LONMARK® Resource File Developer's Guide

Introduction

LonMark resource files are files that define the components of the external interface for one or more LONWORKS® devices. These files allow installation tools and operator interface applications to interpret data produced by a device and to correctly format data sent to a device. They also help a system integrator or system operator to understand how to use a device and to control the LONMARK objects on a device. LONMARK resource files are available that define the standard components used in the external interface of a device. Device manufacturers must create user-defined resource files for any user-defined components defined within the external interface of a device.

There are four types of LonMark resource files. These are described in this guide, and are summarized in the following table:

Type File	Defines network variable, configuration property, and enumerated types. LONMARK standard network variable and configuration property types are defined in the STANDARD.TYP file. Type files have a .TYP extension.
Functional Profile Template	Defines functional profiles that are used for describing LONMARK objects. A functional profile specifies the mandatory and optional network variable and configuration property components of a LONMARK object. Some of the optional network variables and configuration properties may not be present on a particular LONMARK object derived (re-defined) from the standard functional profile template of that LONMARK object. LONMARK standard functional profiles are defined in the STANDARD.FPT file. Functional profile templates have a .FPT extension.
Format File	Defines display and input formats for network variable and configuration property types defined in a type file. Formats for the LONMARK standard network variable and configuration property types are defined in the STANDARD.FMT file. Format files have a .FMT extension.

Copyright ©

Language File	Defines language-dependent strings. There is a separate language file for each supported language. The language the file supports determines the extension of a language file. Two language files are currently available for the LONMARK standard type files; these are STANDARD.ENU for American English and STANDARD.ENG for British English.

The LONMARK standard resource files are updated periodically as the LONMARK Interoperability Association defines new standard types and functional profiles. The definitions in these files are described in the Standard Network Variable Type (SNVT) Master List, the Standard Configuration Property Type (SCPT) Master List, and standard functional profile documents. The latest version of the resource files, master lists, and functional profile documents are available at www.lonmark.org.

This guide describes how to create user resource files that may contain definitions of user-defined network variable types (UNVTs), configuration property types (UCPTs), enumerated types, functional profiles, and language-dependent strings. Every set of user resource files must contain exactly one type file, functional profile template, format file; and one or more language files.

Using Standard Types

A number of standard network variable and configuration property types (SNVTs and SCPTs) have been defined for use in applications which cover most of the data formats in the controls industries, as well as a number of standard functional profiles corresponding to specific functions which are common in specific controls industries (e.g. temperature sensor, lamp actuator). These standard definitions should be used in your applications whenever possible.

In some cases, you may find that there is a network variable or configuration property type that you want to use which is not defined in the SNVT and SCPT lists, perhaps a data type which is specific to your implementation or company. In this case, you may create user resource files, which allow you to define user network variable and configuration property types (UNVTs and UCPTs), as well as user functional profile types (UFPTs). If you believe that the UNVTs, UCPTs, and UFPTs you create would be useful to others in your industry, you may propose them for adoption by the LONMARK Association.

Resource File Utilities

The following utilities, available at www.lonmark.org, can be used to create and manage LONMARK resource files. Their use is described under *Creating LonMark User Resource Files* and *Format File Format*.

DRFUSRNV Type Preprocessor	Converts one or more header files into a preliminary user definition file called USER.DEF. The input header files define the structures, unions, and type definitions used in defining network variables, configuration properties, and application messages. This file is an intermediate file that you must edit, and rename, using a text editor, before converting to binary

	form.
TYPFILW Resource File Compiler	Converts a user definition file into a set of LONMARK resource files consisting of a type file, a language file, and a functional profile template file. The user definition file is a text file.
TYPFILR Resource File Reader	Converts a set of LONMARK resource files (i.e., a type file, a language file, and a functional profile template file) into a user definition file (.DEF extension). Any or all of these files can be empty, or contain data, but they all must exist. They all must contain the same scope selector and program ID reference.
FMT Format File Converter	Converts version 3 format files to version 2 format files. For use with legacy network tools that do not support version 3 format files. See *Format File Format* later in this document for more information.
MKCAT Resource Catalog Builder	Creates a resource file catalog. For use with legacy network tools that do not include a resource file catalog browser.

A new, easier to use utility, is also available in beta form. The beta version of this utility is called the LONMARK Wizard. It is also available at www.lonmark.org. This utility will be released in final form in 2001.

Managing Resource Files

There may be multiple sets of LONMARK resource files on a PC. In addition to the standard resource files, there may be one or more sets of user resource files from one or more manufacturers. Each set of resource files must be contained in a single folder, but there may be multiple sets of resource files. For example, a network may contain devices from several different manufacturers, and each manufacturer may supply their own set of resource files with type, functional profile, format, and language information specific to their devices. Each set of resource files may be kept in a separate folder. These folders are typically installed in the LONWORKS "\Types\User" folder, each identified by the manufacturer name.

An open application-programming interface (API) is available for accessing these files. This API is called the LONMARK Resource File API. The LONMARK Resource File API maintains a resource file catalog with a filename of LDRF.CAT. The resource file catalog contains a list of all resource files and their locations to be used by the LONMARK Resource File API.

To be able to associate a resource file with a network variable, configuration property, or LONMARK object on a device, each set of resource files must be associated with a particular program ID, a range of program IDs, or with all program IDs. The type of association is called the *scope* of the resource file, and the scope is specified using a *scope selector*. The scope selector for a resource file specifies what part or parts of a device's program ID should be used when selecting the resource file.

The scope selector is an integer value between 0 and 6 as defined in the following table:

Scope Selector	*Scope Definition*
0	Used for resource files containing standard definitions for all devices from any manufacturer. This selector value can only be used for resource files created by the LONMARK Interoperability Association.
1	Used for resource files containing standard definitions for all devices of a specified device class from any manufacturer. This selector value can only be used for resource files created by the LONMARK Interoperability Association.
2	Used for resource files containing standard definitions for all devices of a specified device class and device subclass from any manufacturer. This selector value can only be used for resource files created by the LONMARK Interoperability Association.
3	Used for resource files containing user definitions for all devices of a specified manufacturer. This selector value can be used by a manufacturer for resource files that apply to all of the manufacturer's devices.
4	Used for resource files containing user definitions for all devices of a specified manufacturer and device class. This selector value can be used by a manufacturer for resource files that apply to all of the manufacturer's devices of a specific device class.
5	Used for resource files containing user definitions for all devices of a specified manufacturer, device class, and device subclass. This selector value can be used by a manufacturer for resource files that apply to all of the manufacturer's devices of a specific device class and subclass.
6	Used for resource files containing user definitions for all devices of a specified manufacturer, device class, device subclass, and model. This selector value can be used by a manufacturer for resource files that apply to a single device type.

