AMX Corporation

White Paper

CONVERTING AXCESS CODE TO NETLINX CODE

MARCH 2003

Table of Contents

Overview

When AMX designed the NetLinx control system, the goal was to upgrade the processor, bus and to greatly improve the power of the programming language. Originally named Axcess2, the new language was designed to be a superset of the Axcess programming language. NetLinx contains all of the elements of Axcess.

You cannot compile NetLinx code on an Axcess compiler, and likewise, you cannot download NetLinx code to an Axcess control system. To upgrade existing Axcess control system to NetLinx you have to 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.

Compiling Axcess code for NetLinx

In order to compile your existing Axcess code on NetLinx, minor modifications will most likely be required. These include identifier names that conflict with NetLinx identifiers, warning on variable type conversion, and stricter syntax rules.

Reserved Identifiers

Netlinx includes many new identifiers that may have been used in your Axcess program. When converting your Axcess to code to NetLinx, you will need to replace any use of these NetLinx identifiers with a different identifier. Below is a list of identifiers new to NetLinx.

NetLinx Pre-processor

```
__LDATE__
__NETLINX__
#INCLUDE
```

NetLinx Constants

```
DO_PUSH_TIMED_INFINITE
FIRST_LOCAL_PORT
FIRST_VIRTUAL_DEVICE
IP_Addr_Flg_DHCP
NETLINX_AXI_VERSION
SOURCE_TYPE_AXLINK
SOURCE_TYPE_IP_ADDRESS
SOURCE_TYPE_IP_SOCKET_ADDRESS
SOURCE_TYPE_NEURON_ID
SOURCE_TYPE_NEURON_SUBNODE_ICSP
SOURCE_TYPE_NEURON_SUBNODE_PL
SOURCE_TYPE_NO_ADDRESS
```

SOURCE_TYPE_RS232
TIMELINE_ABSOLUTE
TIMELINE_ONCE
TIMELINE_RELATIVE
TIMELINE_REPEAT
URL_Flg_Stat_Connected
URL_Flg_Stat_Connecting
URL_Flg_Stat_Lookup
URL_Flg_Stat_Mask
URL_Flg_Stat_PrgNetLinx
URL_Flg_Stat_Waiting
URL_Flg_TCP
URL_Flg_Temp

NetLinx Structures

DEV_INFO_STRUCT
DNS_STRUCT
IP_ADDRESS_STRUCT
TBUTTON
TCHANNEL
TDATA
TLEVEL
TTIMELINE
URL_STRUCT

NetLinx Variable types

NON_VOLATILE
VOLATILE
PERSISTENT
CHAR
WIDECHAR
INTEGER
SINTEGER
LONG
SLONG
FLOAT
DOUBLE
DEV
DEVCHAN
DEVLEV

NetLinx Variables

BUTTON
CHANNEL
DATA
DV_CHANNEL
LDATE
LEVEL
MASTER_SN
PUSH_DEVCHAN
RELEASE_DEVCHAN
SYSTEM_NUMBER
TIMELINE

NetLinx Keywords

BREAK
BUTTON_EVENT
CASE
CHANNEL_EVENT

COMBINE_CHANNELS
COMBINE_DEVICES
COMBINE_LEVELS
COMMAND
COMPARE_STRINGS
CONSTANT
DATA_EVENT
DEFAULT
DEFINE_EVENT
DEFINE_FUNCTION
DEFINE_MODULE
DEFINE_TYPE
DEVICE_ID_STRING
DO_PUSH_TIMED
FIRST_LOCAL_PORT
FOR
HOLD
LDATE
LEVEL_EVENT
LSHIFT
MASTER_SN
MOD
MODULE_NAME
OFFLINE
ONERROR
ONLINE
PUSH_DEVCHAN
REBOOT
RELEASE_DEVCHAN
REPEAT
RETURN
RSHIFT
STACK_VAR
STRING
STRUCTURE
SWITCH
SYSTEM_NUMBER
TIMED_WAIT_UNTIL
TIMELINE_EVENT
UNCOMBINE_CHANNELS
UNCOMBINE_DEVICES
UNCOMBINE_LEVELS

