_RECORD (DEV device, LONG payloadID, char fields[][], DATA_RECORD rec)
```

**Parameters:**
- **device:** NetLinx device event is coming from
- **payloadID:** Payload identifier supplied in custom event value 1
- **fields:** array of strings specifying what fields from the record to return
- **rec:** DATA_RECORD structure containing the desired field values from the data feed event

**Result:**
- 1 = Record values retrieved
- -1 = Failed to retrieve values due to invalid parameter

**Example:**
```
CUSTOM_EVENT[devTP,btnListview,LISTVIEW_ON_ROW_SELECT_EVENT]
{
  SLONG payloadID
  SLONG payloadType
  CHAR fields[2][16]
  CHAR name[DATA_MAX_VALUE_LENGTH]
  CHAR number[DATA_MAX_VALUE_LENGTH]
  DATA_RECORD record
  // Get the data access ID from the custom event
  payloadID = custom.value1
```
### DATA_GET_EVENT_RECORD (Cont.)

```c
// Get the data type from the custom event payloadType = custom.value2
Listview Buttons & Dynamic Data
if (payloadID > 0 && payloadType == DATA_STRUCTURE_DATARECORD) {
  // Specify which fields we want to retrieve from the payload fields[1] = 'name'
  fields[2] = 'number'

  // Populate a record with the requested fields from the event if (DATA_GET_EVENT_RECORD(dvTP, payloadID, fields, record) > 0) {
    // All is well so far so retrieve the values that we are interested in from the selection that the user made on the panel.
    name = record.content[1].value
    number = record.content[2].value
    // Put the name and number that was selected on a popup and show the popup.
    SEND_COMMAND dvTP, "^TXT-50,0,',name"
    SEND_COMMAND dvTP, "^TXT-51,0,',number"
    SEND_COMMAND dvTP, "^PPN-Calling"
  }
}
}
```

**NOTE:** See the "Listview Buttons & Dynamic Data" section of the TP5 Instruction Manual for a detailed description of format and usage of the WC_DATA_RECORD structure.

---

### WC_DATA_CREATE_FEED

The WC_DATA_CREATE_FEED function creates a WIDECHAR NetLinx data feed with the supplied values.

**Syntax:**
```c
SINTEGER WC_DATA_CREATE_FEED (WC_DATA_FEED FEED)
```

**Parameters:**
- FEED: A WC_DATA_FEED structure populated with the desired data feed identification values

**Result:**
- 1 = Data feed created
- -1 = Data feed create failed due to invalid parameter

**NOTE:** See the "Listview Buttons & Dynamic Data" section of the TP5 Instruction Manual for a detailed description of format and usage of the WC_DATA_RECORD structure.

---

### WC_DATA_ADD_RECORD

The WC_DATA_ADD_RECORD function adds a new record to a WIDECHAR data feed.

**Syntax:**
```c
SINTEGER WC_DATA_ADD_RECORD (CHAR FEED[], WIDECHAR [] RECORDSET_ID, WC_DATA_RECORD REC)
```

**Parameters:**
- FEED: A string containing the name of the data feed to add the record to
- RECORDSET_ID: A WIDECHAR string containing the name of the record set the record belongs to
- REC: A WC_DATA_RECORD containing the record values to add to the data feed

**Result:**
- 1 = Record added
- -1 = Record failed to add due to invalid parameter

**NOTE:** See the "Listview Buttons & Dynamic Data" section of the TP5 Instruction Manual for a detailed description of format and usage of the WC_DATA_RECORD structure.

---

### WC_DATA_GET_EVENT_RECORD

The WC_DATA_GET_EVENT_RECORD function retrieves WIDECHAR data feed event record values from a data feed custom event.

**Syntax:**
```c
SINTEGER WC_DATA_GET_EVENT_RECORD (DEV device, LONG payloadID, WIDECHAR fields[][], WC_DATA_RECORD rec)
```

**Parameters:**
- device: NetLinx device event is coming from
- payloadID: Payload identifier supplied in custom event value
- fields: array of WIDECHAR strings specifying what fields from the record to return
- rec: WC_DATA_RECORD structure containing the desired field values from the data feed event

**Result:**
- 1 = Record values retrieved
- -1 = Failed to retrieve values due to invalid parameter

**NOTE:** See the "Listview Buttons & Dynamic Data" section of the TP5 Instruction Manual for a detailed description of format and usage of the WC_DATA_RECORD structure.
### Log Keywords

The NetLinx programming language supports the following Log keywords:

<table>
<thead>
<tr>
<th><strong>Log Keywords</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SET_LOG_LEVEL</strong></td>
<td>Sets the current log level for the program. All subsequent logs at the specified level or lower will cause a log message out the NetLinx master’s logging facilities.</td>
</tr>
<tr>
<td></td>
<td><strong>NOTE:</strong> The final terminal output is further filtered based on the terminal session &quot;msg on&quot; level. See the MSG ON</td>
</tr>
<tr>
<td></td>
<td>Syntax: <code>SET_LOG_LEVEL(CONSTANT INTEGER LEVEL)</code></td>
</tr>
<tr>
<td></td>
<td>The four valid log levels are:</td>
</tr>
<tr>
<td></td>
<td>• INTEGER AMX_ERROR = 1</td>
</tr>
<tr>
<td></td>
<td>• INTEGER AMX_WARNING = 2</td>
</tr>
<tr>
<td></td>
<td>• INTEGER AMX_INFO = 3</td>
</tr>
<tr>
<td></td>
<td>• INTEGER AMX_DEBUG = 4</td>
</tr>
<tr>
<td><strong>GET_LOG_LEVEL</strong></td>
<td>Retrieves the current log level for the program.</td>
</tr>
<tr>
<td></td>
<td><code>INTEGER GET_LOG_LEVEL()</code></td>
</tr>
<tr>
<td></td>
<td>Where the returned value will be one of the following levels:</td>
</tr>
<tr>
<td></td>
<td>• INTEGER AMX_ERROR = 1</td>
</tr>
<tr>
<td></td>
<td>• INTEGER AMX_WARNING = 2</td>
</tr>
<tr>
<td></td>
<td>• INTEGER AMX_INFO = 3</td>
</tr>
<tr>
<td></td>
<td>• INTEGER AMX_DEBUG = 4</td>
</tr>
<tr>
<td><strong>AMX_LOG</strong></td>
<td>Sends the specified message to the NetLinx master's logging facilities if the current log level setting is at least as large as LEVEL. For example, if the current log level setting is AMX_WARNING, calling log with level of AMX_ERROR will cause a log, while AMX_INFO would not.</td>
</tr>
<tr>
<td></td>
<td><strong>NOTE:</strong> This command is supported by NetLinx Controller firmware version 4 or higher.</td>
</tr>
<tr>
<td></td>
<td><strong>NOTE:</strong> The final terminal output is further filtered based on the terminal session &quot;msg on&quot; level. See the MSG ON</td>
</tr>
<tr>
<td></td>
<td>The AMX_LOG function is meant to replace the &quot;SEND_STRING 0,...&quot; log method.</td>
</tr>
<tr>
<td></td>
<td><code>AMX_LOG(CONSTANT INTEGER LEVEL, CHAR MSG[])</code></td>
</tr>
<tr>
<td></td>
<td>Where level is one of the following values:</td>
</tr>
<tr>
<td></td>
<td>• INTEGER AMX_ERROR = 1</td>
</tr>
<tr>
<td></td>
<td>• INTEGER AMX_WARNING = 2</td>
</tr>
<tr>
<td></td>
<td>• INTEGER AMX_INFO = 3</td>
</tr>
<tr>
<td></td>
<td>• INTEGER AMX_DEBUG = 4</td>
</tr>
<tr>
<td></td>
<td>Example: <code>AMX_LOG(AMX_ERROR,&quot;'FAILURE OCCURRED, VALUE=',ITOA(ERR_VAL)&quot;)</code></td>
</tr>
</tbody>
</table>
Math Functions

Math functions are supported by NetLinx Controller firmware version 4 or higher. The NetLinx programming language supports the following Math Function keywords:

<table>
<thead>
<tr>
<th>Math Function Keywords</th>
<th>Description</th>
</tr>
</thead>
</table>
| EXP_VALUE              | Returns the base-e exponential function of x, which is the e number raised to the power of x.  
  `EXP_VALUE(CONSTANT VARIANT X)`  
  Where X can be any intrinsic type. (INTEGER, FLOAT, DOUBLE, etc) |
| LOG_VALUE              | Returns the natural logarithm of x. The natural logarithm is the base-e logarithm, the inverse of the natural exponential function (EXP_VALUE).  
  `LOG_VALUE(CONSTANT VARIANT X)`  
  Where X can be any intrinsic type (INTEGER, FLOAT, DOUBLE, etc) with a value greater than 0. |
| LOG10_VALUE            | Returns the common (base-10) logarithm of x.  
  `LOG10_VALUE(CONSTANT VARIANT X)`  
  Where X can be any intrinsic type (INTEGER, FLOAT, DOUBLE, etc) with a value greater than 0. |
| POWER_VALUE            | Returns BASE raised to the power EXPONENT.  
  `POWER_VALUE(CONSTANT VARIANT BASE, CONSTANT VARIANT EXPONENT)`  
  Where BASE and EXPONENT can be any intrinsic type (INTEGER, FLOAT, DOUBLE, etc). The following combinations are error conditions which will return 0:  
  - BASE = 0 and EXPONENT=negative  
  - BASE=positive and EXPONENT=non-integral value |
| SQRT_VALUE             | Returns the square root of x.  
  `SQRT_VALUE(CONSTANT VARIANT X)`  
  Where X can be any intrinsic type (INTEGER, FLOAT, DOUBLE, etc) with a value greater than 0. |
Module Keywords

NetLinx Modules

The ability to reuse code is a desirable goal in software development; however, code reuse takes careful planning and organization. As discussed earlier, NetLinx provides tools such as functions and modules to promote reusability. Modules are NetLinx sub-programs designed to be "plugged into" a main program.

Defining a Module

The MODULE_NAME entry on the first line of the file defines the module. The syntax is:

```
MODULE_NAME = '<module name>' [(<parameter list>)]
```

The MODULE_NAME entry identifies the file as containing a NetLinx module, as opposed to a standard NetLinx source code file. The module name is any valid string literal not to exceed 64 characters. A file can contain only one module and the file name must be the same as the module name with the addition of the "AXS" extension. Module parameters behave exactly like subroutine parameters; the parameter list is optional. The value for each parameter is set either by the main program or another module. If the value of a parameter is changed, both the main program and module see the change.

NOTE: Constants and expressions cannot be used as arguments in the parameter list.

The example below defines a module named ModuleExample. Aside from the MODULE_NAME entry, the code looks like any standard NetLinx source code file. All parameters to a module must be one of the intrinsic types: CHAR, INTEGER, SINTEGER, LONG, SLONG, FLOAT, DOUBLE, DEV, DEVCHAN or DEVLEV. Also, any of the above array types can be used.

```
MODULE_NAME='ModuleExample'(DEV dvDECK, DEVCHAN dcTRANPORTS[], INTEGER nFIRST)
(*{{PS_SOURCE_INFO(PROGRAM STATS)
    *([ORPHAN_FILE_PLATFORM: 1]
    *)PS_SOURCE_INFO
    *)}
DEFINE_DEVICE
(* ******************** CONSTANT DEFINITIONS GO BELOW *)
DEFINE_CONSTANT
NO_BUTTON = 0
NO_FUNCTION = 256
PLAY = 1
STOP = 2
PAUSE = 3
FFWD = 4
REW = 5
SFWD = 6
SREV = 7
REC = 8
PLAY_FB = 241
STOP_FB = 242
PAUSE_FB = 243
FFWD_FB = 244
REW_FB = 245
SFWD_FB = 246
SREV_FB = 247
REC_FB = 248
(* vcr will go into stop after rewinding for a certain time *)
VCR1_REW_TO_STOP = 1800 (* 3 min *)
(* vcr will go into stop after search rewinding for a certain time *)
VCR1_SREV_TO_STOP = 12000 (* 20 min *)
(* vcr will go into stop after being paused for a certain time *)
VCR1_PAUSE_TO_STOP = 6000 (* 10 min *)
(* button feedback flag *)
VCR1_DEFEAT_FEEDBACK = 0
```
DEFINE_TYPE

(* VARIABLE DEFINITIONS GO BELOW *)

DEFINE_VARIABLE

VOLATILE INTEGER nOFFSET_FN        (* FUNCTION OFFSET *)
VOLATILE INTEGER nOFFSET_FB        (* FEEDBACK OFFSET *)
VOLATILE INTEGER nFUNC             (* FUNCTION THAT WAS PRESSED *)

DEFINE_CALL 'ALL OFF'
{
  OFF [dvDECK,nOFFSET_FN+PLAY]
  OFF [dvDECK,nOFFSET_FN+STOP]
  OFF [dvDECK,nOFFSET_FN+PAUSE]
  OFF [dvDECK,nOFFSET_FN+FFWD]
  OFF [dvDECK,nOFFSET_FN+REW]
  OFF [dvDECK,nOFFSET_FN+SFWD]
  OFF [dvDECK,nOFFSET_FN+SREV]
  OFF [dvDECK,nOFFSET_FN+REC]
}

DEFINE_CALL 'FEEDBACK' (INTEGER nFUNCTION)
{
  [dvDECK,nOFFSET_FB+PLAY_FB]  = (nFUNCTION=PLAY)
  [dvDECK,nOFFSET_FB+STOP_FB]  = (nFUNCTION=STOP)
  [dvDECK,nOFFSET_FB+PAUSE_FB] = (nFUNCTION=PAUSE)
  [dvDECK,nOFFSET_FB+FFWD_FB]  = (nFUNCTION=FFWD)
  [dvDECK,nOFFSET_FB+REW_FB]   = (nFUNCTION=REW)
  [dvDECK,nOFFSET_FB+SFWD_FB]  = (nFUNCTION=SFWD)
  [dvDECK,nOFFSET_FB+SREV_FB]  = (nFUNCTION=SREV)
  [dvDECK,nOFFSET_FB+REC_FB]   = (nFUNCTION=REC)
}

(* STARTUP CODE GOES BELOW *)

DEFINE_START

(* SELECT OFFSETS IF ANY *)
IF (nFIRST BAND $00FF)
  nOFFSET_FN=(nFIRST BAND $00FF)-PLAY
ELSE
  nOFFSET_FN=0

IF (nFIRST BAND $FF00)
  nOFFSET_FB=((nFIRST BAND $FF00)/$FF)-PLAY_FB
ELSE
  nOFFSET_FB=0

DEFINE_EVENT

(* BUTTON_EVENT - TRANSPORT CONTROLS *)

BUTTON_EVENT[dcTRANSPORTS]
{
  PUSH:
    
    #IF_DEFINED_SYSCALL_NOTIFY
    SEND_STRING 0,"'IN MODULE ',39,'ModuleExample',39"
    #END_IF
(* RUN A FUNCTION *)
nFUNC = GET_LAST(dcTRANSPORTS)
SWITCH (nFUNC)
{
    CASE PLAY:
        IF (! [dvDECK, nOFFSET_FB+REC_FB])
            CANCEL_WAIT 'VCR1 REW TO STOP'
            CANCEL_WAIT 'VCR1 PAUSE TO STOP'
            CANCEL_WAIT 'VCR1 SREV TO STOP'
            CALL 'ALL OFF'
            MIN_TO [dvDECK, nOFFSET_FN+PLAY]
            CALL 'FEEDBACK' (PLAY)
    }

    CASE STOP:
        CANCEL_WAIT 'VCR1 REW TO STOP'
        CANCEL_WAIT 'VCR1 PAUSE TO STOP'
        CANCEL_WAIT 'VCR1 SREV TO STOP'
        CALL 'ALL OFF'
        MIN_TO [dvDECK, nOFFSET_FN+STOP]
        CALL 'FEEDBACK' (STOP)
    }

    CASE PAUSE:
        SELECT
        {
            ACTIVE ([dvDECK, nOFFSET_FB+PAUSE_FB]
                AND [dvDECK, nOFFSET_FB+REC_FB]
                AND dcTRANSPORTS[8].CHANNEL=NO_FUNCTION):
                CANCEL_WAIT 'VCR1 REW TO STOP'
                CANCEL_WAIT 'VCR1 PAUSE TO STOP'
                CANCEL_WAIT 'VCR1 SREV TO STOP'
                CALL 'ALL OFF'
                MIN_TO [dvDECK, nOFFSET_FN+REC]
                CALL 'FEEDBACK' (REC)
            }
            ACTIVE ([dvDECK, nOFFSET_FB+PAUSE_FB]
                AND dcTRANSPORTS[1].CHANNEL<NO_FUNCTION):
                CANCEL_WAIT 'VCR1 REW TO STOP'
                CANCEL_WAIT 'VCR1 PAUSE TO STOP'
                CANCEL_WAIT 'VCR1 SREV TO STOP'
                CALL 'ALL OFF'
                MIN_TO [dvDECK, nOFFSET_FN+PLAY]
                CALL 'FEEDBACK' (PLAY)
            }
            ACTIVE ([dvDECK, nOFFSET_FB+PLAY_FB]):
                CANCEL_WAIT 'VCR1 REW TO STOP'
                CANCEL_WAIT 'VCR1 PAUSE TO STOP'
                CANCEL_WAIT 'VCR1 SREV TO STOP'
                WAIT VCR1_PAUSE_TO_STOP 'VCR1 PAUSE TO STOP'
                SYSTEM_CALL 'FUNCTION' (dvDECK,STOP,nFIRST)
                CALL 'ALL OFF'
                MIN_TO [dvDECK, nOFFSET_FN+PAUSE]
                CALL 'FEEDBACK' (PAUSE)
            }
            ACTIVE ([dvDECK, nOFFSET_FB+REC_FB]):
                CANCEL_WAIT 'VCR1 REW TO STOP'
                CANCEL_WAIT 'VCR1 PAUSE TO STOP'
                CANCEL_WAIT 'VCR1 SREV TO STOP'
                WAIT VCR1_PAUSE_TO_STOP 'VCR1 PAUSE TO STOP'
                SYSTEM_CALL 'FUNCTION' (dvDECK,STOP,nFIRST)
                CALL 'ALL OFF'
                MIN_TO [dvDECK, nOFFSET_FN+PAUSE]
                CALL 'FEEDBACK' (PAUSE)
            ON [dvDECK, nOFFSET_FB+REC_FB]
        }
CASE FFWD:
{
  SELECT
  {
    ACTIVE ({dvDECK,nOFFSET_FB+STOP_FB]} OR {dvDECK,nOFFSET_FB+FFWD_FB} OR {dvDECK,nOFFSET_FB+REW_FB} OR (dcTRANPORTS[6].CHANNEL
    AND ({dvDECK,nOFFSET_FB+PLAY_FB] OR {dvDECK,nOFFSET_FB+SREV_FB} OR {dvDECK,nOFFSET_FB+SFWD_FB}])):
    {
      CANCEL_WAIT 'VCR1 REW TO STOP'
      CANCEL_WAIT 'VCR1 PAUSE TO STOP'
      CANCEL_WAIT 'VCR1 SREV TO STOP'
      CALL 'ALL OFF'
      MIN_TO [dvDECK,nOFFSET_FN+FFWD]
      CALL 'FEEDBACK' (FFWD)
    }
    ACTIVE (dcTRANSPORTS[6].CHANNEL=NO_BUTTON
      AND ({dvDECK,nOFFSET_FB+PLAY_FB] OR {dvDECK,nOFFSET_FB+SREV_FB} OR {dvDECK,nOFFSET_FB+SFWD_FB}])):
      {
        CANCEL_WAIT 'VCR1 REW TO STOP'
        CANCEL_WAIT 'VCR1 PAUSE TO STOP'
        CANCEL_WAIT 'VCR1 SREV TO STOP'
        CALL 'ALL OFF'
        MIN_TO [dvDECK,nOFFSET_FN+SFWD]
        CALL 'FEEDBACK' (SFWD)
      }
  }
}

CASE SFWD:
{
  IF ({dvDECK,nOFFSET_FB+PLAY_FB] OR {dvDECK,nOFFSET_FB+STOP_FB} OR {dvDECK,nOFFSET_FB+REW_FB} OR {dvDECK,nOFFSET_FB+FFWD_FB} OR {dvDECK,nOFFSET_FB+SREV_FB} OR {dvDECK,nOFFSET_FB+SFWD_FB})
    {
      CANCEL_WAIT 'VCR1 REW TO STOP'
      CANCEL_WAIT 'VCR1 PAUSE TO STOP'
      CANCEL_WAIT 'VCR1 SREV TO STOP'
      CALL 'ALL OFF'
      MIN_TO [dvDECK,nOFFSET_FN+SFWD]
      CALL 'FEEDBACK' (SFWD)
    }
}

CASE REW:
{
  SELECT
  {
    ACTIVE ({dvDECK,nOFFSET_FB+STOP_FB]} OR {dvDECK,nOFFSET_FB+FFWD_FB} OR {dvDECK,nOFFSET_FB+REW_FB} OR (dcTRANSPORTS[7].CHANNEL
      AND ({dvDECK,nOFFSET_FB+PLAY_FB] OR {dvDECK,nOFFSET_FB+SREV_FB} OR {dvDECK,nOFFSET_FB+SFWD_FB}])):
      {
        CANCEL_WAIT 'VCR1 REW TO STOP'
        CANCEL_WAIT 'VCR1 PAUSE TO STOP'
        CANCEL_WAIT 'VCR1 SREV TO STOP'
        WAIT VCR1_REW_TO_STOP 'VCR1 REW TO STOP'
        SYSTEM_CALL 'FUNCTION' (dvDECK,STOP,nFIRST)
        CALL 'ALL OFF'
        MIN_TO [dvDECK,nOFFSET_FN+REW]
        CALL 'FEEDBACK' (REW)
ACTIVE (dcTRANPORTS[7].CHANNEL=NO_BUTTON
AND ([(dvDECK,OFFSET_FB+PLAY_FB] OR [(dvDECK,OFFSET_FB+SREV_FB] OR [(dvDECK,OFFSET_FB+SFWD_FB]})];
{
CANCEL_WAIT 'VCR1 REM TO STOP'
CANCEL_WAIT 'VCR1 PAUSE TO STOP'
CANCEL_WAIT 'VCR1 SREV TO STOP'
WAIT VCR1_SREV_TO_STOP 'VCR1 SREV TO STOP'
SYSTEM_CALL 'FUNCTION' (dvDECK,STOP,nFIRST)
CALL 'ALL OFF'
MIN_TO [dvDECK,OFFSET_FN+SREV]
CALL 'FEEDBACK' (SREV)
}
)

CASE SREV:
{
{
CANCEL_WAIT 'VCR1 REM TO STOP'
CANCEL_WAIT 'VCR1 PAUSE TO STOP'
CANCEL_WAIT 'VCR1 SREV TO STOP'
WAIT VCR1_SREV_TO_STOP 'VCR1 SREV TO STOP'
SYSTEM_CALL 'FUNCTION' (dvDECK,STOP,nFIRST)
CALL 'ALL OFF'
MIN_TO [dvDECK,OFFSET_FN+SREV]
CALL 'FEEDBACK' (SREV)
}
)

CASE REC:
{
IF ([(dvDECK,OFFSET_FB+STOP_FB] OR [(dvDECK,OFFSET_FB+REC_FB]})
{
CANCEL_WAIT 'VCR1 REM TO STOP'
CANCEL_WAIT 'VCR1 PAUSE TO STOP'
CANCEL_WAIT 'VCR1 SREV TO STOP'
CALL 'ALL OFF'
MIN_TO [dvDECK,OFFSET_FN+REC]
CALL 'FEEDBACK' (REC)
}
)

[************************************************]
(*            THE ACTUAL PROGRAM GOES BELOW                *)
[************************************************]

DEFINE_PROGRAM

[dcTRANPORTS[1]] = [(dvDECK,OFFSET_FB+PLAY_FB]
[dcTRANSPORTS[2]] = [(dvDECK,OFFSET_FB+STOP_FB]
[dcTRANSPORTS[3]] = [(dvDECK,OFFSET_FB+PAUSE_FB]
[dcTRANSPORTS[4]] = [(dvDECK,OFFSET_FB+FFWD_FB] OR (dcTRANSPORTS[6].CHANNEL=NO_BUTTON AND [(dvDECK,OFFSET_FB+SFWD_FB])
[dcTRANSPORTS[5]] = [(dvDECK,OFFSET_FB+REW_FB] OR (dcTRANSPORTS[7].CHANNEL=NO_BUTTON AND [(dvDECK,OFFSET_FB+SREV_FB])
[dcTRANSPORTS[6]] = [(dvDECK,OFFSET_FB+SFWD_FB]
[dcTRANSPORTS[7]] = [(dvDECK,OFFSET_FB+SREV_FB]
[dcTRANSPORTS[8]] = [(dvDECK,OFFSET_FB+REC_FB] AND ([(dvDECK,OFFSET_FB+PAUSE_FB])

[************************************************]
(*        DO NOT PUT ANY CODE BELOW THIS COMMENT         *)
[************************************************]
Using a Module in a Program

To use a module in a program, you must declare it using the DEFINE_MODULE keyword. This tells the NetLinx compiler to add the module to the program, effectively merging the module's event handling and mainline code with the containing program (or module). In other words, the program will have one event table and one mainline routine consisting of code from the main program and all modules declared using the DEFINE_MODULE statement.

Technically, modules can contain declarations to other modules, provided that no circular references are involved. However, because different instances of the same module must not be separated by instances of a different module, it is highly recommended that you do not declare modules from within other modules - if you have multiple declarations of the parent module they will then be separated by the declarations of the child module.

FIG. 3 demonstrates how a NetLinx module is incorporated into a main program. In this example, the main program has no event table or mainline code.

