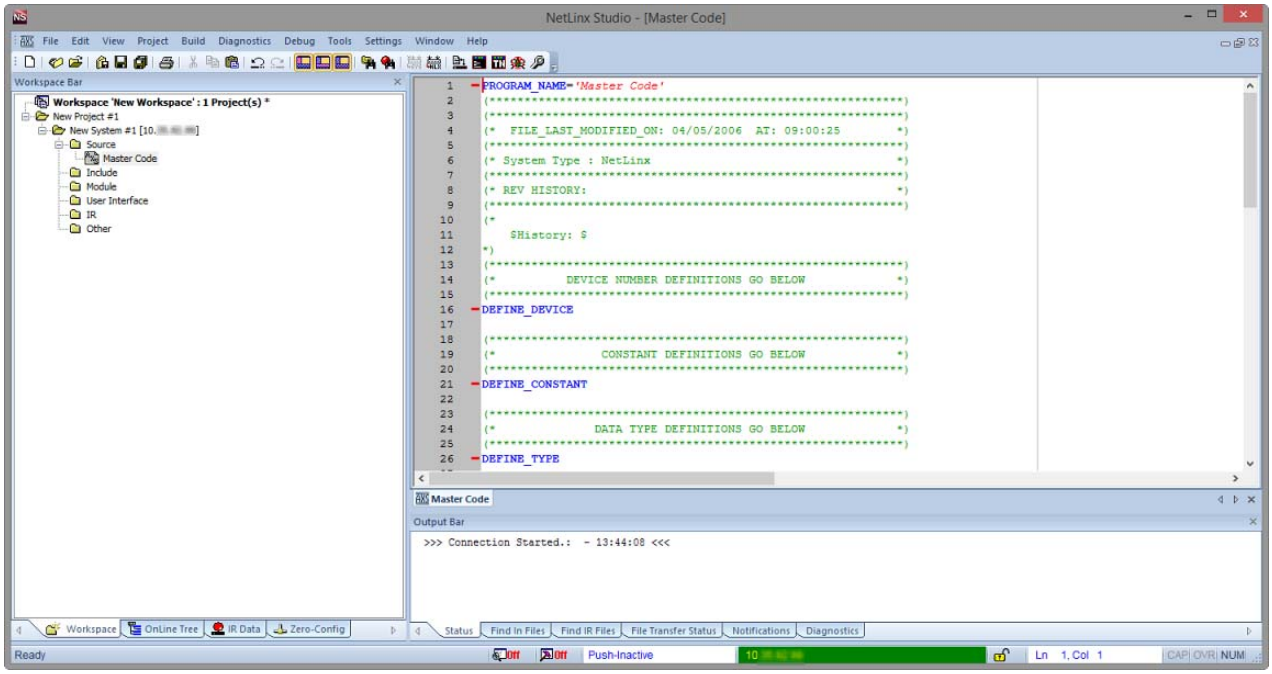
STYLE GUIDE

NETLINX PROGRAMMING



AV FOR AN IT WORLD

COPYRIGHT NOTICE

LIABILITY 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 Style Guide Overview

The personal nature of coding style can make it difficult to collaborate in a team atmosphere and support a hardware manufacturer that relies on custom programming. By providing a style guide for the NetLinx language it is our goal to drive standardization and consistency among the programming community.

The intent of this document is not to convey *how* to program a NetLinx control system but rather establish a standardized convention of the way code is formatted within any given program. In its most basic terms, coding style is how your code looks, plain and simple. Coding style can be extremely personal and everyone has their preferred style whether it's because that's how they were originally taught, they picked it up from another programming language, or it's just something that they developed on their own. It is not uncommon to have to context switch between different programming languages and it is always preferred to use the defined style of the language you are working with if a style guide is available.

While the hardware your code is running on doesn't care how the code looks, other programmers and support staff certainly do. The way code looks adds to our understanding of it. Don't confuse the rules of the style with the rules of the language. The rules of the language will allow you to write code in a large variety of styles, which is part of the problem. A lot of programmers, when faced with issues in code someone else wrote, will reformat it in their own style in order to better understand what's going on and resolve the issue. When everyone is writing code that looks different, everyone else is constantly trying to visually parse the code before being able to understand it. When everyone is writing code that looks the same, your brain can relax a bit as the understanding comes faster and some potential issues can be avoided entirely just by using a consistent programming style.

