AMX Professional Services

NetLinx Programming Standards **Revision 0.6**

Document Control Data

Revision Level	Date	Initials	Comments
0.6	3/20/02	CWR	Added notes on FIRST_LOCAL_PORT
0.5	2/20/02	CWR	Additions/changes from meeting
0.4	8/10/01	CWR	Additions
0.31	11/30/00	DKA	Cleanup
0.3	11/20/00	CWR	Additions
0.2	10/31/00	CWR	Changes from meeting
0.1	12/14/99	RDW	Initial Draft

1. Device/System Numbering

Device Numbering

Description:
Device numbering should remain standard. Recommendations are:

Device:	Types:	Comments:
1-255	AXCESS Devices	Use AXCESS standards
301-3072	NetLinx card frames	Start at frame number 25 (frame# * 12) + 1
5001-6000	ICSNet NetLinx devices	NXI, NXM-COM2, NXM-IRS4, etc.
6001-7000	ICSNet Landmark devices	PLH-VS8, PLH-AS16, PLB-AS16
7001-8000	P3 Devices	
8001-10000	Internet Inside applications	Database Plus, PCLink, etc.
10001-32000	ICSNet Panels	DMS, IMS, Web panels
33001-36863	Virtual devices	For simplicity, start at 33001
Device:	Types:	Comments:
32001-32767	Dynamic devices	Actual range used by master
32768-36863	Virtual devices	Actual range used by master

System Numbering

Description:
Each NetLinx master can be assigned a unique system number (between 1 and 65535). If dealing with a single NetLinx master system, this system number is not an important factor and could typically be left at 1. However, when dealing with a multiple NetLinx master system and implementing master-to-master (M2M), this system number becomes extremely important. Each of these NetLinx masters would need to have a unique system number assigned. (It should be noted that even when implementing multiple NetLinx masters that do not require M2M, it would still be a good idea to provide unique system numbers. This way, if M2M is ever needed in the future, it could be more easily implemented.)

System 0 is not a valid system number to assign to a master. However, with programming the NetLinx system, system 0 is extremely useful to use. When you define a device as system 0, it implies that this device is on the master being programmed. Then, the system number for that master could be any value and the code would still work.

Recommended:

```
DEFINE_DEVICE
dvRelay = 5001:7:0     // System 0 implies THIS NetLinx master
```

With this example, if for some reason, the system number of this master had to change, this program would still work. It is also easier to identify that this device is on my NetLinx system.

Not Recommended:

```
DEFINE_DEVICE
dvRelay = 5001:7:1     // System 1 refers explicitly to system 1,
                       // could be me or another master
```

With this example, if for some reason this system number had to change from 1 to 2, then the system would need to be reprogrammed with the updated system number in the DEFINE_DEVICE section. It is also not easy to identify that this device is on my NetLinx system.

2. IP Configuration of NetLinx Master

Description

In order for the NetLinx system to run on a network, the network settings must be configured for the target network. You will need to have detailed information about your network on hand before starting.
In order for the NetLinx master to operate correctly, it must have the IP address, subnet mask, gateway address, domain name and name server addresses according to the Network Administrator-supplied values. Many Network administrators setup a DHCP server to automatically supply these values to network devices when they reboot. The NetLinx master does support DHCP and if your network supports DHCP, the network setup process will be greatly simplified.
If not using DHCP, it is important to have values for all parameters listed below. Entering just one network parameter incorrectly can cause communication to the network to fail. Please contact your Network Administrator if you do not have the information necessary to complete the network settings for your system.
After changing the network settings, you must reboot the NetLinx master for the settings to take effect.

Network Setup

- In Amx Studio, select the ***Tools->Master Comm Settings*** Menu item. Select ***NetLinx*** for the platform and then select either a COM port (if the networking is not already configured) or an IP address (if it is currently on the network). Select OK when complete.
- Select the ***Tools->NetLinx Diagnostics*** Menu item. Select the ***Networking*** tab from the tabs along the top. Enter ***0*** for both ***System*** and ***Device***. Then press ***Get IP Info***. Then press ***Get DNS Info***
- Is there a supplied Host name for the IP address assigned to the NetLinx master? If so, enter under ***Host Name***. If no name is supplied, enter ***NetLinx***.
- Is your IP address dynamic (DHCP) or fixed (Static)? Select the ***Use DHCP*** or ***Specify IP Address*** according to your network information.
- If you selected ***Use DHCP***, then press ***Set IP Info*** and then press ***Reboot***. When the master restarts, it will contact the DHCP server to fill in any missing network information. You are now finished with your NetLinx Network Setup.
- If you selected ***Specify IP Address***, enter the IP address supplied by the network administrator. If you do not have an IP address, contact the network administrator to obtain one. If you enter the incorrect number, the NetLinx master and very likely another device on the network will not be able to communicate on the network.
- Enter your ***Subnet Mask***. This value determines the scope of your network. If you did not receive a value for this, enter ***255.255.255.0***. It is a very common default. Do not leave this blank.
- Enter your ***Gateway***. This value determines a path to computers not on your network. If you did not receive a value for this, enter your IP address. Do not leave this blank.
- Enter your ***Domain Suffix***. If you did not receive a value for this, leave it blank.
- Enter up to 3 ***DNS IP***. This values help your master find other network resources. If you did not receive a value for these, leave them blank.
- Press ***Set IP Info***, followed by ***Set DNS Info*** and then press ***Reboot***. When the master restarts, it will use the new network settings. You are now finished with your NetLinx Network Setup.