For example, if a manufacturer released a set of LONMARK resource files with all type, format, and language information for all its devices, this set of files would have a scope selector of 3. If a LonMark resource file had a program ID of 80:00:01:05:01:02:04:00 and a scope selector of 3, all applications with 0:00:01 ("Echelon") as the manufacturer ID portion of their program ID would use the types in this file.

If a manufacturer was involved in making devices in more than one industry (such as HVAC and Lighting), they could release two sets of LONMARK resource files with a scope selector of 4. The standard LONMARK resource files discussed above have a scope selector of 0.

By using scope, LONMARK resource files are treated as a hierarchy of type definitions, with scope 0 at the top. Resource files may refer to other resource files above them in the scope hierarchy. For example, a file with a scope selector of 5 could contain references to scope 4, 3, and 0 resource files which match the relevant parts of the program ID.

Creating and Listing LONMARK Resource File Catalogs

Network tools may use the LONMARK Resource File API to automatically create and manage the resource file catalog, or to provide a resource file catalog browser. You can also use the MKCAT Resource Catalog Builder utility to manually create a resource file catalog, or list the contents of a resource file catalog.

To user the Resource Catalog Builder, place the MKCAT.EXE program into the same directory as the TYPFILW Resource File Compiler and the LCADRF32.DLL library. If the STANDARD.TYP file is in the same directory just type "mkcat" at the Windows command prompt. To list the contents of the catalog, type "mkcat -l". Any time you run Resource Catalog Builder it also refreshes the catalog. To add type files in other directories, enter the following command:

```
mkcat –a<type file path>
```

Be sure NOT to put a space after the "-a" and use a full path name for directories you add. Multiple add and list commands can be combined in one call to the Resrouce Catalog Builder.

For example, the following command adds the c:\lonworks\types directory to the catalog and list the contents of the catalog:

```
mkcat -ac:\lonworks\types -l
```

To delete entries, delete the LDRF.CAT catalog file and rebuild the catalog.

Creating LONMARK User Resource Files

To create user LONMARK resource files, follow these steps (these steps are described in more detail in the following sections):

1. Create user type header files that contain the C declarations of your custom network variable and configuration property types. The user type header files are Neuron C files with a .H extension.

2. Use the DRFUSRNV type preprocessor to convert the user type header files to a preliminary user definition file with the name USER.DEF.

3. Rename the user definition file to the name desired for the resource files.

4. Edit the preliminary user definition file to specify the scope selector and fill in information regarding the strings, network variable types, configuration property types, and functional profile templates.

5. Use the TYPFILW resource file compiler to convert the user definition file into user type, language, and functional profile template files.

6. Create a user format file with format definitions for the network variable and configuration property types defined in the user type file.

Step 1: Creating User Type Header Files

User type header files are used as the starting point for creating resource files. The header files are text files containing Neuron C type declarations. The files consist of a main header file and optional include files. Create a separate user type header file for each structure, union, and typedef declaration as described under *User Type Header File Format* at the end of this guide.

Step 2: Using the DRFUSRNV Type Preprocessor

The DRFUSRNV type preprocessor converts the user type header files into a preliminary user definition file. To use the DRFUSRNV type preprocessor, open a Windows command prompt window and type `drfusrnv` followed by the name of the main user type header file. Press the Enter key to continue. The type preprocessor creates a preliminary user definition file with a filename of USER.DEF.

Step 3: Renaming the User Definition File

Rename the preliminary user definition file that you created in step 2 to the name that you want to use for your resource file set. The default filename is USER.DEF. For example, if you are creating a manufacturer specific resource file, you may want to use your company name as the name of the resource file. The extension of the user definition file must be ".DEF".

Step 4: Editing the User Definition File

The user definition file is a text file containing definitions for user network variable types, user configuration property types, and user functional profiles. To create a user definition file, edit the preliminary user definition file that you created in step 2 and renamed in step three using any text editor. Add definitions as described under *User Type Definition File Format* at the end of this guide.

Step 5: Run the TYPFILW Resource File Compiler

The TYPFILW resource file compiler converts a user definition file into a set of LONMARK resource files consisting of a type file, a language file, and a functional profile template. Any or all of these files can be empty, or contain data independently. To use the resource file compiler, open a Windows command prompt window and type `typefilw` followed by the name of the user definition file. Press the Enter key to continue.

The resource file compiler reads the user definition file and any enumeration definition files and constructs a user type file, a language file, and a functional profile template. The user definition file and the enumeration definition files must be in the same folder.

The output file names will have the same base name as the input user definition file. So, if you use resource file compiler to convert a file named MYSTUFF.DEF, and you select US English, you will get output files named MYSTUFF.TYP, MYSTUFF.ENU, and MYSTUFF.FPT. The filename extension of the language file is set by the language chosen.

For example, a US English language file has the extension .ENU. None of the output files can exist prior to running the resource file compiler, or else it will print an error message and stop.

Step 6: Create a User Format File

A format file defines display and input formats for network variable and configuration property types defined in a type file. The format file references the type names in a type file so that format definitions can be associated with the appropriate data types, structures, and enumeration names.

The user format file is a text file, and you may use any text editor to create and edit a user format file. See ***Format File Format*** at the end of this guide for a description of the format of a Format file.

Resource File Format Reference

The following sections define the formats of the input files required for generating resource files.

User Type Header File Format

User type header files are used as the starting point for creating resource files. The header files are text files containing Neuron C type declarations. The files consist of a main header file and optional include files. Each structure, union, and typedef declaration is contained in a separate file. If the types have been previously defined in a Neuron C application, those declarations can be used as a starting point for creating the type declaration files.