The concept of coding "best practice" is not covered as it pertains to CPU and memory efficiency. Using this style guide will certainly help you develop more consistent, readable code but it is not designed to entirely avoid programmatic issues like unnecessarily complicated string parsing operations or writing code that is not easily adaptable to system changes. Coding best practice can be learned best in a Harman Professional University classroom, interacting with Technical Support, or AMX technical documentation.

FORMATTING

CAPITALIZATION

The NetLinx language is not case sensitive. Traditionally AMX programming has been done in all capital letters. Underscores are used in constants and variables in place of spaces (SYSTEM_POWER_STATE). For reasons of ergonomics and not having to jump in and out of caps lock states, lower case is now *preferred*[1]. Alternatively, you can still use the all caps method but don't switch between the two within a single piece of code. Pick one style and stick to it. It is understood that the current NetLinx templates and code wizard still use all caps but it should not influence which method you use throughout the rest of the code.

INDENTATION

Use the default NetLinx Studio indentation. Viewing someone else's code that uses different indentation can be very messy.

- Set tab stops to every 8 characters (default).
- Indent before text with 4 characters (default).
- Enable auto-indentation.

Matching braces should always line up.

Statements within the same encapsulation should always line up.

SPACING

All function names should be immediately followed by a left parenthesis. Do not use a space.

Examples:

```
define_function integer fnSystemOn(integer nInput)
{
    pulse[vdvProj,PWR_ON];
    pulse[dvRelays,SCREEN_DOWN];
    send_command dvDVX, "'VI',itoa(nInput),'O1'";
}

button_event[dvTP,1]
{
    push:
   {
        fnSystemOn(nDvdInput);
   }
}
```

All array references should be immediately followed by an open bracket. Do not use a space.

mples:

```
non_volatile integer nSwitcherOut[8];

nSwitcherOut[3];
```

1. *Unless otherwise described as being capitalized later in this document (such as constant names).

Commas and semicolons are always followed by whitespace. Exceptions are only allowed in string expressions (concatenation), devchan, or devlev.

Examples:

```
for (nCount = 1; nCount <= 8; nCount++)

non_volatile integer nTpButtons[] = {41, 56, 82, 108};

send_level dvVolume, 41, 100;

send_command dvDVX, "'VI',itoa(nInput),'O1'";

pulse[dvRelay,5];
```

Arithmetic and relational operators should have spaces on either side. Ex: (X + 2) > (Y - 6)

Increment/decrement operators should be immediately preceded by their operand, no spaces. Ex. nCount++;

The keywords *if*, *while*, *for*, *switch*, *case*, and *active* should be followed by a space.

DEFINE_ SECTION ORDERING

*DEFINE_* sections should only be listed in the order they appear in the Edit menu's Insert Section dialogue or the default NetLinx template (both of which have the same order). Do not place sections out of order or interleaved throughout the code, this makes things hard to find and not intuitive.

You can repeat the same DEFINE_ statement within its proper location to assist with code folding.

Always use NetLinx template standard commenting on the lines directly preceding the DEFINE_ section. This follows the default formatting of the Edit/Insert Section feature in NetLinx Studio.

Example:

```
(***********************************************************)
(*                THE EVENTS GO BELOW                      *)
(***********************************************************)
DEFINE_EVENT
```

BRACES, SEMICOLONS, AND PARENTHESIS

Braces encapsulating compound statements should always appear on the next line of code (outline formatting) and never at the end of a line (Egyptian or K&R formatting). Though the NetLinx compiler ignores extra spaces, carriage returns, and line feeds and will allow either method, it is best to follow the outline format.

Example:

```
// K&R formatting. Not recommended!
button_event[dvTP,1] {
    push: {
        min_to[dvRelay,3];
    }
}

// Outline formatting. Braces line up.
button_event[dvTP,1]
{
    push:
    {
        min_to[dvRelay,3];
    }
}
```

Use braces on all *if*, *while*, *for* and *wait* statements as well as all event handlers even if they contain just one statement. This makes it easier to read and debug as the associated statements are more clearly visible.

Semicolons at the end of a statement should be used at all times even though the NetLinx compiler does not require them. Since the NetLinx compiler ignores extra spaces, carriage returns, and line feeds an error may occur depending on the statement(s) on the next line. Proper use of semicolons helps avoid these types of compiler errors.