Dynamic / Static considerations

Note that 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 Dynamic DNS/DHCP servers are available then the NetLinx master's host name may be used in

the URL list. As of this writing, the author is aware that only Windows 2000's DNS server/DHCP servers support the required dynamic capabilities.

It is also possible to have the network administrator provide the NetLinx master's host name. Then the network administrator would enter this host name into a host table (Windows NT). This method would require the NetLinx master to be configured as a dynamic IP. You should contact the network administrator for more details and configuration settings.

If the network cannot resolve the host name of the NetLinx master, then it will be necessary to obtain a static IP address to use for the NetLinx master. Then the URL entry would be configured with this static IP.

3. Master Port Numbering for IP Clients

Master Port Numbering

Description:
When defining a device that will use a port on the master, it is recommended to use the FIRST_LOCAL_PORT when declaring the device. This will ensure that the NetLinx program will use the first available local port on the master with future firmware versions. It is also recommended to keep the ports in consecutive order. For example, using ports FIRST_LOCAL_PORT, FIRST_LOCAL_PORT+1, AND FIRST_LOCAL_PORT+2 would be better than using FIRST_LOCAL_PORT, FIRST_LOCAL_PORT+1, AND FIRST_LOCAL_PORT+200. Doing it this way will still allocate ports up to FIRST_LOCAL_PORT+200, even when they are not used. The more ports defined, the more the master has to keep track of, so keep the numbers low and consecutive.

When writing code for IP devices, it is recommended to use DEFINE_DEVICE just like all other devices. This way, it is easy to see the IP devices without having to search through the DEFINE_CONSTANT section for master port numbers. Then when doing the IP_CLIENT_OPEN, you simply use the DEVICE.PORT. This is preferred over creating a constant to define the master port. Also note that the definition of the DATA_EVENT must explicitly declare the DEV as 0:dvIP.PORT:0. See the example below.

For example:

```
DEFINE_DEVICE
dvPcom      = 0:FIRST_LOCAL_PORT:0   // Polycom Viewstation    (IP Device)
dvARQ       = 0:FIRST_LOCAL_PORT+1:0 // Audio Request ARQ1-Pro (IP Device)

DEFINE_START
IP_CLIENT_OPEN(dvPcom.PORT,'255.255.255.200',24,1)
IP_CLIENT_OPEN(dvARQ.PORT,'255.255.255.201',3663,1)

DEFINE_EVENT
DATA_EVENT[0:dvPcom.PORT:0]   // NOTE: Can't use [dvPcom] here!
{
  ONLINE :
  {
  }
}

DATA_EVENT[0:dvARQ.PORT:0]    // NOTE: Can't use [dvARQ] here!
{
  ONLINE :
  {
  }
}
```

Sample IP port listings:

Master Device:	Master Port:	IP Port:	Device:
0	FIRST_LOCAL_PORT	24	Polycom ViewStation
0	FIRST_LOCAL_PORT+1	3663	Audio Request ARQ1-Pro

4. IP Ports Reference

IP Ports Reference List

Description:
IP Ports currently being used are:

IP Port:	Types:	Comments:
1-1024 are assigned port numbers		
21	FTP	File transfer
23	Telnet	Telnet
25	SMTP	Send Mail
80	HTTP	Web
110	POP3	Receive Mail
A complete listing of IP Port numbers can be found at: http://www.iana.org/assignments/port-numbers		
1319	NetLinx Master	NetLinx Master communications
2000-3999	Client/Server sessions	Used with AXB-NET connectivity
10510	WebLinx	Reserved for WebLinx & Internet Inside

5. Symbol Names

Symbols Conventions

Description:
Symbols names, such as constants, devices, and variables, have typically been uppercase in AXCESS since symbols are not case sensitive. NetLinx has not added case-sensitivity to symbols. However, we will move to a new standard of uppercase and mixed case. When using all uppercase, it is more readable to use underscores, such as nMY_VAR. When using mixed case, the underscore is not needed, such as nMyVar. Here is a summary, followed by examples:

KEYWORDS	USE ALL UPPERCASE
RUN-TIME FUNCTIONS	USE ALL UPPERCASE
DEFINE_DEVICE	Use Mixed Case
DEFINE_CONSTANT	USE ALL UPPERCASE
DEFINE_TYPE	Use Mixed Case
DEFINE_VARIABLE	Use Mixed Case
DEFINE_CALL	Use Mixed Case or ALL UPPERCASE
DEFINE_FUNCTION	Use Mixed Case or ALL UPPERCASE
COMMENTS	Use Mixed Case

Devices

Description:
Devices in AXCESS were always a single integer and could be assigned to variables without type casting problem. In NetLinx, all devices have a DEV structure to contain all of the device data (D:P:S). All entries in the DEFINE_DEVICE section are implicitly defined as DEV types. Therefore, we should add the prefix ‘dv’ to all device definitions. An example would be:

```
DEFINE_DEVICE
dvVCR = 5001:1:0        // IR #1    Sony SVO-1630     HC:RMV200   s216.irl
                        //          MODE: IR    Carrier: Off      FG10-006
```

In NetLinx, the use of virtual devices will become more widely used. Virtual devices have a range of 32768-36863, but for simplicity, it is recommended to start virtual devices at 33001. The prefix 'vdv' should be added to all virtual device definitions. Virtual devices will be used in DEFINE_COMBINE, COMBINE_DEVICES, and UNCOMBINE_DEVICES (keywords). An example would be:

```
DEFINE_DEVICE
vdvVirtual  = 33001:1:0 // Virtual control over this system

DEFINE_COMBINE
(vdvVirtual, dvSomeTP)  // Note: first device must be virtual
```

Constants

Description:

Constants in NetLinx will not carry the Hungarian notation that variables will. Constants will be defined with all uppercase using underscores. This will help constants stand out from variables within the code.
Here are some recommendations:

```
DEFINE_CONSTANT

// Source selections
SRC_VCR     =     1
SRC_DVD     =     2

// House zones
ZN_MBR      =     1           // Master Bedroom
ZN_KITCHEN  =     2           // Kitchen


// DVD Specific IR functions
DVD_CUR_UP  =     51          // Cursor up
DVD_CUR_DN  =     52          // Cursor down
DVD_CUR_L   =     53          // Cursor left
DVD_CUR_R   =     54          // Cursor right
DVD_ENTER   =     21          // Enter
```

User Defined Types (Structures)

Description:
When defining a structure, use the _s notation as the structure type definition. The actual name of the variable that uses a structure will carry a Hungarian notation of a prefixed 'u', and each data type within the structure will carry it's own prefix (see variables). Structures can be easily seen, because they use the dot operator (uUserType.sName = 'Test'). Here is the recommended notation for NetLinx programming user-defined data types:

```
DEFINE_TYPE

STRUCTURE _sUSER_TYPE
{
  CHAR      sName[20]               // String array
  INTEGER   nFlag                   // Integer
}

DEFINE_VARIABLE

_sUSER_TYPE uUserType               // User defined type
_sUSER_TYPE uUserTypeArray[]        // Array of user defined types
```

Variables

Description:
Variables in AXCESS were always 1 of 3 types: integers, 8-bit arrays and integer (16-bit) arrays. With NetLinx, we now have many more variable types including user-defined types. To accommodate these new types, NetLinx programming standard will adopt Hungarian notation style. Hungarian notation is a method of specifying the variable type in the name of the variable name itself. This helps to eliminate confusion when writing code.
Variables should be declared as VOLATILE (or defined as STACK_VAR in subroutines/functions) wherever a non-volatile variable is not needed. The NetLinx master contains more volatile memory than non-volatile and when the programs use large chunks of memory, you can run out of non-volatile more quickly.

Here is the recommended notation for NetLinx programming intrinsic data types:

```
DEFINE_VARIABLE

CHAR        cSomeChar               // 8 bit unsigned integer
WIDECHAR    wcSomeWideChar          // 16 bit unsigned integer (for Unicode)
INTEGER     nSomeInt                // 16 bit unsigned integer
SINTEGER    snSomeSignedInt         // 16 bit signed integer
LONG        lSomeLong               // 32 bit unsigned integer
SLONG       slSomeSignedLong        // 32 bit unsigned integer
FLOAT       fSomeFloat              // 32 bit signed floating point
DOUBLE      dSomeDouble             // 64 bit signed floating point
DEV         dvSomeDev               // Device Variable
DEV         vdvSomeVirtualPnl       // Virtual Panel
DEVCHAN     dcSomeDevChan           // Device / Channel variable
DEVLEV      dlSomeDevLev            // Device / Level variable

CHAR        cSomeChar[]             //  Character array – binary data
CHAR        sSomeString[]           //  String array – ASCII data
WIDECHAR    wsSomeWideChar[]        // Wide String array
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
DEV         dvSomeDev[]             // Array of Devices
DEVCHAN     dcSomeDevChan[]         // Array of Device / Channel
DEVLEV      dlSomeDevLev[]          // Array of Device / Level

_sUSER_TYPE uUserType               // User defined type
_sUSER_TYPE uUserTypeArray[]        // Array of user defined types
```