The main header file contains one or more user types, in the form of C type definitions, using names beginning with "`UNVT_`" and "`UCPT`" (one of these prefixes must be used for user types). For example, this is a user type definition for cubic dimensions in millimeters:

```
typedef struct {
    unsigned long   mmWidth;
    unsigned long   mmHeight;
    unsigned long   mmDepth;
} UNVT_cubic_dimns;
```

LONMARK type declaration files must follow Neuron C declaration syntax as defined in the *Neuron C Programmer's Guide* and *Neuron C Reference Guide*, and must also meet the following additional requirements:

- Do not include Neuron C statements that are not typedef, structure, union, or enumeration definitions, or an include directive.
- All user types must begin with "UNVT_" or "UCPT".
    - Define each enumeration in a separate include file. Reference this file with an include directive. The format of this file must be similar to the format of the standard SNVT_???.H include files included with the Neuron C development tools

and available in the LONMARK standard resource files archive at www.lonmark.org. They must contain the "/* 0 */" comments as shown in this example, otherwise they will not be compile properly:

```
typedef enum hvac_hvt_t {
      /*  0 */    HVT_GENERIC,
      /*  1 */    HVT_FAN_COIL,
      /*  2 */    HVT_VAV,
      /*  3 */    HVT_HEAT_PUMP,
      /*  4 */    HVT_ROOFTOP,
      /*  5 */    HVT_UNIT_VENT,
      /*  6 */    HVT_CHILL_CEIL,
      /*  7 */    HVT_RADIATOR,
      /*  8 */    HVT_AHU,
      /*  9 */    HVT_SELF_CONT,
      /* -1 */    HVT_NUL = -1,
} hvac_hvt_t;
```

- Do not include nested structure or union declarations. Expand any nested declarations in the type declaration file. For example, the following table illustrates a nested declaration and how it may be expanded.

Nested Declaration	***Expanded Declaration***

struct a {
int f1;
int f2;
struct b {
int f3;
int f4;
} f5;
};
	struct b {
	int f3;
	int f4;
	};
	
	struct a {
	int f1;
	int f2;
	struct b f5;
	};

- Do not include compiler preprocessor commands. Expand any preprocessor commands in the type declaration file. For example, the following table illustrates a declaration with preprocessor commands and how it may be expanded.

Declaration with Preprocessor Commands	***Expanded Declaration***
#define uint unsigned int	typedef unsigned int uint;

struct a {
uint f1;
uint f2;
#ifdef TEST
int f3;
#endif
};
	struct a {
	uint f1;
	uint f2;
	};

- Do not include comma-separated typedef declarations. Expand any typedef declarations in the type declaration file. For example, the following table illustrates a declaration with a comma-separated typedef and how it may be expanded.

Declaration with Comma-Separated Typedef	*Expanded Declaration*
typedef unsigned int uint, *uint_p;	typedef unsigned int uint; typedef unsigned int *uint_p;

- Do not include multiple definitions for the same literal or type name. Rename any duplicate types.
- Do not declare unnamed bitfields. Instead, use names such as "reserved1" and "reserved2" for unused bitfields.

User Definition File Format

The user definition file is a text file containing definitions for user network variable type, user configuration property types, and user functional profiles. The user definition file consists of 5 sections, in the following order:

- Header
- Strings
- Network variable types
- Configuration property types
- Functional profile templates.

The contents of these sections are detailed in the following sections.

The exclamation point ("!") is used to denote the beginning of a comment. A linefeed automatically denotes the end of a comment, thus a multi-line comment would need to be preceded by a "!" at the beginning of every line.

Header Section

The header section consists of the following:

- Scope selector specification line
- Program ID specification line
- Language file specification line
- Four ASCII string lines: file creator, contact phone #, internet address, and uniform resource locator (URL)
- Type file specification line
- Four ASCII string lines: file creator, contact phone #, internet address, and URL
- Function profile template specification line
- Four ASCII string lines: file creator, contact phone #, internet address, and URL

The scope selector specification line sets the files' scope, using the following syntax:

<SEL *n*>

The *n* field is an integer from 0 to 6. See *Managing Resource Files*, earlier in this guide, for more information.

The program ID specification line sets the files' program ID value. The program ID always consists of eight hexadecimal bytes with the same format as the program ID of a LONMARK Association-compliant application, even though some of the values may not be used depending on the chosen scope—which determines the relevant bits of the program ID. For example, if the scope is set to 3, only applications that matched digits 2-6 of the program ID can use the types defined in this file. The program ID specification line uses the following syntax, where *nn* is one hexadecimal byte (the digits '0' to '9' and either the characters 'a'-'f' or 'A'-'F' may be used).

```
<ID nn nn nn nn nn nn nn nn>
```

The language file specification line sets the information needed to create a language file. This line consists of two integers for major and minor data version number, a three-letter language code (that becomes the file extension of the language file), and two integer pairs (scope and index references) specifying the strings to use for file description information and file creator information, respectively. When editing a user type definition file for another language, change the language code to the appropriate 3-letter extension. Typically, these language-dependent strings would be in the string file being created, so the scopes here would match the scope specified above. The resource file specification line uses the following syntax:

```
<RES majorVer minorVer language descSel descIndex creSel creIndex>
```

The fields are defined as follows:

Field	Description
MajorVer	Major version number for the language file. Set to 1 for the first version, and increment by one for each new version of the resource files containing a new type definition.
MinorVer	Minor version number for the language file. Set to 1 for the first version for a new major version, and increment by one for each new version of the resource file that does not contain a new type definition. For example, a new resource file containing a change to a validation range or a comment string would require a new minor version number, but not a new major version number.
language	Three-letter language code. This code becomes the extension for the language resource file.
descSel *descIndex*	Scope selector value and string index for a language file and a string within it that contains a description of this language file. The string will typically be in the string file being created, so the scope selector value will typically be equal to the scope selector defined in the scope selector specification line.
creSel *creIndex*	Scope selector value and string index for a language file and a string within it that contains the creator of this language file. The string will typically be in the string file being created, so the scope selector value will typically be equal to the scope selector defined in the scope selector specification line.

For example, the following language file specification line specifies data version 1.0 of a US English language file, with a description given by scope = 3, index = 2 (i.e. the second string in the string section), and creator given by string scope = 3, index = 1 (i.e. the first string in the string section):

```
<RES 1 0 ENU 3 2 3 1>
```

To create multiple language files, follow these steps:

1. Create the user type definition file as described in this section.
2. Compile using the `TYPFILW` resource file compiler, as described in the next section.
3. Uncompile using the `TYPFILR` resource file reader. This step causes any implicitly defined stings to explicitly appear in the user definition file in the strings section (see below).
4. Remove the body (but not the headers) of the <NVTs>, <CPTs>, and <FPTs> sections.
5. Translate the strings section into the new language and change the `language` field of the `RES` header.
6. Recompile using the `TYPFILW` resource file compiler to produce a new string file, but discard empty user type files and functional profile templates.

The resource file specification line is followed by four strings (each string beginning with a dollar sign ("$") on a single and separate line—there are no continuation characters). The strings are ASCII strings. The first three strings become the creator string in the file, and the fourth becomes the uniform resource locator (URL) string. The description string is automatically set in accordance with the type of the file. The first three strings, as stated above, are components of the creator string. The first string is the creator name, the second is the creator contact info (an address or phone number), and the third string is an Internet ID for an e-mail or web site address. An asterisk ("*") may be used to indicate a null string.