Parenthesis should be used in mathematical and relational expressions not only to specify order of precedence, but to also help simplify the expression. When in doubt, parenthesize.

MAXIMUM LINE LENGTH

Avoid making lines longer than 120 characters. You should not have to scroll left or right to read your code. If something is off-screen you can't see it and mistakes can be made. Break up long lines at logically appropriate lengths to fit within this character limit.

COMMENTING

Your code should utilize comments to describe complicated operations or anything not self-evident.

- Instead of excessive comments on how you perform a complex operation, first try to make it easier to read by using more descriptive identifiers. This helps in the future in case the operation changes but someone forgets to change the comments.

Use line comments after device declarations to describe makes and models of controlled equipment in the DEFINE_DEVICE section.

Although /* */ can be used to create multi-line comments, (* *) is preferred due to its widespread use in NetLinx templates, help files, sample code, and RMS SDK.

Comments using // should always be followed by a space before the comment for more readability.

EMPTY LINES

Use empty lines to separate events, local scope variable declarations, conditionals, waits, and loops. Everything is easier to see when there is space surrounding it.

Example:

```
button_event[dvTP_Main,1]
{
    push:
    {
        stack_var integer nLoop;

        if (!nSystemPower)
        {
            fnSystemOn();
        }

        for (nLoop = 1; nLoop <= 4; nLoop++)
        {
            [dvTP,nLoop] = (nCurrentSource == nLoop);
        }
    }
}

button_event[dvTP_BluRay,0]
{
    push:
    {
        pulse[dvBluRay,button.input.channel];
    }
}
```

IDENTIFIERS

NAMING

HUNGARIAN NOTATION

Traditionally, Hungarian Notation is used when naming devices, variables, and subroutines. Use of Hungarian Notation with descriptive names and Camel Case makes code easier to read, understand, and helps prevent type based compiler issues.

Examples:

```
volatile integer nTransportStatus;

dvRelays = 5001:8:0;

non_volatile long lFeedbackTime[] = {300};

fnSystemOff();
```

Device names use the Hungarian Notation of *dv*, virtual devices use *vdv*, and device arrays use *da*.

NOTE: *Refer to addendum at the end of this document for suggested notation of user defined and intrinsic data types and devices.*

UNDERSCORES AND CAMEL CASE

Constant definitions that occur in the Standard NetLinx API (SNAPI) use all capital letters and underscores to imply spaces. User created constants should also use this method.

Variable names should use Camel Case. Except for the notation, every first letter of each word is capitalized (no underscores).

VARIABLE DECLARATIONS

Group variables for specific devices or subroutines together.

Variable names for specific devices, processes, and subroutines should share a common prefix. The descriptor should always follow the subject.

Example:

```
non_volatile integer nBargraphMic1;
non_volatile integer nBargraphMic2;

volatile char sButtonLabelSource1[15];
volatile char sButtonLabelMacro2[15];
```

Declare variables as close as possible where they are used. Use local scope unless global scope is absolutely required.

Global scope variable declarations should always include the persistence and data type.

Examples:

```
DEFINE_VARIABLE
integer nCount;                 // No!
non_volatile y;                 // No!
x;                              // Are you kidding me???

volatile integer nSomeNumber;   // YES!
```

Local scope variable declarations should always include the persistence and data type.

Example:

```
data_event [dvSwitcher]
{
    string:
    {
        char sTempBuffer[69];         // No!
        local_var nCount;               // No!
        stack_var integer nInputNum;    // YES!
    }
}
```

CODING

CONSTRUCTS TO AVOID

AVOID NESTING TOO MANY OPERATIONS IN ONE LINE.

It can be very confusing and difficult to fix errors.

Example:

```
// Instead of this:
button_event[dvTpSwitcher,49] // Take button
{
    push:
    {
       send_command dvDGX,"format("'CL',itoa(nLevType),'I%1d'",
       nInput),'O',itoa(nOutput)";
    }
}

// Do this:
button_event[dvTpSwitcher,49] // Take button
{
    push:
    {
        send_command dvDGX,"format("'CL',itoa(nLevType),  // Add level
                                   'I%1d'",nInput),      // Add input
                                   'O',itoa(nOutput)";   // Add output
    }
}
```

EQUALS SIGNS

The NetLinx compiler allows the use of a single equals sign (=) as both a relational operator and an assignment. It is preferable to use a single equals sign for assignment operations only and the double equals (==) for relational operations. The easy way to remember it is if you say "gets" when referring to the single equals and "equal to" when referring to the double equals.