6. Define Device Section

Sample DEFINE_DEVICE

Description:
The DEFINE_DEVICE section should include as much data about the devices as possible. Each declaration should include the dv (or vdv) notation. The device names should be descriptive and include any notation for the device's location (example: dvTPMbr1A…..denotes TP # (1), first device of panel (A), location (master bedroom)).

The comments to the right of the device should include information such as AMX device type, make/model of equipment being controlled, FG# of cable being used, hand control, dos filename of IR file, baud rate, etc. It is sometimes necessary to break up all of this information on it's own line to make it more readable. Remember that the more information put in this section will greatly increase technical support later down the road when this project is completed.

Example of define device section is below:

```
DEFINE_DEVICE

// All axlink devices are to be 1-255
dvTpMbr1a     = 128:1:0   // AXT-CA10 (main control panel)
dvTpMbr1b     = 129:1:0   // AXT-CA10 (second device of main control panel)
dvTPI         = 132:1:0   // AXB-TPI  (second physical control panel)

// Netlinx master card frames NXF-?? Start at 301 (Frame #25)
dvVCR         = 301:1:0   // NXC-COM2 PORT 1
                          // Device: JVC HRS365U
                          // Parameters: 9600,N,8,1
dvDVD         = 301:2:0   // NXC-COM2 PORT 2
                          // Device: Pioneer DVP4700
                          // Parameters: 9600,N,8,1
dvDSS         = 302:1:0   // NXC-IRS4 PORT 1
                          // Device: Sony SAT-B1
                          // HC:RMY129
                          // DOS: sony0358.irl
                          // Parameters: IR/CAROF CONTROL-S

// NXI and NXM start at 5001
dvPROJ        = 5001:1:0  // NXC-COM2 PORT 1
                          // Device: SONY VPH90
                          // Parameters: 38400,N,8,1
dvVolume      = 5096:1:0  // NXMC-VOL4
                          // Vol channel 1&2: Program volume
                          // Vol channel 3: Microphone volume

// Any Landmark switcher start at 6001
dvSwitcherVid = 6001:1:0  // Landmark VS8
dvSwitcherAud = 6002:1:0  // Landmark AS8

// Any P3 device start at 7001
dvMixer       = 7001:1:0  // Yamaha mixer

// Any Internet Inside device start at 8001
dvDBASE       = 8001:1:0  // Database Plus

// Any ICSP panel or web panel start at 10001
dvIMS         = 10001:1:0 // Landmark IMS
dvWeb1A       = 10002:1:0 // Internet Inside web panel (client 1)

// VIRTUAL DEVICES START AT 33001
vdvTpMbr1A    = 33128:1:0 // Virtual panel for dvTpMbr1a
vdvTpMbr1B    = 33129:1:0 // Virtual panel for dvTpMbr1b

vdvTPI        = 33132:1:0 // Virtual panel for dvTPI


// Outside of DEFINE_DEVICE, all comments start at column 37
DEFINE_COMBINE                      // First device must be a virtual device
( vdvTPI,dvTPI )                    // Combines virtual Tp with axlink Tp
```

Panel Combining Recommendations

Description:
In an Axcess system, it was common practice to DEFINE_COMBINE touch panels. This is still available in NetLinx, however it is recommended to use DEV arrays of panels instead. There is a sample program below.

The most common problem DEFINE_COMBINE is when all panels are "almost" identical, but one panel needs to route preview video to this destination, while another panel needs to route preview video to another destination. In this circumstance, with the panels combined, you would need to assign unique channel codes to each button to make it work and hope that other button was never added to the other panel.

To eliminate this problem, it is now recommended to put panels in a DEV array. This will provide DEFINE_COMBINE functionality, as well as let the program determine which panel actually pressed the button to perform these "almost" identical functions. This is always recommended, even when the system only has one panel, because if a web panel was ever added (for example), it could simply be added to the DEV array.