```
FIG. 3 Mainline and Event Table Organization

PROGRAM_NAME="ModuleExampleTest"

(*{{PS_SOURCE_INFO(PROGRAM STATS)                          *)
(* ORPHAN_FILE_PLATFORM: 1                                *)
(*]|PS_SOURCE_INFO                                         *)
(*          DEVICE NUMBER DEFINITIONS GO BELOW             *)
(*]|)}PS_SOURCE_INFO                                     *)

DEFINE_DEVICE
dvVCR     = 1:7:0
dvTP      = 128:1:0

(* VARIABLE DEFINITIONS GO BELOW *)

DEFINE_VARIABLE
VOLATILE
DEVCHAN dcTRANPORTS[] = {
  { dvTP,1 }, { dvTP,2 }, { dvTP,3 }, { dvTP,4 },
  { dvTP,5 }, { dvTP,6 }, { dvTP,7 }, { dvTP,8 }
}

VOLATILE
INTEGER nVCR_FIRST = 0

(* MODULE CODE GOES BELOW *)

DEFINE_MODULE 'ModuleExample' mdlVCR(dvVCR,dcTRANPORTS,nVCR_FIRST)

(* END OF PROGRAM *)
(* DO NOT PUT ANY CODE BELOW THIS COMMENT *)
```
Module Keywords

NetLinx supports the following Module keywords:

<table>
<thead>
<tr>
<th>Module Keywords</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DEFINE_MODULE</strong></td>
<td>This keyword declares a module that will be used by either the main program or another module. It is the counterpart to the MODULE_NAME entry that appears as part of the implementation of the module.</td>
</tr>
<tr>
<td></td>
<td><strong>DEFINE_MODULE '&lt;module name&gt;' InstanceName(&lt;parameter list&gt;)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Parameters:</strong></td>
</tr>
<tr>
<td></td>
<td>• &lt;module name&gt;: The name of the module as specified in the MODULE_NAME statement in the module implementation file.</td>
</tr>
<tr>
<td></td>
<td>• InstanceName: The name to assign to the instance of the module.</td>
</tr>
<tr>
<td></td>
<td>• &lt;parameter list&gt;: The list of parameters available to the module.</td>
</tr>
<tr>
<td><strong>DUET_MEM_SIZE_GET</strong></td>
<td>Display the amount of memory allocated for Duet Java pool.</td>
</tr>
<tr>
<td></td>
<td>This is the current Java memory heap size as measured in Megabytes.</td>
</tr>
<tr>
<td></td>
<td>An example is a value of 5 = 5 MB.</td>
</tr>
<tr>
<td><strong>DUET_MEM_SIZE_SET</strong></td>
<td>Set the amount of memory allocated for Duet Java pool.</td>
</tr>
<tr>
<td></td>
<td>This is the current Java memory heap size as measured in Megabytes.</td>
</tr>
<tr>
<td></td>
<td>This feature is used so that if a NetLinx program requires a certain size of memory be allotted for its currently used Duet Modules, it can be reserved on the target Master.</td>
</tr>
<tr>
<td></td>
<td>Valid values are:</td>
</tr>
<tr>
<td></td>
<td>2 - 8 for 32MB systems</td>
</tr>
<tr>
<td></td>
<td>2 - 36 for 64MB systems</td>
</tr>
<tr>
<td></td>
<td>This setting does not take effect until the next reboot.</td>
</tr>
<tr>
<td></td>
<td><strong>NOTE:</strong> &quot;DUET_MEM_SIZE_SET(int)&quot; should call REBOOT() following a set.</td>
</tr>
<tr>
<td><strong>MODULE_NAME</strong></td>
<td>This keyword introduces the definition of a module.</td>
</tr>
<tr>
<td></td>
<td><strong>MODULE_NAME = '&lt;module name&gt;' (&lt;parameter list&gt;)</strong></td>
</tr>
<tr>
<td></td>
<td>See <strong>DEFINE_MODULE</strong>, on page 116, for more information.</td>
</tr>
</tbody>
</table>
Operator Keywords

Overview
An Operator is a character or group of characters that performs a specific mathematical or relational function. Each operator type is described below.

Arithmetic Operators
Arithmetic operators create a numeric value from one or more operations such as addition, multiplication, and division.

<table>
<thead>
<tr>
<th>Arithmetic Operators</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>%</td>
<td>Modulo (remainder after division)</td>
</tr>
</tbody>
</table>

Relational Operators
A relational operator is a conditional statement; it tells NetLinx whether to execute a particular function(s) in the program.

<table>
<thead>
<tr>
<th>Relational Operators</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Less Than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater Than</td>
</tr>
<tr>
<td>=</td>
<td>Equal To</td>
</tr>
<tr>
<td>==</td>
<td>Equal To</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less Than or Equal To</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater Than or Equal To</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not Equal To</td>
</tr>
</tbody>
</table>

Logical Operators
Logical operators compare two conditions or, in the case of NOT, invert one condition. A true or false result is produced.

<table>
<thead>
<tr>
<th>Logical Operators</th>
<th>Function</th>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>Logical And</td>
<td>AND (see page 118)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>^^</td>
<td>Logical Xor</td>
<td>XOR (see page 118)</td>
</tr>
<tr>
<td>!</td>
<td>Logical Not</td>
<td>NOT (see page 118)</td>
</tr>
</tbody>
</table>

Bitwise Operators
Bitwise operators are keywords or symbols that perform a bit-by-bit operation between two items.

<table>
<thead>
<tr>
<th>Bitwise Operators</th>
<th>Function</th>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Bitwise And</td>
<td>BAND (see page 118)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bitwise Or</td>
</tr>
<tr>
<td>^</td>
<td>Bitwise Xor</td>
<td>BXOR (see page 118)</td>
</tr>
<tr>
<td>~</td>
<td>Bitwise Not</td>
<td>BNOT (see page 118)</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Shift Left</td>
<td>LSHIFT (see page 118)</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Shift Right</td>
<td>RSHIFT (see page 118)</td>
</tr>
</tbody>
</table>

Assignment Operators
The assignment operators may appear only once in a single NetLinx statement.

<table>
<thead>
<tr>
<th>Assignment Operators</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Assignment</td>
</tr>
<tr>
<td>++</td>
<td>Increment by 1</td>
</tr>
<tr>
<td>--</td>
<td>Decrement by 1</td>
</tr>
</tbody>
</table>

The following rules apply to the use of assignment operators:

- The "=" operator may be used to assign:
  - Expressions to intrinsic type variables (see the Data Type Keywords section on page 55)
  - Arrays to other arrays of matching size and type
  - Structures to other structures of the same type
The "++" and "--" operators are statements and cannot appear within expressions. For example:

\[
\text{FOR } (I=1; I<10; I++) \quad // \text{ Legal}
I = j++; \quad // \text{ Illegal}
\]

**NOTE:** Refer to the Structure Keywords section on page 128 for information on structures.

**Operator Precedence**

The table below shows the inherent precedence assigned to the operators.

<table>
<thead>
<tr>
<th>Level</th>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>! ~</td>
<td>Left To Right</td>
</tr>
<tr>
<td>2</td>
<td>* / %</td>
<td>Left To Right</td>
</tr>
<tr>
<td>3</td>
<td>&lt;&lt; &gt;&gt;</td>
<td>Left To Right</td>
</tr>
<tr>
<td>4</td>
<td>+ -</td>
<td>Left To Right</td>
</tr>
<tr>
<td>5</td>
<td>&lt; &lt;= &gt; &gt;= == &lt;&gt;</td>
<td>Left To Right</td>
</tr>
<tr>
<td>6</td>
<td>&amp;</td>
<td>^</td>
</tr>
<tr>
<td>7</td>
<td>&amp;&amp;</td>
<td></td>
</tr>
</tbody>
</table>

**NOTE:** As noted in the chart, the NOT(!) operator has the highest precedence in NetLinx systems but the lowest precedence in Axcess systems. Axcess programs that are converted to NetLinx may exhibit logic problems if they use statements that combine NOT(!) and other operators. Contact AMX Technical Support for help resolving these issues.

**Operator Keywords**

NetLinx supports the following Operators:

- **AND (&&)**: This logical operator evaluates two logical conditions. Both conditions must be true for the entire expression to be true.
- **BAND (&)**: This operator performs a bitwise AND on two data items, which can be constants or variables.
- **BNOT (~)**: This operator performs a bitwise NOT on a constant or variable.
- **BOR (|)**: This operator performs a bitwise OR on two data items, which can be constants or variables.
- **BXOR (^)**: This operator performs a bitwise XOR operation between two data items, which can be constants or variables.
- **LSHIFT**: This keyword causes the bits in the associated integer field to be shifted left. This has the effect of multiplying by \(2^n\), where \(n\) is the number of bit positions to shift. The symbol \(<\) is equivalent to LSHIFT.

For example:

\[
\text{INT2} = \text{INT1 LSHIFT 2}
\]

is equivalent to:

\[
\text{INT2} = \text{INT1 << 2}
\]

Both statements shift \(\text{INT1}\) left two positions. Either statement could be replaced with the following:

\[
\text{INT2} = \text{INT1} * 4
\]

- **MOD (%)**: This keyword is used to generate the remainder of a division function. You cannot take the mod of an integer without first loading the value into a variable.

For example:

\[
\text{VRAM_LSB} = ((2 \% 16)+$30) \quad (* \text{ does not work } *)
\]

However,

\[
\text{ID} = 2
\]
\[
\text{OTHER} = 16
\]
\[
\text{VRAM_LSB} = (((\text{ID} \% \text{OTHER}) + $30) \quad (* \text{ works } *)
\]

- **NOT (!)**: This keyword is used to negate a given expression.

**IF** \(\text{NOT} (X > 10)\)

\[
\{ \quad // \text{ statements to execute if } X \leq 10
\}

- **OR (||)**: This keyword evaluates two conditions. If one or both conditions are true, the entire expression evaluates to true.

- **RSHIFT**: This keyword causes the bits in the associated value field to be shifted right. This has the effect of dividing by \(2^n\) where \(n\) is the number of bit positions to shift. The symbol \(>>\) is equivalent to RSHIFT.

For example:

\[
\text{INT2} = \text{INT1 RSHIFT 2}
\]

is equivalent to:

\[
\text{INT2} = \text{INT1} >> 2
\]

Both statements shift \(\text{INT1}\) right two positions. Either statement could be replaced with:

\[
\text{INT2} = \text{INT1} / 4
\]

- **XOR (^^)**: This keyword evaluates two conditions. One and only one condition can be true for the entire expression to be true.
Port Keywords

The NetLinx programming language supports the following Port keywords:

<table>
<thead>
<tr>
<th>Port Keywords</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DYNAMIC_POLLED_PORT</td>
<td>Designates a NetLinx serial port that should be polled for dynamic device detection. This API must be called for each serial port that can dynamically have a device plugged into it. DYNAMIC_POLLED_PORT (DEV netlinxDevice)</td>
</tr>
<tr>
<td>FIRST_LOCAL_PORT</td>
<td>This keyword contains the lowest number that may be assigned as a local port number.</td>
</tr>
<tr>
<td>STATIC_PORT_BINDING</td>
<td>Designates an application device along with its SDK class and the physical interface it is bound to.</td>
</tr>
</tbody>
</table>
## Push and Release Keywords

NetLinx supports the following PUSH and RELEASE keywords:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DO_PUSH</strong></td>
<td>This keyword causes an input change from OFF to ON to occur on a specified device-channel without the device-channel being activated by external means. To prevent the program from stalling mainline too long, there is a 0.5 second timeout on DO_PUSH. <strong>DO_PUSH</strong> defaults to a 0.5 second push on a channel before issuing a <strong>DO_RELEASE</strong> for you (unless another <strong>DO_PUSH</strong> is executed for the same channel). NetLinx will forcibly exit the <strong>DO_PUSH</strong> after 0.5 seconds, regardless of the operation it is executing. If the channel is already ON, no event is generated. <strong>NOTE</strong>: The timeout feature is used to prevent un-released pushes and out of control ramping. <strong>DO_PUSH</strong>(DEVICE, CHANNEL)</td>
</tr>
</tbody>
</table>
| **DO_PUSH_TIMED** | Similar to **DO_PUSH**, except **DO_PUSH_TIMED** lets you specify the timeout, so you can control the length of time that will pass before the automatic **DO_RELEASE** is generated. **DO_PUSH_TIMED**(DEV Device, INTEGER Channel, LONG Timeout) Parameters:  
  • Device: The device to PUSH.  
  • Channel: The channel to PUSH.  
  • Timeout: The time (in 1/10ths of seconds) the PUSH remains active. If zero is specified as the timeout then the timeout is 0.5 seconds. If **DO_PUSH_TIMED_INFINITY** is specified as the timeout then the push never times out. **DO_PUSH_TIMED**(dvTouchPanel, 5, 10) // push button 5 for 1.0S |
| **DO_RELEASE**  | This keyword causes an input change from ON to OFF to occur on a specified device and channel without the channel being deactivated by external means. If the channel is already OFF, no event is generated. **DO_RELEASE**(DEVICE, CHANNEL) |
| **MIN_TO**      | This keyword operates just like the TO keyword, except that the specified channel or variable stays on for a minimum length of time, even if the corresponding channel is released. The minimum length of time is set by **SET_PULSE_TIME**. **MIN_TO** follows the same conditions of operation as the TO keyword. See **SET_PULSE_TIME**, on page 122, for more information. |
| **PUSH**        | This keyword declares a block of code to be executed when a push event is received for the associated device and channel. An example **PUSH** statement is shown below:  
  
  **PUSH**(DEVICE,CHANNEL)PUSH[DEVCHAN[ ]]  
  {// statements}  
  This keyword also defines a section in the **BUTTON_EVENT** handler for processing **PUSH** events. |
| **PUSH_CHANNEL**| This keyword contains the channel number that was just turned on due to an input change. The value remains valid for one pass through mainline. The inactive state of this variable is all fields equal to zero. |
| **PUSH_DEVCHAN**| This keyword contains the device-channel (a DEVCHAN structure) that was just turned on due to an input change. Individual fields of this DEVCHAN structure can be accessed using dot-operator syntax, as shown below:  
  
  **PUSH_DEVCHAN**.Device  
  **PUSH_DEVCHAN**.Device.Number  
  **PUSH_DEVCHAN**.Device.Port  
  **PUSH_DEVCHAN**.Device.System  
  **PUSH_DEVCHAN**.Channel  
  • These fields remain valid for one pass through mainline.  
  • The inactive state of this variable is **all fields equal to zero**. |
| **PUSH_DEVICE** | This keyword contains the number of the device that was just turned on due to an input change. The value remains valid for one pass through mainline. The inactive state of this variable is all fields equal to zero. |
| **RELEASE**     | This keyword declares a block of code to be executed when a release event is received for the associated device and channel.  
  
  **RELEASE**(DEVICE,CHANNEL)RELEASE[DEVCHAN[ ]]  
  {// statements}  
  This keyword also defines a section in a **BUTTON_EVENT** handler for processing **RELEASE** events. |
| **RELEASE_CHANNEL** | This keyword contains the number of the channel that was just turned off due to an input change.  
  • The value remains valid for one pass through mainline.  
  • The inactive state of this variable is **all fields equal to zero**. |
| **RELEASE_DEVCHAN** | This keyword contains the device-channel (a DEVCHAN structure) that was just turned off due to an input change. Individual fields of this DEVCHAN structure can be accessed using dot-operator syntax, as shown below:  
  
  **RELEASE_DEVCHAN**.Device  
  **RELEASE_DEVCHAN**.Device.Number  
  **RELEASE_DEVCHAN**.Device.Port  
  **RELEASE_DEVCHAN**.Device.System  
  **RELEASE_DEVCHAN**.Channel  
  • These fields remain valid for one pass through mainline.  
  • The inactive state of this variable is **all fields equal to zero**. |
| **RELEASE_DEVICE** | This system variable contains the number of the device associated with the channel that was just turned off due to an input change.  
  • The value remains valid for one pass through mainline.  
  • The inactive state of this variable is **all fields equal to zero**. |
### PUSH and RELEASE Keywords (Cont.)

| **TO** | This keyword activates a channel or variable for as long as the corresponding channel of its PUSH statement is activated. When the channel referenced by the PUSH statement changes from off to on, the TO command starts activating the associated channel or variable. When the channel is released, the TO command stops activating of the channel or variable. Therefore, a TO statement must be found underneath a PUSH statement only.

| **Syntax** | The syntax is shown below:

TO (DEVICE, CHANNEL) TO (DEVCHAN[ ]) TO (Variable)

**Several conditions apply to the use of the TO command:**

- It must be used only below a PUSH statement.
- It cannot be used as part of a set of WAIT statements.
- It cannot be placed in the DEFINE_START section.

The channel or variable will act under the rules set by DEFINE_LATCHING, DEFINE_MUTUALLY_EXCLUSIVE, and DEFINE_TOGGLING. |
## SET Keywords

NetLinx supports the following SET keywords:

<table>
<thead>
<tr>
<th>SET Keywords</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SET_DNS_LIST</strong></td>
<td>See page 100.</td>
<td>------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>SET_IP_ADDRESS</strong></td>
<td>See page 100.</td>
<td>------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>SET_LENGTH_ARRAY</strong></td>
<td>See page 32.</td>
<td>------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>SET_LENGTH_STRING</strong></td>
<td>See page 127.</td>
<td>------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>SET_OUTDOOR_TEMPERATURE</strong></td>
<td>This function establishes the value for the outdoor temperature.</td>
<td>SET_OUTDOOR_TEMPERATURE (INTEGER Temp)</td>
</tr>
<tr>
<td></td>
<td>• This value is broadcast to all devices periodically.</td>
<td>Parameters:</td>
</tr>
<tr>
<td></td>
<td>• A value of 32768 indicates that no outdoor temperature is available.</td>
<td>• Temp: The outdoor temperature as it shall be displayed. It is up to the programmer to provide the correct temperature scale whether it is Celsius or Fahrenheit.</td>
</tr>
<tr>
<td></td>
<td>SET_OUTDOOR_TEMPERATURE (32) // show 32 degrees</td>
<td>------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>SET_PULSE_TIME</strong></td>
<td>This function sets the PULSE time in 1/10th second units. The default PULSE time is 5 (0.5 seconds).</td>
<td>SET_PULSE_TIME (TIME)</td>
</tr>
<tr>
<td><strong>PULSE</strong></td>
<td>This keyword turns a channel or variable on for the length of time set by the function SET_PULSE_TIME.</td>
<td>Parameters:</td>
</tr>
<tr>
<td></td>
<td>Once the pulse time elapses, the channel or variable is turned off.</td>
<td>• Device, Channel: The device and channel the pulse function is applied to.</td>
</tr>
<tr>
<td></td>
<td>PULSE [DEVICE, CHANNEL]</td>
<td>------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>PULSE [DEVCHAN[]]</td>
<td>------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>PULSE [Variable]</td>
<td>------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>SET_SYSTEM_NUMBER</strong></td>
<td>Sets the system number of the NetLinx master. The new system number will take effect after the system has been rebooted.</td>
<td>SLONG SET_SYSTEM_NUMBER (INTEGER newSystemNum)</td>
</tr>
<tr>
<td>Parameters:</td>
<td>Desired new system number</td>
<td>Parameters:</td>
</tr>
<tr>
<td>Result:</td>
<td>• 0: Operation was successful.</td>
<td>• newSystemNum: The new system number.</td>
</tr>
<tr>
<td></td>
<td>• -1: System number is invalid.</td>
<td>------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>• -2: Assignment of system number causes conflict.</td>
<td>------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>This function only affects the master’s system number, not the system number of any attached devices. Therefore, any devices with pre-programmed system numbers will no longer communicate with this master.</td>
<td>------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>SET_SYSTEM_NUMBER (3) // set new system number</td>
<td>------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>SET_TIMER</strong></td>
<td>This keyword resets the system timer. The system timer counts up in 1/10th second units. The value passed to this function (TIME) may be any unsigned 32-bit variable or constant. This provides a timer with a maximum range of over 13 years.</td>
<td>SET_TIMER (TIME)</td>
</tr>
<tr>
<td><strong>NOTE:</strong></td>
<td>The system timer is reset to zero on power up.</td>
<td>------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>This function lets the programmer override the default number of channels that a virtual device port maintains. By default every virtual device port maintains the state of channels 1-255 inclusive.</td>
<td>SET_VIRTUAL_CHANNEL_COUNT (DEV Device, INTEGER Count)</td>
</tr>
<tr>
<td>Parameters:</td>
<td>Device: The virtual device port to modify.</td>
<td>Parameters:</td>
</tr>
<tr>
<td></td>
<td>Count: The number of channels that the specified virtual device port should maintain.</td>
<td>• Device: The virtual device port to modify.</td>
</tr>
<tr>
<td></td>
<td>SET_VIRTUAL_CHANNEL_COUNT (dvVirtual, 1024) // 1024 channels</td>
<td>• Count: The number of channels that the specified virtual device port should maintain.</td>
</tr>
<tr>
<td><strong>SET_VIRTUAL_LEVEL_COUNT</strong></td>
<td>This function lets the programmer override the default number of levels that a virtual device port maintains. By default every virtual device port maintains the state of levels 1-8 inclusive.</td>
<td>SET_VIRTUAL_LEVEL_COUNT (DEV Device, INTEGER Count)</td>
</tr>
<tr>
<td>Parameters:</td>
<td>Device: The virtual device port to modify.</td>
<td>Parameters:</td>
</tr>
<tr>
<td></td>
<td>Count: The number of levels that the specified virtual device port should maintain.</td>
<td>• Device: The virtual device port to modify.</td>
</tr>
<tr>
<td></td>
<td>SET_VIRTUAL_LEVEL_COUNT (dvVirtual, 10) // make it have 10 levels</td>
<td>• Count: The number of levels that the specified virtual device port should maintain.</td>
</tr>
<tr>
<td><strong>SET_VIRTUAL_PORT_COUNT</strong></td>
<td>This function lets the programmer override the default number of ports that a virtual device port maintains. By default every virtual device port maintains the state of a single port (port 1).</td>
<td>SET_VIRTUAL_PORT_COUNT (DEV Device, INTEGER Count)</td>
</tr>
<tr>
<td>Parameters:</td>
<td>Device: The virtual device to modify.</td>
<td>Parameters:</td>
</tr>
<tr>
<td></td>
<td>Count: The number of ports that the specified virtual device should maintain.</td>
<td>• Device: The virtual device to modify.</td>
</tr>
<tr>
<td></td>
<td>SET_VIRTUAL_PORT_COUNT (dvVirtual, 2) // 2 ports</td>
<td>• Count: The number of ports that the specified virtual device should maintain.</td>
</tr>
</tbody>
</table>
SMTP Keywords

Overview

SMTP functionality is supported by NetLinx Controller firmware version 4 or higher. NetLinx supports the following SMTP keywords:

<table>
<thead>
<tr>
<th>SMTP Keywords</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMTP_SERVER_CONFIG_SET</td>
<td>Set a configuration value for the current SMTP server.</td>
</tr>
<tr>
<td></td>
<td>• SMTP Server configuration will be retained between boots of the master.</td>
</tr>
<tr>
<td></td>
<td>• Once the server configuration values have been set, email can be sent using the SMTP_SEND() API.</td>
</tr>
<tr>
<td>Syntax:</td>
<td>SMTP_SERVER_CONFIG_SET(CONSTANT CHAR CONFIG_NAME, CONSTANT CHAR CONFIG_VALUE)</td>
</tr>
<tr>
<td></td>
<td>Where CONFIG_NAME is one of the following:</td>
</tr>
<tr>
<td></td>
<td>• SMTP_ADDRESS - Used to set the address of the SMTP server (e.g. 'mail.acme.com')</td>
</tr>
<tr>
<td></td>
<td>• SMTP_PORT_NUMBER - Used to set the IP port number to connect to on the SMTP server (e.g. '25').</td>
</tr>
<tr>
<td></td>
<td>• SMTP_USERNAME - Used to set the username for server authentication. If username length is set to 0, authentication is not attempted when connecting to the server.</td>
</tr>
<tr>
<td></td>
<td>• SMTP_PASSWORD - Used to set the password for server authentication. If password length is set to 0, authentication is still attempted but a zero-length password (NULL_STR) is sent.</td>
</tr>
<tr>
<td></td>
<td>• SMTP_FROM - Used to set the 'Mail-From:' field in outgoing emails.</td>
</tr>
<tr>
<td></td>
<td>• SMTP_REQUIRE_TLS - Used to set whether TLS authentication security should be required when connecting to the server. Valid values are SMTP_TLS_TRUE and SMTP_TLS_FALSE.</td>
</tr>
<tr>
<td>Example:</td>
<td>SMTP_SERVER_CONFIG_SET(SMTP_ADDRESS, 'mail.acme.com')</td>
</tr>
<tr>
<td></td>
<td>SMTP_SERVER_CONFIG_SET(SMTP_PORT_NUMBER,'25')</td>
</tr>
<tr>
<td></td>
<td>SMTP_SERVER_CONFIG_SET(SMTP_USERNAME, '<a href="mailto:john.doe@acme.com">john.doe@acme.com</a>')</td>
</tr>
<tr>
<td></td>
<td>SMTP_SERVER_CONFIG_SET(SMTP_PASSWORD, 'mypassword')</td>
</tr>
<tr>
<td></td>
<td>SMTP_SERVER_CONFIG_SET(SMTP_FROM, '<a href="mailto:john.doe@acme.com">john.doe@acme.com</a>')</td>
</tr>
<tr>
<td>SMTP_SERVER_CONFIG_GET</td>
<td>Get a configuration value for the current SMTP server.</td>
</tr>
<tr>
<td>Syntax:</td>
<td>CHAR[] SMTP_SERVER_CONFIG_GET(CONSTANT CHAR CONFIG_NAME)</td>
</tr>
<tr>
<td></td>
<td>Where CONFIG_NAME is one of the following:</td>
</tr>
<tr>
<td></td>
<td>• SMTP_ADDRESS - Used to get the address of the SMTP server (e.g. 'mail.acme.com').</td>
</tr>
<tr>
<td></td>
<td>• SMTP_PORT_NUMBER - Used to get the IP port number to connect to on the SMTP server (e.g. '25').</td>
</tr>
<tr>
<td></td>
<td>• SMTP_USERNAME - Used to set the username for server authentication. If username length is set to 0, authentication is not attempted when connecting to the server.</td>
</tr>
<tr>
<td></td>
<td>• SMTP_PASSWORD - Used to set the password for server authentication. If password length is set to 0, authentication is still attempted but a zero-length password (NULL_STR) is sent.</td>
</tr>
<tr>
<td></td>
<td>• SMTP_FROM - Used to set the 'Mail-From:' field in outgoing emails.</td>
</tr>
<tr>
<td></td>
<td>• SMTP_REQUIRE_TLS - Used to set whether TLS authentication security should be required when connecting to the server. Valid return values are SMTP_TLS_TRUE and SMTP_TLS_FALSE.</td>
</tr>
<tr>
<td>NOTE: Query of SMTP_USERNAME and SMTP_PASSWORD is disabled for security reasons.</td>
<td></td>
</tr>
<tr>
<td>Example:</td>
<td>CURRENT_ADDRESS = SMTP_SERVER_CONFIG_GET(SMTP_ADDRESS)</td>
</tr>
<tr>
<td></td>
<td>CURRENT_PORT = SMTP_SERVER_CONFIG_GET(SMTP_PORT_NUMBER)</td>
</tr>
<tr>
<td></td>
<td>CURRENT_FROM = SMTP_SERVER_CONFIG_GET(SMTP_FROM)</td>
</tr>
<tr>
<td></td>
<td>CURRENT_TLS = SMTP_SERVER_CONFIG_GET(SMTP_REQUIRE_TLS)</td>
</tr>
<tr>
<td>SMTP_SEND</td>
<td>Sends an email to a single destination. It returns an identifier associated with the email event. The email is sent asynchronously by the on-board SMTP client. If the mail transmission fails, an ONERROR DATA event will be sent to the supplied status DPS with DATA.NUMBER set to the error code and DATA.TEXT set to the string representation of the email identifier.</td>
</tr>
<tr>
<td>Syntax:</td>
<td>SINTEGER SMTP_SEND(DEV DPS, CONSTANT CHAR TO_ADDRESS[], CONSTANT CHAR SUBJECT[], CONSTANT CHAR BODY[], CONSTANT CHAR TEXT_ATTACHMENT[])</td>
</tr>
<tr>
<td>Where:</td>
<td>• DPS is a DEV to receive asynchronous send status</td>
</tr>
<tr>
<td></td>
<td>• TO_ADDRESS is a string containing the email address of the destination. String must be less than 127 characters.</td>
</tr>
<tr>
<td></td>
<td>• SUBJECT is a string containing the email subject line</td>
</tr>
<tr>
<td></td>
<td>• BODY is a string containing the email body text</td>
</tr>
<tr>
<td></td>
<td>• TEXT_ATTACHMENT is a string containing the filename of a text file to be attached to the email. Filename must be less than 256 characters and file size must be under 65536 bytes. Can be specified as NULL_STR when no attachment is desired.</td>
</tr>
<tr>
<td>Example:</td>
<td>MAIL_IDX1 = SMTP_SEND (0:3:0, '<a href="mailto:john.doe@acme.com">john.doe@acme.com</a>','Mail Subject','This is the mail text','attachment.txt')</td>
</tr>
<tr>
<td></td>
<td>MAIL_IDX2 = SMTP_SEND(0:3:0,'<a href="mailto:jane.doe@acme.com">jane.doe@acme.com</a>','Mail Alert', 'This is an email alert!',NULL_STR)</td>
</tr>
<tr>
<td></td>
<td>DEFINE_EVENT DATA_EVENT [0:3:0]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
String Keywords

Overview

A string is an array of characters of known length. This length may be less than the dimensioned length. For example:

```plaintext
DEFINE_VARIABLE
CHAR MyString[32]
INTEGER StrLen

DEFINE_START
MyString = 'STOP'
StrLen = LENGTH_STRING(MyString)
```

In the example above, StrLen holds the value 4, the length of MyString. The length of MyString can range from 0 to 32. If an attempt is made to assign a string longer than the capacity of the destination string, the copied string is truncated to fit. The string length is implicitly set when a string literal, string expression, or variable is assigned to the string. The function SET_LENGTH_STRING can be used to explicitly set the length of a string to any arbitrary length between 0 and the dimension of the character array. For example:

```plaintext
SET_LENGTH_STRING(MyString, 3)
```

causes the contents of MyString to read 'STO', even though the character 'P' still resides in MYSTRING[4].

String Expressions

A string expression is a string enclosed in double quotes containing a series of constants and/or variables evaluated at run-time to form a string result. String expressions can contain up to 16000 characters consisting of string literals, variables, arrays, and ASCII values between 0 and 255. For example:

```plaintext
CHAR StrExp[6]
StrExp = "STOP, 25, 'OFF', X"
```

In the example above, the string expression contains the constant STOP, the value 25, the string literal 'OFF', and the variable X. Assuming STOP is 2 and X = 5, the string expression will evaluate to "2, 25, 'OFF', 5".

Wide Strings

The wide string (wide character string data type) is provided for dealing with Unicode fonts, which use 16-bit character codes, used for many Far-Eastern fonts (instead of the standard 8-bit codes used with most Western fonts). Here's a syntax sample for a wide character string:

```plaintext
WIDECHAR WChar[40]
```

The statement above declares a wide character string containing 40 elements, for a total of 80 bytes. A wide character string can be used in the same manner as other character strings. It maintains a length field that can be retrieved using LENGTH_STRING and set using SET_LENGTH_STRING. For example:

```plaintext
WIDECHAR StrExp[6]
INTEGER StrLen

StrExp = {STOP, 500, 'OFF', X}
StrLen = LENGTH_STRING(StrExp)
```

In the example above, if STOP is 2 and X is a wide character whose value is 1000, the string expression will evaluate to "2, 500, 79, 70, 70, 1000" and StrLen is 6. Each array element can now assume a value of up to 65,535, rather than the limit of 255 imposed by the standard character string. A CHAR string may be assigned or compared to a wide character string. For example:

```plaintext
WChar = 'FFWD'

- or -

IF (WChar = 'REV')
{
    (* statements *)
}
```

Each 8-bit character in the CHAR string is converted to 16-bit before the assignment or comparison operation is performed.
## STRING Keywords

NetLinx supports the following STRING keywords:

<table>
<thead>
<tr>
<th>STRING Keywords</th>
<th>Description</th>
</tr>
</thead>
</table>
| **CHARD**       | Sets the delay between all transmitted characters to that specified in 100-microsecond increments.  
The syntax:  
```
CHARD-<time in 100 microsecond increments>
```
Example:  
```
SEND_COMMAND device,'CHARD-100'
```
Sets a 10mS delay between all transmitted characters. |
| **CHARDM**      | Sets the delay between all transmitted characters to that specified in 1-millisecond increments.  
The syntax:  
```
CHARDM-<time in 1 millisecond increments>
```
Example:  
```
SEND_COMMAND device,'CHARDM-100'
```
Sets a 10 mS delay between all transmitted characters. |
| **COMPARE_STRING** | This keyword compares two character strings. If either string contains a '?' character, the matching character in the other string is not compared. The '?' is equivalent to a wildcard.  
Example:  
```
DEFINE_LIBRARY_FUNCTION LONG COMPARE_STRING(CHAR A[], CHAR B[])
```
Here is some useful debugging code:  
```
tstStr = 'ALEXERICRYAN'
ulError = COMPARE_STRING ( tstStr, 'ALEX' )
if( ulError == 0 )
   SEND_STRING dvDebug, 'ALEXERICRYAN != ALEX... BAD!'
else
   SEND_STRING dvDebug, 'ALEXERICRYAN != ALEX... BAD!'
```
```
tstStr = 'ALEXERICRYAN'
ulError = COMPARE_STRING ( tstStr, 'ALEXERICRYAN' )
if ( ulError == 0 )
   SEND_STRING dvDebug, 'ALEXERICRYAN != ALEXERICRYAN...BAD!'
else
   SEND_STRING dvDebug, 'ALEXERICRYAN != ALEXERICRYAN'
```
```
tstStr = 'ALEXERICRYAN'
ulError = COMPARE_STRING ( tstStr, 'ALEX????RYAN' )
if ( ulError == 0 )
   SEND_STRING dvDebug, 'ALEXERICRYAN != ALEX????RYAN...BAD!'
else
   SEND_STRING dvDebug, 'ALEXERICRYAN == ALEX????RYAN'
```
Another example of a use for this feature is if you want an event to occur every hour. You would enter a time string that would contain a '??;00 ;00' (hours/minute/sec) for the recurring event that in this case would occur every hour.  
Result: The returned result can only be True (1) or False (0).  
- 0 = the strings don't match  
- 1 = the strings are the same |
| **FIND_STRING** | This function searches through a string for a specified sequence of characters.  
```
INTEGER FIND_STRING (CHAR STRING[], CHAR Seq[], INTEGER Start)
```
```
INTEGER FIND_STRING (WIDECHAR STRING[], WIDECHAR Seq[], INTEGER Start)
```
Parameters:  
- STRING: The string of character to search.  
- Seq: The sequence of characters to search for.  
- Start: The starting character position for the search.  
Result: A 16-bit unsigned integer representing the character location of Seq in STRING. If the character string is found at the beginning of the string, this function returns 1; any error condition returns 0.  
```
POS = FIND_STRING('STRING', 'ABC', 1)
```
|
| **LEFT_STRING** | This function returns the specified number of characters from the beginning of a string.  
```
CHAR[] LEFT_STRING (CHAR STRING[], LONG Count)
```
```
WIDECHAR[] LEFT_STRING (WIDECHAR STRING[], LONG Count)
```
Parameters:  
- STRING: The string from which to extract the characters.  
- Count: The number of character to copy from the beginning of the string.  
The result is a string containing a copy of the first Count characters from STRING.  
```
STRING = 'ABCDEFG'Substr = LEFT_STRING('ABCDEFG', 3)
// Substr = 'ABC'
```
### LENGTH_STRING
This function returns the length of a CHAR or WIDECHAR string. This function is retained for compatibility with previous versions of Axcess and provides the same information as LENGTH_ARRAY.