Example:

```
x = 4;          // Say, "X gets four".

if (x == 4)     // Say, "If x is equal to four".
```

DEFINE_PROGRAM

Avoid using DEFINE_PROGRAM whenever possible.

- Due to differences in the underlying architecture of the NX-Series masters, changing variables in the DEFINE_PROGRAM section of code can negatively impact program performance.
- See "Differences in DEFINE_PROGRAM Program Execution" section of the NX-Series Controllers WebConsole & Programming Guide for additional and alternate coding methodologies.
- See also Tech Note #993

INITIALIZATION

CONSTANTS

Per the rules of the NetLinx language, constants must be initialized to a value when they are declared.

VARIABLES

An initialization value of a variable does not need to be used if the intended value is null or 0.

If the intended initial value(s) of a variable or array are constants then the variable should be initialized with those values upon its declaration.

If the intended initial value(s) of a variable are not constants then the variable should be initialized with those values within the *DEFINE_START* section.

Variable arrays containing string expressions for device control should also be initialized within the DEFINE_START section.

INCLUDES

Use conditional compiling to prevent errors.

Example:

```
DEFINE_CONSTANT
#if_not_defined DEVICE_COMMUNICATING
integer  DEVICE_COMMUNICATING = 251
#end_if
```

Do not use include files to replace typical sections of main code. Ex. - Don't use one include for all of your device definitions, another for your global variables, etc.

- Without device and variable declarations for a block of code in the same file you would have to copy or modify too many other files to get it working.
- Managing changes between files can be difficult.

MODULES

All user created NetLinx modules should follow the SNAPI guidelines for channels and commands. This allows for more consistency between NetLinx, Duet, and Driver Design modules as well as making it easy to swap out modules or add RMS programming hooks as needed.

FUNCTIONS

Local scope variables should be declared within the encapsulating braces, never before it.

For portability, global scope variables should not be referenced within a function but rather passed in as a parameter whenever possible. This makes copying functions between programs easier and uses only a marginal amount of memory to reference the initial variable.

Addendum

Hungarian Notation for NetLinx programming

Intrinsic types

```
char cSomeChar                  //8 bit unsigned integer
char sSomeString[]              // String array – ASCII data
widechar wcSomeWideChar         // 16 bit unsigned integer (array of characters)
widechar wsSomeWideChar[]       // Wide String array
integer nSomeInt                // 16 bit unsigned integer
sinteger snSomeSignedInt        // 16 bit signed integer
long lSomeLong                  // 32 bit unsigned integer
slong slSomeSignedLong          // 32 bit signed integer
float fSomeFloat                // 32 bit signed floating point
double dSomeDouble              // 64 bit signed floating point
devchan dcSomeDevChan           // Device / Channel variable
devlev dlSomeDevLev             // Device / Level variable
integer nSomeInt[]              // Integer array
sinteger snSomeSignedInt[]      // Signed integer array
long lSomeLong[]                // Long integer array
slong slSomeSignedLong[]        // Signed long integer array
float fSomeFloat[]              // Float array
double dSomeDouble[]            // Double float array
devchan dcSomeDevChan[]         // Array of Device / Channel
devlev dlSomeDevLev[]           // Array of Device / Level
```

Other Types

```
struct_type _SomeStruct         // DEFINE_TYPE for a Structure
user_type uSomeUserType         // Variable of type user defined*
user_type uUserTypeArray[]      // Array of user defined types
dev dvSomeDev                   // Device Variable
dev vdvSomeVirtualDevice        // Virtual Device
dev daSomeDevArray[]            // Array of Devices
function fnSomeFunction         // Function names
```

*Members of a structure use the Hungarian notation from the list of intrinsic types above.

3000 RESEARCH DRIVE, RICHARDSON, TX 75082
AMX.com | 800.222.0193 | 469.624.8000 | +1.469.624.7400 | fax 469.624.7153

Last Revised:
12/21/2015