Here is the example:

```
DEFINE_DEVICE
dvTp1a = 128:1:0        // AXT-CV10 #1
dvTp2a = 132:1:0        // AXT-CV10 #2


DEFINE_VARIABLE
dvPnls[] = {dvTp1a, dvTp2a}


BUTTON_EVENT[dvPnls,1]              // Select source and preview
{
  PUSH:
  {
    SetRoomSource (SRC_VCR)

    IF(BUTTON.INPUT.DEVICE = dvTp1a)
      SetSourcePreview (SRC_VCR, 1)
    ELSE IF(BUTTON.INPUT.DEVICE = dvTp2a)
      SetSourcePreview (SRC_VCR, 2)
  }
}
```

7. Events Sections

Data Events

Description:
All device configurations should take place in the online event for each individual device. With this, the system should be configured for correct operation at device online events without any other technician intervention.

For IR devices, setup of mode (IR/Serial) and carrier (on/off) should be configured. It is also customary to provide any default feedback for IR controlled devices as well. For example, the SYSTEM_CALL 'FUNCTION' (dvVCR,STOP,0) should be used in the IR Online event.

```
DATA_EVENT[dvVCR]
{
  ONLINE:
  {
    SEND_COMMAND Data.Device,'SET MODE IR'
    SEND_COMMAND Data.Device,'CARON'
    SYSTEM_CALL 'FUNCTION' (Data.Device,2,0)  // Stop deck
  }
}
```

For 232 devices, setup of baud rate, parity, data bits, stop bits, and 485 status should be configured. With 232 devices, it is recommended to use CREATE_BUFFER in startup. Then parse over the buffer under the STRING event. All buffer parsing should be done with a separate subroutine, usually passing the device number, the buffer, and sometimes an index pointer as parameters. A single buffer parsing routine could then be used to parse data for many different devices of the same type.

```
DATA_EVENT[dvVideoSwt]
{
  ONLINE:
  {
    SEND_COMMAND Data.Device,'SET BAUD 9600,N,8,1 485 DISABLED'
  }
  OFFLINE:
  {
  }
  STRING:
  {
    WHILE(FIND_STRING(VideoSwtBuff,”13,10”,1))
    {
      CALL 'Parse Video Swt Buff' (dvVideoSwt,1,VideoSwtBuff)
    }
  }
}
```

Button Events

Description:
All button presses should incorporate the BUTTON_EVENT. No pushes/releases should occur in DEFINE_PROGRAM section. Where it is applicable to group common buttons, use a DEVCHAN set. Where it is not applicable, simply use the [device,channel].

```
BUTTON_EVENT[dcSrcSelect]           // Source selections
{
  PUSH :
  {
    SetRoomSource (GET_LAST(dcSrcSelect))
  }
}

BUTTON_EVENT[dvTpMbr1a,1]           // System power on
{
  PUSH :
  {
    SetRoomPower (1)
  }
}
```

Channel Events

Description:
There are no special requirements for channel events. However, it may require a touch panel bargraph to be updated with a previous level in a mute on/off condition.

```
CHANNEL_EVENT[dvVOL,3]
{
  ON :
  {
    SEND_LEVEL dvTpMbr1a,1,0
  }
  OFF :
  {
    SEND_LEVEL dvTpMbr1a,1,nPgmLvl
  }
}
```

Level Events

Description:
Under a level event is where you should update the level on a touch panel. This used to be done in Mainline with Axcess. It will still work in mainline, but it is recommended to move it under the level event.

```
LEVEL_EVENT[dvVOL,1]
{
  nPgmLvl = LEVEL.VALUE       // NOTE: CREATE_LEVEL not needed

  IF([dvVOL,3])
    SEND_LEVEL dvTpMbr1a,1,0
  ELSE
    SEND_LEVEL dvTpMbr1a,1,nPgmLvl
}
```

8. Functions and Subroutines

Functions

Description:
It is recommended that a function call be used with a switch...case to iterate through all of the available sources (VCR, DVD, camera, etc) supported in the program. Each of these sources should be declared as a constant to make the calls easier to read. Although case is ignored, it is recommended the name of the function be mixed case. For example:

```
DEFINE_FUNCTION INTEGER SetRoomSource (INTEGER nSRC)
{
  SWITCH (nSRC) :
  {
    CASE SRC_VCR :
    {
      SEND_STRING dvSWT,"'CI1O1T'"
      CALL 'SetProjFn' (dvPROJ,1,FN_PWR_ON)
      PULSE[dvRelay,SCREEN_DOWN]

      RETURN (SRC_VCR)
    }
    CASE SRC_CASS :
    {
      SEND_STRING dvSWT,"'CI2O1T'"
      CALL 'SetProjFn' (dvPROJ,1,FN_PWR_OFF)
      PULSE[dvRelay,SCREEN_UP]

      RETURN (SRC_CASS)
    }
  }
}
```