The type file specification line sets the information needed to create a type file. This line consists of two integers for major and minor data version number and two integer pairs (scope and index references) specifying the strings to use for file description information and file creator information, respectively. Typically, these language-dependent strings would be in the string file being created, so the scopes here would match the scope specified above. The type file specification line uses the following syntax:

```
<TYP majorVer minorVer descSel descIndex creSel creIndex>
```

The fields are defined as follows:

Field	Description
MajorVer	Major version number for the type file. Set to 1 for the first version, and increment by one for each new version of the resource files containing a new type definition.
MinorVer	Minor version number for the type file. Set to 1 for the first version for a new major version, and increment by one for each new version of the resource file that does not contain a new type definition. For example, a new resource file containing a change to a validation range or a comment string would require a new minor version number, but not a new major version number.
descSel	Scope selector value and string index for a language file and a string within it that contains a description of this type file. The

descIndex	string will typically be in the string file being created, so the scope selector value will typically be equal to the context selector defined in the scope selector specification line.
creSel *creIndex*	Scope selector value and string index for a language file and a string within it that contains the creator of this type file. The string will typically be in the string file being created, so the scope selector value will typically be equal to the scope selector defined in the scope selector specification line.

For example, the following type file specification line specifies data version 1.0 of a type file, with description given by scope = 3, index = 4 (i.e. the fourth string in the string section), and creator given by string scope = 3, index = 1 (i.e. the first string in the string file):

```
<TYP 1 0 3 4 3 1>
```

The type file specification line is followed by four strings (each string beginning with a dollar sign ("$") on a single and separate line—there are no continuation characters). The strings are ASCII strings. The first three strings become the creator string in the file, and the fourth becomes the URL string. The description string is automatically set in accordance with the type of the file. The first three strings, as stated above, are components of the creator string. The first string is the creator name, the second is the creator contact info (an address or phone number), and the third string is an Internet ID for an e-mail or web site address. An asterisk ("*") may be used to indicate a null string.

The functional profile template specification line sets the information needed to create a functional profile template. This line consists of two integers for major and minor data version number and two integer pairs (scope and index references) specifying the strings to use for file description information and file creator information, respectively. Typically, these language-dependent strings would be in the string file being created, so the scopes here would match the scope specified above. The functional profile template file specification line uses the following syntax:

<FPT *majorVer minorVer descSel descIndex creSel creIndex*>

The fields are defined as follows:

Field	**Description**
MajorVer	Major version number for the functional profile template file. Set to 1 for the first version, and increment by one for each new version of the resource files containing a new type definition.
MinorVer	Minor version number for the functional profile template file. Set to 1 for the first version for a new major version, and increment by one for each new version of the resource file that does not contain a new type definition. For example, a new resource file containing a change to a validation range or a comment string would require a new minor version number, but not a new major version number.
descSel *descIndex*	Scope selector value and string index for a language file and a string within it that contains a description of this functional profile template file. The string will typically be in the string file being created, so the scope selector value will typically be equal to the scope selector defined in the scope selector specification line.
creSel *creIndex*	Scope selector value and string index for a language file and a string within it that contains the creator of this functional profile

template file. The string will typically be in the string file being created, so the scope selector value will typically be equal to the scope selector defined in the scope selector specification line.

For example, the following functional profile specification line specifies data version 1.0 of a functional profile template, with description given by string scope = 3, index = 4 (i.e. the fourth string in the string section), and creator given by string scope = 3, index = 1 (i.e. the first string in the string file):

```
<FPT 1 0 3 4 3 1>
```

The functional profile template file specification line is followed by four strings (each string on a single and separate line—there are no continuation characters). The strings are ASCII strings. The first three strings become the creator string in the file, and the fourth becomes the URL string. The description string is automatically set in accordance with the type of the file. The first three strings, as stated above, are components of the creator string. The first string is the creator name, the second is the creator contact info (an address or phone number), and the third string is an Internet ID for an e-mail or web site address.

Strings Section

The strings section of the user definition file allows you to keep all language-dependent strings for a type file in one place. This allows you to easily create language files by translating the list of strings (i.e. it is only necessary to edit the strings section of the user definition file, and then recompile). These strings are referenced by the RES, TYP, and FPT header components (see *Header Section*). At any point in the file where a string would be used, a reference using the following format can be used instead:

^*scope*:*index*

The list of language-dependent strings begins with a single line marking the beginning of the section. This line is of the following form:

```
<STRs>
```

The strings are listed one per line (there may be intervening blank lines and comments). Each string must begin with a dollar sign ("$"), and that character is not part of the text of the string. Strings may not contain the exclamation point ("!"), since that character is used to begin a comment. The list of strings may be empty, but the line marking the beginning of the section must be present. In this case, an empty string file is created. Whatever strings are added to the string file in this section; in later sections there are string fields as well, and the TYPFILW resource file compiler will, in those later sections, reuse the same language-dependent string index wherever it appears. If any string in those later sections has not already been defined in the string file, it will be added to the string file at that time, using the next available index.

Therefore, no strings are required to be added in this section, though it is advisable to do so to more explicitly control the ordering of strings. The control of the string order permits the same indices to be used in each string file created for each language desired. This is a requirement for easy management of multiple language files.

Each language file has a unique extension, which identifies the language of the strings in that file. The following languages currently have defined extensions:

Czech	"csy"
Danish	"dan"
Dutch (Belgian)	"nlb"
Dutch (default)	"nld"
English (UK)	"eng"
English (US)	"enu"
Finnish	"fin"
French (Belgian)	"frb"
French (Canadian)	"frc"
French (default)	"fra"
French (Swiss)	"frs"
German (Austrian)	"dea"
German (default)	"deu"
German (Swiss)	"des"
Greek	"ell"
Hungarian	"hun"
Icelandic	"isl"
Italian (default)	"ita"
Italian (Swiss)	"its"
Norwegian (Bokmal)	"nor"
Polish	"plk"
Portuguese (Brazilian)	"ptb"
Portuguese (default)	"ptg"
Russian	"rus"
Slovak	"sky"
Spanish (default)	"esp"
Spanish (Mexican)	"esm"
Swedish	"sve"
Turkish	"trk"

There is no API support for two-byte (wide) character sets.

Network Variable Types Section

This section lists the network variable types defined in the user definition file. The network variable types section begins with the following line:

```
<NVTs>
```

This line may be followed by zero or more network variable type definitions. Each network variable type definition begins with a line that contains an index, the network variable type name, and the top-level type information. If the network variable type is a simple scalar type, this top-level type information is the only type information needed. Otherwise, more type information follows on subsequent lines in a C-style depth-first ordering.