```
LONG LENGTH_STRING (CHAR STRING[ ])  
LONG LENGTH_STRING (WIDECHAR STRING[ ])  
```

**Parameters:**
- STRING: The input character string.

The result is the length of STRING. The string length can be set implicitly through a literal or variable string assignment or explicitly by calling SET_LENGTH_STRING.

**For example:**
```
IF (LENGTH_STRING(STRING) > 0)  
{  
  // process string  
}
```

### LOWER_STRING
This function changes all alphabetic characters in the specified string to lower case.

```
CHAR[ ] LOWER_STRING (CHAR STRING[ ])  
WIDECHAR[ ] LOWER_STRING (WIDECHAR STRING[ ])  
```

**Parameters:**
- STRING: The character string to convert to lower case.

The result is the converted character string.
```
LCString = LOWER_STRING(STRING)  
```

### MAX_LENGTH_STRING
This function returns the dimensioned length of a CHAR or WIDECHAR string. This function is retained for compatibility with previous versions of Axcess. It provides the same information as MAX_LENGTH_ARRAY (see page 32).

```
LONG MAX_LENGTH_STRING (CHAR STRING[ ])  
LONG MAX_LENGTH_STRING (WIDECHAR STRING[ ])  
```

**Parameters:**
- STRING: The input character string.

**Result:** The dimensioned length of STRING.
```
MaxLen = MAX_LENGTH_STRING(STRING)  
Len = LENGTH_STRING(STRING)  
IF (MaxLen > Len)  
{  
  // append character to STRING  
}
```

### MID_STRING
This function returns the specified number of characters, starting at the specified location in the source string.

```
CHAR[ ] MID_STRING (CHAR STRING, LONG Start, LONG Count)  
WIDECHAR[ ] MID_STRING (WIDECHAR STRING, LONG Start, LONG Count)  
```

**Parameters:**
- STRING: The input character string.  
- Start: Starting location in the string.  
- Count: Number of characters to extract.

The result is a character string containing the specified characters.
```
STRING = 'ABCDEFGHIJK'  
Substr = MID_STRING(STRING, 5, 4)  
(* Substr = 'EFGH' *)  
```

### REDIRECT_STRING
This keyword is used to pass all strings from device 1 to device 2 and all strings from device 2 to device 1. This is called a redirection and you can assign up to eight at one time.

```
REDIRECT_STRING (Number, DEV1, DEV2)  
```

The parameter Number identifies the particular redirection (1-8). To cancel a redirection, pass zero for Device1 and Device2.

**NOTE:** Redirections are lost if system power is turned off.

### REMOVE_STRING
This function removes characters from the specified string. All characters up to and including the first occurrence of the specified sequence are removed.

```
CHAR[ ] REMOVE_STRING (CHAR STRING, CHAR Seq[ ], LONG Start)  
WIDECHAR[ ] REMOVE_STRING (WIDECHAR STRING, WIDECHAR Seq[ ], LONG Start)  
```

**Parameters:**
- STRING: String from which to find and remove characters.  
- Seq: Sequence of characters to find.  
- Start: Starting position in the string to begin search.

The result is a string containing the removed characters. If the character sequence was not found, an empty string is returned.
```
STRING = 'ABCDEF'  
Substr = REMOVE_STRING(STRING, 'BC', 1)  
(* Substr = 'ABC' *)  
(* STRING = 'DEF' *)  
```
<table>
<thead>
<tr>
<th>STRING Keywords (Cont.)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RIGHT_STRING</strong></td>
</tr>
</tbody>
</table>
| Returns the specified number of characters from the end of a string.  
  CHAR[ ] RIGHT_STRING (CHAR STRING[ ], LONG Count)  
  WIDECHAR[ ] RIGHT_STRING (WIDECHAR STRING[ ], LONG Count)  
  Parameters:  
  • STRING: The string from which to extract the characters.  
  • Count: The number of character to copy from the end of the string.  
  The return is a string containing a copy of the last Count characters from STRING.  
  STRING = 'ABCDEFG'  
  Substr = RIGHT_STRING(STRING, 3)  // Substr = 'EFG' |
| **SEND_STRING**         |
| This keyword sends a string to a NetLinx device/port.  
  The syntax is:  
  SEND_STRING DEV, '<string>'  
  - or -  
  SEND_STRING DEV[ ], '<string>'  
  When sending to an IP socket, you may receive the following error (via ONERROR event):  
  17 Local Port Not Open  
  This error means you are trying to send a string to a local port on which IP_CLIENT_OPEN (page 98) or IP_SERVER_OPEN (page 99) has not been called. |
| **SET_LENGTH_STRING**   |
| This function sets the length of a CHAR or WIDECHAR string. This function is retained for compatibility with previous versions of Axcess. It provides the same functionality as SET_LENGTH_ARRAY (page 32).  
  SET_LENGTH_STRING (CHAR STRING[ ], LONG Len)  
  SET_LENGTH_STRING (WIDECHAR STRING[ ], LONG Len)  
  Parameters:  
  • STRING: The input character string.  
  • Len: The new string length.  
  SET_LENGTH_STRING(STRING, 10) |
| **STRING**              |
| This keyword defines a section in a DATA event handler for processing SEND_STRING instructions. |
| **STRING_TO_VARIABLE**  |
| See page 70. |
| **UPPER_STRING**        |
| This function changes all alphabetic characters in the specified string to upper case.  
  The syntax:  
  CHAR[ ]UPPER_STRING (CHAR STRING[ ])  
  WIDECHAR[ ]UPPER_STRING (WIDECHAR STRING[ ])  
  Parameters:  
  • STRING: The character string to convert to upper case.  
  Result: The converted character string.  
  UCString = UPPER_STRING(STRING) |
| **VARIABLE_TO_STRING**  |
| See page 53. |
Structure Keywords

Overview

Structures group different data types together as one data unit. Structures also group arrays of structures together so that each element of the array contains all of the elements of the structure.

NOTE: Arrays are limited by their inability to have multiple data-types within one array. NetLinx supports Structures to remove this limitation. Refer to the Array Keywords on page 30 for information on Arrays.

Structures are defined within the DEFINE_TYPE section. The DEFINE_TYPE section appears between the DEFINE_CONSTANT section and the DEFINE_VARIABLE section.

NOTE: Since structures cannot be used within the DEFINE_CONSTANT section but must be declared before they are used within the DEFINE_VARIABLE section, placing DEFINE_TYPE between DEFINE_CONSTANT and DEFINE_VARIABLE is the logical location.

The standard format for structures is:

```
STRUCTURE <name>
{
    [<type>] <data1>
    [<type>] <data2>
    ...
}
```

Example:

```
DEFINE_TYPE
STRUCTURE NEWSTRUCT
{
    INTEGER Number
    CHAR    Text[20]
}
```

In the example above, a structure named NEWSTRUCT is declared to contain two data types, a 16-bit number and a 20-character array. Once declared, a structure may be used in the same way as any other data type. Here is a syntax sample:

```
DEFINE_VARIABLE
NEWSTRUCT MyNewStruct
NEWSTRUCT MyNewStructArray[3]
```

Structures can be initialized using set notation as in the two examples below. Notice that the members of each structure, as well as the entire array, are enclosed in braces:

```
MyNewStruct.Number = 0
MyNewStruct.Text = 'Copyright by Company X'

MyNewStructArray[1].Number = 1
MyNewStructArray[1].Text = 'Line 1'
MyNewStructArray[2].Number = 2
MyNewStructArray[2].Text = 'Line 2'
MyNewStructArray[3].Number = 3
MyNewStructArray[3].Text = 'Line 3'
```

Structure members are referenced using dot-operator syntax as shown below:

```
MyNewStruct.Number = 0
MyNewStructArray[1].Number = 20
SET_LENGTH_STRING (MyNewStruct.Text, 16)
```

Example - Using Structures to Define a Database Table

A database table is an array of structures; the database table is an array of records - each record is a structure. Each record contains data of different types.

Consider the elements of a database table. We then show how to define the structure and create a variable that uses the data structure in an array. We show how to access the individual elements of the structure.

```
Employee Number            (* INDEX - Integer Value *)
Employee National Insurance Number (* National Insurance Number - Long *)
Employee First Name        (* First Name - Character Array *)
Employee Last Name          (* Last Name - Character Array *)
Contribution to Pension    (* Contribution in % - Float *)
```

Using the standard format shown above, the 'employee' structure is defined in the DEFINE_TYPE section:

```
DEFINE_TYPE
STRUCTURE EMP
{
    INTEGER EMP_NUM
    CHAR NI_NUM[9]
    CHAR F_NAME[16]
    CHAR L_NAME[16]
    FLOAT CONT_PENSION
}
```
Within the DEFINE_VARIABLE section, an instance of the structure and an array of the structure is defined as follows:

```
DEFINE_VARIABLE
EMP JOHN_DOE
EMP AMX_EMP[1000]
```

Within the program, information is assigned to the structure, using the information stored within the structure:

```
JOHN_DOE.EMP_NUM = 101
JOHN_DOE.NI_NUM = '155426367'
JOHN_DOE.F_NAME = 'JOHN'
JOHN_DOE.L_NAME = 'DOE'
JOHN_DOE.CONT_PENSION = 0.01

EMP_INDEX = JOHNDOE.EMP_NUM (* EMP_INDEX = 101 *)
AMX_EMP[101] = JOHNDOE (* AMX_EMP[101] = {101, '155426367', 'JOHN', 'DOE', 0.01}*)
AMX_EMP[60].EMP_NUM = 60
AMX_EMP[60].F_NAME = 'BOB'
```

**NOTE:** Other uses for arrays of structures include channel listings, speed-dial lists, and user password lists.

### Data Sets

NetLinx predefines several structures designed to work with NetLinx device numbers, channels, and levels. Data sets allow you to group and combine certain elements of NetLinx devices. There are three data set structures supported by NetLinx:

- **DEV** (Device Sets)
- **DEVCHAN** (Device-Channel Sets)
- **DEVLEV** (Device-Level Sets)

You have already seen the structure DEV structure in the DEFINE_DEVICE section. If we were to define the structure DEV in the DEFINE_TYPE section, it would look like this:

```
STRUCTURE DEV
{
    INTEGER DEVICE
    INTEGER PORT
    INTEGER SYSTEM
}
```

The actual instancing of the structure is unique to the DEV structure because you separate the individual structure’s elements with colons (:) instead of enclosing the structure with braces {} and separating the elements with commas (,). For example:

```
DEV PANEL_A = 128:1:0       (* correct *)
DEV PANEL_B = {128, 1, 0}   (* wrong *)
```

Using the DEV structure, you create the structures DEVCHAN and DEVLEV like this:

```
STRUCTURE DEVCHAN
{
    DEV DEVICE
    INTEGER CHANNEL
}
```

```
STRUCTURE DEVLEV
{
    DEV DEVICE
    INTEGER LEVEL
}
```

DEVCHAN and DEVLEV instance and initialize similarly to other NetLinx structures:

```
DEV PANEL_A = 128:1:0
DEV PANEL_B = 129:1:0
DEVCHAN BUTTON_A = { PANEL_A, 1 }
DEVCHAN BUTTON_B = { 128:1:0, 2 }
DEVLEV LEVEL_1 = { PANEL_A, 1 }
DEVLEV LEVEL_2 = { 128:1:0, 2 }
```

**DEV, DEVCHAN, and DEVLEV** are structures built into the NetLinx language. You can do more with DEV, DEVCHAN, and DEVLEV than you could with structures you create within the code.

```
DEV PANEL_GROUP1[] = { 128:1:0, 129:1:0, 130:1:0 }
DEV MSP_GROUP[5] = { MSP1, MSP2, MSP3 }
DEVCHAN PRESET1_BUTTONS[S] = { (TP1, 21), (MSP1, 1), {134:1:0, 1} }
DEVLEV VOL1_LEVEL[1] = { (TP1, 1), {MSP1, 1}, (192:1:0, 1) }
```

You can use the structures and arrays of the structures within many commands and situations where you would use a device number, a device and channel combination, or a device and level combination. These data sets allow you to combine devices, devices and channels, and devices and levels without using the DEFINE_COMBINE or DEFINE_CONNECT_LEVEL sections. This gives you the ability to combine certain pages of panels or to combine panels under certain conditions. In Axcess, once the panels were combined you were locked into that system configuration. Instead of writing the following statements:

```
PUSH[MSP1, 1]
PUSH[MSP2, 1]
PUSH[MSP3, 1]
[RELAY, 1] = ![RELAY, 1]
[MSP1, 1] = [RELAY, 1]
[MSP2, 1] = [RELAY, 1]
[MSP3, 1] = [RELAY, 1]
```
You can use device sets or channel sets to accomplish the same functionality:

```
PUSH[MSP_GROUP,1]                    (* MSP_GROUP IS A DEV SET *)
    [RELAY, 1] = ![RELAY, 1]
    [MSP_GROUP, 1] = [RELAY, 1]
- or -
PUSH[MSP_PRESET1]                    (* MSP_PRESET1 IS A DEVCHAN SET *)
    [RELAY,1] = ![RELAY, 1]
    [MSP_PRESET1] = [RELAY, 1]
```

## STRUCTURE Keywords

NetLinx supports the following STRUCTURE keywords:

<table>
<thead>
<tr>
<th>STRUCTURE Keywords</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DEFINE_TYPE</strong></td>
<td>See page 63.</td>
</tr>
<tr>
<td><strong>STRUCT</strong></td>
<td>This is an abbreviated form of the STRUTURE keyword, and has the same functionality.</td>
</tr>
</tbody>
</table>
| **STRUCTURE**      | This keyword introduces the declaration of a STRUCTURE data type. 
|                    | STRUCTURE <name> 
|                    | {'data1'} 
|                    | {'data2'} 
|                    | 

```
NetLinx supports the following Terminal keywords:

<table>
<thead>
<tr>
<th>Terminal Keywords</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SSH_CLIENT_CLOSE</strong></td>
<td>Closes an open SSH communication port with a server.</td>
<td><code>slong SSH_CLIENT_CLOSE(INTEGER LocalPort)</code></td>
</tr>
<tr>
<td><strong>Parameters:</strong></td>
<td></td>
<td><strong>Parameters:</strong></td>
</tr>
<tr>
<td>• LocalPort - A user-defined (non-zero) integer value representing the local port on the client machine to use for this conversation. This local port number must be passed to SSH_CLIENT_OPEN to open the conversation.</td>
<td>• LocalPort - A user-defined (non-zero) integer value representing the local port on the client machine to use for this conversation. This local port number must be passed to SSH_CLIENT_OPEN to open the conversation.</td>
<td></td>
</tr>
<tr>
<td><strong>Returns:</strong></td>
<td></td>
<td><strong>Returns:</strong></td>
</tr>
<tr>
<td>This function always returns 0. Errors are returned via the DATA_EVENT ONERROR method. The following errors may be returned from the call:</td>
<td>This function always returns 0. Errors are returned via the DATA_EVENT ONERROR method. The following errors may be returned from the call:</td>
<td></td>
</tr>
<tr>
<td>2 - General failure (out of memory)</td>
<td>2 - General failure (out of memory)</td>
<td></td>
</tr>
<tr>
<td>4 - Unknown host</td>
<td>4 - Unknown host</td>
<td></td>
</tr>
<tr>
<td>6 - Connection refused</td>
<td>6 - Connection refused</td>
<td></td>
</tr>
<tr>
<td>7 - Connection timed out</td>
<td>7 - Connection timed out</td>
<td></td>
</tr>
<tr>
<td>8 - Unknown connection error</td>
<td>8 - Unknown connection error</td>
<td></td>
</tr>
<tr>
<td>9 - Already closed</td>
<td>9 - Already closed</td>
<td></td>
</tr>
<tr>
<td>14 - Local port already used</td>
<td>14 - Local port already used</td>
<td></td>
</tr>
<tr>
<td>16 - Too many open sockets</td>
<td>16 - Too many open sockets</td>
<td></td>
</tr>
<tr>
<td><strong>Example:</strong></td>
<td><code>SSH_CLIENT_CLOSE(5000)</code></td>
<td><strong>Example:</strong></td>
</tr>
<tr>
<td><strong>SSH_CLIENT_OPEN</strong></td>
<td>Opens a port for SSH communication with a server.</td>
<td><code>slong SSH_CLIENT_OPEN(INTEGER LocalPort, CHAR ServerAddress[], INTEGER remotePort, CHAR username[], char password[], char privateKeyPathname[], char privateKeyPassphrase[])</code></td>
</tr>
<tr>
<td><strong>Parameters:</strong></td>
<td></td>
<td><strong>Parameters:</strong></td>
</tr>
<tr>
<td>• LocalPort - A user-defined (non-zero) integer value representing the local port on the client machine to use for this conversation. This local port number must be passed to SSH_CLIENT_CLOSE to close the conversation.</td>
<td>• LocalPort - A user-defined (non-zero) integer value representing the local port on the client machine to use for this conversation. This local port number must be passed to SSH_CLIENT_CLOSE to close the conversation.</td>
<td></td>
</tr>
<tr>
<td>• ServerAddress - A string containing either the IP address (in dotted-quad-notation) or the domain name of the server to which you want to connect.</td>
<td>• ServerAddress - A string containing either the IP address (in dotted-quad-notation) or the domain name of the server to which you want to connect.</td>
<td></td>
</tr>
<tr>
<td>• remotePort - The port number on the server that identifies the program or service that the client is requesting, typically 22</td>
<td>• remotePort - The port number on the server that identifies the program or service that the client is requesting, typically 22</td>
<td></td>
</tr>
<tr>
<td>• username - Login user name</td>
<td>• username - Login user name</td>
<td></td>
</tr>
<tr>
<td>• password - Password for the user name, null if using PKI</td>
<td>• password - Password for the user name, null if using PKI</td>
<td></td>
</tr>
<tr>
<td>• privateKeyPathname - Path to private key</td>
<td>• privateKeyPathname - Path to private key</td>
<td></td>
</tr>
<tr>
<td>• privateKeyPassphrase - Password for private key.</td>
<td>• privateKeyPassphrase - Password for private key.</td>
<td></td>
</tr>
<tr>
<td><strong>Returns:</strong></td>
<td></td>
<td><strong>Returns:</strong></td>
</tr>
<tr>
<td>This function always returns 0. Errors are returned via the DATA_EVENT ONERROR method. The following errors may be returned from the call:</td>
<td>This function always returns 0. Errors are returned via the DATA_EVENT ONERROR method. The following errors may be returned from the call:</td>
<td></td>
</tr>
<tr>
<td>2 - General failure (out of memory)</td>
<td>2 - General failure (out of memory)</td>
<td></td>
</tr>
<tr>
<td>4 - Unknown host</td>
<td>4 - Unknown host</td>
<td></td>
</tr>
<tr>
<td>6 - Connection refused</td>
<td>6 - Connection refused</td>
<td></td>
</tr>
<tr>
<td>7 - Connection timed out</td>
<td>7 - Connection timed out</td>
<td></td>
</tr>
<tr>
<td>8 - Unknown connection error</td>
<td>8 - Unknown connection error</td>
<td></td>
</tr>
<tr>
<td>9 - Already closed</td>
<td>9 - Already closed</td>
<td></td>
</tr>
<tr>
<td>14 - Local port already used</td>
<td>14 - Local port already used</td>
<td></td>
</tr>
<tr>
<td>16 - Too many open sockets</td>
<td>16 - Too many open sockets</td>
<td></td>
</tr>
<tr>
<td><strong>Example:</strong></td>
<td><code>SSH_CLIENT_OPEN(5000, '192.168.0.1', 22, 'user1', 'password', '/certs/id_rsa', '')</code></td>
<td><strong>Example:</strong></td>
</tr>
</tbody>
</table>

Example:

```
SSH_CLIENT_CLOSE(5000)
```
# Time and Date Keywords

NetLinx supports the following Time & Date keywords:

<table>
<thead>
<tr>
<th>Time and Date Keywords</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ASTRO_CLOCK</strong></td>
<td>This routine calculates the time of sunset and sunrise at a specified location (longitude and latitude) on a specified date.</td>
</tr>
<tr>
<td>Parameters:</td>
<td></td>
</tr>
<tr>
<td>• Longitude: Longitude in Degrees. Fraction of Degrees. West longitudes must be negative.</td>
<td></td>
</tr>
<tr>
<td>• Latitude: Latitude in Degrees. Fraction of Degrees. South latitudes must be negative.</td>
<td></td>
</tr>
<tr>
<td>• HoursFromGMT: Number of hours from GMT. Hours West of GMT can be entered as negative (e.g., -5 for EST, -4 for EDT).</td>
<td></td>
</tr>
<tr>
<td>• Date: In mm/dd/yyyy format.</td>
<td></td>
</tr>
<tr>
<td>• Sunrise: Value gets filled in by the function in 24-hour format.</td>
<td></td>
</tr>
<tr>
<td>• Sunset: Value gets filled in by the function in mm/dd/yyyy format.</td>
<td></td>
</tr>
<tr>
<td>Result:</td>
<td></td>
</tr>
<tr>
<td>0: Success</td>
<td></td>
</tr>
<tr>
<td>-1: Latitude entry error</td>
<td></td>
</tr>
<tr>
<td>-2: Longitude entry error</td>
<td></td>
</tr>
<tr>
<td>-3: Hours entry error</td>
<td></td>
</tr>
<tr>
<td>-4: Date entry error</td>
<td></td>
</tr>
<tr>
<td><strong>CLOCK</strong></td>
<td>Sets the date and time on the Master. The date and time settings are propagated over the local bus.</td>
</tr>
<tr>
<td>Example:</td>
<td></td>
</tr>
<tr>
<td>SEND_COMMAND 0,&quot;'CLOCK 04-12-05 09:45:31'&quot;</td>
<td></td>
</tr>
<tr>
<td><strong>DATE</strong></td>
<td>The system variable DATE returns the current date in (mm/dd/yy) string format. The wildcard character '?' is not allowed for string comparisons because the actual date is needed.</td>
</tr>
<tr>
<td>Example:</td>
<td></td>
</tr>
<tr>
<td>IF (DATE = '12/25/00')</td>
<td></td>
</tr>
<tr>
<td>You can replace the wildcard feature by using the COMPARE_STRING function.</td>
<td></td>
</tr>
<tr>
<td><strong>DATE_TO_DAY</strong></td>
<td>This function returns an sinteger representing the day portion of a date string. The S in SINTEGER allows a negative value to be returned.</td>
</tr>
<tr>
<td>Parameters:</td>
<td></td>
</tr>
<tr>
<td>• LDATE: [Input] string containing the date in mm/dd/yyyy format.</td>
<td></td>
</tr>
<tr>
<td>If successful, this function returns an integer (1-31) representing the day portion of the date string. If the specified date is invalid, this function returns -1.</td>
<td></td>
</tr>
<tr>
<td>SINTEGER nDay = DATE_TO_DAY ('2/9/1999') // nDay = 9</td>
<td></td>
</tr>
<tr>
<td><strong>DATE_TO_MONTH</strong></td>
<td>This function returns an sinteger representing the month portion of a date string.</td>
</tr>
<tr>
<td>Parameters:</td>
<td></td>
</tr>
<tr>
<td>• LDATE: [Input] string containing the date in mm/dd/yyyy format.</td>
<td></td>
</tr>
<tr>
<td>If successful, this function returns an integer (1-12) representing the month portion of the date string. If the specified date is invalid, this function returns -1.</td>
<td></td>
</tr>
<tr>
<td>SINTEGER nMonth = DATE_TO_MONTH ('2/9/1999') // nMonth = 2</td>
<td></td>
</tr>
<tr>
<td><strong>DATE_TO_YEAR</strong></td>
<td>This function returns an sinteger representing the year portion of a date string.</td>
</tr>
<tr>
<td>Parameters:</td>
<td></td>
</tr>
<tr>
<td>• LDATE: [Input] string containing the date in mm/dd/yyyy format.</td>
<td></td>
</tr>
<tr>
<td>If successful, this function returns a 4-digit integer representing the year portion of the date string. If the specified date is invalid, this function returns -1.</td>
<td></td>
</tr>
<tr>
<td>SINTEGER nYear = DATE_TO_YEAR ('2/9/1999') // nYear = 1999</td>
<td></td>
</tr>
<tr>
<td><strong>DAY</strong></td>
<td>This system variable returns the current day of the week as one of the following strings: 'MON', 'TUE', 'WED', 'THU', 'FRI', 'SAT' or 'SUN'.</td>
</tr>
<tr>
<td>Example:</td>
<td></td>
</tr>
<tr>
<td>IF (DAY = 'SUN')</td>
<td></td>
</tr>
<tr>
<td><strong>DAY_OF_WEEK</strong></td>
<td>This function returns the day of the week for the specified date.</td>
</tr>
<tr>
<td>Parameters:</td>
<td></td>
</tr>
<tr>
<td>• LDATE: String containing the date in mm/dd/yyyy format.</td>
<td></td>
</tr>
<tr>
<td>This function returns an SINTEGER representing the day of the week (1 = Sunday, 2 = Monday, etc.).</td>
<td></td>
</tr>
<tr>
<td>SINTEGER nDay = DAY_OF_WEEK ('2/13/1999') // nDay = 7 (Saturday)</td>
<td></td>
</tr>
<tr>
<td><strong>LDATE</strong></td>
<td>This system variable returns the current date in (mm/dd/yyyy) string format.</td>
</tr>
<tr>
<td>Example:</td>
<td></td>
</tr>
<tr>
<td>IF (LDATE = '12/25/2000')</td>
<td></td>
</tr>
</tbody>
</table>

---

**Time and Date Keywords**

ASTRO_CLOCK

This routine calculates the time of sunset and sunrise at a specified location (longitude and latitude) on a specified date.

```c
SINTEGER ASTRO_CLOCK(DOUBLE Longitude,DOUBLE Latitude,DOUBLE HoursFromGMT,CHAR Date,CHAR Sunrise,CHAR Sunset)
```

**Parameters:**

- **Longitude:** Longitude in Degrees. Fraction of Degrees. West longitudes must be negative.
- **Latitude:** Latitude in Degrees. Fraction of Degrees. South latitudes must be negative.
- **HoursFromGMT:** Number of hours from GMT. Hours West of GMT can be entered as negative (e.g., -5 for EST, -4 for EDT).
- **Date:** In mm/dd/yyyy format.
- **Sunrise:** Value gets filled in by the function in 24-hour format.
- **Sunset:** Value gets filled in by the function in mm/dd/yyyy format.

**Result:**

- 0: Success
- -1: Latitude entry error
- -2: Longitude entry error
- -3: Hours entry error
- -4: Date entry error

CLOCK

Sets the date and time on the Master. The date and time settings are propagated over the local bus.

Example:

```c
SEND_COMMAND 0,"'CLOCK 04-12-05 09:45:31'"
```

DATE

The system variable DATE returns the current date in (mm/dd/yy) string format. The wildcard character '?' is not allowed for string comparisons because the actual date is needed.

Example:

```c
IF (DATE = '12/25/00')
{
}
```

You can replace the wildcard feature by using the COMPARE_STRING function.

DATE_TO_DAY

This function returns an sinteger representing the day portion of a date string. The S in SINTEGER allows a negative value to be returned.

```c
SINTEGER DATE_TO_DAY (CHAR LDATE[])
```

**Parameters:**

- **LDATE:** [Input] string containing the date in mm/dd/yyyy format.

If successful, this function returns an integer (1-31) representing the day portion of the date string. If the specified date is invalid, this function returns -1.

```c
SINTEGER nDay = DATE_TO_DAY ('2/9/1999') // nDay = 9
```

DATE_TO_MONTH

This function returns an sinteger representing the month portion of a date string.

```c
SINTEGER DATE_TO_MONTH (CHAR LDATE[])
```

**Parameters:**

- **LDATE:** [Input] string containing the date in mm/dd/yyyy format.

