383-Netlinx Event Programming Rules and Parameters

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 is 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 be defined at compile time. In addition, there are 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.

Example 1

```
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 values were defined at compile time, the event is entered into the event table correctly. Let's look at another example.

Example 2

```
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":

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, channels, 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) variables. At compile time, these variables contain a value of 0. See example below.

Example 4 (Don't Do This!)

```
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'"

  }

}
```

This will have the effect you expect from the button, but probably for a different reason than you expect. 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:

Example 5 (Don't Do This!)

```
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.
- Known limitations:
    - The number of BUTTON_EVENT, CHANNEL_EVENT, or LEVEL_EVENT triggers is limited to 4000 per event statement.
    - The number of effective DEV's is limited to 50, even if the DEV array is larger.
    - The number of effective INTEGER's is limited to 4000, divided by the length of the DEV array - even if the DEV array is larger than 50.

Note: This means if you try to trigger an event with a DEV array of 100 devices, the INTEGER array will be limited to 4000/100 = 40; i.e., the buttons above index 40 won't work.

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

Example 6

```
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 an event using an array as a parameter. 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.