The network variable type indices used in the type file must start at '1' with the first network variable type, and must increase sequentially for each subsequent network variable type. All user network variable type names must start with the five characters "UNVT_", and must not be longer than 16 total characters.

The type information consists of one or more keywords and parameters as described in the following table:

Type Keyword and Parameters	Definition
quad signed	A 32-bit signed value.
long signed	A Neuron C 16-bit long signed integer.
long unsigned	A Neuron C 16-bit long unsigned integer.
short signed	A Neuron C 8-bit short signed integer.
short unsigned	A Neuron C 8-bit short unsigned integer.
char signed	A Neuron C 8-bit signed character.
char unsigned	A Neuron C 8-bit unsigned character.
float	An IEEE standard 32-bit float.
enum *tagName fileName*	A Neuron C 8-bit signed enumeration. The *tagName* parameter is the enumeration tag name. The *fileName* parameter is the enumeration definition file name.
bitfield *type offset size*	A Neuron C bitfield. The *type* parameter must be the keyword `signed` or `unsigned`. The *offset* parameter is the bitfield offset in bits from the MSB. The *size* parameter is the bitfield size in bits; valid values are 1 to 8. The valid values of the *offset* parameter are 0 to (8 - *size*).

After the type line for a scalar, there are three string lines (any or all can be a single asterisk ("*"), indicating NULL string, instead). The first string line is the language-dependent name of the type (as opposed to the network variable type name, or the field name, both of which are programmatic constructs used in Neuron C). The second string line is additional commentary information about the type. Often, this is empty, but occasionally is used for hints about what sort of data is expected or typical. Finally, the third string is the language-dependent name of the units the data represents. For example, a long unsigned data type might be represented with the following lines:

```
15    UNVT_flow_micr    long  unsigned
$Flow volume
*
$microliters/second    ! my units
MIN MAX
NO SCALE
```

In the above example, the language-dependent string for the name is "Flow volume". The second string is the null string, denoted by a single asterisk. The third string (the units string) is "microliters/second". There are two more lines in the type definition seen above, and these are the range line and scaling factor line.

The range line exists for all scalar types, and can be coded as **MIN MAX** in the case of no range limitation, or as numeric integer (or floating point, where applicable) values representing the minimum and maximum legal values for the type. Any of the following is acceptable, provided the minimum and maximum values are representable by the underlying data type:

```
MIN MAX
MIN maxval
minval MAX
minval maxval
```

The scaling factors line is required for all scalar data types except enum and float; in these two cases scaling factors are not allowed. The scaling factors line consists of three integer constants, *a*, *b*, and *c*. When converting raw data to scaled data, the formula is *scaledValue* = $a * (10^b) * (rawValue + c)$). The default scaling, that is no scaling, is thus when $a = 1$, $b = 0$, and $c = 0$. Instead of specifying 1 0 0 for a scaling factors line, the more readable "NO SCALE" can be used instead.

The following Neuron C aggregate types are also supported:

array *numFields*	The array keyword is followed by the number of elements
struct *numFields*	The struct keyword is followed by the number of fields
union *numFields*	The union keyword is followed by the number of fields

When an array aggregate type is specified, it is followed by the three strings (name, comment, units) and then is followed by another type descriptor for the element of the array.

When a struct or union type is specified, it is followed by the three strings (name, comment, units) and then is followed by a number of type descriptors for the fields of the structure or union. If any of those type descriptors are arrays, structures, or unions, they will be first followed by their component types before resuming the list of the field types. This is what is meant by "depth-first" ordering, and is completely analogous to the way types are specified in the Neuron C language.

Finally, a type descriptor can be a reference to a network variable type. This is done with the **reference** keyword, followed by the name of the type being referenced. The referenced type must already exist in this type file. This descriptor is also followed by the three strings. When the type is read by the type utilities, the reference type can be replaced by the referent type. At that time, any of the strings from the reference, if they exist, will replace the strings in the referent type.

The strings, referenced types, and enumerations can be in other more generic type files rather than in the type file currently being defined. For example, if the type file being defined has a scope selector of 3, there may exist—in addition to the standard type and language files (scope selector 0) —type files and language files at scope selectors 1 and 2.

Besides the automatic behavior documented above, and as discussed earlier, the TYPFILW resource file compiler permits explicit references to be used in place of strings, types, and enums. These explicit references start with a caret character ("^"), and then are followed by an integer scope selector, a colon, and an integer index. So, for the UNVT_flow_micr type example shown above, recoded to use an explicit reference for the "microliters/second" string, might appear as follows:

```
15    UNVT_flow_micr    long  unsigned
$Flow volume
*
^1:14
MIN MAX
NO SCALE
```

Configuration Property Types Section

This section lists the configuration property types defined in the user definition file. The configuration property types section begins with the following line:

```
<CPTs>
```

This line may be followed by zero or more configuration property type definitions. A configuration property type is like a network variable type, with additional data fields. The configuration property type indices, like the network variable type indices, start with 1 and are consecutive and increasing. The user configuration property type names must start with the four characters "UCPT" and must not exceed 63 characters in total length. There is an additional field on the first line of each configuration property type definition, following the index and preceding the name. This field consists of either the characters "fxd" or "inh". A **fxd** configuration property type is a fixed type, regardless of the context in which it is used.

An **inh** configuration property type is a type that can be inherited from a network variable in the context in the device where it is used. This is the implementation of a configuration property type denoted as "SNVT_xxx". When a configuration property is declared in a device, it is specified as being a property of a network variable, or of a LONMARK object, or of the entire device. When an **inh** configuration property type is a property of a network variable, it inherits the type of the network variable. When an **inh** configuration property type is a property of a LONMARK object, it inherits the type of the principal network variable in the LONMARK object. The principal network variable in the LONMARK object is determined via the functional profile template declaration associated with the LONMARK object—although it is possible that there is no network variable designated as the principal one.

The configuration property type declaration proceeds like the network variable type declaration. At the end of that information, the configuration property type then has three additional lines of data, the first two of which are optional (omission is denoted by lines with a single asterisk), and the third of which is mandatory. The three lines are all byte array values (groups of two hexadecimal digits separated by whitespace). The length of each byte array is such that the number of groups of bytes is the same as the length of the configuration property type. The first of the three lines is the minimum-range override value, and the second is the maximum-range override value. These override values can replace the validation range of referenced types and make the configuration property type either more or less restrictive than the underlying network variable type. The third byte array line gives the default, or initial, value for the configuration property type.

