DATA CLASSIFICATION

Product Preview

This heading on a data sheet indicates that the device is in the formative stages or under development at the time of printing of this data book. Please check with Motorola for current status. The disclaimer at the bottom of the first page reads: “This document contains information on a product under development. Motorola reserves the right to change or discontinue this product without notice.”

Advance Information

This heading on a data sheet indicates that the device is in sampling, preproduction, or first production stages at the time of printing of this data book. Please check with Motorola for updated information. The disclaimer at the bottom of the first page reads: “This document contains information on a new product. Specifications and information herein are subject to change without notice.”

Fully Released

A fully released data sheet contains neither a classification heading nor a disclaimer at the bottom of the first page. This document contains information on a product in full production. Guaranteed limits will not be changed without written notice to your local Motorola Semiconductor Sales Office.

Technical Summary

The Technical Summary is an abridged version of the complete device data sheet that contains the key technical information required to determine the correct device for a specific application. Complete device data sheets for these more complex devices are available from your Motorola Semiconductor Sales Office or authorized distributor.

MOTOROLA

LONWORKS

Technology Device Data, Rev. 5
Volume II — Engineering Bulletins

Through the LONWORKS program, Motorola offers the MC143120 and MC143150 Neuron integrated circuits. These integrated circuits are sophisticated VLSI devices for network applications. The *LONWORKS Technology Device Data Book* combines specifications on these parts with a large selection of applications literature. The applications literature is included in this book in order to provide you with suggestions and hints for implementing network solutions in your own applications.

This book is divided into three volumes:

- Volume I — Device Data
- Volume II — Engineering Bulletins from Echelon
- Volume III — Applications Literature

For the most current copy of this data book, please visit our web site at:
http://motorola.com/lonworks

Increasingly sophisticated LONWORKS devices are constantly under development. For the latest releases, additional technical information, and pricing, please contact your nearest Motorola Semiconductor Sales Office or authorized distributor. A complete listing of sales offices and distributors is included at the back of this book.

Motorola LONWORKS Application Support
Austin, Texas: (512) 934-7610 FAX: (512) 934-7991
http://motorola.com/lonworks
Toulouse, France: 33-561-199175
Hong Kong: 852-2-666-8470
Tokyo, Japan: 81-33-280-8414

CONTENTS

CONTENTS (CONTINUED)

CONTENTS (CONTINUED)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (CONTINUED)

LIST OF FIGURES (CONTINUED)

NOTE: For the latest versions of the EB documents above, download them from Echelon's LonLink.

Engineering Bulletins EB

NEURON® CHIP Quadrature Input Function Interface

August 1991

LONWORKS™ Engineering Bulletin

Introduction

The NEURON CHIP quadrature input function provides a simple means to process external data encoded in quadrature format.

Quadrature encoding is used in position sensing applications where only two external characteristics are needed to accurately determine the position of an object relative to its last position: magnitude and direction of change. The quadrature encoding allows both of the above attributes to be conveyed using only two signals, thereby simplifying the circuitry and eventual decoding that will be needed.

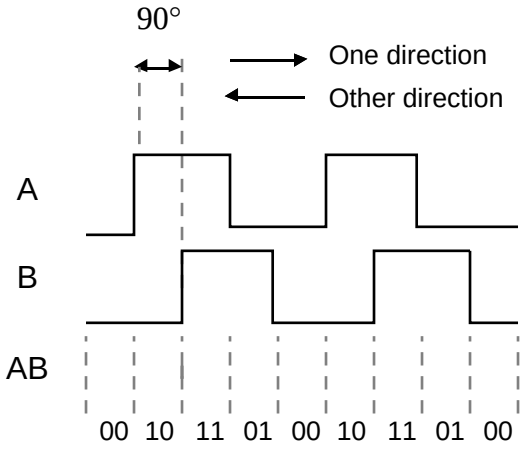


Figure 1. Ideal quadrature relationship.

A quadrature relationship between two signals, as the name implies, assumes a 90 degree phase difference between them, as shown in figure 1. Note that the resulting composite code (AB) follows the reflected (Gray) code sequence which has the property that only one bit is changed between consecutive values.

Position sensing can be divided into two broad classes: rotational and linear. Encoders for both classes exist in the market with a wide range of resolution and reliability. The resolution of a quadrature encoder refers to the number of different composite output code transitions it can generate for a given amount of movement (e.g., 16 counts per revolution for a rotational encoder).