If successful, this function returns an integer (1-12) representing the month portion of the date string. If the specified date is invalid, this function returns -1.

```c
SINTEGER nMonth = DATE_TO_MONTH ('2/9/1999') // nMonth = 2
```

DATE_TO_YEAR

This function returns an sinteger representing the year portion of a date string.

```c
SINTEGER DATE_TO_YEAR (CHAR LDATE[])
```

**Parameters:**

- **LDATE:** [Input] string containing the date in mm/dd/yyyy format.

If successful, this function returns a 4-digit integer representing the year portion of the date string. If the specified date is invalid, this function returns -1.

```c
SINTEGER nYear = DATE_TO_YEAR ('2/9/1999') // nYear = 1999
```

DAY

This system variable returns the current day of the week as one of the following strings: 'MON', 'TUE', 'WED', 'THU', 'FRI', 'SAT' or 'SUN'.

Example:

```c
IF (DAY = 'SUN')
{
}
```

DAY_OF_WEEK

This function returns the day of the week for the specified date.

```c
SINTEGER DAY_OF_WEEK (CHAR LDATE[])
```

**Parameters:**

- **LDATE:** String containing the date in mm/dd/yyyy format.

This function returns an SINTEGER representing the day of the week (1 = Sunday, 2 = Monday, etc.).

```c
SINTEGER nDay = DAY_OF_WEEK ('2/13/1999') // nDay = 7 (Saturday)
```
### Time and Date Keywords (Cont.)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
</table>
| **TIME**        | This keyword holds the current time as a string in the form “hh:mm:ss”. The time is represented in 24-hour format. | ```
IF (TIME = '23:59:59')
{
}
``` |
| **TIME_TO_HOUR**| This function returns an integer representing the hour portion of a time string.                   | ```
SINTEGER TIME_TO_HOUR (CHAR TimeStr[
 Parameters:  
• TimeStr: Input string containing the time in hh:mm:ss format.  
If successful, this function returns an integer (0-23) representing the hour portion of the time string. The specified time is invalid, this function returns -1.  
CHAR TimeStr[] = '9:30:08'  
SINTEGER nHour  
nHour = TIME_TO_HOUR (TimeStr) // nHour = 9
``` |
| **TIME_TO_MINUTE**| This function returns an integer representing the minute portion of a time string.                 | ```
SINTEGER TIME_TO_MINUTE (CHAR TimeStr[
 Parameters:  
• TimeStr: Input string containing the time in hh:mm:ss format.  
If successful, this function returns an integer (0-59) representing the minute portion of the time string. If the specified time is invalid, this function returns -1.  
CHAR TimeStr[] = '9:30:08'  
SINTEGER nMinute  
nMinute = TIME_TO_MINUTE (TimeStr) // nMinute = 30
``` |
| **TIME_TO_SECOND**| This function returns an integer representing the second portion of a time string.                | ```
SINTEGER TIME_TO_SECOND (CHAR TimeStr[
 Parameters:  
• TimeStr: Input string containing the time in hh:mm:ss format.  
If successful, this function returns an integer (0-59) representing the second portion of the time string. If the specified time is invalid, this function returns -1.  
CHAR TimeStr[] = '9:30:08'  
SINTEGER nSecond  
nSecond = TIME_TO_SECOND (TimeStr) // nSecond = 8
``` |
Timeline Keywords

Overview

NetLinx timeline functions provide a mechanism for triggering events based upon a sequence of times. The sequence of times is passed into the timeline functions as an array of `LONG` values, with each value representing a time period (in milliseconds) that is either relative to the start time of the timeline or to the previously triggered event.

**NOTE:** Timelines introduce the capability to dynamically set up a timed sequence, provide the user with a mechanism to modify the sequence, and allow the user to create, delete, and modify sequences.

The old way of programming timed sequences was to cascade or nest WAITs. Using nested WAITs hard-coded the timed sequence; so, the only way to modify the timing was to modify the NetLinx program, recompile, and download. Timelines make adding, deleting and editing the sequence much simpler for the programmer. Timeline functions and debugging allow the timings to be modified without the modify/compile/download cycle because the array of times may be modified via NetLinx debugging. Once the timings have been tweaked, the changes can be incorporated in the NetLinx program.

Creating a Timeline

Timelines are represented by the illustration in (FIG. 4). When the TIMELINE_CREATE function is executed, the timeline starts at zero and begins counting. When the timer value equals a value in the TIMES array, a TIMELINE_EVENT is triggered. Within the timeline event, a TIMELINE structure is available to get information about the specific time from the TIMES array that generated the event. When a relative timeline is created, the NetLinx Master converts the provided relative times into absolute times that are stored internally.

The TIMELINE structure contains the following members:

```c
STRUCTURE TIMELINE {
    INTEGER ID //user supplied ID
    INTEGER SEQUENCE //index in Times array
    LONG TIME //time since start of timeline
    INTEGER RELATIVE //0=absolute 1=relative
    LONG REPETITION //# of loops for repeating timeline
}
```

Each TIMELINE data member is defined as follows:

- **ID:** The ID that the user assigned to the timeline in the TIMELINE_CREATE function.
- **SEQUENCE:** The index of the time in the Times array that was passed to the TIMELINE_CREATE function. The SEQUENCE data member is used to determine what action to take for the event and is normally decoded with a SWITCH/CASE structure (as shown in the example).
- **TIME:** The amount of time that has elapsed since the timeline started. For repeating timelines, the TIME and REPETITION data members can be used to calculate the total amount of time it has been running.
- **RELATIVE:** If the timeline is operating in relative mode, this data member is equal to TIMELINE_RELATIVE. If the timeline is absolute, it is equal to TIMELINE_ABSOLUTE.
- **REPETITION:** If the timeline was created with TIMELINE_REPEAT, this data member holds the number of times the timeline has been executed. REPETITION contains zero for the first pass through the timeline. Thus, the calculation to determine the total amount of time the timeline has been running is simply: `TIMELINE.TIME * TIMELINE.REPETITION`.

**Return Values:**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful</td>
</tr>
<tr>
<td>1</td>
<td>Timeline ID already in use</td>
</tr>
<tr>
<td>2</td>
<td>Specified array is not an array of LONGs</td>
</tr>
<tr>
<td>3</td>
<td>Specified length is greater than the length of the passed array</td>
</tr>
<tr>
<td>4</td>
<td>Out of memory</td>
</tr>
</tbody>
</table>
Example:

DEFINE_VARIABLE
LONG TimeArray[100]

DEFINE_CONSTANT
TL1 = 1
TL2 = 2

DEFINE_EVENT
TIMELINE_EVENT[TL1] // capture all events for Timeline 1
{
  switch(Timeline.Sequence) // which time was it?
  {
    case 1: { SEND_COMMAND dvPanel,"'TEXT1-1 1'" }
    case 2: { SEND_COMMAND dvPanel,"'TEXT1-1 2'" }
    case 3: { SEND_COMMAND dvPanel,"'TEXT1-1 3'" }
    case 4: { SEND_COMMAND dvPanel,"'TEXT1-1 4'" }
    case 5: { SEND_COMMAND dvPanel,"'TEXT1-1 5'" }
  }
}

TIMELINE_EVENT[TL2]
{
  switch(Timeline.Sequence)
  {
    case 1: { SEND_COMMAND dvPanel,"'TEXT2-2 1'" }
    case 2: { SEND_COMMAND dvPanel,"'TEXT2-2 2'" }
    case 3: { SEND_COMMAND dvPanel,"'TEXT2-2 3'" }
    case 4: { SEND_COMMAND dvPanel,"'TEXT2-2 4'" }
    case 5: { SEND_COMMAND dvPanel,"'TEXT2-2 5'" }
  }
}

DEFINE_PROGRAM
PUSH[dvPanel,1]
{
  TimeArray[1] = 1000
  TimeArray[3] = 3000
  TimeArray[4] = 4000
  TimeArray[5] = 5000
  TIMELINE_CREATE(TL1, TimeArray, 5, TIMELINE_ABSOLUTE, TIMELINE_REPEAT)
}

PUSH[dvPanel,2]
{
  TimeArray[1] = 1000
  TimeArray[2] = 1000
  TimeArray[3] = 1000
  TimeArray[4] = 1000
  TimeArray[5] = 1000
  TIMELINE_CREATE(TL2, TimeArray, 5, TIMELINE_RELATIVE, TIMELINE_ONCE)
}

The example above creates two timelines (TL1 and TL2) that trigger events at the same rate (once per second).

- TL1 uses TIMELINE_ABSOLUTE to specify that the times in TimeArray are absolute with respect to the start of the timeline. Since TL1 specifies the TIMELINE_REPEAT, it is also repeating and will generate a TIMELINE_EVENT every second iterating through all five times in a round-robin fashion: 1,2,3,4,5,1,2,3,4,5,1,2,3, and so on.
- TL2 uses TIMELINE_RELATIVE to specify that the times in TimeArray are relative to each other (i.e. each event occurs 1000 milliseconds after the previous). Since TL2 specifies the TIMELINE_ONCE parameter, it will execute the entire timeline once, then stop: 1,2,3,4,5.

**TIMELINE Example**

The following code is an example of how to use TIMELINE functions.

PROGRAM_NAME='TimelineExample'
(*{{PS_SOURCE_INFO(PROGRAM STATS)                          *)
  (***********************************************************)
  (*  FILE CREATED ON: 05/22/2001 AT: 12:05:56               *)
  (***********************************************************)
  (*  FILE_LAST_MODIFIED_ON: 05/22/2001 AT: 12:15:56         *)
  (***********************************************************)
  (*  ORPHAN_FILE_PLATFORM: 1                                *)
  (***********************************************************)
DEFINE_DEVICE
dvPanel    = 128:1:0
dvDebug    = 0:0:0

DEFINE_CONSTANT
MY_LINE_1 = 1
MY_LINE_2 = 2

DEFINE_VARIABLE
LONG TimeArray[100]
INTEGER iLoop

DEFINE_START

DEFINE_EVENT
TIMELINE_EVENT[MY_LINE_1]
{
    switch(Timeline.Sequence)
    {
        case 1: { SEND_COMMAND dvPanel,"'TEXT1-1 1'" }
        case 2: { SEND_COMMAND dvPanel,"'TEXT1-1 2'" }
        case 3: { SEND_COMMAND dvPanel,"'TEXT1-1 3'" }
        case 4: { SEND_COMMAND dvPanel,"'TEXT1-1 4'" }
        case 5: { SEND_COMMAND dvPanel,"'TEXT1-1 5'" }
    }
    SEND_STRING dvDebug,"'Timer ',ITOA(Timeline.ID),' Event ',ITOA(Timeline.Sequence),
        ' Time = ',ITOA(Timeline.Time),
        ' Repetition = ',ITOA(Timeline.Repetition),
        ' Relative = ',ITOA(Timeline.Relative)"
}
TIMELINE_EVENT[MY_LINE_2]
{
    switch(Timeline.Sequence)
    {
        case 1: { SEND_COMMAND dvPanel,"'TEXT2-2 1'" }
        case 2: { SEND_COMMAND dvPanel,"'TEXT2-2 2'" }
        case 3: { SEND_COMMAND dvPanel,"'TEXT2-2 3'" }
        case 4: { SEND_COMMAND dvPanel,"'TEXT2-2 4'" }
        case 5: { SEND_COMMAND dvPanel,"'TEXT2-2 5'" }
    }
    SEND_STRING dvDebug,"'Timer ',ITOA(Timeline.ID),' Event ',ITOA(Timeline.Sequence),
        ' Time = ',ITOA(Timeline.Time),
        ' Repetition = ',ITOA(Timeline.Repetition),
        ' Relative = ',ITOA(Timeline.Relative)"
}

DEFINE_PROGRAM
(* create will sort the order of the times but index stays *)
(* with the time. This example will execute 1 2 4 3 5 *)
(* sequence numbers *)

PUSH[dvPanel,1]
{
  TimeArray[1] = 1000
  TimeArray[4] = 3000
  TimeArray[3] = 4000
  TimeArray[5] = 5000
  TIMELINE_CREATE(MY_LINE_1,TimeArray,5,TIMELINE_ABSOLUTE,TIMELINE_ONCE)
}

PUSH[dvPanel,2]
{
  TimeArray[1] = 1000
  TimeArray[3] = 3000
  TimeArray[4] = 4000
  TimeArray[5] = 5000
  TIMELINE_CREATE(MY_LINE_2,TimeArray,5,TIMELINE_ABSOLUTE,TIMELINE_REPEAT)
}

(* Modify the timeline my kill, pause and restarting *)

PUSH[dvPanel,3]
{
  IF (TIMELINE_ACTIVE(MY_LINE_1)) TIMELINE_KILL(MY_LINE_1)
  IF (TIMELINE_ACTIVE(MY_LINE_2)) TIMELINE_KILL(MY_LINE_2)
}

PUSH[dvPanel,4]
{
  IF (TIMELINE_ACTIVE(MY_LINE_1)) TIMELINE_PAUSE(MY_LINE_1)
  IF (TIMELINE_ACTIVE(MY_LINE_2)) TIMELINE_PAUSE(MY_LINE_2)
}

PUSH[dvPanel,5]
{
  IF (TIMELINE_ACTIVE(MY_LINE_1)) TIMELINE_RESTART(MY_LINE_1)
  IF (TIMELINE_ACTIVE(MY_LINE_2)) TIMELINE_RESTART(MY_LINE_2)
}

(* Force time to a different value *)

PUSH[dvPanel,6]
{
  IF (TIMELINE_ACTIVE(MY_LINE_1))
    TIMELINE_SET(MY_LINE_1,0)
}

(* Get the current time from create *)

PUSH[dvPanel,7]
{
  SEND_COMMAND dvPanel,"'TEXT3-','Timer 1 Time is ',ITOA(TIMELINE_GET(MY_LINE_1))"
  SEND_COMMAND dvPanel,"'TEXT4-','Timer 2 Time is ',ITOA(TIMELINE_GET(MY_LINE_2))"
}

(* Pause and restart the timeline at new locations *)

PUSH[dvPanel,8]
{
  TIMELINE_PAUSE(MY_LINE_1)
  TIMELINE_PAUSE(MY_LINE_2)
  TIMELINE_SET(MY_LINE_1,0)
  TIMELINE_SET(MY_LINE_2,0)
  TIMELINE_RESTART(MY_LINE_1)
  TIMELINE_RESTART(MY_LINE_2)
}
TIMELINE IDS

When creating a TIMELINE_EVENT, the timeline ID must be a user defined long constant. The compiler will not semantic check the type of the timeline ID, and the NetLinx runtime system will attempt to cast the contents of the timeline ID constant, to a long constant. A runtime error will occur if the cast is unsuccessful.

Here's an example of TIMELINE code:

```c
DEFINE_VARIABLE
CONSTANT LONG TimelineID_1 = 1
CONSTANT LONG TimelineID_2 = 2
CONSTANT LONG TimelineID_3 = 3
CONSTANT LONG TimelineID_4 = 4
LONG TimeArray[4] =
{ 1000, // 1 second
  2000, // 2 seconds
  3000, // 3 seconds
  4000 // 4 seconds
}
DEFINE_START
TIMELINE_CREATE (TimelineID_1, TimeArray, LENGTH_ARRAY(TimeArray), TIMELINE_RELATIVE, TIMELINE_REPEAT)
TIMELINE_CREATE (TimelineID_2, TimeArray, LENGTH_ARRAY(TimeArray), TIMELINE_RELATIVE, TIMELINE_REPEAT)
TIMELINE_CREATE (TimelineID_3, TimeArray, LENGTH_ARRAY(TimeArray), TIMELINE_RELATIVE, TIMELINE_REPEAT)
TIMELINE_CREATE (TimelineID_4, TimeArray, LENGTH_ARRAY(TimeArray), TIMELINE_RELATIVE, TIMELINE_REPEAT)
DEFINE_EVENT
// typical TIMELINE_EVENT statement
TIMELINE_EVENT[TimelineID_1] // capture all events for Timeline 1
{
  SEND_STRING 0, "'TL ID = ', itoa(timeline.id), ', sequence = ',itoa(timeline.sequence)"
}
// example of "stacked" TIMELINE_EVENT statements
TIMELINE_EVENT[TimelineID_2] // capture all events for Timeline 2
TIMELINE_EVENT[TimelineID_3] // capture all events for Timeline 3
TIMELINE_EVENT[TimelineID_4] // capture all events for Timeline 4
{
  SEND_STRING 0, "'TL ID = ', itoa(timeline.id), ', sequence = ',itoa(timeline.sequence)"
}
// end
```

NetLinx supports the following TIMELINE keywords:

### TIMELINE KEYWORDS

<table>
<thead>
<tr>
<th>TIMELINE Keywords</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TIMELINE_ACTIVE</strong></td>
<td>This function is used to determine if a timeline has been created. If the timeline does not exist (i.e. TIMELINE_CREATE has not been called) this function returns zero.</td>
</tr>
<tr>
<td>INTEGER TIMELINE_ACTIVE(LONG Id)</td>
<td></td>
</tr>
<tr>
<td>Parameters:</td>
<td></td>
</tr>
<tr>
<td>• Id: A user defined value that uniquely identifies this timeline. Each timeline must be assigned a unique identifier starting with number one.</td>
<td></td>
</tr>
<tr>
<td>Returns:</td>
<td></td>
</tr>
<tr>
<td>• 0: Not created.</td>
<td></td>
</tr>
<tr>
<td>• Non-zero: The timeline has been created.</td>
<td></td>
</tr>
<tr>
<td>IF(TIMELINE_ACTIVE(TL1)) // if timeline 1 is running</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>// do something</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

| **TIMELINE_CREATE** | Creates an initial timeline and specifies the attributes of the timeline. Time is measured in millisecond (1/1000 of a second) increments. |
| INTEGER TIMELINE_CREATE(LONG Id, LONG Times[], LONG Length, LONG Relative, LONG Repeat) |
| Parameters: |
| • Id: A user defined value that uniquely identifies this timeline. Each timeline must be assigned a unique identifier starting with number one. |
| • Times: An array of times where each time specifies when a TIMELINE_EVENT will be triggered. The times in the array may be relative to each other or relative to the start of the timeline depending upon the Relative parameter. For an absolute timeline, it is not necessary for the times in the array to be sorted in any particular order (the NetLinx master does this internally for you). The NetLinx master makes an internal copy of the values in the array allowing the user to modify the passed in array as desired without affecting the operation of the timeline. |
| • Length: The count of times in the Times array. |
| • Relative: Indicates whether the Times array contains relative times or absolute times. |
| • Relative indicates the each time given is relative to the last event time (i.e. the time delay in between the triggered events). |
| • Absolute indicates that each time given is absolute with respect to the start of the timeline. |
| • Repeat: Indicates whether the timeline should automatically start over again when Length events have been triggered. |

**TIMELINE Keywords (Cont.)**
**TIMELINE_EVENT**

These events are generated when a timeline's internal timer is equal to one of the specified times in the times array. The TIMELINE_EVENT must be placed in the DEFINE_EVENT section of the program.

```
TIMELINE_EVENT[timelineID]
```

See the TIMELINE_CREATE function (above) for a more detailed description.

**TIMELINE_GET**

This function returns the value of the specified timeline's timer. The timer indicates the number of milliseconds that have passed since the timeline started. If the timeline is paused the timer is also paused and subsequent calls to TIMELINE_GET will return the same value.

```
LONG TIMELINE_GET (LONG Id)
```

Parameters:
- `Id`: A user defined value that uniquely identifies this timeline. Each timeline must be assigned a unique identifier starting with number one.

Result: This function returns the specified timeline's internal timer. The timer value represents the number of milliseconds that have passed since the timeline started.

```
TIMELINE_SET (TL1,TIMELINE_GET (TL1)+1000)  // jump ahead 1 second
```

**TIMELINE_KILL**

This function is used to terminate a timeline. Any further references to the specified timeline ID are invalid.

```
INTEGER TIMELINE_KILL(LONG Id)
```

Parameters:
- `Id`: A user defined value that uniquely identifies this timeline. Each timeline must be assigned a unique identifier starting with number one.

Result:
- 0: Successful
- 1: Specified timeline ID invalid

```
TIMELINE_KILL(TL1)   // permanently destroy the timeline
```

**TIMELINE_PAUSE**

This function is used to suspend the execution of a timeline. It may be restarted from where it left off with the TIMELINE_RESTART function.

```
INTEGER TIMELINE_PAUSE(LONG Id)
```

Parameters:
- `Id`: A user defined value that uniquely identifies this timeline. Each timeline must be assigned a unique identifier starting with number one.

Result:
- 0: Successful
- 1: Specified timeline ID invalid

```
TIMELINE_PAUSE(TL1)   // momentarily suspend the timeline
```

**TIMELINE_RELOAD**

This function is used to change the array times of a timeline. The new array of times takes affect immediately even if the timeline is currently executing. If the timeline is executing when this function is called the timeline continues to execute and the next matching time from the new array triggers an event.

```
INTEGER TIMELINE_RELOAD(LONG Id, LONG Times[],LONG Length)
```

Parameters:
- `Id`: A user defined value that uniquely identifies this timeline. Each timeline must be assigned a unique identifier starting with number one.
- `Times`: An array of times where each time specifies when a TIMELINE_EVENT will be triggered. The times in the array must utilize the same time base (TIMELINE_RELATIVE or TIMELINE_ABSOLUTE) as determined by the original call to TIMELINE_CREATE. The NetLinx master makes an internal copy of the values in the array allowing the user to modify the passed in array as desired without affecting the operation of the timeline.
- `Length`: The count of times in the Times array.

Result:
- 0: Successful
- 1: Timeline ID already in use
- 2: Specified array is not an array of LONGs.
- 3: Specified length is greater than the length of the passed array.
- 4: Out of memory

```
TimeArray[1] = 1000
TimeArray[2] = 1500
TIMELINE_RELOAD(TL1,TimeArray,3) // Modify the timeline
```

**TIMELINE_RESTART**

This function is used to continue execution of a timeline that was suspended with TIMELINE_PAUSE.

```
INTEGER TIMELINE_RESTART(LONG Id)
```

Parameters:
- `Id`: A user defined value that uniquely identifies this timeline. Each timeline must be assigned a unique identifier starting with number one.

Result:
- 0: Successful
- 1: Specified timeline ID invalid

```
TIMELINE_RESTART(TL1)   // continue the timeline
```
### TIMELINE_SET

This function is used to modify the current timer value of a timeline. The timeline’s timer is immediately set to the new value regardless of whether the timeline is executing or not.

```
INTEGER TIMELINE_SET (LONG Id, LONG Timer)
```

**Parameters:**
- **Id**: A user defined value that uniquely identifies this timeline. Each timeline must be assigned a unique identifier starting with number one.
- **Timer**: The new value for the timeline’s internal timer.

**Result:**
- 0: Successful
- 1: Specified timeline ID invalid
- 2: Specified timer value out of range

```
TIMELINE_SET (TL1, 0) // start it over again
```
UniCode Keywords

Overview

NetLinx UniCode Functions allow programmers to embed Unicode String literals in their NetLinx programs, manipulate them using run-time functions and send them to touch panels and other user interfaces.

Working With UniCode in NetLinx Studio

NetLinx supports 16-bit Unicode characters. You can type Unicode character literals strings into your program, assign them to variables, manipulate them using string operations, read and write Unicode characters to the file system and send Unicode strings to user interfaces for display.

Configuring NetLinx Studio

Before you begin to work with Unicode, you must enable the UTF-8 Unicode option and the Unicode Compile option in NetLinx Studio. The UTF-8 Unicode option will tell Studio to store your file as UTF-8, which will support Unicode characters. The Unicode Compile option will tell Studio to process the _WC pre-processor statements to properly handle Unicode embedded in your source files at compile time.

Enabling UTF-8 in NetLinx Studio

1. Choose Settings -> Preferences from the menu bar.
2. Select the Editor - Display and Indentations (FIG. 5).

3. Under Display, check the Enable UTF-8 format to display Hebrew, Arabic, Cyrillic and Han characters checkbox.
4. Click OK to save changes and close the Preferences dialog.

Enabling Unicode Compiling in NetLinx Studio

1. Choose Settings > Preferences from the menu bar
2. Select the NetLinx Compiler tab (FIG. 6).
3. Under Options, check the Enable _WC Preprocessor checkbox.
4. Click OK to save changes and close the Preferences dialog.

Including the Unicode Library

The Unicode Library is implemented in a NetLinx Include file, UnicodeLib.axi, that must be included in your program in order to access the Unicode functions.

The Unicode Library is located in an Include file located in the C:\Program Files\Common Files\AMXShare\AXIs directory. Because this location is the default Include search path, you do not need to specify the directory in the include statement.

To include the Unicode Library to your program add these lines to your program:

```c
#INCLUDE 'UnicodeLib.axi'
```

Defining a Unicode String Literal

To enter Unicode characters into your program, enclose the characters in single quotes, like you would any other string, and wrap the string literal in the Unicode macro _WC. Here is an example:

```c
_WC('Your string goes here')
```

- All Unicode string literals must be wrapped in the _WC macro.
- Failing to wrap a Unicode string in the _WC macro will result in a compiler error.

Storing a Unicode String

Unicode strings are stored in WIDECHAR arrays, similar to the way ASCII strings are stored in CHAR arrays. To define a WIDECHAR constant or variable and initialize it using a Unicode string literal, use the following syntax:

```c
WIDECHAR wcMyString[] = _WC('My String')
```

**NOTE:** The "wc" prefix is Hungarian notation for widechar. This is simply a programming convention and is completely optional. Hungarian notation helps you better identify your variables while you are programming and is a general recommended standard. For more information, see Wikipedia’s Hungarian Notation page.

Working with WIDECHAR Arrays and Unicode Strings

Working with WIDECHAR arrays and Unicode strings is very similar to working with CHAR arrays and ASCII strings. Most operations that can be performed on a CHAR array can be performed on a WIDECHAR array.

For instance, to assign a string to a variable use this syntax:

```c
wcMyString = _WC('My String')
```

The string functions defined for CHAR arrays have been defined for WIDECHAR array for use in Unicode programming. These functions allow you to operate on strings similar to the way you would with CHAR array. For instance, to remove the first 3 characters from a WIDECHAR array and return those characters as a WIDECHAR array, use WC_GET_BUFFER_STRING:

```c
wcRemoved = WC_GET_BUFFER_STRING(wcMyString, 3)
```
You will find that most other function work exactly as their CHAR counterpart do except they work on and return WIDECHAR arrays. The list of Unicode compatible functions is:

- WC_COMPARE_STRING
- WC_GET_BUFFER_CHAR
- WC_GET_BUFFER_STRING
- WC_LEFT_STRING
- WC_FIND_STRING
- WC_LENGTH_STRING
- WC_LOWER_STRING
- WC_MAX_LENGTH_STRING
- WC_MID_STRING
- WC_REMOVE_STRING
- WC_RIGHT_STRING
- WC_SET_LENGTH_STRING
- WC_UPPER_STRING

**Character Case Mappings**

Converting between upper and lower case is accomplished by using the Unicode.org character database to determine the mapping between upper case and lower case characters. Not all Unicode characters have an upper or lower case equivalent; these characters will not be affected by WC_UPPER_STRING and WC_LOWER_STRING. Only the characters defined by Unicode.org as having an upper or lower case mapping are affected by these functions.

For more information on Unicode character conversion, see the Unicode.org character conversion FAQ.

**Concatenating String**

Unicode strings and WIDECHAR array cannot be concatenated using the same syntax that ASCII strings use. In NetLinx, string expressions are enclosed in double quotes and can only contain 8-bit strings. To concatenate Unicode strings and WIDECHAR arrays, you must use the WC_CONCAT_STRING function:

```c
wcMyString = WC_CONCAT_STRING(_WC('First name'),_WC(' Surname'))
```

If you attempt to concatenate Unicode strings or WIDECHAR arrays using NetLinx string expressions, expect data loss.

**Converting Between WIDECHAR and CHAR**

On occasion, you may need to convert a CHAR array to a WIDECHAR array or a WIDECHAR array to a CHAR array. The CH_TO_WC and WC_TO_CH functions can be used to accomplish these conversions. For example:

```c
wcMyString = CH_TO_WC('Any ASCII string')
cMyString = CH_TO_WC(cMyString)
cMyString = WC_TO_CH (_WC('Any Unicode string'))
cMyString = WC_TO_CH (wcMyString)
```

- When converting from WIDECHAR to CHAR, Unicode characters are converted to '?'.
- Any ASCII or extended ASCII characters, i.e. 8-bit characters, contained in the WIDECHAR array will appear in the CHAR array.
- Converting from CHAR to WIDECHAR never results in loss of data.

**Using FORMAT**

The NetLinx Unicode library does not include a Unicode compatible FORMAT function. In NetLinx, the FORMAT function is used to convert numbers to text. To use FORMAT with Unicode string, use FORMAT to convert the number to a CHAR array and then use CH_TO_WC and WC_CONCAT_STRING to combine the result with an existing WIDECHAR array.

The following two syntaxes are functionality equivalent:

```c
fTemperature = 98.652
fTemperature = 98.652
mwString = FORMAT('The current temperature is %3.2f',fTemperature)
cTempString = FORMAT('%3.2f',fTemperature)
cTempString = wcMyString
```

```c
chMyString = CH_TO_WC('The current temperature is %3.2f',fTemperature)
```

**Reading and Writing to Files**

The NetLinx Unicode library supports reading and writing of WIDECHAR arrays. The WC_FILE routines operate the same as the FILE routines with the exception of FILE_OPEN. WC_FILE_OPEN takes an additional parameter; the file format. The WC_FILE_OPEN returns a special file handle so it is important to only use the file handle returned by WC_FILE_OPEN with other WC_FILE functions and the file handle used with WC_FILE functions must have been obtained by calling WC_FILE_OPEN.

The NetLinx Unicode library supports three different file formats for compatibility with files created on a computer. Windows Notepad supports the same three file formats so files created in Notepad can be read using the WC_FILE routines and files created using the WC_FILE routines can be read with Notepad.