The following example shows the SCPT number 15, for "Input value feedback delay":

```
15 fxd      SCPT_in_fb_dly    reference   SNVT_elapsed_tm

$Input value feedback delay
```

```
$The time period between feedback output updates
*
*
00 00 00 00 3B 03 E7
00 00 00 00 00 00 00
```

Functional Profile Template Section

The next section lists the functional profiles defined in the user definition file. The functional profiles section begins with the following line:

```
<FPTs>
```

This line may be followed by zero or more functional profile definitions. Each functional profile definition begins with a line that contains the functional profile index, the functional profile numeric key, the functional profile name, and five integers which give the counts for number of mandatory network variables, number of optional network variables, number of mandatory configuration properties, number of optional configuration properties, and the index (starting from 1) of the principal network variable in the functional profile. If there is no principal network variable, the principal network variable field should contain a 0. Following this line are two string lines, for the language-dependent name of the functional profile, and a language-dependent comment. The definition for standard functional profile 10, the CO_2 Sensor, is given below as an example:

```
10    1070  SFPTCO2Sensor     2 0 3 1 1

$CO2 sensor object
$Carbon dioxide (CO2) sensor
```

Immediately following the base definition of the functional profile are network variable records, each defining a network variable member of the functional profile. There is a network variable record for each mandatory and optional network variable as specified in the first definition line. Then, following the network variable records are configuration property records, each defining a configuration property member of the functional profile. There is a configuration property record for each mandatory and optional configuration property, again as specified in the first definition line for the functional profile.

Each network variable record consists of five lines. The first line begins with an **input** or **output** keyword followed by a programmatic name for the network variable member. Following the name is a **man** or **opt** keyword indicating a mandatory or optional network variable, and then the name of a network variable type. An optional keyword can follow the name of the network variable type, and this is used to specify the protocol service type for output network variables. This can be **unack**, **unackr**, **reqrsp**, or **polled** (**ackd** is the default when no keyword is supplied).

The second and third lines are the language-dependent name of the network variable and a language-dependent comment, respectively. The fourth and fifth lines are optional byte arrays used as minimum and maximum values for the network variable, used to override the base network variable type if desired. If there is no override, an asterisk ("*") is used. As an example, the following is the first network variable from the CO_2 sensor object whose functional profile base definition appears above:

```
output      nvoCO2ppm   man   SNVT_ppm

$CO2 level
$The CO2 level in parts per million
*
*
```

Each configuration property record consists of six lines. The first line begins with the programmatic name for the configuration property member. Following the name is a **man** or **opt** keyword (indicating a mandatory or optional configuration property), and then the name of the configuration property type, and finally an index indicating to which network variable member the property applies (if the property applies to the whole object, 0 is used).

Optional keywords can follow the configuration property index, and are used to specify the control for when a configuration property may be modified. These flags are the following:

Flag	Definition
const	Designates a constant configuration property. These configuration properties are never changed by a network tool. However, network tools may write such configuration properties when residing in value file index 1 as long as the value is not changed. Configuration properties with the `const` flag but without the `devspec` flag can be assumed to have the same value on all devices using the same program ID.
devspec	Designates a configuration property that is constant, but may vary by device. Network tools must therefore always read this configuration property from the device instead of relying upon the value in an external interface file or a value stored in a network database. Network tools must never change this configuration property except as a side effect of a new program download. The `const` flag must also be specified whenever the `devspec` flag is specified.
mfg	Designates factory settings that need to be read or written when the device is manufactured, but are not normally (or ever) modified in the field. In this way, a standard network tool may be used during manufacture to calibrate a device, while a field network tool would observe the flag and prevent or require a password to modify the value.
offline	Designates a configuration property that can only be modified while the device is offline. This flag is only applicable to direct-memory read/write configuration parameters or network variable configuration properties. This flag is ignored for file-transfer configuration parameters. This is because file transfer cannot function while an application is in the offline state. In fact, an offline application must be placed into the online state for the duration of any file-transfer configuration parameter operations.

reset	Designates that the device should be reset after updating this configuration property.
obj_disable	Designates that the associated LONMARK object must be disabled while updating this configuration property.

The second and third lines are the language-dependent name of the configuration property and a language-dependent comment, respectively. The fourth, fifth, and sixth lines are byte arrays. The fourth line specifies an optional ("*" if omitted) minimum-value range restriction that overrides the minimum value from the configuration property definition. The fifth line specifies an optional ("*" if omitted) maximum-value range restriction that overrides the maximum value from the configuration property definition. The sixth line specifies an optional ("*" if omitted) default value that overrides the default value from the configuration property definition.

```
nciMaxSendTime    man   SCPTmaxSendTime   0

$Maximum send time
$The maximum period of time between output value transmission
*
*
01 2C
```

The end of the functional profile section is also the end of the user definition file. The last line of the file must be the following:

```
<End>
```

Format File Format

A format file defines display and input formats for network variable and configuration property types defined in a type file. The format file references the type names in a type file so that format definitions can be associated with the appropriate data types, structures, and enumeration names.

This section describes the format for version 3 format files. Version 1 was the original specification for format files. Version 2 added unit conversion, alternate formats, and default formats. Alternate and default formats were used to support different formats for US and SI unit types, but required two versions of every format file to support different US and SI defaults. Version 3 adds improved localization support that works with Windows localization support, and therefore no longer requires default formats. The new support includes locale-specific list separator, date, and time formats. Some legacy network tools may not support version 3 format files. The FMTCONV Format File Conversion utility, available from www.lonmark.org, is used to convert version 3 format files to version 2 format for use with legacy tools.

For integer, floating point, and enumeration types, the formatting is almost always a simple conversion to a text string, although you may want to decide otherwise. For structured types (for example, time-of-day expressed in hours, minutes, and seconds), you can specify how the data is to be formatted. Each data type present in a format file must have a corresponding type defined in a standard or user type file.

A format file is a text file, and you may use any text editor to create and edit a format file. You should never modify the standard format file since it is periodically updated by Echelon.

Two lines are placed at the beginning of a format file to identify the file content and to identify to which devices the file applies. The first line sets the program ID for the file, and the second line defines the scope selector. The syntax for the two lines is the following:

```
set program_id nn:nn:nn:nn:nn:nn:nn:nn;
set selector selectorName;
```

The keywords in the two lines are case insensitive. The semicolon terminating characters are required.