NetLinx Functions

ABS_VALUE
ADD_URL_ENTRY
ASTRO_CLOCK
ATOF
ATOL
COMPARE_STRING
DATE_TO_DAY
DATE_TO_MONTH
DATE_TO_YEAR
DAY_OF_WEEK
DELETE_URL_ENTRY
DEVICE_ID_STRING
DEVICE_INFO
DO_PUSH_TIMED
FILE_CLOSE
FILE_COPY
FILE_CREATEDIR
FILE_DELETE
FILE_DIR

```
FILE_GETDIR
FILE_OPEN
FILE_READ
FILE_READ_LINE
FILE_REMOVEDIR
FILE_RENAME
FILE_SEEK
FILE_SETDIR
FILE_WRITE
FILE_WRITE_LINE
FORMAT
FTOA
GET_BUFFER_STRING
GET_DNS_LIST
GET_IP_ADDRESS
GET_LAST
GET_SERIAL_NUMBER
GET_SYSTEM_NUMBER
GET_UNIQUE_ID
GET_URL_LIST
HEXTOI
IP_CLIENT_CLOSE
IP_CLIENT_OPEN
IP_MC_SERVER_OPEN
IP_SERVER_CLOSE
IP_SERVER_OPEN
LENGTH_ARRAY
LENGTH_VARIABLE_TO_STRING
LENGTH_VARIABLE_TO_XML
MAX_LENGTH_ARRAY
MAX_LENGTH_STRING
MAX_VALUE
MIN_VALUE
RAW_BE
RAW_LE
REBOOT
REDIRECT_STRING
SET_DNS_LIST
SET_IP_ADDRESS
SET_LENGTH_ARRAY
SET_OUTDOOR_TEMPATURE
SET_OUTDOOR_TEMPERATURE
SET_SYSTEM_NUMBER
SET_VALIDATION_CODE
SET_VIRTUAL_CHANNEL_COUNT
SET_VIRTUAL_LEVEL_COUNT
SET_VIRTUAL_PORT_COUNT
STRING_TO_VARIABLE
TIME_TO_HOUR
TIME_TO_MINUTE
TIME_TO_SECOND
TIMELINE_ACTIVE
TIMELINE_CREATE
TIMELINE_DYNAMIC_ID
TIMELINE_GET
TIMELINE_KILL
TIMELINE_PAUSE
TIMELINE_RELOAD
TIMELINE_RESTART
TIMELINE_SET
TYPE_CAST
VARIABLE_TO_STRING
VARIABLE_TO_XML
XML_TO_VARIABLE
```

DEFINE_DEVICE

The DEFINE_DEVICE section of your Axcess will likely require modification. AXlink devices can be declared just like in Axcess programs. If a device is declared in a NetLinx program with just the device number, the NetLinx Compiler assumes that it has a Port number of 1 and a System number of 0. However, you should convert all existing device declaration using the D:P:S (Device:Port:System) notation. Doing so enabled certain NetLinx specific debugging features and can help pinpoint other, possible obscure, errors.

Axcess Code

```
DEFINE_DEVICE

VPROJ = 1                    (* RS232 controlled Projector *)
VCR   = 8                    (* IR controlled VCR *)
IO    = 14                   (* Power sensing via VSS2 for VCR *)
RADIO = 96                   (* AXR-RF for TXC-32+ *)
TP    = 128                  (* Touch Panel *)
```

NetLinx Code

```
DEFINE_DEVICE

VPROJ = 1:1:0                (* RS232 controlled Projector *)
VCR   = 8:1:0                (* IR controlled VCR *)
IO    = 14:1:0               (* Power sensing via VSS2 for VCR *)
RADIO = 96:1:0               (* AXR-RF for TXC-32+ *)
TP    = 128:1:0              (* Touch Panel *)
```

In NetLinx, a system number of 0 means “local system”. When referencing a device on the local system, you should use a value of 0 instead of the actual system number. This allows the code to be more portable; it can run on multiple NetLinx systems the same way without modification. If you want to reference a device on another master, you must use the System number of the other master.

DEFINE_COMBINE statements require the use of a Virtual Device. Virtual Devices are declared in the DEFINE_DEVICE section and have a value between 32768 and 36863. The Virtual Device must be the first device in the DEFINE_COMBINE list. The master treats the Virtual Device just like any other device, except the Virtual Device can never go offline. This resolves some problems associated with the first device in the DEFINE_COMBINE list falling offline.

Axcess Code

```
DEFINE_DEVICE

TP1          = 128           (* Touch Panel 1 *)
TP2          = 132           (* Touch Panel 2 *)
```

```
DEFINE_COMBINE (TP1,TP2)
```

NetLinx Code

```
DEFINE_DEVICE

TP1         = 128             (* Touch Panel 1 *)
TP2         = 132             (* Touch Panel 2 *)
TP_VIRTUAL  = 33001:1:0       (* Virtual Touch Panel for combine *)

DEFINE_COMBINE (TP_VIRTUAL ,TP1,TP2)
```

DEFINE_CONSTANT

Axcess defines Constants as either a fixed integer value between 0 and 65,535 or an array with a maximum length of 255 bytes where each element can hold a value from 0 to 255. These values can be expressed in ASCII, Decimal or Hexadecimal. An expression can include any previously defined symbol and can be built using the double-quote syntax.