When reading or appending to file, the file format is automatically determined when the file is opened. You can pass in a variable to WC_FILE_OPEN and the function will set the variable to the file format that was detected. When writing files, the file format parameter will determine how data is written to the file. The following constants can be used for specifying or checking the file format: WC_FORMAT_UNICODE, WC_FORMAT_UNICODE_BE, WC_FORMAT_UTF8.
The Unicode file format, specified by the constant `WC_FORMAT_UNICODE`, is the fastest to encode and decode. You should use this format unless you have a particular application that requires either UTF-8 or Unicode BE encoding.

The `WC_FILE_READ/WRITE` functions take the number of characters that will be read or written to the file. However, the functions return the number of bytes read or written to the file, not the number of characters. For Unicode and Unicode BE encoding, there are 2 bytes for every character.

For UTF-8 encoding, the number of bytes for every character varies depending on the character.

Unicode filenames are not supported. The parameter for the file name is a `CHAR` array. Always use a non-Unicode name for the file.

The following file functions support `WIDECHAR` arrays:

- `WC_FILE_OPEN`
- `WC_FILE_CLOSE`
- `WC_FILE_READ`
- `WC_FILE_READ_LINE`
- `WC_FILE_WRITE`
- `WC_FILE_WRITE_LINE`

**Send Strings to a User Interface**

Sending a `WIDECHAR` array to a user interface is accomplished using `WC_TP_ENCODE`. `WC_TP_ENCODE` takes a `WIDECHAR` array and returns a `CHAR` array formatted for a user interface UNI or BAU command.

```c
    cMyString = WC_TP_ENCODE(wcMyString)
    SEND_COMMAND dvTP,"'^UNI-1,0,',cMyString "
```

**Right-to-Left Unicode Strings**

Right-to-Left Unicode languages are stored in memory the same way left-to-right language are. The first memory position of an array contains the first logical character.

You can access the right-most character of a Right-to-Left Unicode string using this notation:

```c
    wchChar = wcString[1]
```

Right-to-left languages are not stored differently than left-to-right languages, they are simply rendered differently than right to left languages. However, note that the functions `WC_LEFT_STRING` and `WC_RIGHT_STRING` remove a number of characters from the start and end of a string respectively.

Using `WC_LEFT_STRING` on a right-to-left language will return the number of right-most, i.e. first, characters you requested, not the left-most, i.e. end, characters.

`WC_LEFT_STRING` returns the number of characters request from the front of the string and `WC_RIGHT_STRING` return the number of characters requested from the end of the string, regardless of the language's orientation.

**Compiler Errors**

The most common type of compiler errors you will encounter while programming for Unicode are caused by not wrapping Unicode string literals in `WC`, passing a `WIDECHAR` to a function that take a `CHAR` array or passing a `CHAR` array to a function that takes a `WIDECHAR` array.

If you forget to wrap a Unicode string in `WC`, expect to see the following compiler error:

On the line where the string is defined:

```c
    C10571: Converting type [string] to [WIDECHAR]
```

On the line where the constant or variable is used:

```c
    C10585: Dimension mismatch: [1] vs. [0] and C10533: Illegal assignment statement
```

If you try to pass a `CHAR` array to a function that expects a `WIDECHAR` array, expect to see the following compiler error:

On the line where the function call is made

```c
    C10585: Dimension mismatch: [1] vs. [0] and Type mismatch in call for parameter [WCDATA]
```

If you try to pass a `WIDECHAR` array to a function that expects a `CHAR` array, expect to see the following compiler error:

On the line where the function call is made

```c
    C10585: Dimension mismatch: [1] vs. [0] and Type mismatch in call for parameter [A]
```

**NOTE:** Parameter names might not match those listed above.
## Unicode Keywords

The NetLinx programming language supports the following Unicode keywords:

<table>
<thead>
<tr>
<th>Unicode Keywords</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>_WC</strong></td>
<td>This keyword supports Unicode strings. All Unicode string literals must be contained in single quotes and in the _WC macro.</td>
</tr>
<tr>
<td><strong>WC_COMPARE_STRING</strong></td>
<td>This keyword compares two Unicode strings. If either string contains a '?' character, the matching character in the other string is not compared. The '?' is equivalent to a wildcard. For example:</td>
</tr>
<tr>
<td><strong>WC_CONCAT_STRING</strong></td>
<td>This keyword concatenates two WIDECHAR arrays.</td>
</tr>
<tr>
<td><strong>WC_DECODE</strong></td>
<td>This function decodes Unicode string from a character string using one of 4 formats.</td>
</tr>
<tr>
<td><strong>WC_ENCODE</strong></td>
<td>This function encodes a Unicode string to a character string using one of 4 formats.</td>
</tr>
</tbody>
</table>

**_WC**

This keyword is a macro for Unicode strings. All Unicode string literals must be contained in single quotes and in the _WC macro.

```
WIDECHAR wcData[] = WC('Unicode String')
```

**WC_COMPARE_STRING**

This keyword compares two Unicode strings. If either string contains a '?' character, the matching character in the other string is not compared. The '?' is equivalent to a wildcard. For example:

```
INTEGER WC_COMPARE_STRING(WIDECHAR STR1[], WIDECHAR STR2[])
```

**Parameters:**
- `STR1` - the first widechar string to be compared.
- `STR2` - the first widechar string to be compared.

**Result:**
The returned result can only be True (1) or False (0).
- 0 = the strings don't match
- 1 = the strings are the same

See `COMPARE_STRING` (page 125) for a code example.

**WC_CONCAT_STRING**

This keyword concatenates two WIDECHAR arrays.

```
WIDECHAR[] WC_CONCAT_STRING(WIDECHAR STR1[], WIDECHAR STR2[])
```

**Parameters:**
- `STR1` - the first widechar string to be concatenated.
- `STR2` - the first widechar string to be concatenated.

**Result:**
A widechar string which concatenates STR1 and STR2

```
wcMyString = WC_CONCAT_STRING(wcString1,wcString2)
```

**WC_DECODE**

This function decodes Unicode string from a character string using one of 4 formats.

```
WIDECHAR[] WC_DECODE(CHAR cData[], INTEGER Format, LONG Start)
```

**Parameters:**
- `cData`: String containing the encoded Unicode string
- `Format`:
  - 1 Unicode: The data is encoded as a Unicode formatted stream. The constant `WC_FORMAT_UNICODE` is defined as a value of 1 for specifying this format.
  - 2 Unicode BE: The data is encoded as a Unicode BE (Big Endian) formatted stream. The constant `WC_FORMAT_UNICODE_BE` is defined as a value of 2 for specifying this format.
  - 3 UTF-8: The data is encoded as a UTF-8 formatted stream. The constant `WC_FORMAT_UTF8` is defined as a value of 3 for specifying this format.
  - 4 TP: The data is encoded for use with the UNI TP command. The constant `WC_FORMAT_TP` is defined as a value of 4 for specifying this format.
- `Stat`: Position in Data from which to start reading

**Result:**
A WIDECHAR array containing the Unicode data.

```
wData = WC_DECODE(cData, WC_FORMAT_UNICODE,1)
```

**WC_ENCODE**

This function encodes a Unicode string to a character string using one of 4 formats.

```
WIDECHAR[] WC_ENCODE(WIDECHAR STRING[], INTEGER Format, LONG Start)
```

**Parameters:**
- `STRING`: String containing the Unicode string to encode
- `Format`:
  - 1 Unicode: Encode the data as a Unicode formatted stream. The constant `WC_FORMAT_UNICODE` is defined as a value of 1 for specifying this format.
  - 2 Unicode BE: Encode the data as a Unicode BE (Big Endian) formatted stream. The constant `WC_FORMAT_UNICODE_BE` is defined as a value of 2 for specifying this format.
  - 3 UTF-8: Encode the data as a UTF-8 formatted stream. The constant `WC_FORMAT_UTF8` is defined as a value of 3 for specifying this format.
  - 4 TP: Encode the data for use with the UNI TP command. The constant `WC_FORMAT_TP` is defined as a value of 4 for specifying this format.
- `Stat`: Position in STRING from which to start reading

**Result:**
Result is a CHAR array containing the encoded Unicode data.

```
cData = WC_ENCODE(wcMyString, WC_FORMAT_UNICODE,1)
```
UniCode Keywords (Cont.)

**WC_FILE_CLOSE**

This function closes a file opened with **WC_FILE_OPEN**. This function should be called when all reading or writing to the file is completed.

```c
SLONG WC_FILE_CLOSE (LONG hFile)
```

- **Parameters:**
  - `hFile`: Handle to the file returned by **WC_FILE_OPEN**.

- **Result:**
  - 0: Operation was successful
  - -1: Invalid file handle
  - -5: Disk I/O error
  - -7: File already closed

There is a limit to the number of file handles available from the system. If files are not closed, it may not be possible to open a file.

```c
Result = WC_FILE_CLOSE (hFile)
```

**WC_FILE_OPEN**

This function opens a file for reading or writing.

```c
SLONG FILE_OPEN (CHAR FilePath[], LONG IOFlag, LONG Format)
```

- **Parameters:**
  - `FilePath`: String containing the path to the file to be opened
  - `IOFlag`:
    - 1: Read: The file is opened with read only status. The constant **FILE_READ_ONLY** is defined as a value of 1 for specifying this flag.
    - 2: R/W New: The file is opened with read write status. If the file currently exists, its contents are erased. The constant **FILE_RW_NEW** is defined as a value of 2 for specifying this flag.
    - 3: R/W Append: The file is opened with read write status. The current contents of the file are preserved and the file pointer is set to point to the end of the file. The constant **FILE_RW_APPEND** is defined as a value of 3 for specifying this flag.
  - `Format`:
    - 1: Unicode The file is opened as a Unicode formatted file. If the file is opened as Read or R/W Append and the file is a Unicode formatted file, this parameter will be set to this value by the function. The constant **WC_FORMAT_UNICODE** is defined as a value of 1 for specifying this format.
    - 2: Unicode BE The file is opened as a Unicode BE (big Endian) formatted file. If the file is opened as Read or R/W Append and the file is a Unicode BE formatted file, this parameter will be set to this value by the function. The constant **WC_FORMAT_UNICODE_BE** is defined as a value of 2 for specifying this format.
    - 3: UTF-8 The file is opened as a UTF-8 formatted file. If the file is opened as Read or R/W Append and the file is a UTF-8 formatted file, this parameter will be set to this value by the function. The constant **WC_FORMAT_UTF8** is defined as a value of 3 for specifying this format.

If the open operation is successful, this function returns a non-zero integer value representing the handle to the file. This handle must be used in subsequent read, write, and close operations.

```c
>0: Handle to file (open was successful)
-2: Invalid file path or name
-3: Invalid value supplied for IOFlag
-5: Disk I/O error
-9: End-of-file reached
```

If the file is opened successfully, it must be closed after all reading or writing is completed, by calling **WC_FILE_CLOSE**. If files are not closed, subsequent file open operations may fail due to the limited number of file handles available.

```c
// Open MYFILE.TXT for reading
INTEGER nFormatSLONG hFilehFile = WC_FILE_OPEN("MYFILE.TXT", FILE_READ_ONLY,nFormat)
// nFormat will be set to detected file type
```

**WC_FILE_READ**

This function reads a block of widechar data from the specified file.

```c
SLONG WC_FILE_READ (LONG hFile, WIDECHAR Buffer[], LONG BufLen)
```

- **Parameters:**
  - `hFile`: Handle to the file returned by **WC_FILE_OPEN**
  - `Buffer`: Buffer to hold the data to be read
  - `BufLen`: Maximum number of characters to read

- **Result:**
  - >0: The number of bytes actually read
  - -1: Invalid file handle
  - -5: Disk I/O error
  - -6: Invalid parameter
  - -9: End-of-file reached

This function reads (from the current location of the file pointer) the number of characters specified by `BufLen` (or fewer bytes if the end of file is reached). The characters are read from the file identified by `hFile` and are stored in `Buffer`. The file pointer will automatically be advanced the correct number of bytes so the next read operation continues where the last operation left off.

```c
WIDECHAR wcBuffer[1024]nBytes = WC_FILE_READ (hFile, wcBuffer, 1024)
```
### WC_FILE_READ_LINE
This function reads a line of widechar data from the specified file.

\[
\text{SLONG WC_FILE_READ_LINE (LONG hFile, WIDECHAR Buffer[], LONG BufLen) }
\]

**Parameters:**
- **hFile**: Handle to the file returned by `WC_FILE_OPEN`
- **Buffer**: Buffer to hold the data to be read
- **BufLen**: Maximum number of characters to read

**Result:**
- 0: The number of bytes actually read
- -1: Invalid file handle
- -5: Disk I/O error
- -6: Invalid parameter (buffer length must be greater than zero)
- -9: End-of-file reached

This function reads from the current location of the file pointer up to the next carriage return or to the end-of-file (EOF), whichever comes first. A complete line will not be read if the buffer length is exceeded before a carriage return (or EOF) is encountered. The characters are read from the file identified by file and are stored in Buffer. The `<CR>` or `<CR><LF>` pair will not be stored in Buffer.

If a complete line is read, the file pointer is advanced to the next character in the file after the `<CR>` or `<CR><LF>` pair or to the EOF if the last line was read.

```c
WIDECHAR wcBuffer[80]nBytes = WC_FILE_READ_LINE (hFile, wcBuffer, 80)
```

### WC_FILE_WRITE
This function writes a block of widechar data to the specified file.

\[
\text{SLONG WC_FILE_WRITE (LONG hFile, WIDECHAR Buffer[], LONG BufLen) }
\]

**Parameters:**
- **hFile**: Handle to the file returned by `WC_FILE_OPEN`
- **Buffer**: Buffer containing the data to write.
- **BufLen**: Number of characters to write.

**Result:**
- >0: The number of bytes actually written
- -1: Invalid file handle
- -5: Disk I/O error
- -6: Invalid parameter (buffer length must be greater than zero)
- -11: Disk full. The data will overwrite or append to the current contents of the file depending on the current position of the file pointer.

```c
WIDECHAR wcBuffer[1024]Result = WC_FILE_WRITE (hFile, wcBuffer, 1024)
```

### WC_FILE_WRITE_LINE
This function writes a line of widechar data to the specified file.

\[
\text{SLONG WC_FILE_WRITE_LINE (LONG hFile, WIDECHAR Line[], LONG LineLen) }
\]

**Parameters:**
- **hFile**: Handle to the file returned by `WC_FILE_OPEN`
- **Line**: Buffer containing the line of data to write.
- **LineLen**: Number of characters to write.

**Result:**
- >0: The number of bytes actually written
- -1: Invalid file handle
- -5: Disk I/O error
- -6: Invalid parameter (LineLen must be greater than zero)
- -11: Disk full. A `<CR><LF>` character pair is automatically appended to the end of the line.

```c
WIDECHAR wcLine[80]Result = FILE_WRITE_LINE (hFile, wcLine, 80)
```

### WC_FIND_STRING
This function searches through a string for a specified sequence of characters.

\[
\text{INTEGER WC_FIND_STRING (WIDECHAR STRING[], WIDECHAR Seq[], INTEGER Start) }
\]

**Parameters:**
- **STRING**: The string of character to search.
- **Seq**: The sequence of characters to search for.
- **Start**: The starting character position for the search.

**Result:**
- A 16-bit unsigned integer representing the character location of `Seq` in `STRING`. If the character string is found at the beginning of the string, this function returns 1; any error condition returns 0.

```c
POS = WC_FIND_STRING(STRING, _WC('ABC'), 1)
```

### WC_GET_BUFFER_CHAR
This keyword removes a character from a buffer.

\[
\text{WIDECHAR WC_GET_BUFFER_CHAR (WIDECHAR A[])}
\]

The result is a WIDECHAR value.

**WC_GET_BUFFER_CHAR** has a two-part operation:
1. Retrieve the first character in the buffer.
2. Remove the retrieved character from the buffer and shift the remaining characters by one to fill the gap.

```c
wchChar = GET_BUFFER_STRING(wcString)  // wchChar contains first character of wcString
// wcString is now one character smaller in length and
// starts with what used to be the 2nd character
```
### WC_GET_BUFFER_STRING
This function removes characters from a buffer.

```c
WIDECHAR WC_GET_BUFFER_STRING (WIDECHAR A[], Length)
```

- **Length**: The number of characters to remove.
- **Result**: A `WIDECHAR` value.

**Operation**:
1. Retrieve `<length>` number of characters from the buffer.
2. Remove the retrieved character from the buffer and shift the remaining characters up to fill the gap.

```c
wcSubStr = GET_BUFFER_STRING(wcString, 3)
// wcSubStr contains first 3 characters of wcString
// wcString is now three characters smaller in length and
// starts with what used to be the 4th character
```

### WC_LEFT_STRING
This function returns the specified number of characters from the beginning of a string.

```c
WIDECHAR[] WC_LEFT_STRING (WIDECHAR STRING[], Long Count)
```

- **Parameters**:
  - `STRING`: The string from which to extract the characters.
  - `Count`: The number of characters to copy from the beginning of the string.

**Result**: A string containing a copy of the first `Count` characters from `STRING`.

```c
wcSTRING = _WC('ABCDEFG')
wCSubstr = WC_LEFT_STRING(wcSTRING, 3) // wcSubstr = 'ABC'
```

### WC_LENGTH_STRING
This function returns the length of a `WIDECHAR` string. This function provides the same information as `LENGTH_ARRAY` (page 32).

```c
LONG WC_LENGTH_STRING (WIDECHAR STRING[])
```

- **Parameters**: `STRING`: The input character string.
- **Result**: The length of `STRING`. The string length can be set implicitly through a literal or variable string assignment or explicitly by calling `SET_LENGTH_STRING`.

```c
MAXLEN = WC_MAX_LENGTH_STRING(wcSTRING)
Len = WC_LENGTH_STRING(wcSTRING)
IF (MAXLEN > Len){ // append character to wcSTRING}
```

### WC_LOWER_STRING
This function changes all alphabetic characters in the specified string to lower case using the case mapping defined by Unicode.org.

```c
WIDECHAR[] WC_LOWER_STRING (WIDECHAR STRING[])
```

- **Parameters**: `STRING`: The `WIDECHAR` string to convert to lower case.
- **Result**: The result is the converted `WIDECHAR` string.

```c
wcLCString = WC_LOWER_STRING(wcSTRING)
```

### WC_MAX_LENGTH_STRING
This function returns the dimensioned length of a `WIDECHAR` string. This function provides the same information as `MAX_LENGTH_ARRAY` (page 32).

```c
LONG WC_MAX_LENGTH_STRING (WIDECHAR STRING[])
```

- **Parameters**: `STRING`: The input widechar string.
- **Result**: The result is the dimensioned length of `STRING`.

```c
MAXLEN = WC_MAX_LENGTH_STRING(wcSTRING) Len = WC_LENGTH_STRING(wcSTRING)
IF (MAXLEN > Len){ // append character to wcSTRING}
```

### WC_MID_STRING
This function returns the specified number of characters, starting at the specified location in the source string.

```c
WIDECHAR[] WC_MID_STRING (WIDECHAR STRING[], Long Start, Long Count)
```

- **Parameters**:
  - `STRING`: The input character string.
  - `Start`: Starting location in the string.
  - `Count`: Number of characters to extract.

**Result**: The result is a widechar string containing the specified characters.

```c
wcSTRING = _WC('ABCDEFGHIJK')
wCSubstr = WC_MID_STRING(wcSTRING, 5, 4) // wcSubstr = 'EFGH'
```

### WC_REMOVE_STRING
This function removes characters from the specified string. All characters up to and including the first occurrence of the specified sequence are removed.

```c
WIDECHAR[] WC_REMOVE_STRING (WIDECHAR STRING[], WIDECHAR Seq[], Long Start)
```

- **Parameters**:
  - `STRING`: String from which to find and remove characters.
  - `Seq`: Sequence of characters to find.
  - `Start`: Starting position in the string to begin search.

**Result**: The result is a string containing the removed characters. If the character sequence was not found, an empty string is returned.

```c
wcSTRING = _WC('ABCDEF')
wCSubstr = WC_REMOVE_STRING(wcSTRING, _WC('BC'), 1)
// wcSubstr = 'ABC' // wcSTRING = 'DEF'
```
<table>
<thead>
<tr>
<th>UniCode Keywords (Cont.)</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>WC_RIGHT_STRING</strong></td>
<td>Returns the specified number of characters from the end of a string.</td>
<td>WC_RIGHT_STRING(WIDECHAR STRING[], LONG Count)</td>
</tr>
<tr>
<td>Parameters:</td>
<td></td>
<td>STRING: The string from which to extract the characters.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Count: The number of character to copy from the end of the string.</td>
</tr>
<tr>
<td>Result:</td>
<td></td>
<td>The return is a string containing a copy of the last Count characters from STRING.</td>
</tr>
<tr>
<td>wcSTRING = WC('ABCDEFG')</td>
<td>wcSubstr = WC_RIGHT_STRING(wcSTRING, 3) // wcSubstr = 'EFG'</td>
<td></td>
</tr>
</tbody>
</table>

| **WC_SET_LENGTH_STRING** | This function sets the length of a WIDECHAR string. This function provides the same functionality as SET_LENGTH_ARRAY (page 32). | WC_SET_LENGTH_STRING(WIDECHAR STRING[], LONG Len) |
| Parameters: | | STRING: The input widechar string. |
| | | Len: The new string length. |
| WC_SET_LENGTH_STRING(wcSTRING, 10) | |

| **WC_TO_CH** | This keyword converts a WIDECHAR array to a CHAR array. | WC_TO_CH(WIDECHAR wcSTRING[]) |
| Parameters: | | STRING: The widechar string to convert to a character string. |
| Result: | | A character string version of the widechar string. |
| wcUCString = WC_UPPER_STRING(wcSTRING) | |

| **WC_TP_ENCODE** | This function encodes a WIDECHAR array into a CHAR array formatted for the UNI and BAU user interface commands. | WC_TP_ENCODE(WIDECHAR STRING[]) |
| Parameters: | | STRING: The widechar string to send to a user interface. |
| Result: | | The result is an encoded character string. |
| cString = WC_TP_ENCODE(wcSTRING)SEND_COMMAND dvTY,"'^UNI-1,0,',cString" |

| **WC_UPPER_STRING** | This function changes all alphabetic characters in the specified string to upper case using the case mapping specified by Unicode.org. | WC_UPPER_STRING(WIDECHAR wcSTRING[]) |
| Parameters: | | STRING: The widechar string to convert to upper case. |
| Result: | | The result is the converted widechar string. |
| wcUCString = WC_UPPER_STRING(wcSTRING) | |
Overview

NetLinx defaults non-array variables to the integer data types and defaults array variables to character data type array. The variable must be explicitly declared if using any other data type. NetLinx provides support for several different types of variables distinguished by attributes, such as:

- **Scope**
- **Constancy**
- **Persistence**

Scope

Scope is a term used in reference to program variables that describe where in the program they can be accessed. There are two types:

- **Local scope**: a variable can only be accessed in the subroutine or method that it is declared.
- **Global scope**: a variable can be accessed anywhere in the program.

Scope differentiates the two basic classes of NetLinx variables:

- **Local variable**: a variable declared within a subroutine or function whose scope is limited to that subroutine or function.
- **Global variable**: a variable declared in the `DEFINE_VARIABLE` section; its scope extends throughout the module in which it is declared.

Local Variables

Local variables are restricted in scope to the statement block in which they are declared. A statement block is one or more NetLinx statements enclosed in a pair of braces, like the blocks following subroutines, functions, conditionals, loops, waits, and so on. Local variables must be declared immediately after the opening brace of a block but before the first executable statement. To provide compatibility with the Axcess language, local variables may be declared right before the opening brace for `DEFINE_CALL` declarations only. For example, both formats shown below are legal in NetLinx:

```netlinx
DEFINE_CALL 'My Subroutine' (INTEGER INT1)
    LOCAL_VAR INTEGER INT2
    { (* body of subroutine *) }
```

```netlinx
DEFINE_CALL 'My Subroutine' (INTEGER INT1)
    { *
        LOCAL_VAR INTEGER INT2
        (* body of subroutine *)
    }
```

The scope of a local variable is restricted to the statement block in which it is declared. A local variable is either static or non-static, depending on whether it is declared as `LOCAL_VAR` (page 152) or `STACK_VAR` (page 153).

**NOTE:** A static variable maintains its value throughout the execution of the program, regardless of whether it is within scope of the current program instruction.

- The keyword `LOCAL_VAR` specifies a static variable. A static variable's value is initialized the first time the statement block in which it is declared is executed and retained after execution of the statement block has finished.
- The `STACK_VAR` keyword specifies a non-static variable. A non-static variable's value is re-initialized every time the statement block in which it is declared is executed.
- If neither the `LOCAL_VAR` nor the `STACK_VAR` keyword is specified, `STACK_VAR` is assumed (default).

```netlinx
IF (X > 10)
    { *
        LOCAL_VAR INTEGER INT2  // static (permanent)
        STACK_VAR CHAR ARRAY1[10] // non-static (temporary)
        (* statements *)
    }
```

**NOTE:** Variable declarations outside of `DEFINE_VARIABLE` will default to `STACK_VAR` if neither "local" or "stack" is specified.

Local_VAR and STACK_VAR can be used interchangeably in any statement block except for waits. Only LOCAL_VAR variables may be declared inside a wait block.

```netlinx
WAIT 10, 'My Wait Name'
    { *
        LOCAL_VAR CHAR TempBuf[80]
        (* statements *)
    }
```

A name assigned to a local variable must be unique within the statement block in which it is declared and any statement block enclosing that block. Therefore, non-nested statement blocks can define the same local variable name without conflict. For example:
DEFINE_FUNCTION integer MyFunc(INTEGER nFlag)
{
    LOCAL_VAR INTEGER n
    IF (nFlag > 0)
    {
        LOCAL_VAR INTEGER n          // illegal declaration
        .
        .
    }
    .
    .
}

DEFINE_FUNCTION integer MyFunc(INTEGER nFlag)
{
    IF (nFlag > 0)
    {
        LOCAL_VAR INTEGER n
        .
        .
    }
    else
    {
        LOCAL_VAR INTEGER n           // legal declaration
        .
        .
    }
}

The general form of a static local variable declaration is:

[LOCAL_VAR] [VOLATILE | PERSISTENT] [CONSTANT] [<type>] name

The general form of the non-static local variable declaration is:

[STACK_VAR] [<type>] name

Since non-static local variables are allocated on the program stack (a block of memory reserved for allocation of temporary variables), the keywords VOLATILE (page 153), PERSISTENT (page 153), and CONSTANT (page 152) do not apply.

Global Variables

Global variables are defined in the DEFINE_VARIABLE (page 63) section of any program module.

For example:

DEFINE_VARIABLE
CONSTANT INTEGER MAXLEN = 64
CHAR STR[MAXLEN] = 'No errors were found.'
INTEGER ARRAY[] = {100, 200, 300}

A global variable is accessible throughout the module or program in which it is defined. Global variables retain their value as long as the program runs. They may retain their value after powering down or reloading the system, depending on the variable's persistence attributes (VOLATILE and PERSISTENT).

If a local variable shares the same name as a global variable, the local variable always takes precedence.

The general form of a global variable definition is:

[NON_VOLATILE | VOLATILE | PERSISTENT] [CONSTANT] [<type>] name [= <value>]

Constancy

Any variable may also be assigned the attribute CONSTANT(page 152). This declares a variable to be immutable (cannot change at run-time). The variable must be initialized as part of its declaration if this keyword is used.

Persistence

The persistence of a variable is controlled through the NON_VOLATILE (page 153), VOLATILE (page 153), and PERSISTENT (page 153) keywords.

Non-Volatile Variables

A variable declared with the NON_VOLATILE keyword is stored in non-volatile memory. It will retain its value in the event of a system power-down, but is reset to zero if the program is reloaded. Unless specified otherwise, all variables are stored in non-volatile memory.

Volatile Variables

A variable declared with the VOLATILE keyword is stored in volatile memory and resets to zero after either a power-down or reload. Voltatile memory is generally faster and more plentiful than non-volatile memory. For this reason, you should use the VOLATILE keyword when declaring large data arrays where persistence of the data is not a requirement.

Persistent Variables

If a variable is declared with the PERSISTENT keyword, it is initialized to zero the first time the program is loaded but will retain its value after either power-down or reload. If the data type is omitted from the variable definition, the following defaults are assumed:

- Single variables are PERSISTENT type.
- Arrays are CHAR type.
You can define a variable to be persistent using the PERSISTENT storage modifier as shown below:

```netlink
DEFINE_VARIABLE
PERSISTENT CHAR cMyString[100]
```

All persistent variables are automatically non-volatile, and it's not legal to define a variable as VOLATILE and PERSISTENT. Any time after a NetLinx program that has a persistent variable declared subsequent downloads of new NetLinx programs that contain the same persistent variable will automatically be set to contain the same value as it previously did. By default, non-persistent variables are set to zero after a NetLinx program downloads. Persistence overrides this behavior by setting the variable in the newly downloaded program to be the same as it was before the download.

Typically, persistent variables are used for saving preset information. Suppose you have a system that contains several PosiTrack camera positioning systems and that the user interface to the system allows the user to set the position of any of the cameras and record that position for recalling later. The position presets are stored in a non-volatile array variable so they are maintained during a power cycle. Without persistent variables, an update to the NetLinx program would zero out all of the presets that the user had stored. With persistent variables, the new NetLinx program can be downloaded and all of the presets remain intact.

When a new NetLinx program is downloaded to the Master, the Master iterates through all non-volatile variables from the new program looking for persistent ones. When it finds a persistent variable in the new program, it searches the old programs persistent variable space for the "same variable". When it finds the same variable, the value of the new variable is set to the same value as the old programs variable. It is important to note what is considered to be the "same variable".