The program ID specification line sets the file's program ID value. The program ID always consists of eight hexadecimal bytes with the same format as the program ID of a LONMARK Association-compliant application, even though some of the values may not be used, depending on the chosen scope. Each *nn* in the program ID specification line is one hexadecimal byte (the digits '0' through '9' and either the characters 'a'-'f' or 'A'-'F' may be used).

The scope selector specification line sets the format file's scope as described under *Managing Resource Files* earlier in this guide. The choices for *selectorName* are shown below, and they correspond to the scope selector numeric values (used in the definition file) of 1-6, respectively:

```
set selector DEVICE_CLASS;
set selector DEVICE_CLASS and SUB_CLASS;
set selector MANUFACTURER;
set selector MANUFACTURER and DEVICE_CLASS;
set selector MANUFACTURER and DEVICE_CLASS and SUB_CLASS;
set selector MANUFACTURER and DEVICE_CLASS and SUB_CLASS and MODEL;
```

For example, the following two lines specify a match in the manufacturer field for manufacturer code 0002A:

```
set program_id 80:00:2A:00:00:00:00:00;
set selector MANUFACTURER;
```

The body of the format file consists of a series of records, one or more for each type whose format is being defined. A format record can span several lines and contain spaces. Only spaces within the quotes of a text format type are significant; all other spaces are ignored.

Following is the syntax for each record:

```
typeName[#alternateFormat]: formatSpecifier
```

The *typeName* field specifies the type name as defined in the type file. Type names are case-sensitive.

The *alternateFormat* field allows you to specify an alternate format for this type. Each type may have multiple formats defined. Each format must have it's own definition. See *Alternate Formats*, below, for more information about alternate formats.

The *formatSpecifier* field can be any of the following:

real A single-precision, 32-bit, IEEE floating-point number.

int A signed, 32-bit integer number

discrete An 8-bit value that contains 0 or 1.

text(…) A text string.

The Text Format Specifier

The text format specifier should be used for data that is not a simple number (enumerations, strings, characters, and structures), or where data formatted as text is preferred. The STANDARD.FMT format file consists entirely of text format specifications, since most network tools are adept at handling text-formatted data, and text-formatted data may be specified for every data type.

The Grammar of the Text Format Specifier

Within the parentheses of the text format specifier, there are several constructs that may appear — for examples, see the following sections and the STANDARD.FMT file. The grammar is as follows:

<text format group> = '(' <text format list> ')'
= <text format>

<text format list> = <text format list> ',' <text format>
= <text format>

<text format> = '(' <condition> '?' <text format group> ':' <text format group> ')'
= '(' <text format string> ',' <field spec list> ')'
= 'time' '(' < field spec string > ',' < field spec string > [',' < field spec string >]
[',' < field spec string >] ')'
= 'date' '(' <field spec string> ',' <field spec string> ',' <field spec string> ')'

<condition> = '(' <field spec string> <conditional operator> <decimal const> ')'

<conditional operator> = '=='
= '!='

<field spec list> = <field spec list> ',' <field spec with modifiers>
= <field spec with modifiers>

<field spec with modifiers> = <field spec with multiplier and adder> <string resource reference>
= <field spec with multiplier and adder>

<field spec with multiplier and adder> = <field spec string> <multiplier> <adder>
= <multiplier><adder>
= <field spec string>

<field spec string> = <field spec string> '.' <field name>
= <field name>

<string resource reference> = '(' <mode> ':' <index> ')'

The Components of the Text Format String

Specifying a text format string within a text format is similar to specifying a formatting string to the C programming language printf() function, with some simplifications and extensions. A format specification string is a quoted string within a text format specifier. If the format string contains a type code (a percent sign followed by a letter), the corresponding field data argument is formatted according to the type-code specification. The following type codes may be used:

%c A single character. The base type in Neuron C must be char, int, or enum.

%d A signed or unsigned decimal number (based on the signedness defined in the type file). The base type must be a Neuron C char, int, or long or a structure or array. If it is a structure or an array of at least four bytes in length, it is assumed to be a Neuron C signed 32-bit number of s32_type.

%f A floating point number. The base type must be a structure, an array, or a fixed point Neuron C int or long. If it is a structure or array of at least four bytes in length, it is assumed to be a Neuron C floating-point number of float_type or SNVT_xxx_f type.

%m An enumeration. The base type must be an enumerated list. If an enumeration does not exist for the value, the format string is processed as if it were %d.

%s A null-terminated string. The base type must be an array of 8-bit data. String data must be null terminated.

%x An unsigned hexadecimal integer. The size is determined from the type file. The data are always treated as unsigned. The base type must be char, int, or long. If it is a structure, or an array of at least four bytes in length, it is assumed to be a Neuron C signed 32-bit number of s32_type.

The text format string may also include a vertical bar "|" character to specify a locale-specific list-separator character. This character is translated to the operating system list-separator character for the current operating system default locale. The current setting of the Windows list-separator character may be found in the List Separator setting on the Number tab of the Regional Options in the Windows Control Panel. The list-separator character can only be used with LO class alternate formats, as described under *Alternate Formats*.

A backslash ("\") is used as an escape character to include other format characters as text. For example, the following characters can be included in a format string:

\% The % character.

\\ The \ character.

\" The " character.

\| The | character.

A text format string may also contain conditional formats, unit conversion factors, and localized time and date format specifiers as desribed in the following sections.

Following is an example of a text format definition including string, floating point, and list-separator format specifiers from the SCPTrefrigType#LO format definition:

```
SCPTrefrigType#LO:text("%s %f|%f|%f", refrigerant, A, B, C);
```

Conditional Formats

Text formats that contain the ternary "?" operator allow you to display different formats depending on the value of a particular field. The ternary operator is used much the same way that it is used in the C programming language (i.e. `<condition> ? <format if condition is true> : <format if condition is false>`), however, only the "equal to" ('==') and "is not equal to" ('!=') comparison operators are supported in the condition. In order to ensure that the format specified may be formatted and unformatted for all data values, the field that appears in the conditional statement should appear in a regular format specification string *before* it appears in the conditional statement. Formats are processed in left-to-right order.

Following is an example of a format definition with conditional format specifiers from the SNVT_earth_pos#SI format definition:

```
SNVT_earth_pos#SI: text( ("%d %d ", latitude_direction,
                  longitude_direction),
                  ( (latitude_direction == 0) ? ("S") : ("N") ),
                  (" %d %d ", latitude_deg, latitude_min),
                  ( (longitude_direction == 0) ? ("E") : ("W") ),
                  (" %d %d %f", longitude_deg, longitude_min,
                  height_above_sea) ); ! meters above sea level
```

Unit Conversion Factors

The text format may specify unit conversion factors (a multiplier and an adder) for simple data types, and also for fields of a structured type. The unit conversion factors are applied as a multiplication and an addition when data is converted for output, and they are applied in the reverse order, as a subtraction and a division when data is input. If a localized string reference is specified, it will override the unit description string found in the type file.