Subroutines

Description:
It is recommended that a subroutine be used with a switch...case to iterate through all functions of this particular device supported in the program. For example, if a 232-controlled video projector is included in the program, there may be a subroutine to provide specific functions for that projector such as power on, power off, video mute on, video mute off, and input selects. All of these functions should be declared in the constant section to make the calls easier to read. This subroutine should contain feedback and the actual 232 command. It should not contain any wait statements or logic. This will keep the call generic and useable for many different devices. The logic should be included where the subroutine is called.

```
DEFINE_CALL 'SetProjFn' (DEV dvPROJ, INTEGER nIDX, INTEGER nFN)
{
  SWITCH (nFN)
  {
    CASE FN_PROJ_ON :
    {
      ON[dvPROJ,FN_PWR]
      SEND_STRING dvPROJ,"'C01',13"

      nVidMute[nIDX] = 0
      nVidInp[nIDX] = 0
    }
    CASE FN_PROJ_OFF :
    {
      OFF[dvPROJ,FN_PWR]
      SEND_STRING dvPROJ,"'C00',13"

      nVidMute[nIDX] = 0
      nVidInp[nIDX] = 0
    }
    CASE FN_PROJ_INP_VID1 :
    {
      SEND_STRING dvPROJ,"'C07',13"

      nVidMute[nIDX] = 0
      nVidInp[nIDX] = FN_PROJ_INP_VID1
    }
    CASE FN_PROJ_INP_RGB1 :
    {
      SEND_STRING dvPROJ,"'C05',13"

      nVidMute[nIDX] = 0
      nVidInp[nIDX] = FN_PROJ_INP_RGB1
    }
    CASE FN_PROJ_VID_MUTE_ON :
    {
      SEND_STRING dvPROJ,"'C0N',13"

      nVidMute[nIDX] = 1
    }
    CASE FN_PROJ_VID_MUTE_OFF :
    {
      SEND_STRING dvPROJ,"'C0F',13"

      nVidMute[nIDX] = 0
    }
  }
}

BUTTON_EVENT[vdvVIRTUAL,1]          // Projector video mute
{
  PUSH:
  {
    IF([dvPROJ,FN_PWR])
      CALL 'SetProjFn' (dvPROJ,1,FN_VID_MUTE_ON)
  }
}
```

9. Master to Master (M2M)

Brief Description

Description:
With an M2M system, the NetLinx master needs a path to communicate with other masters. This is accomplished with entries in a NetLinx master's URL list of the other NetLinx masters. Then when a program defines a device that is on another system, the NetLinx master will open up communication with that other master to exchange data such as strings, commands, channels, pushes, etc between each master. Therefore, you as a programmer are not even aware that the device is on another system.

External device classifications

When using external devices on other masters, you should determine whether this device has a low-overhead classification in regards to communication or whether this external device will require a large amount of data to be transferred , i.e. a high-overhead classification.

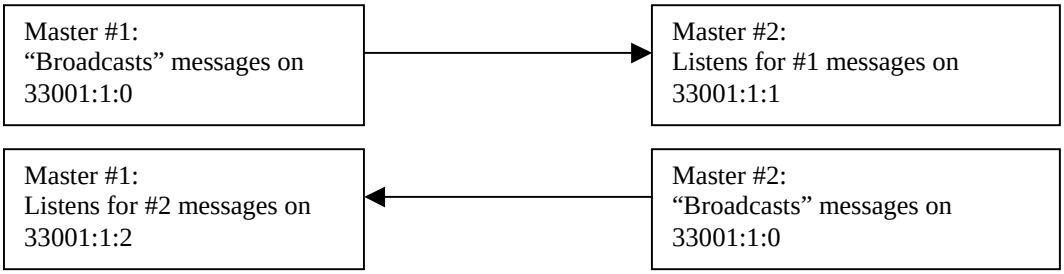
First, let's take a low-overhead device such as an Extron SW6 switcher. It is considered low-overhead because it has a limited number of inputs and outputs and it would be uncommon to try and make several switches back to back with this device.

Next, a high-overhead device would be something like an Extron Matrix 6400. It is considered high-overhead because it has a large number of inputs and outputs and it would be common to make several switches back to back with this device. With this in mind, assume that 5 masters had to make I/O switches on this device, you could run the risk of simultaneously trying to make several I/O routes on this switcher from all 5 masters. It would be a better idea to only let the NetLinx master that is physically attached to this device, do all of the communications with it. Then, you would have all masters add commands to a command queue on this master, where all of the switches could be made through this queue. How do other masters add commands to this queue? Simple, use a virtual device and send commands to this virtual device from all of the masters. This virtual device then processes the data.text and throws this switching command into the queue.