The master identifies the "same variable" by verifying for duplicity the following:

- Variable name
- Variable source location
- Variable type

Therefore, in order for persistence to function properly, the name, type, and file declared in must be the same as the previously downloaded NetLinx program. If you changed any of the three, the new persistent variable will not be set with the old variable's value.

### Constants

Constants are defined in the DEFINE_CONSTANT (page 61) section.

### Variables Keywords

The NetLinx programming language supports the following Variables keywords:

<table>
<thead>
<tr>
<th>Keywords</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS_VALUE</td>
<td>Provides the absolute value of a variable. It will take any intrinsic variable type and return the same type.</td>
</tr>
<tr>
<td>MAX_VALUE</td>
<td>Provides the value of the highest of two variables. It will take any intrinsic variable type and return the same type of the highest variable.</td>
</tr>
<tr>
<td>MIN_VALUE</td>
<td>Provides the value of the lowest of two variables. It will take any intrinsic variable type and return the same type of the lowest variable.</td>
</tr>
</tbody>
</table>

**Example Usage**:

```netlink
DEFINE_VARIABLE
SLONG Var1, Var2
DEFINE_START
Var1 = -1
DEFINE_PROGRAM
Var2 = ABS_VALUE(Var1)        // Var2 = 1
```

```netlink
DEFINE_VARIABLE
SLONG Var1, Var2, VarMax
DEFINE_START
Var1 = 100
Var2 = 200
DEFINE_PROGRAM
VarMax = MAX_VALUE (Var1,Var2)     // VarMax = 200
```

```netlink
DEFINE_VARIABLE
SLONG Var1, Var2, VarMin
DEFINE_START
Var1 = 100
Var2 = 200
DEFINE_PROGRAM
VarMin = MIN_VALUE (Var1,Var2)         // VarMin = 100
```
### Variables Keywords (Cont.)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NON_VOLATILE</strong></td>
<td>A variable declared with the <code>NON_VOLATILE</code> keyword is stored in non-volatile memory. It retains its value in the event of a system power-down, but is reset to zero if the program is reloaded. Unless specified otherwise, all variables are stored in non-volatile memory.</td>
</tr>
<tr>
<td><strong>OFF</strong></td>
<td>See page 37</td>
</tr>
<tr>
<td><strong>ON</strong></td>
<td>See page 37</td>
</tr>
</tbody>
</table>
| **PERSISTENT** | If a variable is declared with the `PERSISTENT` keyword, it is initialized to zero the first time the program is loaded but will retain its value after power-down or reload.  
  • The `PERSISTENT` attribute does not apply to non-static local variables, since non-static local variables are allocated on the program stack (a block of memory reserved for allocation of temporary variables).  
  • The `PERSISTENT` attribute does not apply to the individual members of a structure. |
| **RANDOM_NUMBER** | This function returns a random number \( X \) in the range \( 0 \leq X < \text{Max} \).  
  \[
  \text{LONG RANDOM_NUMBER (LONG Max)}
  \]  
  Parameters:  
  • \( \text{Max} \): An unsigned long integer (must be greater than zero) that will serve as the upper limit for the random number generator.
  
  The result is an unsigned long integer \( \geq 0 \) and \( < \text{Max} \).
  
  \[
  \text{Num} = \text{RANDOM_NUMBER(1000)}  
  \]  
  \[
  \quad \text{//} \quad 0 \leq \text{Num} < 1000
  \] |
| **STACK_VAR**  | This keyword specifies a non-static variable.  
  A non-static variable's value is re-initialized every time the statement block in which it is declared is executed. References to `STACK_VAR` variables are not allowed within waits. `STACK_VAR` are temporary variables that cease to exist when the block in which they are declared is exited. If neither the `LOCAL_VAR` nor the `STACK_VAR` keyword is specified, `STACK_VAR` is assumed. |
| **TOTAL_OFF**  | See page 37. |
| **TYPE_CAST**  | This routine eliminates compiler type cast warnings by casting the passed intrinsic variable type to the type assigned by the return value.  
  \[
  \text{IntrinsicVariableNewType TYPE_CAST (IntrinsicVariableType)}
  \]  
  It is possible to eliminate the compiler warnings related to type casting. The `TYPE_CAST` library function converts any non-array intrinsic type to any other non-array intrinsic type. The type conversion still happens and follows the standard Type Conversion Rules, but any warnings related to the type cast are eliminated. Type casting causes potential loss of data when a variable or constant is assigned to a variable of smaller type. |
| **VOLATILE**   | This keyword is used as part of a variable declaration to specify that storage space for the variable be allocated in volatile memory. Variables stored in volatile memory are not retained when the system is powered-down, as are variables stored in non-volatile memory. The trade-off is that volatile memory is generally more plentiful and therefore a good choice for storing large data arrays. |
Wait Keywords

Overview
Wait instructions allow delayed execution of one or more program statements. When a wait statement is executed, it is added to a list of currently active wait requests and the program continues running.

Types of Waits
Types of Wait statements include:

- **Timed Waits** have an associated parameter that indicates the amount of time that must elapse before the associated wait instruction(s) are to be executed. See page 155.
- **Conditional Waits** require that a specified condition be met before the instructions are executed. See page 155.
- **Timed Conditional Waits** have a timeout parameter; if the condition is not met before the specified time elapses, the wait request is canceled. See page 155.

Naming Waits
Supplying a unique name in the wait statement allows the wait to be identified for purposes of canceling, pausing, or restarting the wait request. The name must not conflict with previously defined constants, variables, buffers, subroutines, or functions. Unlike other NetLinx identifiers, wait names may contain spaces. If a wait instruction that uses a name currently in the wait list is encountered, the new wait instruction is thrown away so as not to conflict with the one currently in progress. If this feature is not desired, the current wait must be canceled before processing the new request.

Nesting Waits
The wait time for a nested wait is the sum of it’s own wait time, plus that of the enclosing waits. In the example below, SECOND WAIT occurs 0.5 seconds after FIRST WAIT is executed, or 1.5 seconds after FIRST WAIT is added to the wait list.

\[
\text{WAIT 10 'FIRST WAIT'}
\]
\[
\{\text{ (* FIRST WAIT statements *)}
\]
\[
\text{WAIT 5 'SECOND WAIT'}
\]
\[
\{\text{ (* SECOND WAIT statements *)}
\}
\]

To execute the inner wait of a nested conditional wait, the conditions must be met in the order specified (condition 1, then condition 2) but not necessarily at the same time.

\[
\text{WAIT UNTIL <condition 1> 'FIRST WAIT'}
\]
\[
\{\text{ (* FIRST WAIT statements *)}
\]
\[
\text{WAIT UNTIL <condition 2> 'SECOND WAIT'}
\]
\[
\{\text{ (* SECOND WAIT statements *)}
\}
\]

Using Waits - Limitations
- References to STACK_VAR variables are not allowed within waits (STACK_VAR are temporary variables that cease to exist when the block in which they are declared is exited).
- Variable copies are made of functions and subroutine parameters. This can have speed/execution penalties.
- A RETURN is not allowed within a WAIT within functions and subroutines.
- A BREAK or CONTINUE cannot appear within a WAIT if it takes execution out of the scope of the WAIT.
- The code within a WAIT cannot reference a function or subroutine array parameter whose bounds are unspecified.

WAIT keywords
The NetLinx programming language supports the following WAIT keywords:

<table>
<thead>
<tr>
<th>WAIT Keywords</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANCEL_ALL_WAIT</td>
<td>This keyword cancels all WAITs (named or unnamed) from the appropriate Wait list. Syntax: CANCEL_ALL_WAIT</td>
</tr>
<tr>
<td>CANCEL_ALL_WAIT_UNTIL</td>
<td>This keyword cancels all WAIT_UNTILs and TIMED_WAIT_UNTILs (named or unnamed) from the appropriate Wait list. Syntax: CANCEL_ALL_WAIT_UNTIL</td>
</tr>
<tr>
<td>CANCEL_WAIT</td>
<td>This keyword removes the wait specified by name from the appropriate wait list. Syntax: CANCEL_WAIT '&lt;wait name&gt;'</td>
</tr>
</tbody>
</table>
WAIT Keywords (Cont.)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANCEL_WAIT_UNTIL</td>
<td>This keyword cancels a specified WAIT_UNTIL or TIMED_WAIT_UNTIL. Only named WAIT_UNTIL and named TIMED_WAIT_UNTIL commands can be canceled.</td>
</tr>
<tr>
<td>Syntax:</td>
<td>CANCEL_WAIT_UNTIL '&lt;wait name&gt;'</td>
</tr>
<tr>
<td>PAUSE_ALL_WAIT</td>
<td>This keyword suspends all waits currently in effect.</td>
</tr>
<tr>
<td>Syntax:</td>
<td>PAUSE_ALL_WAIT</td>
</tr>
<tr>
<td>PAUSE_WAIT</td>
<td>Puts a scheduled wait on hold. The wait being paused is identified by the parameter name. The wait timer stops counting down until it is resumed with a RESTART_WAIT command.</td>
</tr>
<tr>
<td>Syntax:</td>
<td>PAUSE_WAIT '&lt;wait name&gt;'</td>
</tr>
<tr>
<td>RESTART_ALL_WAIT</td>
<td>This command resumes all waits that were previously paused. This includes both named and unnamed waits.</td>
</tr>
<tr>
<td>Syntax:</td>
<td>RESTART_ALL_WAIT</td>
</tr>
<tr>
<td>RESTART_WAIT</td>
<td>RESTART_WAIT resumes the countdown for a wait suspended with PAUSE_WAIT. The wait to be restarted is identified by the parameter name.</td>
</tr>
<tr>
<td>Syntax:</td>
<td>RESTART_WAIT '&lt;name&gt;'</td>
</tr>
<tr>
<td>WAIT</td>
<td>This keyword delays execution of one or more statements for a specified period of time.</td>
</tr>
<tr>
<td>Syntax:</td>
<td>WAIT time ['&lt;name&gt;']</td>
</tr>
<tr>
<td>Parameters:</td>
<td>• time: A constant or variable indicating the wait time. Time is expressed in 1/10th second units. The statement below specifies a wait time of 5 seconds for the wait named FIRST_WAIT.</td>
</tr>
<tr>
<td></td>
<td>• &lt;name&gt;: The name to assign to the wait. This name must be a literal string. The wait name is optional, although unless a wait is named it cannot be individually canceled, paused, or restarted.</td>
</tr>
<tr>
<td></td>
<td>If greater precision is required, the time parameter can be expressed as a decimal fraction, for example 0.1 to specify a wait time of 1/100th of a second. The range is 0.1 to 0.9.</td>
</tr>
<tr>
<td></td>
<td>WAIT 50 'FIRST WAIT'</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>(* wait statements *)</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>WAIT_UNTIL</td>
<td>This is a conditional Wait request. This keyword is used to delay execution of one or more statements until a specified condition is met.</td>
</tr>
<tr>
<td>Syntax:</td>
<td>WAIT_UNTIL &lt;condition&gt; ['&lt;name&gt;']</td>
</tr>
<tr>
<td>Parameters:</td>
<td>• &lt;condition&gt;: Any single or compound expression that can be evaluated as a logical expression. The Wait statements are executed if and when the wait condition becomes True.</td>
</tr>
<tr>
<td></td>
<td>• &lt;name&gt;: The name to assign to the Wait. This name must be a literal string. The Wait name is optional, although unless a Wait is named it cannot be individually canceled, paused, or restarted.</td>
</tr>
<tr>
<td>TIMED_WAIT_UNTIL</td>
<td>This is a Timed Conditional Wait request. This keyword delays execution of one or more statements until a particular condition is met. It is similar to WAIT_UNTIL except that this instruction provides for a timeout parameter to be specified.</td>
</tr>
<tr>
<td>Syntax:</td>
<td>TIMED_WAIT_UNTIL &lt;condition&gt; timeout ['&lt;name&gt;']</td>
</tr>
<tr>
<td>Parameters:</td>
<td>• &lt;condition&gt;: Any single or compound expression that can be evaluated as a logical expression. The Wait statements are executed if and when the Wait condition becomes true.</td>
</tr>
<tr>
<td></td>
<td>• timeout: A constant or variable indicating the timeout value in 1/10th seconds. If the Wait condition is not met within the time indicated by this parameter, the Wait is canceled, in which case no wait statements are executed.</td>
</tr>
<tr>
<td></td>
<td>• &lt;name&gt;: The name to assign to the Wait. This name must be a literal string. The Wait name is optional, although unless a Wait is named it cannot be individually canceled, paused, or restarted.</td>
</tr>
</tbody>
</table>
Appendix A - Compiler Warning & Errors

Compiler Warnings

Sometimes the compiler generates a warning message instead of an error message; these warning messages always start with w.

- A warning about a particular statement means that the statement is not technically an error, but you should be careful doing it.
- Warnings, unlike errors, do not stop the program from compiling.

### Compiler Warnings

<table>
<thead>
<tr>
<th>Warning</th>
<th>Description</th>
</tr>
</thead>
</table>
| (w) Cannot assign unlike types | This warning occurs when a variable or value of one type is assigned to a variable of a different type. Here are some examples:  
  - Assigning a string literal, string expression, or array to a non-array variable  
  - Assigning a non-array variable to an entire array  
  - Assigning an integer array to a non-integer array  
  - Assigning a two-dimensional array to a one-dimensional array, or vice versa  
  - Assigning the result of a function that returns an array type to a non-array variable or to a two-dimensional array variable (for example, X = ITOA(12), where X is a non-array variable or two-dimensional array variable)  
  - Assigning the result of a function that returns a non-array type to a one- or two-dimensional array variable (for example, X = ATOI('AMX'), where X is a one- or two-dimensional array variable)  
  This message is a warning and not an error, because X = ITOA(12) works correctly when X is a simple variable, since the result is a single value between 0 and 65,535. |

| (w) DEFINE_CALL is not used | This warning occurs at the end of program compilation for each DEFINE_CALL (page 58) subroutine that was declared but never used. |

| (w) Integer applies to arrays only | This warning appears when the keyword INTEGER (page 55) is applied to a non-array type of variable. Doing this is not an error, because non-array variables are already integers, but it is redundant. |

| (w) Long_While within While | This warning occurs if the compiler finds a LONG_WHILE (page 53) or MEDIUM_WHILE (page 53) inside a block of code following a WHILE (page 53) keyword. This warning exists because the WHILE command has a 1/2 second timeout period, and the LONG_WHILE and MEDIUM_WHILE keywords do not. This could create a hard-to-find logic error.  
  The solution is to change the WHILE to a LONG_WHILE. |

| (w) Possibly too many nested levels | This warning appears if there is a large amount of nesting in the program. This can happen with a long chain of IF...ELSE IF statements (page 47).  
  The solution is to use the SELECT...ACTIVE (page 47) set of statements. |

| (w) Variable is not used | This warning occurs at the end of compilation for each variable that was declared but never used. |

Compiler Errors

The compiler informs you when it finds an error during the compilation process. Most of the time these errors occur due to a typographical error or incorrect syntax of a particular command. Unlike warnings, errors must be corrected before your NetLinx program can be executed.

### Compiler Errors

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A &quot;&lt;symbol&gt;&quot; was expected</td>
<td>The compiler is expecting a certain symbol at this particular place.</td>
</tr>
<tr>
<td>ACTIVE keyword expected</td>
<td>An ACTIVE keyword is not present after a SELECT (page 52) keyword.</td>
</tr>
<tr>
<td>Allowed only in DEFINE_START</td>
<td>A keyword that is only allowed to appear in the DEFINE_START (page 62) section of the program was encountered elsewhere.</td>
</tr>
<tr>
<td>Attempted CALL to undefined subroutine</td>
<td>A CALL statement refers to a subroutine that has not been defined with a DEFINE_CALL (page 61) statement.</td>
</tr>
<tr>
<td>Comment never ends, EOF encountered</td>
<td>A comment begins but never ends. Place a close comment, * ) at the end of the unfinished comment.</td>
</tr>
<tr>
<td>Conditional compile nesting too deep</td>
<td>There are too many nested #IF_DEFINED (page 29) or #IF_NOT_DEFINED (page 29) conditional compilation statements. The limit is 20 nested conditional compilation statements.</td>
</tr>
</tbody>
</table>
| Constant type not allowed | A constant value was declared as latching, toggling, or mutually exclusive, as shown below:  
  DEFINE_CONSTANT  
  PLAY = 1  
  DEFINE_LATCHING  
  PLAY (* Error: PLAY is a constant *) |
| DEFINE_CALL must have a name | DEFINE_CALL (page 61) must have a name after it. For example,  
  DEFINE_CALL 'VHS'. |
<table>
<thead>
<tr>
<th>Compiler Errors (Cont.)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFINE_CALL name already used</td>
<td>The name of the DEFINE_CALL (page 61) has already been used. This name cannot be the same as an already declared identifier of any type.</td>
</tr>
<tr>
<td>Device values must be equal</td>
<td>In a range specification, the devices (or their defined identifiers) must be equal. For example, ([1.1]..[1.5]) is valid; ([1.1]..[2.5]) is not.</td>
</tr>
<tr>
<td>Duplicate symbol</td>
<td>Duplicate definitions of variables or constants are found. All variables and constants must have unique identifiers.</td>
</tr>
<tr>
<td>Evaluation stack overflow</td>
<td>The expression is too complicated. Try breaking it up into smaller pieces.</td>
</tr>
<tr>
<td>Evaluation stack underflow</td>
<td>The expression is too complicated. Try breaking it up into smaller pieces.</td>
</tr>
<tr>
<td>Identifier expected</td>
<td>The compiler is expecting an identifier after a #DEFINE (page 29) statement or after an integer declaration in the DEFINE_VARIABLE (page 63) section.</td>
</tr>
<tr>
<td>Identifier is not an array type</td>
<td>A non-array variable was treated as an array.</td>
</tr>
<tr>
<td>Include file not found</td>
<td>An INCLUDE (page 52) statement was encountered, but the specified include file could not be found.</td>
</tr>
<tr>
<td>Invalid include file name</td>
<td>A string literal enclosed in single quotes must follow the INCLUDE (page 52) keyword.</td>
</tr>
<tr>
<td>Library file not found</td>
<td>The library file containing the specified SYSTEM_CALL (page 28) could not be found.</td>
</tr>
<tr>
<td>Maximum string length exceeded</td>
<td>String literals are limited in length to 132 characters, including spaces.</td>
</tr>
<tr>
<td>Must be char array reference</td>
<td>An array type variable was expected in CREATE_BUFFER (page 35), CREATE_MULTI_BUFFER (page 35), or CLEAR_BUFFER (page 35).</td>
</tr>
<tr>
<td>Must be integer reference</td>
<td>The identifier in question must be an integer. This error occurs when the third parameter of CREATE_LEVEL (page 102) is an array or array element.</td>
</tr>
<tr>
<td>Out of memory</td>
<td>The compiler has run out of memory. Free up memory either by removing any pop-up programs or drivers, by using extended memory, or by breaking your program into one or more Include files.</td>
</tr>
<tr>
<td>Parameter mismatch in CALL</td>
<td>A value or variable passed to a CALL (page 28) as a parameter is of the wrong type as defined by the DEFINE_CALL (page 61) statement.</td>
</tr>
<tr>
<td>PROGRAM_NAME must be on line 1</td>
<td>Move the PROGRAM_NAME= (page 63) statement to the first line of the program.</td>
</tr>
<tr>
<td>Push/Release not allowed within Push/Release</td>
<td>A PUSH (page 120) or RELEASE (page 120) statement was found within a block of code headed by a PUSH or RELEASE statement.</td>
</tr>
<tr>
<td>Push/Release not allowed within Wait</td>
<td>These keywords are not allowed in a section of code which will be executed due to a WAIT (page 155) keyword.</td>
</tr>
<tr>
<td>PUSH_CHANNEL not allowed within Wait</td>
<td>These keywords are not allowed in a section of code which will be executed due to a WAIT (page 155) keyword.</td>
</tr>
<tr>
<td>RELEASE_CHANNEL not allowed within Wait</td>
<td>These keywords are not allowed in a section of code which will be executed due to a WAIT (page 155) keyword.</td>
</tr>
<tr>
<td>PUSH_DEVICE not allowed within Wait</td>
<td>These keywords are not allowed in a section of code which will be executed due to a WAIT (page 155) keyword.</td>
</tr>
<tr>
<td>RELEASE_DEVICE not allowed within Wait</td>
<td>These keywords are not allowed in a section of code which will be executed due to a WAIT (page 155) keyword.</td>
</tr>
<tr>
<td>String constant expected</td>
<td>A string is required for the particular operation. This error occurs if a string literal enclosed in single quotes does not follow the PROGRAM_NAME (page 63) keyword.</td>
</tr>
<tr>
<td>String constant never ends, EOF encountered</td>
<td>A string literal is started but never ends. Add a closing single quotation mark (‘) to the end of the string.</td>
</tr>
<tr>
<td>String literal expected</td>
<td>A string is required for the particular operation. This error occurs if a string literal enclosed in single quotes does not follow the #WARN (page 29) keyword.</td>
</tr>
<tr>
<td>Subroutine may not call itself</td>
<td>A subroutine (page 26) cannot call itself. It can, however, call a different subroutine.</td>
</tr>
<tr>
<td>Syntax error</td>
<td>A syntax error is found in an expression. In most cases, this error means that a character is out of place or something is misspelled.</td>
</tr>
<tr>
<td>SYSTEM_CALL name not same as PROGRAM_NAME in &lt;file&gt;</td>
<td>This error occurs when a library file is compiled and the name of the subroutine in the library file does not match the PROGRAM_NAME (page 63) string on the first line of the file.</td>
</tr>
<tr>
<td>This variable type not allowed</td>
<td>This error occurs when an attempt is made to use an array variable with DEFINE_LATCHING (page 62), DEFINE.Toggle (page 62), or DEFINE_MUTUALLY_EXCLUSIVE (page 62).</td>
</tr>
</tbody>
</table>
Appendix A - Compiler Warning & Errors

158 Language Reference Guide - NetLinx Programming

Run-Time Errors

In many cases, a program is compiled and sent to the Central Controller error-free, but the system does not act in the way it should. If the program code is correct, you should check for run-time errors. These errors occur in the Central Controller, usually when it could not perform a particular operation in the program.

### Run-Time Errors

<table>
<thead>
<tr>
<th>Error Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bad assign 2dim...</td>
<td>These errors occur if an attempt is made to assign a two-dimensional array to a different type (such as a variable or one-dimensional array), and vice versa.</td>
</tr>
<tr>
<td>Bad assign Call...</td>
<td>These errors occur if the Central Controller cannot assign a parameter in a CALL (page 16) statement to the parameter in the corresponding DEFINE_CALL (page 16) statement.</td>
</tr>
<tr>
<td>Bad element assign...</td>
<td>These errors occur if an assignment is attempted past the end of an array, or to the Ø location of an array (for example, ARRAY[Ø]).</td>
</tr>
<tr>
<td>Bad Off... Bad On... Bad To...</td>
<td>These errors indicate that the device-channel or variable that is being referenced by an OFF, ON, or TO keyword is out of range.</td>
</tr>
<tr>
<td>Bad re-assign Call...</td>
<td>These errors occur when the Central Controller attempts to re-assign the parameter variables in a DEFINE_CALL (page 16) to the parameters in the corresponding CALL (page 16) statement, and the assignment cannot be made.</td>
</tr>
<tr>
<td>Bad run token</td>
<td>This error occurs when the Central Controller does not understand a piece of data in the program it is executing.</td>
</tr>
<tr>
<td>Bad Set_Length...</td>
<td>These errors occur if the SET_LENGTH_STRING (page 149) keyword tries to set the length value of an array to a value greater than the array's storage capacity.</td>
</tr>
<tr>
<td>Bad While</td>
<td>This error occurs whenever a WHILE (page 48) loop terminates due to the half-second timeout imposed on WHILE loops.</td>
</tr>
</tbody>
</table>
Overview

This section explains the concept of Master-to-Master ("M2M") systems, and provides the information that must be understood to successfully deploy M2M systems. Most M2M systems can be successfully deployed using "route mode direct" and the appropriate topology. These two items are explained in detail in the subsections "Master routing" (page 159) and "Topologies" (page 162).

Master-to-Master

The functionality of M2M consists of master routing and inter-system control. Master routing is the ability to route messages to any other master or device and is the foundation of all M2M functionality. Inter-system control allows a master, or its NetLinx program, to control and get status of any other device (or master) that is connected to any other master. FIG. 7 depicts a typical system of two interconnected NetLinx control systems with several devices connected to each one:

FIG. 7 Two Interconnected NetLinx Control Systems

The top portion of the illustration in FIG. 7 shows the physical connections and the devices represented. The bottom portion shows the logical connections that have been assigned. In this example the NI-3100 will not communicate with the ENV-VST-C unless defined in the DEFINE_DEVICE section of its program code running on NI-3100 using the appropriate system number, for example 128:1:1. The first port on the MVP-8400i could be defined on system 1 using 10001:1:7, and on system 7 using 10001:1:7 or 10001:1:0.

Master Routing

By design, all NetLinx masters do not automatically make a M2M connection with other NetLinx masters by virtue of being on the same network. The connection between them must be made intentionally by adding them to a list. This connection list is called the "URL List". The URL List on the NetLinx master is used to force the master to initiate a TCP connection to the specified URL/IP address.

Any TCP/IP device, including NetLinx masters, which utilize DHCP to obtain its TCP/IP configuration, are subject to having their IP address change at any time. Therefore, NetLinx master’s IP address must be static unless the network supports Dynamic DNS AND a DHCP server capable of updating the DNS tables on behalf of the DHCP client. If a Dynamic DNS/DHCP server is available then the NetLinx master’s host name may be used in the URL List. Therefore, the first step in assembling a M2M system is to set unique system numbers on each master.

- Valid system numbers are 1 - 65535
- System 0 is a wildcard referring to the local system and is used within DEFINE_DEVICE and NetLinx Studio connections
The next step is to configure the URL List in either of the masters, but not both, to point to the other master. For example, in Illustration 1 NetLinx master system #1 could have its URL List configured with a single entry that contains the IP address of the NetLinx master system #7; this will establish a two-way connection.

The system #7 master does not need to have a URL entry to communicate with system #1. If the system #7 master’s URL List does contain the IP address for system #1 a routing loop will be created which will lead to problems (FIG. 8).

Once the systems are connected to each other they exchange routing information such that each master will learn about all the masters connected to each other. The implementation of master routing primarily involves the communication of routing tables between masters. The routing table is built using the entries within the local URL List, the DPS entries in the DEFINE_DEVICE section of the code, and from the routing tables exchanged between connected masters. Routing tables are exchanged between masters upon their initial connection and updates to the routing tables are exchanged periodically. Route table transmission has a certain amount of randomization built in to prevent flooding the network with routing table transmissions when a master reports online/offline. Each master in a network will add a minor random delay (1-5 seconds) so that they don’t all transmit at the same time.

There is no fixed limit on the number of entries in a routing table. The number of routes is dependent on the number of systems in the network for which there is no set limit. The only limit is the memory space in each master to maintain all of the system information of the various systems throughout the network.

**Route Modes (Normal and Direct)**

There are two route modes in which masters can be configured to share their routing table. The first and default is "normal", in this mode the master will share the entire routing table built from all interconnected masters. The second is "direct"; in this mode the master will share a routing table that only contains itself. When using "direct" mode the master will only connect with the masters that are one hop away. As a diagnostic aid, the "show route" command can be issued from a telnet session to show paths to other masters. Consider the following system of interconnected NetLinx masters (FIG. 9):

```
> show route
Route Data:
System Route Metric PhyAddress
---------------------------------
1  2  2  TCP Socket=18 IP=192.168.12.76 Index=3
2  2  1  TCP Socket=18 IP=192.168.12.76 Index=3
3  2  2  TCP Socket=18 IP=192.168.12.76 Index=3
4  4  1  TCP Socket=16 IP=192.168.12.80 Index=1
-> 5  5  0  Axlink
106 106 1  TCP Socket=19 IP=192.168.12.106 Index=2
111 106 2  TCP Socket=19 IP=192.168.12.106 Index=2
```
The "Route Data:" indicates which routing mode the master is using. When the master is configured for route mode "normal", nothing additional will be presented. When the master is configured for route mode "direct", the following note will appear. "Direct Connect Only Mode"

- The "->:" to the left of system number 5 indicates that system number 5 is the local system (i.e. the system that the telnet session is connected to).

- The Route column indicates which system number packets are to be routed to in order to get to their destination. For example, to send a message from system #5 to system #1 the message must be sent to/through system #2. You can see this visually in FIG. 9, or by examining the Route entry for System #1 in the "show route" table.

- The Metric column indicates the number of system masters that the message must transverse/hop in order to get to its destination. For the example above, the metric is 2 because the message must enter system #2, then system #1. Note that a metric of 16 or "Dead" indicates a route that is expected but does not exist. Further, since the maximum usable metric is 15 there is a limit of 16 masters in the width plus height of the master topology (see the Design Considerations, Constraints, and Topologies section on page 162).

- The PhyAddress column indicates the internal connection parameters used by the master to maintain the connection information.

  - "TCP Socket=" - This is the IP socket that is used for this connection. Refer to "show TCP" for additional information.
  - "IP=" - This is the IP address of the masters used for this connection point.
  - "Index=" - This is the order in which the connection was established. When the master contains the entry in its URL List this often represents the order they were entered into the list.

"Show Route" supports the "/v", verbose, parameter which will enable additional information about the routing table. This information is typically meaningful only to firmware engineering when diagnosing issues involving route table transmissions. The additional information available is described as:

- Current Time: The number of milliseconds since boot.
- Update Time: The milliseconds since boot when the next route table sync will occur.
- Normal Update Time: The milliseconds since boot when the next route table sync will occur.
- Triggered Update Time: The last time a triggered update occurred (ex. a new master came online, forcing a table update). If no triggered update has occurred, the field will say Max Time (effectively -1)
- Timeout Time: The time that the next route table sync should have occurred by.
- Next Update: • "Normal" This indicates the next update will occur at the "Normal Update Time"
  • "Triggered" indicates the next update is occurring due to a triggered event.
- Flags column: The Flags column indicates if the route to that master has "changed" during the last route reporting cycle. Upon the next reporting cycle with no new change, the field will be empty.

