Marshalling Protocol (Group of Bytes)

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 known lengths do not have a size prefix). The specific formats of all the supported types are described below.

Type	Description	Stream Format
`BYTE`	Unsigned char/byte value.	`0xE1`
`WORD`	Unsigned short value.	`0xE2`
`DWORD`	4-byte value (could be an unsigned long integer or a float).	`0xE3`
`QWORD`	8-byte value (could be an unsigned Quad-word or a double).	`0xE4`
`BYTESTR`	Sequence of BYTE's whose element count is <= 64K.	`0xE5`
`WORDSTR`	Sequence of WORD's whose element count is <= 64K.	`0xE6`
`DWORDSTR`	Sequence of DWORD's whose element count is <= 64K.	`0xE7`
`QWORDSTR`	Sequence of QWORD's whose element count is <= 64K.	`0xE8`
`LBYTESTR`	Large sequence of BYTE's whose element count can be > 64K (larger version of BYTESTR).	`0xE9`
`STRUCT`	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.	`0xEA`
`ENDSTRUCT`	Byte indicator for end of structure – not really a data type prefix.	`0xEB`
`ARRAY`	Array of any one of the types in this table whose element count can be > 64K. Each element in an array is self-descriptive. The type of the first element (byte after Length LSB) is the type of the entire array.	`0xEC`
`SKIP`	Byte indicator for space to be skipped in the input and NULL'ed in the marshalled output. This can be viewed as a NULL data type prefix.	`0xED`

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, and 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:

Type	Description	Stream Format
BYTE	Unsigned char/byte value. If var is an element of an array, name is replaced with <index></index> and the index value, and the type is optional. Typical, only <var><data>Data</data></var> is needed.	<var> <name>MyName</name> <type>ui1</type> <data>255</data> </var>
UWORD	Unsigned short value. If var is an element of an array, name is replaced with <index></index> and the index value, and the type is optional. Typical, 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> and the index value, and the type is optional. Typical, 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> and the index value, and the type is optional. Typical, 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> and the index value, and the type is optional. Typical, only <var><data>Data</data></var> is needed.	<var> <name>MyName</name> <type>ui4</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> and the index value, and the type is optional. Typical, 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> and the index value, and the type is optional. Typical, only <var><data>Data</data></var> is needed.	<var> <name>MyName</name> <type>float.IEEE.754.64</type> <data>4.56</data> </var>
STRUCT	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 <index></index> and the index value.	<struct> <name><MyName></name> <var>…</var> </struct>

ARRAY	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.	<array> <name><MyName></name> <type>Type</type> <curLength>100</curLength> <var><index>1</index>…</var> </array>...or ...<array> <name><MyName></name> <type>Type</type> <curLength>100</curLength> <struct><index>1</index>…</struct> </array>... or ...<array><name> <MyName></name> <type>Type</type> <curLength>100</curLength> <array> <index>1</index>…</array> </array>
Array – String encoding (Strings)	Array of unsigned characters. Data is encoded using String encoding (Section 2.2). Type and length are optional.	<array> <name><MyName></name> <type>Type</type> <curLength>100</curLength> <string>MyString</string> </array>
ARRAY – Binary Encoded	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.	<array> <name><MyName></name> <type>Type</type> <curLength>100</curLength> <encoded> <style>LE or BE</ style > <size>1,2,4,8</size> <data>01020304</data> </encoded> </array>

Marshalling Protocol: 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 string type, printable ASCII characters appear as ASCII, and non-printable characters appear as escaped decimal or hex code, **&#<decimal code>;** or **&#x<hex code>;**. An example string would be:

<data>My Name is Jimmy Buffet
</data>

or:

<data>My Name is Jimmy Buffet
</data>

Additionally, some characters have a more readable syntax. These characters are invalid in XML so the following characters can be also be encoded in this format:

Character	Escape Version
<	<
>	>
&	&
‘	'
"	"

Binary Array Encoding

Arrays can optionally be encoded/decoded as pairs of ASCII-encoded HEX. The pairs of ASCII-encoded HEX need 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>

<style>BE</ style >
<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.

Encoding and Decoding: Binary and XML

There are six special 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 an ASCII text only XML document.

The binary routines are ideal when sending data from one NetLinx system to another NetLinx system over RS232 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 do intend to edit the file later since it is entirely ASCII text. It should be noted that while the XML is more universal, the XML format however is not very compact. The XML routines encode and decode a variable non-sequentially meaning that the order and type of the variables do not need to match on both the encoding and decoding side.

```
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 *)
(***********************************************************)
DEFINE_DEVICE
dvTP = 128:1:0
(***********************************************************)
(* CONSTANT DEFINITIONS GO BELOW *)
(***********************************************************)
DEFINE_CONSTANT
nFileRead = 1
nFileWrite = 2
(***********************************************************)
(* DATA TYPE DEFINITIONS GO BELOW *)
(***********************************************************)
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
}
(***********************************************************)
(* VARIABLE DEFINITIONS GO BELOW *)
(***********************************************************)
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
(***********************************************************)
(* STARTUP CODE GOES BELOW *)
(***********************************************************)
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
(***********************************************************)
(* THE EVENTS GOES BELOW *)
(***********************************************************)
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 DICS *)
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 DICS *)
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)"
}
}
(* 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 DICS *)
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 *)
(***********************************************************)
```