Note, the bottleneck is not an issue with the NetLinx master, but rather an issue with the 232 communications with the external device. Typically the 232 buffer of an external device is not designed to take on all of the traffic that an M2M system could send to it.

M2M Inner-Communications

Sometimes it is necessary to communicate between NetLinx programs. With an Axcess system, we did this with a 422 data link between masters. To do this in NetLinx, we use virtual devices defined on each master in the M2M system. It is a good idea to create a "broadcast" virtual device for each master. That master would "broadcast" messages out and if another master wanted to listen, that master would create the same virtual device from the other master's system. This master would also have it's own "broadcast" virtual device. NOTE: You have to be careful that the replies to these commands don't create other replies. When this happens, your NetLinx master's input and output LED will come on solid because these masters are replying to each other's message. This is an undesirable condition and should be avoided.



A common use would be sharing variables between masters. This is accomplished with VARIABLE_TO_STRING for sending and STRING_TO_VARIABLE to receive. You should refer to the "NetLinx Programming Language" instruction manual for more details.

For further information regarding M2M, please refer to the "NetLinx Programming Language" instruction manual. It can be found on the AMX website.

10. Modules

Overview

Modules are NetLinx programs that are to be easily dropped into a NetLinx program to communicate with a particular device. The module takes all of the complex code and hides it behind the module as a *.tko file. Then, a simpler API is exposed to the NetLinx program for external control. Note that it is common to label the NetLinx program as the caller, since that program makes "calls" to the module (via send commands, channels, etc).

When creating modules, they should never define other modules. Instead, all modules should be defined in the same location.

For further information regarding modules, please refer to the "NetLinx Programming Language" instruction manual. It can be found on the AMX website.

Module Naming

The naming convention of a module is 'MFG_MODEL_<type>' where type would be a classification of module such as COMM (Communication) or UI (User Interface). There could be different UI <types> based upon each device being controlled.

Using an AXI for Parameters

Some modules may require large parameter lists, for button channel arrays especially. Therefore, it is a good idea to create an include file where all of these parameters are defined. This include file should also contain the definition of the modules. This AXI file would typically be included after the DEFINE_DEVICE section of the caller's NetLinx program. Within the AXI file, the module definitions typically would come after the DEFINE_START section. Other parameter lists may be within the DEFINE_VARIABLE section.

Note, some UI modules may allow the caller to enter a NO BUTTON condition, which really means this UI doesn't require the functionality of the button. In system calls, NO BUTTON is defined as 0. In NetLinx, using 0 under a button event provides the functionality that ALL pushes will iterate through the button event, regardless of whether they are included in the DEVCHAN or INTEGER button array. This could be undesirable, since all pushes would iterate through any button events that had 0 as a channel. So, instead of using 0 for NO BUTTON, you should choose a button that is out of the range of 1-255. I recommend using 257 for NO BUTTON.

Note, as of now, all common module definitions should be included one after the other. For example, this is the proper way to add multiple module definitions:

```
PROGRAM NAME ‘AMX_COMM MODULES INCLUDE FILE’

(***********************************************************)
(*               CONSTANT DEFINITIONS GO HERE              *)
(***********************************************************)
DEFINE_CONSTANT

NL_NO_BUTTON = 257


(***********************************************************)
(*               VARIABLE DEFINITIONS GO HERE              *)
(***********************************************************)
DEFINE_VARIABLE

(*** AMX_UI button parameters ***)
INTEGER nUiBtnList1[] =
{
  201,                  // UI Button #1
  202,                  // UI Button #2
  NL_NO_BUTTON          // UI Button #3 (Not required)
}

(***********************************************************)
(*               STARTUP CODE GOES HERE                    *)
(***********************************************************)
DEFINE_START


(***********************************************************)
(*              MODULE DEFINITIONS GOE HERE                *)
(***********************************************************)
DEFINE_MODULE ‘AMX_COMM’ mdlCOMM1(vdvSomeDev1, dvSomeDev1)
DEFINE_MODULE ‘AMX_COMM’ mdlCOMM2(vdvSomeDev2, dvSomeDev2)

DEFINE_MODULE ‘AMX_UI’ mdlUI1(vdvSomeDev1, dvSomePnl1, nUiBtnList1)
DEFINE_MODULE ‘AMX_UI’ mdlUI2(vdvSomeDev2, dvSomePnl2, nUiBtnList2)
```