NetLinx processes Constants slightly differently than Axcess programs. NetLinx allows you to define an expression in the DEFINE_CONSTANT section. However, double-quote expressions are not allowed, all values must be constants and the constant must declared as an array by including a set of empty brackets following the constant name. NetLinx also includes.

Since NetLinx is a strongly typed language, constants are casts to types as well. Occasionally, this may cause problems when constants are assigned to the results of a mathmatic operation. For instance, if a constant is defined as 10 * 40, the value may get cats to a CHAR resulting in a value of 144(400 MOD 256) instead of 400. In these cases, NetLinx include a new variable type to create constant variables.

Axcess Code

```
DEFINE_CONSTANT

VALUE_MIN     = 40
DEFAULT_NAME  = 'Axcess'
ETX           = "$FE,$FF"
VALUE_MAX     = 140
P_PON         = "$02,'PON',$03"
LARGE_CONST   = 10 * 40
```

NetLinx Code

```
DEFINE_CONSTANT

VALUE_MIN    = 40
DEFAULT_NAME = 'Axcess'
```

```
ETX[]         = {$FE,$FF}
VALUE_MAX     = VALUE_MIN + 100
P_PON[]       = {$02,'P','O','N',$03}

DEFINE_VARIABLE

CONSTANT INTEGER LARGE_CONST   = 10 * 40
```

DEFINE_VARIABLE

In Axcess, there were three kinds of variable: INTEGERs, ARRAYs and INTEGER ARRAY's. Converting between one type of variable to another was usually a simple matter and handled for you automatically. For instance, you could assign an INTEGER ARRAY to an ARRAY without an error or warning that data will be lost.

The NetLinx language is a strongly typed language. This means that each type of variable is predefined as part of the programming language and all constants or variables defined must be described with one of the data types. Certain operations may be allowable only with certain data types. For example, if you try to convert an INTEGER ARRAY to an ARRAY, you will receive warnings from the compiler. This is not an error, but is just a reminder that values above 255 will be truncated. Axcess also truncated the values, it just didn't give you a warning.

NetLinx will assume certain data types for you automatically the way Axcess did. If you create a variable without a type and that variable is not an array, NetLinx will assume you mean the variable is of type INTEGER. If you create a variable without a type and that variable is an array, NetLinx will assume you mean the variable is of type CHAR.

Since NetLinx is a strongly typed language, there are likely to be some variable that required types in order to compile under NetLinx. Most commonly, these occur when you assign a device to a variable. All devices in NetLinx are DEV structures, composed of the Number, Port and System used in the D:P:S notation. In order to assign a device to a variable in NetLinx, you can either change the variable to be of type DEV or modify the code to store only part of the DEV structure to the variable.

Axcess Code

```
DEFINE_VARIABLE

CURRENT_SOURCE                    (* Currently selected source *)

DEFINE_START

CURRENT_SOURCE = VCR              (* Set VCR as currently selected *)
```

NetLinx Code

```
DEFINE_VARIABLE
```

```
DEV CURRENT_SOURCE                          (* Currently selected source
*)

DEFINE_START

CURRENT_SOURCE = VCR                (* Set VCR as currently selected *)
```

or

```
DEFINE_VARIABLE

CURRENT_SOURCE                      (* Currently selected source *)

DEFINE_START

CURRENT_SOURCE = VCR.NUMBER         (* Set VCR as currently selected *)
```

DEFINE_CALL

In NetLinx, the DEFINE_CALL section behaves as it does in Axcess. However, you may need to add variable types to call parameters to eliminate warnings. As with variables used to hold device references, you should declare parameters to be of type DEV.

Axcess Code

```
DEFINE_CALL 'SEND TO DEVICE' (CARD, STR[100])
{
  SEND_STRING CARD,"STR"
}
```

NetLinx Code

```
DEFINE_CALL 'SEND TO DEVICE' (DEV CARD, STR[100])
{
  SEND_STRING CARD,"STR"
}
```

DEFINE_START

In NetLinx, the DEFINE_START section behaves as it does in Axcess. However, a new mechanism in NetLinx, the DEFINE_EVENT section, is more commonly used for device initialization. It is possible that commands issued to device in DEFINE_START of an Axcess program will not work properly in a NetLinx program. This is occurs because at the time DEFINE_START runs, the device has not reported to the NetLinx master and is not online.