The end result of all this routing and connection data is that a device or master can communicate with other devices or masters regardless of the physical connection of the device. Note that masters may only be "connected" to each other via Ethernet/TCP/IP. As an example (see FIG. 7 on page 159), NetLinx Studio is running on a PC that is connected to System #7 as device number 32002. The routing capabilities of the NetLinx master allow NetLinx Studio to download IR codes to the NXC-IRS4 (S=7 D=24), download a master firmware upgrade to NetLinx master #1, and download new touch panel pages to the touch panel on master #1. All of this is possible simply by having NetLinx Studio connected to a NetLinx master with M2M firmware.
Design Considerations, Constraints, and Topologies

Design Considerations
When designing a system that will utilize the M2M functionality, there are multiple points to consider. The first thing to consider is the reason for using M2M. The most common reasons are:
- Expansion of a system to add device ports.
- Expansion of a system to an area the main system cannot reach.
- Sharing of processing load.
- Standalone capability of system areas.
- Isolation of areas for security reasons.
- Dedicate a master to common/shared devices located in a central location.
- Etc… a combination of the above.

The second thing to consider is the code requirements for each master:
- Masters that are only being used to add device ports must have an empty ".tkn" file loaded, otherwise the devices will not be accessible.
- Masters that are used to share the processing load or are intended to provide standalone capability must define its local devices and the specific remote devices needed on the other masters in DEFINE_DEVICE.
- Ports on remote devices declared in DEFINE_DEVICE must exist! For example, adding touch panel port 80 when the panel file that has been loaded only specifies 20 will cause errors in the negotiation.
- Events must be written for remote devices for the program to hear them. Writing events causes the master to negotiate for the transmission of these events over M2M (as reflected in SHOW NOTIFY).

The third thing to consider is the connection topology:
- Is there a main master who all other masters must connect with?
- Do all the masters need to talk to each other?
- Or is there some combination of the above?

Constraints
To properly configure the URL Lists in a multi-master system, there must be an understanding of 3 hard constraints.
1. The first constraint is the maximum number of 200 entries in a URL List. This limit although important will most likely never pertain as the second constraint is far more relevant.
2. The second constraint is the maximum number of 250 simultaneous TCP/IP connections supported by a single master. The maximum number of simultaneous TCP/IP ICSP (NetLinx device) connections supported by a single master is 200. The top ~25 of the remaining 50 are intended to be used for internal services i.e. ftp, telnet, http, etc…. The next 25 are intended to be used for IP connections used in the NetLinx code via IP_CLIENT_OPEN, IP_SERVER_OPEN, and Duet modules.
   If there are more than 25 IP connections made from within the code they will utilize the required number of remaining 200 IP sockets which reduces the number of available socket connections and subsequently the number of available NetLinx device connections which will reduce the number of available entries within the URL List.
3. The third constraint is the routing metric limit of 15 usable hops on the topology of the interconnected NetLinx masters. While the limit of 15 hops may seem very limiting, this is not really the case if you carefully design the topology. FIG. 10 provides a visual of the 15 hop limit.

Chain Topology
This topology shows 16 masters connected to each other such that any master is routeable to any other master.

The URL Lists would be configured like this:

<table>
<thead>
<tr>
<th>System Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master 1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

**NOTE:** The system number is being used here for readability, the actual URL/IP address must be entered into the URL List.

Using this topology can be both network and processor intensive as a message from system 1 to a device/port on system 16 must be passed between the 14 masters. For example, a serial string sent from within the code on system 1 to 5001:1:16 will be passed to system 2, and then to 3, etc. until it reaches system 16. Therefore the single serial string results in 15 messages across the network.
With an IO pulse from system 1 to a port on system 16 the following occurs; an ON message is passed to system 2, then to 3, ... until it reaches system 16, then the feedback on message sent back down the chain from system 16 to system 1, then a PUSH message from system 16 to system 1 following the same chain, then the OFF would be sent from system 1 to system 16, followed by a feedback off message from system 16 to system 1, then the RELEASE message from system 16 to system 1. Therefore that single pulse becomes 90 messages across the network. Another drawback to this topology is if a single master loses communication than all subsequent masters will cease communicating.

**Star Topology**

FIG. 11 shows the M2M system configured in a star topology to take advantage of the fact that each NetLinx master supports multiple connections to masters:

![Star Topology Diagram]

In a Star topology, the URL Lists would be configured as shown below:

<table>
<thead>
<tr>
<th>System Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master 1</td>
</tr>
<tr>
<td>Master 2</td>
</tr>
<tr>
<td>Master 3</td>
</tr>
<tr>
<td>Master 4</td>
</tr>
<tr>
<td>Master 5</td>
</tr>
<tr>
<td>Master 6</td>
</tr>
<tr>
<td>Master 7</td>
</tr>
<tr>
<td>Master 8</td>
</tr>
<tr>
<td>Master 9</td>
</tr>
<tr>
<td>Master 10</td>
</tr>
<tr>
<td>Master 11</td>
</tr>
<tr>
<td>Master 12</td>
</tr>
<tr>
<td>Master 13</td>
</tr>
<tr>
<td>Master 14</td>
</tr>
<tr>
<td>Master 15</td>
</tr>
<tr>
<td>Master 16</td>
</tr>
</tbody>
</table>

**NOTE:** The system number is being used here for readability, the actual URL/IP address must be entered into the URL List.

The largest drawback to this configuration is that if there is a communication issue with master 1 all other masters lose connection with each other.
Cluster Topology

Another possible connection topology is to establish communication hubs by combining the previously discussed topologies that optimize the traffic with adjacent masters but still allow connections to all other masters, as shown in FIG. 12:

![Clustered Master Interconnection Topology](image)

**FIG. 12** Clustered Master Interconnection Topology

In a Cluster topology, the URL Lists would be configured as shown below:

<table>
<thead>
<tr>
<th>System Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

**NOTE:** The system number is being used here for readability, the actual URL/IP address must be entered into the URL List.

When determining the interconnection topology of many NetLinx masters, special consideration should be made to have masters that communicate a lot of information with each other to connect to each other. Thus if you have two systems that share devices, control, or information they should probably be near each other in the topology and not at opposite ends of the connection matrix where each message is forced to pass through several NetLinx masters.

**NOTE:** Utilizing route mode direct will enable masters to isolate themselves from most traffic or to target the messages which will reduce network traffic and processor overhead.
**Cascade Topology**

FIG. 13 shows 16 masters connected to each other such that any master is routeable to any other master using route mode direct.

In a *Cascade* topology, the URL Lists would be configured as shown below:

<table>
<thead>
<tr>
<th>System Number</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>URL Entry</strong></td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>15</td>
</tr>
</tbody>
</table>

**NOTE:** The system number is being used here for readability, the actual URL/IP address must be entered into the URL List.

This topology has many advantages over the previously listed methods:
- Each master is able to see all the other masters, with one hop
- No passing of messages, which reduces the processing load on the master
- Robust, if one master goes down communication is lost with only that master and the devices connected to it
- Reduced network traffic
Cluster Topology Modified

FIG. 14 uses the Cluster concept and direct mode to link specific masters, yet remain isolated from other masters on the network.

The URL Lists would be configured like this:

<table>
<thead>
<tr>
<th>System Number</th>
<th>URL Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2, 7, 16, 12</td>
</tr>
<tr>
<td>2</td>
<td>3, 8, 13</td>
</tr>
<tr>
<td>3</td>
<td>4, 9, 14</td>
</tr>
<tr>
<td>4</td>
<td>5, 10, 15</td>
</tr>
<tr>
<td>5</td>
<td>6, 16</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
</tr>
</tbody>
</table>

NOTE: The system number is being used here for readability, the actual URL/IP address must be entered into the URL List.

Although this topology looks similar to the previous Cluster topology (see FIG. 12 on page 164), by using route mode the communication connections are very specific. Masters will only be able to communicate with masters that have an arrow between them. For example:

- The master with system 1 will only be able to communicate with masters 2, 3, 4, 5, 6, and 11, but will not connect with masters 7, 8, 9, 10, 12, 13, 14, 15, and 16.
- The connection, indicated with the red arrow, between master 10 and master 16 may appear to create a routing loop, but since the masters are configured to use route mode direct a loop is avoided.
- Master 10 will only be able to connect with masters 6 and 16.

The goal when using M2M is to minimize the amount of traffic between masters while providing the required functionality. Using route mode direct with the appropriate topology helps to accomplish this goal because it is the most efficient routing method since it will reduce network traffic and master processing of messages.
Configuring and Programming M2M Systems

Using NetLinx Studio with M2M Systems

NetLinx Studio can be used to configure and diagnose M2M systems. After you have connected NetLinx Studio to the master, and you have configured the master with the proper "Network Address" information, you will need to change the system number on the master via the Device Addressing dialog (FIG. 15):

To access this dialog in NetLinx Studio, select Diagnostics > Device Addressing… (or select the DPS icon from the Diagnostics toolbar).

NOTE: Once the System Number has been changed the master must be rebooted for the change to go into effect.

The next step is to configure the URL List, via the URL Listing and Add URL dialogs (FIG. 16).

The masters and devices in a M2M system can be viewed using the Refresh Network Online Tree option within the Online Tree (click the Display button in the Online Tree tab of the Workspace Bar to access the Online Tree menu). This function will run a recursive process that will connect to the master specified in the Communication Settings dialog, and gather information to populate the Online Tree. If there are any other masters in the routing table, NetLinx Studio will then connect to those masters and get their information until the end of each branch is reached. There are some limitations in diagnosing or watching devices/ports in a M2M system using NetLinx Studio. For example, if NetLinx Studio is connected to master system 1, and a connection is established to master system 2, then only the devices on system 2 defined within the code of system 1 will be accessible to watch via "Asynchronous Notifications".
Using Telnet with M2M Systems

Once the master's system number has been configured via NetLinx Studio, a telnet session can be used to configure and diagnose M2M systems. Note, when troubleshooting M2M systems NetLinx Studio and telnet connections to the master complement each other as the information from one application/interface may not be available from the other. The first command to become familiar with is to set the routing mode on the master. The command is "route mode" followed by the desired mode, direct or normal. The routing mode on the master can be verified by sending the command "route mode" with no parameter, or with the command "show route". The "show route" command is described in a previous section of this document.

To view the entries in the URL List use the command "show url". To modify the entries in the URL List use the command "set url". Both of these commands will accept a <D:P:S> parameter to view or modify URL Lists on other masters.

The command "show system" will display all the systems and devices that are online and tracked in the device manager. The device manager tracks all devices defined in DEFINE_DEVICE or used in DEFINE_EVENT. The "show system" command supports two mutually exclusive parameters. The "<S>" parameter displays the devices on the specified system. For example, when connected to system 1 issue the command "show system 2" to display the devices on System 2. Using the "<min>" parameter will limit the display to a minimal set of information.

There are two commands that are similar yet remain unique, they are "show remote" and "show notify".

- "show remote" displays the devices on a remote master that are being monitored by the local master.
- "show notify" displays the devices on the local master that are being monitored by a remote master.

The outputs of both commands are structured similarly and are described below. In the example below, "show remote" was issued on system 1. "show notify" was issued on system 16:

```
>show remote
Show Remote Device List
-----------------------
Device List of Remote Devices requested by this System
Device  Port  System  Needs
------------------------------------------------------
05001  00001  00016   Channels Commands Strings
05001  00005  00016   Channels Commands
33001  00001  00016   Channels Commands Strings Levels
```

```
>show notify
Show Notification List
----------------------
Device Notification List of devices requested by other Systems
Device:Port  System  Needs
------------------------------------------------------
05001:00001  00001   Channels Commands Strings
05001:00005  00001   Channels Commands
33001:00001  00001   Channels Commands Strings Levels
```

| Device column: The Device column lists the device that is being monitored. |
| Port column: The Port column lists the port on the device that is being monitored |
| System column: The System calling lists the system number that the device is connected to in the case of the "show remote". With "show notify" the system number that is watching the device will be listed. |
| Needs column: The Needs column contains the information that is being tracked. A device defined in "DEFINE_DEVICE" or used in "DEFINE_EVENT" will list the default needs “Channels Commands”. The “Strings” need will be listed if the device is used in a “DATA_EVENT” or “CREATE_BUFFER”. The “Levels” need will be listed if the device is used in a “LEVEL_EVENT” or “CREATE_LEVEL”. |

The command to view all of the TCP connections on a master is "show tcp". This command supports two parameters:

- The first parameter is "<v>" which stands for verbose, this does not appear to change the results.
- The second parameter is "<all>", this will display information about all 200 TCP/IP locations.

Control/NetLinx Language Support

The features of control to M2M include channel control (PUSH/RELEASE/ON/OFF/TO), level control, send commands and send strings. Channel controls allow one NetLinx master to PUSH/RELEASE a channel on a device of another system via the DO_PUSH/DO_RELEASE functions. Additionally, ON, OFF, TO, and feedback statements can control channels on devices of remote systems. If a channel has a characteristic modifier associated with it, that modifier still applies to the channel regardless of whether the channel is manipulated locally or remotely. For example, if a group of channels and variables is mutually exclusive then an ON to one of the channels will turn off all other channels and variables in the group prior to turning on the desired channel. Levels, strings and commands are essentially forwarded to the destination device. Note that control is not limited to physical devices and that NetLinx program defined virtual devices may also be manipulated by a remote system. This allows a local system to define a virtual device that can receive PUSH, RELEASE, ON, OFF, etc. and make programmatic decisions based upon that control. Additionally, notification of control messages is not limited to "main line" functions like PUSH and RELEASE; rather all EVENT based code will operate normally regardless of the source of the original control message/function.
Design Consideration and Constraints
In order to reference devices of other NetLinx systems, the devices **MUST** be defined in the `DEFINE_DEVICE` section of the NetLinx program. Conversely, only devices that are necessary should be placed in the `DEFINE_DEVICE` section to avoid any unnecessary network traffic between NetLinx masters.

- **DEFINE_LATCHING** – A remote device’s channel is not allowed in the `DEFINE_LATCHING` section.
- **DEFINE_MUTUALLY_EXCLUSIVE** – A remote device’s channel is not allowed in the `DEFINE_MUTUALLY_EXCLUSIVE` section.
- **DEFINE_TOGGLING** – A remote device’s channel is not allowed in the `DEFINE_TOGGLING` section.
- The proper way to modify a channel’s behavior is to use `ON/OFF/TO/PULSE`!
- **DEFINE_MODULE** – As a guideline the best practice is to run a UI module on the master that the touch panel or keypad is connected to, and to run the COMM module on the master that the device is connected to. This practice should limit the number of messages across the network as the amount of messages between the UI and COMM modules is generally smaller than the amount of messages between the device and the COMM module.

Inter-Master Variables
Inter-master variables are not implemented at this time. However the value of variables may be passed among the masters in the system using `SEND_COMMAND` or `SEND_STRING` to a common virtual device.

Using Virtual Devices as Moderators
Virtual Devices may be used as moderators to share information between masters that may or may not be related to specific devices, like passing the values of a variable. They can also be used to minimize the network traffic by using them to distribute the information to multiple devices on other masters.

Code Example: Tracking Online/Offline State In a Remote Master
```
DEFINE_DEVICE
SYSTEM4 = 33001:1:4
DEFINE_VARIABLE
INTEGER SYSTEM4_STATUS
DEFINE_EVENT
DATA_EVENT[SYSTEM4]
{
  ONLINE:
  {
    SYSTEM4_STATUS = 1
  }
  OFFLINE:
  {
    SYSTEM4_STATUS = 0
  }
}
```

Modifying the URL List From Within the NetLinx Code
There may be times when viewing or changing the URL List from within the NetLinx code is desired. This can be accomplished using the following functions “GET_URL_LIST”, “ADD_URL_ENTRY”, and “DELETE_URL_ENTRY”. Please refer to the NetLinx Keywords Help within NetLinx Studio for details and examples.

M2M Processing Queues and Troubleshooting
The Route Manager queue is the message queue that receives any inbound route table messages from other masters. These messages are then processed by the Route Manager firmware to update its tables, refer to the section above labeled “Master Routing”.

The Notification Manager queue is the message queue that receives notification requests for device state changes from a remote entity (ex. another master). In M2M communication, two connected masters do not blindly forward all local device state changes to the other master. Instead, they will only forward specifically requested state changes based on the remote master’s needs as defined in the NetLinx code, refer to the telnet commands “show remote” and “show notify”. The Notification manager queue receives these messages and then the Notification manager processes the requests and adds the information to its database, refer to the telnet commands “show remote” and “show notify”, of “requested” state changes. When a state change occurs in the master, it compares the change to its database and if a remote master has requested notification of the change, it forwards the state change to the remote master.

When configuring M2M systems it may be necessary to alter the queue sizes of the above mentioned queues. This can be done using the following telnet commands “show buffers”, “show max buffers”, and “set queue size”. To view the number of messages in each queue at a specific moment use the command “show buffers”. To monitor the largest number of messages in each queue since the master has booted use the command “show max buffers”. Use the command “set queue size” to determine and set the upper limit on each queue. If the information returned from “show max buffers” is equal to the upper limit of the queue, it would be appropriate to increase the upper limit of the queue size.

General M2M Issues
When multiple masters exist within a large NetLinx installation the significance of the System number component cannot be over emphasized. Out of habit it is easy to ignore the system field within NetLinx Studio because its value has not meant anything in standalone systems. A significant source of technical support phone calls will be directly related to invalid or unintentionally incorrect settings of the system number, URL List, or route mode. When NetLinx Studio connects to a single master, yet allows the user to access all other system masters it is inevitable that some confusion will occur. Therefore, it is a good idea to document each master’s system number and the topology of the interconnections!
Appendix C - Marshalling Protocol

Overview

The protocol assumes that every logical field (group of bytes) is prefixed with type/size information. For example, if there is a 4 byte long integer field within a structure, the byte stream representing that field consists of 5 bytes. The first byte (0xE3) specifies that a long integer follows and then the 4 remaining bytes contain the value of the long integer.

This concept is extended to all primitive, structure and array types. The type of a field is always stored as a single byte. The size of a field may or may not be stored depending upon the field type (fields with know lengths do not have a size prefix).

The specific formats of all the supported types are described in the table below.

Marshalled Stream Format

The following table describes the byte format of the various types supported in the marshaller (fields within <>'s indicate actual data bytes):

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Stream Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>Unsigned char/byte value.</td>
<td>0xE1 &lt;BYTE&gt;</td>
</tr>
<tr>
<td>WORD</td>
<td>Unsigned short value.</td>
<td>0xE2 &lt;WORD Hi&gt; &lt;WORD Lo&gt;</td>
</tr>
<tr>
<td>DWORD</td>
<td>4-byte value (could be an unsigned long integer or a float).</td>
<td>0xE3 &lt;DWORD MSB&gt; &lt;DWORD LSB&gt;</td>
</tr>
<tr>
<td>QWORD</td>
<td>8-byte value (could be an unsigned Quad-word or a double).</td>
<td>0xE4 &lt;QWORD MSB&gt; &lt;QWORD LSB&gt;</td>
</tr>
<tr>
<td>BYTESTR</td>
<td>Sequence of BYTE's whose element count is &lt;= 64K.</td>
<td>0xE5 Length Hi Length Lo &lt;BYTE Sequence&gt;</td>
</tr>
<tr>
<td>WORDSTR</td>
<td>Sequence of WORD's whose element count is &lt;= 64K.</td>
<td>0xE6 Length Hi Length Lo &lt;WORD Sequence&gt;</td>
</tr>
<tr>
<td>DWORDSTR</td>
<td>Sequence of DWORD's whose element count is &lt;= 64K.</td>
<td>0xE7 Length Hi Length Lo &lt;DWORD Sequence&gt;</td>
</tr>
<tr>
<td>QWORDSTR</td>
<td>Sequence of QWORD's whose element count is &lt;= 64K.</td>
<td>0xE8 Length Hi Length Lo &lt;QWORD Sequence&gt;</td>
</tr>
<tr>
<td>LBYTESTR</td>
<td>Large sequence of BYTE's whose element count can be &gt; 64K (larger version of BYTESTR).</td>
<td>0xE9 Length MSB Length LSB &lt; BYTE Sequence&gt;</td>
</tr>
</tbody>
</table>
Appendix C - Marshalling Protocol

Marshalling Protocol (Variables)

The protocol assumes that every logical field (variable) is identified with a name or index, type/size information and the actual data. For example, if there is a 4 byte long integer field within a structure, the XML stream representing that field would consist of 3 tags:

- A name tag specifying the name of the variable
- A type tag specifying a 4 byte unsigned value
- The data.

This concept is extended to all primitive, structure and array types. The type of a field is always stored using W3C standard type declarations. The type of the field is optional, as the data will be “stuffed” into whatever type matches the name of the parameter. The specific formats of all the supported types are described below.

Marshalled Stream Format

The following table describes the byte format of the various types supported in the XML marshaller.

### Types Supported in the XML Marshaller

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Stream Format</th>
</tr>
</thead>
</table>
| BYTE   | Unsigned char/byte value. If var is an element of an array, name is replaced with <index></index>. The index value, and the type are optional. Typically, only <var><data>Data</data></var> is needed. | <var>
  <name>MyName</name>
  <type>ui1</type>
  <data>255</data>
</var> |
| UWORLD | Unsigned short value. If var is an element of an array, name is replaced with <index></index>. The index value, and the type are optional. Typically, only <var><data>Data</data></var> is needed. | <var>
  <name>MyName</name>
  <type>ui2</type>
  <data>65535</data>
</var> |
| WORD   | Signed short value. If var is an element of an array, name is replaced with <index></index>. The index value, and the type are optional. Typically, only <var><data>Data</data></var> is needed. | <var>
  <name>MyName</name>
  <type>i2</type>
  <data>-32767</data>
</var> |
| ULONG  | 4-byte unsigned value. If var is an element of an array, name is replaced with <index></index>. The index value, and the type are optional. Typically, only <var><data>Data</data></var> is needed. | <var>
  <name>MyName</name>
  <type>ui4</type>
  <data>4294967295</data>
</var> |
| LONG   | 4-byte signed value. If var is an element of an array, name is replaced with <index></index>. The index value, and the type are optional. Typically, only <var><data>Data</data></var> is needed. | <var>
  <name>MyName</name>
  <type>i4</type>
  <data>-2147483647</data>
</var> |
| FLOAT  | 4-byte IEEE 754 float value. If var is an element of an array, name is replaced with <index></index>. The index value, and the type are optional. Typically, only <var><data>Data</data></var> is needed. | <var>
  <name>MyName</name>
  <type>float.IEEE.754.32</type>
  <data>1.23</data>
</var> |
| DOUBLE | 8-byte IEEE 754 float value. If var is an element of an array, name is replaced with <index></index>. The index value, and the type are optional. Typically, only <var><data>Data</data></var> is needed. | <var>
  <name>MyName</name>
  <type>float.IEEE.754.64</type>
  <data>4.56</data>
</var> |
### Types Supported in the XML Marshaller (Cont.)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>STRUCT</strong></td>
<td>A structure containing one or more fields. Each element within a structure is self-descriptive and can be any of the types in this table. If the struct is the outermost parent, then name is optional. If struct is an element of an array, name is replaced with <code>&lt;index&gt;&lt;/index&gt;</code> and the index value.</td>
<td><code>&lt;struct&gt; &lt;name&gt;&lt;MyName&gt;&lt;/name&gt; &lt;var&gt;… &lt;var&gt; &lt;/struct&gt;</code></td>
</tr>
<tr>
<td><strong>ARRAY</strong></td>
<td>Array of any one of the types in this table. Each element in an array is self-descriptive. The type of the parent is the type of the entire array. Type is optional and generally not included when the array is an array of structures. Current Length is optional. Array can contain a series of items, a series of structures or a series of array. Elements of an array should define an index instead of a name. This is the commonly used format for structures but all types are allowed.</td>
<td><code>&lt;array&gt; &lt;name&gt;&lt;MyName&gt;&lt;/name&gt; &lt;type&gt;Type&lt;/type&gt; &lt;curLength&gt;100&lt;/curLength&gt; &lt;var&gt; &lt;index&gt;1&lt;/index&gt;… &lt;var&gt; &lt;/array&gt;... or... &lt;array&gt; &lt;name&gt;&lt;MyName&gt;&lt;/name&gt; &lt;type&gt;Type&lt;/type&gt; &lt;curLength&gt;100&lt;/curLength&gt; &lt;struct&gt; &lt;index&gt;1&lt;/index&gt;… &lt;/struct&gt; &lt;/array&gt;... or... &lt;array&gt; &lt;name&gt;&lt;MyName&gt;&lt;/name&gt; &lt;type&gt;Type&lt;/type&gt; &lt;curLength&gt;100&lt;/curLength&gt; &lt;struct&gt; &lt;index&gt;1&lt;/index&gt;… &lt;/struct&gt; &lt;/array&gt;</code></td>
</tr>
<tr>
<td>Array - String encoding (Strings)</td>
<td>Array of unsigned characters. Data is encoded using String encoding. Type and length are optional.</td>
<td><code>&lt;array&gt; &lt;name&gt;&lt;MyName&gt;&lt;/name&gt; &lt;type&gt;Type&lt;/type&gt; &lt;curLength&gt;100&lt;/curLength&gt; &lt;string&gt;MyString&lt;/string&gt; &lt;/array&gt;</code></td>
</tr>
<tr>
<td>Array - Binary Encoded</td>
<td>Array of any one of the types in this table except structures. This is the default for all non-CHAR arrays but CHAR arrays can use this encoding as well. The type of the parent is the type of the entire array. Type is optional and generally not included. Current Length is optional. Style is LE for little endian or BE for big endian. BE is the default. Size indicates the byte size but not the type. ByteSize=4 is used for LONG, SLONG, and FLOAT and means that 8 nibbles will be present for each element being encoded/decoded.</td>
<td><code>&lt;array&gt; &lt;name&gt;&lt;MyName&gt;&lt;/name&gt; &lt;type&gt;Type&lt;/type&gt; &lt;curLength&gt;100&lt;/curLength&gt; &lt;encoded&gt; &lt;style&gt;LE or BE&lt;/style&gt; &lt;size&gt;1,2,4,8&lt;/size&gt; &lt;data&gt;01020304&lt;/data&gt; &lt;/encoded&gt; &lt;/array&gt;</code></td>
</tr>
</tbody>
</table>

### Encoding Notes
- The encoding XML will not contain any white space. This includes CR,LF pairs.
- The decoding XML may contain white spaces. They will be ignored according to standard XML rules (i.e. Spaces as between tags are read.)
- Array may be encoded or decoded as binary encoded data
- XML comments, `<!-- -->`, will be ignored in decode.

### String Encoding

NetLinx has no native string type, but since it is a common type the encoding/decoding of the string data will be logically handled so the XML remains concise. CHAR arrays will be encoded/decoded as a string type, printable ASCII characters appear as ASCII, and non-printable characters appear as escaped decimal or hex code, `\&#{decimal code};` or `\&#{hex code};`. An example string would be:

```xml
<data>My Name is Jimmy Buffet &#x0D;</data> - or - <data>My Name is Jimmy Buffet &#13;</data>
```

Additionally, some characters have a more readable syntax. These characters are invalid in XML; so, the following characters can be encoded in the above format or the following format:
Appendix C - Marshalling Protocol

Binary Array Encoding

Arrays can optionally be encoded/decoded as pairs of ASCII-encoded HEX. The pairs of ASCII-encoded HEX needs to be padded to the size of the data so a 4-byte data value needs to have 4 bytes that represent it. There are no spaces between pairs, and the default is Big-Endian. Little Endian can be encoded or decoded as an option. The HEX letters may appear as upper or lower case and are by default upper case. Any example of a 2-byte (signed or unsigned) array containing the value 1,2,3,4,1,12,13,14 is:

<encoded>
  <size>2</size>
  <data>010203040B0C0D0E</data>
</encoded>

This is the default type of encoding for non-CHAR arrays but can be used to encode/decode char arrays as well. The data section must contain BytesSize*Elements nibbles.