This method is wrong:

```
PROGRAM NAME 'AMX_COMM MODULES INCLUDE FILE'

(***********************************************************)
(*               CONSTANT DEFINITIONS GO HERE              *)
(***********************************************************)
DEFINE_CONSTANT

NL_NO_BUTTON = 257      // NetLinx no button


(***********************************************************)
(*               VARIABLE DEFINITIONS GO HERE              *)
(***********************************************************)
DEFINE_VARIABLE

(*** AMX_UI button parameters ***)
INTEGER nUiBtnList1[] =
{
  201,                  // UI Button #1
  202,                  // UI Button #2
  NL_NO_BUTTON          // UI Button #3 (Not required)
}

(***********************************************************)
(*               STARTUP CODE GOES HERE                    *)
(***********************************************************)
DEFINE_START


(***********************************************************)
(*              MODULE DEFINITIONS GOE HERE                *)
(***********************************************************)
DEFINE_MODULE 'AMX_COMM' mdlCOMM1(vdvSomeDev1, dvSomeDev1)
DEFINE_MODULE 'AMX_UI' mdlUI1(vdvSomeDev1, dvSomePnl1)

DEFINE_MODULE 'AMX_COMM' mdlCOMM2(vdvSomeDev2, dvSomeDev2)
DEFINE_MODULE 'AMX_UI' mdlUI2(vdvSomeDev2, dvSomePnl2)
```

COMM Module

This is the module that will communicate with the external device. It typically requires 2 parameters, the first being the virtual device and the second being the actual device. This COMM module will do all of the buffer parsing, device polling, and device control that may be required. This COMM module will also use the virtual device to expose feedback, levels, send commands (for controlling the external device), and string events (for getting data back from the module about the device). This is referred to as the custom API for this device. This API will require a text document regarding how to use the module.

This COMM module should always queue up the commands when it is necessary. That way the caller could stack several commands and expect the module to execute them without any knowledge of the delays required to make all of the commands go out.

UI Module

This is the module that will be the User Interface between the touch panel and the actual device. The parameter list typically contains the virtual device first, a DEV type touch panel, and groups of INTEGER arrays for channel codes and variable text. When multiple panels are required to control the same device, it is up to the caller to determine whether the main NetLinx program should combine devices and pass a virtual device in as the touch panel, or whether multiple calls to the same UI module with different panels (parameters) will be needed.

This UI will be custom for each external device. Some devices may require multiple UI modules. For instance, a security system may require a UI module for a security control and a separate UI module for thermostat control.

These UI modules will communicate with the COMM module through the API. The idea of the UI module is to easily drop in a touch panel page along with the UI module for control of this device with little knowledge of how the device actually works. Of course, if the programmer does not like the functionality of the UI module, they will be able to communicate with the COMM module using the same API and their own custom interface.

When it comes time to define button lists for the UI module, you should consider whether you will require the caller to pass in a DEVCHAN set, or a single DEV for the panel and an INTEGER array for the channel numbers. Each has it's own advantages. The DEVCHAN method allows all channel codes of the list to be on multiple devices of a touch panel. But, this method is not good when you need to see an ONLINE data event for a panel to update variable text. If this is a requirement, the single DEV and an INTEGER array for channel numbers may be a better solution.

It should be noted that a UI module could contain "hidden" features using NL_NO_BUTTON (similar to system calls). For example, a button list may contain both an ON and OFF element for channel codes. This would be for a UI that would require both discrete buttons. However, it is more common that a UI contain a single PWR button that toggles. This can be accomplished in a module by using the NL_NO_BUTTON channel for the OFF element. This way, when the ON button is pressed, the module can see if the OFF button is defined as a no button condition, then the module will make the ON button turn into toggling functionality. You should never assume this is how a module will work. You should review the API to see if it contains notes about these "hidden" features.

Example of "hidden" features:

```
MODULE_NAME='AMX_UI' (DEV vdvComm, DEV dvPnl, INTEGER nBtnList[])

BUTTON_EVENT[dvPnl,nBtnList]
{
  PUSH :
  {
    SWITCH(GET_LAST(nBtnList))
    {
      CASE 1 :          // Button - Power On
      {
      (*** Provide "hidden" toggling functionality ***)
        IF(nBtnList[2] = NL_NO_BUTTON)
        {
          IF([vdvComm,PWR_FB])
            SEND_COMMAND vdvComm,'POF'
          ELSE
            SEND_COMMAND vdvComm,'PON'
        }
        ELSE
        {
          SEND_COMMAND vdvComm,'PON'
        }
      }
      CASE 2 :          // Button - Power Off
      {
        SEND_COMMAND vdvComm,'POF'
      }
    }
  }
}

DEFINE_PROGRAM

(*** UI FEEDBACK ***)
[dvPnl,nBtnList[1]] = [vdvComm,PWR_FB]

IF(nBtnList[2] <> NL_NO_BUTTON)
  [dvPnl,nBtnList[2]] = (![vdvComm,PWR_FB])
```

Module Diagram