If you believe that commands from DEFINE_START do not appear to work in your NetLinx program, you should move them to a DATA_EVENT in the DEFINE_EVENT section.

Axcess Code

```
DEFINE_START

SEND_COMMAND VOL,'P1=P2'
```

NetLinx Code

```
DEFINE_EVENT

DATA_EVENT[VOL]
{
  ONLINE:
    SEND_COMMAND VOL,'P1=P2'
}
```

Syntax Rules

NetLinx has a more robust and strict compiler. Errors that were overlooked in Axcess will be caught in NetLinx. This means that older Axcess code that compiled error free may not compile in NetLinx. This does not mean that the language changed, only that certain errors did not surface under the Axcess Compiler. Axcess allowed the following expressions but they will need to be corrected in order to compile in NetLinx:

Axcess Code

```
IF (X = 1) AND (Y = 1)        (* No () around the expression *)
{
  SEND_STRING 0,"'Debug Statement'13,10" (* Missing comma *)
}
```

NetLinx Code

```
IF ((X = 1) AND (Y = 1))      (* () around the expression required *)
{
  SEND_STRING 0,"'Debug Statement',13,10" (* Comma required *)
}
```

Axcess allow implied string expressions when sending strings and command via SEND_STRING and SEND_COMMAND. Axcess allows a single byte to be interpreted as a string without wrapping it in double-quotes. NetLinx requires that this expression is wrapped in double-quotes:

Axcess Code

```
SEND_STRING CAM,$90           (* INITIALIZE COMM. *)
```

NetLinx Code

```
SEND_STRING CAM,"$90"          (* INITIALIZE COMM. *)
```

Keyword Changes

To exist properly in the NetLinx environment, certain keyword behaviors have changed. Although this is limited, you should note and understand the behaviors of these keywords. These new behaviors may or may not affect your program.

WHILE

In NetLinx, WHILE loops no longer timeout after a half second. Relying on the WHILE timeout was never recommended, but now it can lock you into an endless loop. WHILE loops now behave just like MEDIUM_WHILE loops.

Order of Operation

According to 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 instead of the highest. Usually, this problem is solved by wrapping the NOT expression in parentheses which forces the expression to execute in the desired order. However, your program may be taking advantage of the logic flaw. When these systems are converted to NetLinx, the logic does not work as desired.

```
DEFINE_CALL 'PRECEDENCE' (A,B)
LOCAL_VAR C D E
{
  C = !A && B
  D = B && !A
  E = !B && !A
}

DEFINE_START

CALL PRECEDENCE ' (0,0)
CALL PRECEDENCE ' (1,0)
CALL PRECEDENCE ' (0,1)
CALL PRECEDENCE ' (1,1)
```

Axcess code

A	B	!A && B	B && !A	!B && !A
0	0	**1**	0	1
1	0	**1**	0	1
0	1	1	1	0
1	1	0	0	1

NetLinx Code

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, use parentheses to force the order of operation.

Please be aware of this difference as you support programs that are 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.

Axcess code

```
IF (SYSTEM_POWER && ![VCR,PLAY] || [VCR,RECORD])
{
}
```

NetLinx Code

```
IF (SYSTEM_POWER && !([VCR,PLAY] || [VCR,RECORD]))
{
}
```

Getting your code to run

Once you have downloaded your program to the NetLinx Master, you must reboot the Master. The Master must be rebooted for any changes to the Master to take effect, including program downloads, system number changes, and Network IP address changes.

More Information

The NetLinx language provides many more features to programmers than Axcess. In some cases, you will want to take advantage of these new features when converting a

system from Axcess to NetLinx. If you are starting a program from scratch, it is strongly recommended you take advantage of these new features.

A few of the features you should take advantage of in NetLinx include the following:

- PUSHes and RELEASEs are largely replaced with the new DEFINE_EVENT event handlers.
- Repeating operation occurring as the result of a button pressed can be moved to a BUTTON_EVENT HOLD event handler to simplified code.
- String processing can be moved to an event handler in the DEFINE_EVENT section for more optimal processing.
- WAIT's can be replaced by TIMELINEs proving finer resolution and event processing for repeating processes such as polling.

If you would like more information on NetLinx programming standards and recommendations, AMX has more documents available online. Just point your browser to www.amx.com, click on Dealers->Tech Center->Tech Notes and search for *Standards.*

Related documents and Tech Notes: NetLinx Programming Standards
TN 186
TN 249
TN 261
TN 310