Binary Encoding Result

<table>
<thead>
<tr>
<th>Character</th>
<th>Escape Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>&quot;</td>
<td>&amp;</td>
</tr>
<tr>
<td>'</td>
<td>'</td>
</tr>
<tr>
<td>=</td>
<td>&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Character</th>
<th>Escape Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>&quot;</td>
<td>&amp;</td>
</tr>
<tr>
<td>'</td>
<td>'</td>
</tr>
<tr>
<td>=</td>
<td>&quot;</td>
</tr>
</tbody>
</table>

**Binary Encoding Result**

<table>
<thead>
<tr>
<th>Byte In Encoded String</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$EC</td>
<td>Start of Array Encoding</td>
</tr>
<tr>
<td>$00 $00 $00 $03</td>
<td>Number of Elements in the Array</td>
</tr>
<tr>
<td>$EA</td>
<td>Start of Structure</td>
</tr>
<tr>
<td>$E3</td>
<td>DWORD: LONG or SLONG</td>
</tr>
<tr>
<td>$00 $A9 $63 $48</td>
<td>Data: 11101000</td>
</tr>
<tr>
<td>$E5</td>
<td>Start of CHAR Array (String)</td>
</tr>
<tr>
<td>$00 $0D</td>
<td>Length of Array: 13</td>
</tr>
<tr>
<td>$42 $75 $66 $66 $65 $74 $2C $20 $4A $69 $6D $6D $79</td>
<td>Data: 'Buffet, Jimmy'</td>
</tr>
<tr>
<td>$E5</td>
<td>Start of CHAR Array (String)</td>
</tr>
<tr>
<td>$00 $1A</td>
<td>Length of Array: 26</td>
</tr>
<tr>
<td>$4C $69 $76 $69 $6E $67 $20 $26 $20 $44 $79 $69 $6E $67 $20 $69 $6E $20 $33 $2F $34 $20 $54 $69 $6D $6D</td>
<td>Data: 'Living &amp; Dying in 3/4 Time'</td>
</tr>
<tr>
<td>$E5</td>
<td>Start of CHAR Array (String)</td>
</tr>
<tr>
<td>$00 $03</td>
<td>Length of Array: 3</td>
</tr>
<tr>
<td>$4D $43 $41</td>
<td>Data: 'MCA'</td>
</tr>
<tr>
<td>$E5</td>
<td>Start of CHAR Array (String)</td>
</tr>
<tr>
<td>$00 $03</td>
<td>Length of Array: 3</td>
</tr>
<tr>
<td>$4D $43 $41</td>
<td>Data: 'MCA'</td>
</tr>
<tr>
<td>$E5</td>
<td>Start of CHAR Array (String)</td>
</tr>
<tr>
<td>$00 $04</td>
<td>Length of Array: 4</td>
</tr>
<tr>
<td>$31 $39 $37 $34</td>
<td>Data: '1974'</td>
</tr>
<tr>
<td>$E2</td>
<td>WORD: INTEGER or SINTEGER</td>
</tr>
<tr>
<td>$00 $0B</td>
<td>Data: 11</td>
</tr>
<tr>
<td>$E5</td>
<td>Start of CHAR Array (String)</td>
</tr>
<tr>
<td>$00 $0A</td>
<td>Length of Array: 10</td>
</tr>
<tr>
<td>$33 $31 $33 $32 $33 $33 $33 $34 $33 $34 $33 $35</td>
<td>Data: '3132333435'</td>
</tr>
<tr>
<td>$E2</td>
<td>WORD: INTEGER or SINTEGER</td>
</tr>
<tr>
<td>$00 $5B</td>
<td>Data: 91</td>
</tr>
<tr>
<td>$EB</td>
<td>End of Structure</td>
</tr>
<tr>
<td>$EA</td>
<td>Start of Structure</td>
</tr>
<tr>
<td>$E3</td>
<td>DWORD: LONG or SLONG</td>
</tr>
<tr>
<td>$01 $07 $2F $E5</td>
<td>Data: 17248229</td>
</tr>
</tbody>
</table>
## Binary Encoding Result (Cont.)

<table>
<thead>
<tr>
<th>Byte In Encoded String</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$00 $0D</td>
<td>Length of Array: 13</td>
</tr>
<tr>
<td>$42 $75 $66 $66 $65 $74 $2C $20 $4A $69 $6D $6D $79</td>
<td>Data: 'Buffet, Jimmy'</td>
</tr>
<tr>
<td>$E5</td>
<td>Start of CHAR Array (String)</td>
</tr>
<tr>
<td>$00 $15</td>
<td>Length of Array: 21</td>
</tr>
<tr>
<td>$4F $66 $66 $20 $74 $6F $20 $53 $65 $65 $20 $74 $68 $65 $20 $4C $69 $7A $61 $72 $64</td>
<td>Data: 'Off to See the Lizard'</td>
</tr>
<tr>
<td>$E5</td>
<td>Start of CHAR Array (String)</td>
</tr>
<tr>
<td>$00 $03</td>
<td>Length of Array: 3</td>
</tr>
<tr>
<td>$4D $43 $41</td>
<td>Data: 'MCA'</td>
</tr>
<tr>
<td>$E5</td>
<td>Start of CHAR Array (String)</td>
</tr>
<tr>
<td>$00 $03</td>
<td>Length of Array: 3</td>
</tr>
<tr>
<td>$4D $43 $41</td>
<td>Data: 'MCA'</td>
</tr>
<tr>
<td>$E5</td>
<td>Start of CHAR Array (String)</td>
</tr>
<tr>
<td>$00 $04</td>
<td>Length of Array: 4</td>
</tr>
<tr>
<td>$31 $39 $38 $39</td>
<td>Data: '1989'</td>
</tr>
<tr>
<td>$E2</td>
<td>WORD: INTEGER or SINTEGER</td>
</tr>
<tr>
<td>$00 $0B</td>
<td>Data: 11</td>
</tr>
<tr>
<td>$E5</td>
<td>Start of CHAR Array (String)</td>
</tr>
<tr>
<td>$00 $0A</td>
<td>Length of Array: 10</td>
</tr>
<tr>
<td>$33 $31 $33 $32 $33 $33 $33 $34 $33 $36</td>
<td>Data: '3132333436'</td>
</tr>
<tr>
<td>$E2</td>
<td>WORD: INTEGER or SINTEGER</td>
</tr>
<tr>
<td>$00 $69</td>
<td>Data: 105</td>
</tr>
<tr>
<td>$EB</td>
<td>End of Structure</td>
</tr>
<tr>
<td>$EA</td>
<td>Start of Structure</td>
</tr>
<tr>
<td>$E3</td>
<td>DWORD: LONG or SLONG</td>
</tr>
<tr>
<td>$00 $BC $1E $A4</td>
<td>Data: 12328612</td>
</tr>
<tr>
<td>$E5</td>
<td>Start of CHAR Array (String)</td>
</tr>
<tr>
<td>$00 $0D</td>
<td>Length of Array: 13</td>
</tr>
<tr>
<td>$42 $75 $66 $66 $65 $74 $2C $20 $4A $69 $6D $6D $79</td>
<td>Data: 'Buffet, Jimmy'</td>
</tr>
<tr>
<td>$E5</td>
<td>Start of CHAR Array (String)</td>
</tr>
<tr>
<td>$00 $05</td>
<td>Length of Array: 5</td>
</tr>
<tr>
<td>$41 $2D $31 $2D $41</td>
<td>Data: 'A-1-A'</td>
</tr>
<tr>
<td>$E5</td>
<td>Start of CHAR Array (String)</td>
</tr>
<tr>
<td>$00 $03</td>
<td>Length of Array: 3</td>
</tr>
<tr>
<td>$4D $43 $41</td>
<td>Data: 'MCA'</td>
</tr>
<tr>
<td>$E5</td>
<td>Start of CHAR Array (String)</td>
</tr>
<tr>
<td>$00 $04</td>
<td>Length of Array: 4</td>
</tr>
<tr>
<td>$31 $39 $37 $34</td>
<td>Data: '1974'</td>
</tr>
<tr>
<td>$E2</td>
<td>WORD: INTEGER or SINTEGER</td>
</tr>
<tr>
<td>$00 $0B</td>
<td>Data: 11</td>
</tr>
<tr>
<td>$E5</td>
<td>Start of CHAR Array (String)</td>
</tr>
<tr>
<td>$00 $0A</td>
<td>Length of Array: 10</td>
</tr>
<tr>
<td>$33 $31 $33 $32 $33 $33 $33 $34 $33 $37</td>
<td>Data: '3132333437'</td>
</tr>
<tr>
<td>$E2</td>
<td>WORD: INTEGER or SINTEGER</td>
</tr>
<tr>
<td>$00 $8D</td>
<td>Data: 189</td>
</tr>
<tr>
<td>$EB</td>
<td>End of Structure</td>
</tr>
</tbody>
</table>
XML Encoding Result

```xml
<array>
  <curLength>0</curLength>
  <STRUCT>
    <index>1</index>
    <var>
      <name>LTITLEID</name>
      <data>11101000</data>
    </var>
    <array>
      <name>SARTIST</name>
      <curLength>13</curLength>
      <string>Buffet, Jimmy</string>
    </array>
    <array>
      <name>STITLE</name>
      <curLength>26</curLength>
      <string>Living &amp; Dying in 3/4 Time</string>
    </array>
    <array>
      <name>SCOPYRIGHT</name>
      <curLength>3</curLength>
      <string>MCA</string>
    </array>
    <array>
      <name>SLABEL</name>
      <curLength>3</curLength>
      <string>MCA</string>
    </array>
    <array>
      <name>SRELEASEDATE</name>
      <curLength>4</curLength>
      <string>1974</string>
    </array>
    <var>
      <name>NNUMTRACKS</name>
      <data>11</data>
    </var>
    <array>
      <name>SCODE</name>
      <curLength>10</curLength>
      <string>3132333435</string>
    </array>
    <var>
      <name>NDISCNUMBER</name>
      <data>91</data>
    </var>
  </STRUCT>
  <STRUCT>
    <index>2</index>
    <var>
      <name>LTITLEID</name>
      <data>17248229</data>
    </var>
    ...  
    <var>
      <name>NDISCNUMBER</name>
      <data>105</data>
    </var>
  </STRUCT>
  <STRUCT>
    <index>3</index>
    <var>
      <name>LTITLEID</name>
      <data>12326612</data>
    </var>
    ...  
    <var>
      <name>NDISCNUMBER</name>
      <data>189</data>
    </var>
  </STRUCT>
</array>
```
Appendix D - NetLinx vs. Axcess

Overview

The NetLinx control system was designed to upgrade the processor bus and improve the power of the Axcess programming language. Originally named Axcess2, the NetLinx was designed to be a superset of the Axcess programming language. The relationship between the new language (NetLinx) and Axcess is very similar to the relationship between C++ and C. Just as C++ brought a whole new level of power to C programming, NetLinx offers a variety of new tools and commands to dynamically increase the speed and power of present and future applications. NetLinx contains all of the elements of Axcess. Largely, you can use the same code from Axcess systems in NetLinx systems. Some exceptions include variable names conflicting with new NetLinx keywords; however, Axcess keywords are valid in NetLinx. You cannot compile NetLinx code on an Axcess compiler, or download NetLinx code to an Axcess control system. To upgrade an existing Axcess control system to NetLinx you must upgrade the Axcess Master to a NetLinx Master. You can still use the existing Axcess equipment as long as you can disable the existing Axcess Central Controller.

NOTE: The exceptions are the Axcent, the Axcent2, the AXB-EM232, and the AXB-MPE+ Master Port Expander. None of these integrated controllers allow you to disable the Central Controller. Both Axcess Card Frame Systems and Axcent3 systems allow you to either remove or disable the Axcess Central Controller.

NOTE: If you are using an Axcent3 / Axcent3 Pro, you can disable the Master with the Open Axcess program. You can connect the Axcent3 / Axcent3 Pro to a NetLinx Master Module via AxLink. Then you can compile and download the existing Axcess code.

Several Axcess control limitations have been fixed in NetLinx.

- NetLinx expands the types of data and variables from Axcess.
- NetLinx provides multiple processes and event threads beyond the Mainline in Axcess.
- NetLinx offers more options in distributed processing. NetLinx expands and strengthens Master-to-Master communications and expands the traditional AxLink bus to include ICSNet and Ethernet Network communications.

Axcess is linear in its process. At run time, Axcess runs the DEFINE_START code once when the system is loaded or restarted. Axcess then makes a pass through mainline code, polls the bus for activity, checks the wait and pulse stacks, and repeats the process again. The length of mainline and the activity on the bus affect runtime speeds. The mainline process is considered a single thread. NetLinx runs on multiple threads; mainline and event handlers run on parallel threads. Event handlers are defined within NetLinx and operate like mini-mainlines. They contain far less code and run faster than mainline. If an event occurs, and an event handler has been defined for that event, NetLinx bypasses mainline and runs the event handler.

NetLinx vs. Axcess - Comparison by Structure

<table>
<thead>
<tr>
<th>DEFINE DEVICE</th>
<th>NetLinx Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axcess Language</td>
<td>NetLinx defines the device by Device:Port:System.</td>
</tr>
<tr>
<td>Axcess defines devices with a single number (sometimes called an address) from 1 to 255. Axcess permits a maximum of 255 devices on the AxLink bus.</td>
<td>Device is a 16-bit integer representing the device number. Physical devices range from 1 to 32,000. Virtual devices range from 32,768 to 36,863.</td>
</tr>
<tr>
<td>DEFINE_DEVICE</td>
<td>NOTE: These numbers do not seem so random when represented in hexadecimal. Physical devices range from $0001 to $7FFF. Virtual devices range from $8000 to $8FFF.</td>
</tr>
<tr>
<td>VCR = 1 (* AXC-IRS *)</td>
<td>Port is a 16-bit integer representing the port number in a range of 1 through the number of ports on the device (1 = this port)</td>
</tr>
<tr>
<td>VPROJ= 2 (* AXC-IRS *)</td>
<td>System is a 16-bit integer representing the system number (0 = this system)</td>
</tr>
<tr>
<td>TP = 128 (* AXT-CA10*)</td>
<td></td>
</tr>
<tr>
<td>DEFINE_CONSTANT</td>
<td>NetLinx processes constants just like Axcess. NetLinx also allows you to define an expression in the DEFINE_CONSTANT section. The expression cannot contain any variables.</td>
</tr>
<tr>
<td>Axcess defines constants as either a fixed integer value between 0 and 65,535 or an array with a maximum length of 255 bytes in which each element can hold a value from 0 to 255. These values can be expressed in ASCII, Decimal, or Hexadecimal.</td>
<td></td>
</tr>
<tr>
<td>DEFINE_CONSTANT</td>
<td></td>
</tr>
<tr>
<td>VALUE_MAX = 140</td>
<td>VALUE_MIN = 40</td>
</tr>
<tr>
<td>DEFAULT_NAME = 'Axcess'</td>
<td>DEFAULT_NAME = 'Axcess'</td>
</tr>
<tr>
<td>ETX = &quot;$FE,$FF&quot;</td>
<td>ETX = &quot;$FE,$FF&quot;</td>
</tr>
<tr>
<td>VALUE_MAX = VALUE_MIN + 100</td>
<td>VALUE_MAX = VALUE_MIN + 100</td>
</tr>
</tbody>
</table>
**DEFINE VARIABLES**

### Axcess Language

Axcess supports 5 types of variables:
- **Integer Variables** (default) can contain a value from 0 to 65,535.
- **Character Arrays** are single element arrays, in which each element has a value from 0 to 255 with a maximum of 255 elements.
- **2-Dimensional Arrays** equate to a maximum of 255 single element character arrays. Each element can have a value from 0 to 255.
- **Integer Arrays** are single element arrays, in which each element can contain a value from 0 to 65,535 with a maximum of 255 elements.
- **2-Dimensional Integer Arrays** may have a maximum value of 65,535.

Variables are Non-Volatile (the variable loses its value when the program is loaded, but retains its value if the controller is reset).

**DEFINE_VARIABLE**
- VALUE
- ARRAY[3]
- ARRAY_2DIM[4][6]
- INTEGER INT_ARRAY[6]

### NetLinx Language

NetLinx substantially increased the number of supported variable types. In addition to more data types, NetLinx also supports Sets, Structures, and Multi-dimensional arrays.

Arrays default to Character Arrays. Variables default to Integer Variables. Variables default to Non-Volatile, but can be set as Non-Volatile or Volatile (Volatile variables are initialized when code is loaded or when the system is reset).

**DEFINE_VARIABLE**
- CHAR
- VALUE1
- WIDECHAR
- BIGCHAR
- INTEGER
- VALUE2
- SINTENSOR
- SIGNED1
- LONG
- BIGVALUE
- SLONG
- SIGNED2
- FLOAT
- DECIMAL
- DOUBLE
- VERYBIGVALUE
- INTEGER
- ARRAY[3][3][3]
- VOLATILE
- INTEGER
- RESET_VAR

**DEFINE_CALL (Subroutines)**

### Axcess Language

Axcess provides two methods for incorporating subroutines into your program.
- **DEFINE_CALL** subroutines are defined in the program and support parameter passing into the call. Changing the parameter value inside the call changes the value of the variable passed to the parameter. The DEFINE_CALL can use global variables or defined local variables. DEFINE_CALL is for standalone statements and cannot be used in-line as an expression.
- **SYSTEM_CALL** is an externally defined subroutine with a `.LIB` extension. SYSTEM_CALL programs are produced by AMX and are available on CD-ROM and on the Tech Support Web site at www.amx.com.

**DEFINE_CALL 'SWITCH' (CARD,IN,OUT)**

```
{SEND_STRING CARD,
 "ITOA(IN),'*','ITOA(OUT),'!'"
}
```

**DEFINE_CALL 'MULTIPLY' (X,Y,RESULT)**

```
RESULT = X * Y
```

**DEFINE_PROGRAM**

```
PUSH[TP,11]
{CALL 'SWITCH' (SWITCHER,4,1)
}
PUSH[TP,12]
{CALL 'MULTIPLY' (3,4,VALUE)
}
SYSTEM_CALL [1] 'VCR1'
(VCR,TP,21,22,23,24,25,26,27,28,0)
```

### NetLinx Language

Like Axcess, NetLinx supports DEFINE_CALL and SYSTEM_CALL. NetLinx also supports functions, which are similar to a DEFINE_CALL(s). They can be used standalone or in-line as an expression. Functions are defined in the DEFINE_CALL section of the code as a global function.

Defining a function differs slightly from a DEFINE_CALL:
- The data type of the function's return value must be specified.
- The function name is not enclosed with quotes or case sensitive.

**DEFINE_FUNCTION INTEGER MULTIPLY**

```
(INTEGER X, INTEGER Y)
{
RETURN (X * Y)
}
```

**DEFINE_PROGRAM**

```
PUSH[TP,11]
{CALL 'SWITCH' (SWITCHER,4,1)
}
PUSH[TP,12]
{VALUE = MULTIPLY(3, 4)
}
SYSTEM_CALL [1] 'VCR1'
(VCR,TP,21,22,23,24,25,26,27,28,0)
```

**DEFINE_START**

### Axcess Language

**DEFINE_START** sets the initialization parameters for the Axcess program. This section defines buffers, levels, sets communication settings, and initializes variables. **DEFINE_START** is run once when the program is loaded or the system is reset.

**DEFINE_START**

```
CREATE_BUFFER TP, TP_BUFFER
CREATE_LEVEL VOL, 1, VOL_LEVEL1
SEND_COMMAND SWT,
 "SET BAUD 9600,N,8,1,DISABLE"
ON[CLEAR_TO_SEND]
```

### NetLinx Language

There is no difference between the way Axcess and NetLinx handle the **DEFINE_START** section of the program; however, the role of the **DEFINE_START** section is greatly reduced. Variable initializations are handled in the **DEFINE_VARIABLE** section. Device initializations are handled with a **DATA_EVENT** in the **DEFINE_EVENT** section.

**DEFINE_START**

```
ON[CLEAR_TO_SEND]
```
### DEFINE_EVENT

<table>
<thead>
<tr>
<th>Axcess Language</th>
<th>NetLinx Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axcess does not support events.</td>
<td>Events are a new process in NetLinx. The events thread runs parallel to the mainline thread. Events describe certain types of conditions within the control system. If the conditions are defined as a DEFINE_EVENT, the event code is run and mainline is bypassed. There are five different types of events: Button Events, Channel Events, Data Events, Level Events, and Timeline Events.</td>
</tr>
<tr>
<td>DEFINE_EVENT</td>
<td></td>
</tr>
<tr>
<td>BUTTON_EVENT[TP,21]</td>
<td>DEFINE_PROGRAM</td>
</tr>
<tr>
<td>(* KC REPEAT 'A' *)</td>
<td></td>
</tr>
<tr>
<td>{ PUSH:</td>
<td></td>
</tr>
<tr>
<td>{ SEND_STRING KC, 'A'</td>
<td></td>
</tr>
<tr>
<td>} RELEASE:</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>HOLD[5,REPEAT]:</td>
<td></td>
</tr>
<tr>
<td>{ SEND_STRING KC, 'A'</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

### DEFINE_PROGRAM

<table>
<thead>
<tr>
<th>Axcess Language</th>
<th>NetLinx Language</th>
</tr>
</thead>
</table>
| The DEFINE_PROGRAM or mainline section of the Axcess program is where most of the programming process takes place. Axcess supports 99 reserved identifiers or keywords. 83 of these keywords can be used in the mainline. Axcess runs through a loop where:  
- The AxLink bus is queried for any changes.  
- Mainline code is run.  
- Axcess checks the wait stack and the pulse stacks for any expired waits and pulses.  
- The process is repeated. | The DEFINE_PROGRAM or mainline section of the NetLinx program and the DEFINE_EVENT section of code are responsible for processing events in a NetLinx system. NetLinx has expanded the list of keywords to 194 reserved identifiers. NetLinx also supports loops, data conversions, string processing, and file handling. NetLinx handles mainline in a similar fashion to Axcess, with a couple of differences. Because NetLinx supports multiple bus formats (AxLink, ICSNet, and Ethernet), events and changes in bus status are handled through a connection manager and message queue. NetLinx checks the message queue to see if an event is defined for the message. If not, NetLinx makes a pass through mainline. When NetLinx finishes the event handler or mainline, NetLinx processes the Wait list and Pulse list, and returns to the message queue to start the process again. |
Axcess/NetLinx Incompatibility

According to previous versions of each of their language reference manuals, Axcess and NetLinx each give the operator NOT highest precedence while giving AND and OR lowest. As demonstrated in the following code, however, the two systems behave differently. In reality, Axcess gives the operator NOT lowest precedence.

```
DEFINE_VARIABLE C D E
DEFINE_CALL 'GO' (A,B)
{
   C = !A && B
   D = B && !A
   E = !B && !A
}
DEFINE_PROGRAM
PUSH[1,1]
   CALL 'GO' (0,0)
PUSH[1,2]
   CALL 'GO' (1,0)
PUSH[1,3]
   CALL 'GO' (0,1)
PUSH[1,4]
   CALL 'GO' (1,1)

Axcess RESULTS
A   B   !A && B   B && !A   !B && !A
  0   0     1      0       1
  1   0     0      1       1
  0   1     1      1       0
  1   1     0      0       1

NETLINX RESULTS
A   B   !A && B   B && !A   !B && !A
  0   0     0      0       1
  1   0     0      0       0
  0   1     1      1       0
  1   1     0      0       0
```

The problem applies whether A and B are channels, variables, or expressions, and for OR as well as AND. To solve the problem, AMX always recommends the use of (!A) && B instead of !A && B; however, and this is critical, some programs out there are taking advantage of the logic flaw. Where the Axcess programmer intended the truth table of !(A && B) he/she may have coded !A && B and gotten the desired result. If these systems are converted to NetLinx Masters, the logic will not work as desired.

Please be aware of this difference as you support programs being converted from Axcess to NetLinx. When it occurs, Axcess-like operation can generally be achieved by including all the conditions to the right of the NOT in a single set of parentheses. For example:

```
IF (SYSTEM_POWER && !([VCR,PLAY] || [VCR,RECORD]))
```

becomes:

```
IF (SYSTEM_POWER && !([VCR,PLAY] || [VCR,RECORD]))
```

Combining Devices, Channels and Levels

Axcess allows you to combine devices and levels within the DEFINE_COMBINE and DEFINE_CONNECT_LEVEL sections. This method is static and is fixed when the program compiles. You can combine functionality within mainline by stacking push and release statements. Stacking pushes allows you the flexibility to conditionally change what elements of the program share functionality, but the program can be more difficult to maintain over time than if the panels were combined using DEFINE_COMBINE.

NetLinx provides several new methods for combining the functionality of devices, channels, and levels. Using DEV, DEVCHAN and DEVLEV accomplishes the same thing as stacking pushes in Axcess, and it reduce the overall maintenance associated with stacking pushes; however, data sets are statically implemented within the DEFINE_EVENT section. When the program compiles, the references to the data sets in the DEFINE_EVENT are set and cannot change at run time.

Virtual devices, levels and device/channel sets

One of the drawbacks to combining devices and levels in Axcess is the way the central controller handled the first device in the combine list going online and offline. This resulted in unexpected device behavior and inconsistent feedback.

NetLinx uses virtual devices. Virtual devices carry a device number ranging from 32,768 to 36,863, a port number of 1, and a system number of 0. Virtual Devices are devices that cannot be taken off the bus. By listing a virtual device as the first device in a DEFINE_COMBINE, COMBINE_DEVICES, COMBINE_LEVELS, or COMBINE_CHANNELS statement, the abnormalities seen in Axcess DEFINE_COMBINE statements are eliminated.

Combining and Uncombining devices

NetLinx still recognizes the DEFINE_COMBINE section. This section still operates as it did in Axcess; however, once the DEFINE_COMBINE section has been compiled it remains static. NetLinx introduces two functions: COMBINE_DEVICES and UNCOMBINE_DEVICES. COMBINE_DEVICES and UNCOMBINE_DEVICES dynamically change the devices combined together. When devices are combined the combine list and DEV set lists are reevaluated and updated during run time. COMBINE_DEVICES and UNCOMBINE_DEVICES are used as stand-alone statements in an event, mainline or in assignment statements. COMBINE_DEVICES and UNCOMBINE_DEVICES will return a value of 0 or -1, depending on the success or failure of the operation. The first device in a COMBINE_DEVICES statement should be a virtual device.
The devices, listed after the virtual device, are either a list of individual device numbers, DEV sets, or any combination of devices and DEV sets. The UNCOMBINE_DEVICES statement requires only the first device in the COMBINE_DEVICES list, which should be a virtual device. The format for COMBINE_DEVICES and UNCOMBINE_DEVICES is:

```
COMBINE_DEVICES (<virtual device>, <device1>, <device2>..)
UNCOMBINE_DEVICES (<virtual device>)
```

Devices combined with COMBINE_DEVICES respond like devices combined using the DEFINE_COMBINE section. The central controller recognizes any input from the devices in the combine list as the first device in the list.

**Combining and Uncombining levels**

The NetLinx functions COMBINE_LEVELS and UNCOMBINE_LEVELS work similar to the DEFINE_CONNECT_LEVEL section in Axcess. For compatibility with Axcess code, the DEFINE_CONNECT_LEVEL section is still valid. Like COMBINE_DEVICES, COMBINE_LEVELS and UNCOMBINE_LEVELS can be used within events and mainline code to dynamically change what levels are connected to each other. It is also recommended that a Virtual DEVLEV set be used as the first DEVLEV set in the COMBINE_LEVELS function. The format for COMBINE_LEVELS and UNCOMBINE_LEVELS is:

```
COMBINE_LEVELS (<virtual DEVLEV>, <DEVLEV1>, <DEVLEV2>..)
UNCOMBINE_LEVELS (<virtual DEVLEV>)
```

DEVLEV structures defined within the COMBINE_LEVELS are either individual DEVLEV structures or one dimension of a DEVLEV array. Any reference to the levels is handled through the first device in the list.

**Combining and Uncombining channels**

Combining DEVCHANs is unique to NetLinx. The NetLinx function COMBINE_CHANNELS combines an individual channel on a virtual device to one or more channels on another device (or devices). The format for COMBINE_CHANNELS and UNCOMBINE_CHANNELS is:

```
COMBINE_CHANNELS (<virtual DEVCHAN>, <DEVCHAN1[]>, <DEVCHAN2[]>..)
UNCOMBINE_CHANNELS (<virtual DEVCHAN>)
```

**String Comparisons**

While in Axcess it is possible to perform a string comparison using the '?' wildcard, NetLinx requires the COMPARE_STRING function to be used instead.

**Axcess code - string comparison**

```
IF (TIME = '12:00:??')
(* Do something at noon - evaluation is valid for an entire minute *)
```

**NetLinx code - string comparison**

```
IF (COMPARE_STRING(TIME,'12:00:??'))
// Do something at noon - evaluation is valid for an entire minute
```

**Modules**

There are two ways to reuse code in different Axcess programs: *Include Files and System Calls.*

- Include files redirect the compiler to files with an .AXI extension. The .AXI files can contain the same type of information present within an Axcess program. All data is accessible both within the Include file and within the primary Axcess program. Include files are limited because they are static. Mainline statements within the Include file cannot be adapted from program to program without altering the Include file. To update the Include files in a program, the entire program must be compiled and loaded.

- System calls are external subroutines that can be instanced and referenced in the main program. Like DEFINE_CALL subroutines, System Calls can pass parameters to adapt the System Call to the needs of different programs. System Calls have been one of the primary tools for creating standardized reusable blocks of code. To update the System Calls within a program, the entire program must be compiled and loaded.

Modules are unique to NetLinx. Like Include files, the code within the Module is not limited to the DEFINE_CALL section. Modules can contain variable definitions, functions, subroutines, startup code, events, and mainline. Modules are passed parameters that are used to adapt the information and variables used within the Module (similar to System calls).

Modules are similar to programs loaded into AXB-232++ boxes. They operate as stand-alone programs inside the NetLinx program. Interaction between the Module and the NetLinx Program is done through User Interface (UI) pushes and releases, turning virtual device channels on and off, and passing variables and arrays to the Module. The code in the Module is local, or is restricted to use only within the Module. This means that functions and subroutines defined with Module cannot be directly used with the main NetLinx code.

Modules will eventually replace System calls. Where several system calls are currently needed to provide device initialization, buffer processing, and device functionality, one module will handle all three functions.

The first line of a Module contains the MODULE_NAME keyword, the Module name, and the parameter list. The format is shown below:

```
MODULE_NAME = '<module name>' [(<param1>, <param2>, .. , <paramN>)]
```

The <module name> must match the file name, but has the .AXS extension. The module name can be 64 characters long and contain valid file name characters. The parameter name is optional and follows the same restrictions as subroutine parameters, with the exception that constants and expressions cannot be used as arguments. Within the NetLinx program, the Module is referenced using the following format:

```
DEFINE_MODULE '<module name>' <instance name> [(<pass1>, <pass2>, .. , <passN>)]
```
The `<module name>` must match the module name specified in the Module file, as shown above. The `<instance name>` is a unique name given to each occurrence of the module within the program. If the module is used twice within the program, each occurrence gets a unique instance name. The parameter list passed to the module must match number and types of parameters listed in the module file above. The DEFINE_MODULE statements are listed in the code after the DEFINE_CALL and DEFINE_FUNCTION sections, but before the DEFINE_START section. The DEFINE_MODULE statements cannot appear within the DEFINE_PROGRAM or DEFINE_EVENTS section.

**NOTE:** To use a module, the module must be compiled with the Source Code, and the Master must be rebooted to run the new module.