Alternate formats with unit conversion factors can be used for converting units to the United States (US) measurement system.

For examples of unit conversion of a non-structured type, the following two lines define the Système Internationale (SI) and US formats for the SNVT_temp_f standard network variable type:

```
SNVT_temp_f#SI:   text("%f", *1+0(0:854));      ! degrees C
SNVT_temp_f#US:   text("%f", *1.8+32(0:855));   ! degrees F
```

For examples of unit conversion of individual fields of a structured type, the following lines define the SI and US formats for the SCPTsetPnts standard configuration property type:

```
SCPTsetPnts#SI:   text("%f,%f,%f,%f,%f,%f",
                  occupied_cool, standby_cool, unoccupied_cool,
                  occupied_heat, standby_heat, unoccupied_heat);
SCPTsetPnts#US:   text("%f,%f,%f,%f,%f,%f",     ! degrees F
                  occupied_cool*1.8+32(0:855),
                  standby_cool*1.8+32(0:855),
                  unoccupied_cool*1.8+32(0:855),
                  occupied_heat*1.8+32(0:855),
                  standby_heat*1.8+32(0:855),
                  unoccupied_heat*1.8+32(0:855));
```

Localized Time and Date Formats

Within a text format specification, you may use the time and date format specifiers to format a time or date value as specified by the operating system default locale method. The date format specifier requires three parameters, which specify the data fields where it will find the year, month, and day values to be formatted. The time format specifier requires two to four parameters, specifying hour and minute values to be formatted, and optionally, second and millisecond values.

For the Windows operating system, the current setting of the date format may be found under Short Date Style on the Date tab of Regional Settings in the Windows Control Panel. The current setting of the time format may be found under Time Style on the Time tab of the Regional Settings, with the following exceptions:

1) The time format specifier does not support AM/PM time formats, so this type of time format will be converted to a 24-hour format.

2) The time format specifier supports display of milliseconds, which is not defined in Windows time styles. If supplied, the milliseconds field will be appended to the seconds field, and separated from the seconds field by the Decimal Symbol character from the Number tab of the Regional Settings.

The time and date format specifiers may only be used in LO class alternate formats, as described under *Alternate Formats*.

Following is an example of a format definition with time format specifier from the SCPTmaxSntT#LO format definition:

```
SCPTmaxSndT#LO:   text(("%d ", day),
                  time(hour, minute, second, millisecond));
```

Following is an example of a format definition with date format specifier from the SNVT_date_cal#LO format definition:

```
SNVT_date_cal#LO: text(date(year, month, day));
```

Alternate Formats

Alternate formats are used to specify multiple formats for a single type. Alternate formats are grouped into *alternate format classes*, where an alternate format class defines a set of related alternate formats. Alternate format classes are identified by two-character, upper-case, substrings within an alternate format specifier. Three standard alternate format classes are defined to support locale-specific formatting. The standard alternate formats are the following:

1) SI = Système Internationale measurement system format (also known as *metric*)

2) US = United States measurement system format

3) LO = operating system (e.g. Microsoft Windows) locale-specific format

The SI and US classes have a special relationship: Any type that has at least one format defined with one of these classes must also have at least one format defined with the other one of these classes. Any format definition that specifies one of these classes cannot specify the other one—they are complimentary. Membership in these classes, together with the Windows system default locale, determines the runtime default format as described under *Default Format*.

Any format definition that contains the list-separator, date, or time locale-specific specifier must be a member of the LO alternate format class. This aids in ensuring backward compatibility and also make it obvious to the user that locale-specific data transformations will occur when using this format.

If an alternate format belongs to one or more alternate format classes, the alternate format class names must appear immediately following the pound "#" character. If one of the classes is the US or SI class, the US or SI class name must be listed first, immediately following the pound "#" character. If a format belongs to multiple alternate format classes, the alternate format class names must be separated by an underscore character ("_"), and an underscore character must also separate any alternate format class names from the rest of the alternate format specifier. The rest of the alternate format specifier extension should use lower- or mixed-case letters, and to avoid confusion, should not use two upper-case letters in a row.

Following is an example of a format definition that specifies both an SI and LO alternate format class, from the SCPTsetPnts#SI_LO format definition:

```
SCPTsetPnts#SI_LO:text("%f|%f|%f|%f|%f|%f",
                  occupied_cool, standby_cool, unoccupied_cool,
occupied_heat, standby_heat, unoccupied_heat);
```

Following are examples of two format definitions that both specify US alternate format classes, but one includes a "diff" format name extension to differentiate it from the other. These are from the SNVT_temp#US and SNVT_temp#US_diff format definitions:

```
SNVT_temp#US:     text("%f", *1.8+32(0:855)); ! degrees F
SNVT_temp#US_diff:text("%f", *1.8+0(0:855));
                  ! degrees F (differential, no offset)
```

Default Format

If multiple formats exist for a given type, one of the formats is defined as the default. The default format is the format that will be assigned if the user specifies only the type name, with no alternate format. The default is selected from all formats with a matching type name based on the following criteria, in priority order:

1. A format definition preceded by an asterisk ("*"). A maximum of one default format per type may be specified.

2. A format definition that is not defined as an alternate format.

3. If the measurement system defined by the operating system locale is metric, an SI class alternate format will be selected as the default. For example, the Windows measurement system is defined on the Number tab of the Regional Settings on the Control Panel. If there are multiple SI class alternate formats, the default will be selected as follows:

   i) An SI class format definition preceded by a plus ("+") character. A maximum of one default SI class format per type may be specified.

   ii) If an SI class alternate format default is not specified, the first SI class alternate format in alphabetical order.

4. If the measurement system defined by the operating system locale is U.S., a US class alternate format will be selected as the default. If there are multiple US class alternate formats, the default will be selected as follows:

   i) A US class alternate format name preceded by a plus ("+") character. A maximum of one default US class alternate format per type may be specified.

   ii) If an SI alternate format default is not specified, the first US class alternate format in alphabetical order.

5. The first alternate format in alphabetical order.

Revision 3, October 2000

Echelon, LON, LONWORKS, LONMARK, LonPoint, LONTALK, Neuron, 3120, 3150, and the Echelon logo are registered trademarks of Echelon Corporation. LonMaker and LonSupport are trademarks of Echelon Corporation.