Echelon Part Number 005-0003-01

Both mechanical and electronic (optical) encoders exist in the market allowing for flexibility and range in cost, reliability, accuracy, and size.

Figure 2 shows the contact pattern for a mechanical linear quadrature encoder.

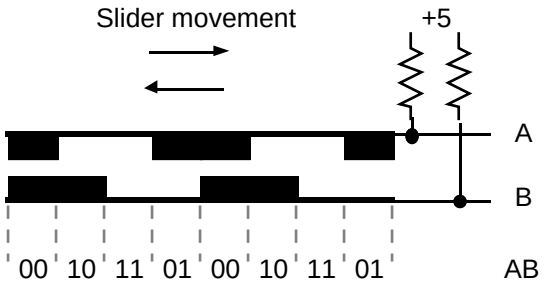


Figure 2. An example of a linear encoder's mechanical contact pattern. The common pin is not shown and is the slider which rides on the "rail" .

A rotational encoder can be realized by simply wrapping the pattern shown in figure 2 so that the two ends are connected and there is therefore no restriction imposed on the movement of the slider. The slider is then hinged on the central axis of the newly formed circle so that a more convenient rotating shaft could be used.

A rotational quadrature encoder, also referred to as a digital pot or a shaft encoder, is often used to replace variable resistor potentiometers or any other rotational dial input device in microprocessor-based systems. This provides an easy interface to the digital domain.

Linear quadrature decoders also simplify such interfaces by replacing linear slide variable resistor potentiometers or any other linear (straight-line) input device.

It is important to note that although these devices are called quadrature encoders, they do not necessarily produce perfect quadrature (90°) outputs at all times.

While at a constant non-zero speed the output of these encoders might look like the one shown in figure 1, it is generally accepted that the encoder is producing a quadrature output as long as the correct relative sequence of level transitions is obeyed and the interface circuitry's timing requirements are met. In other words, the duty cycle of the output waveforms need not be 50%.

An acceptable encoder output might look like the one shown in figure 3. Note that the same sequence of composite output codes is observed in both directions, as in figure 1.

The two characteristics mentioned earlier, namely magnitude and direction of movement, can be extracted from the quadrature output of an encoder in several ways.

The magnitude (i.e. angular or linear displacement) can be observed by simply counting the number of transitions on either output (A or B). The rate of change of magnitude (angular or linear velocity), could then be extracted by performing the same counting operation within a given period of time.

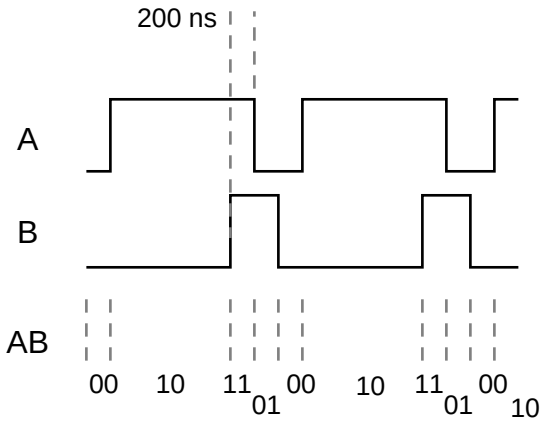


Figure 3. Another "quadrature" relationship.

The determination of direction from the quadrature signal is a bit more subtle. Two popular techniques are: comparing current composite output code to the previous one, and observing the level of one output (e.g., A) at the time when the other output (e.g., B) makes a transition.

Regardless of the technique used in interfacing a quadrature encoder to a system, some amount of circuitry and/or software is usually required on the designer's part. Using the NEURON CHIP quadrature input function block such a need is eliminated.

The quadrature encoder, as shown in figure 4, is connected directly to the NEURON CHIP, allowing the designer to concentrate on the intended application rather than on interface issues.

Usage

The quadrature input function of the NEURON CHIP can be invoked by using the following statement in a NEURON C application program:

```
pin  input quadrature
io_object_name;
```

where:

`pin` is a NEURON CHIP I/O pin. Either IO_4 or IO_6 may be used. The second pin needed for the two quadrature inputs is assumed to be the next higher pin number (IO_5 or IO_7) on the NEURON CHIP.

`io_object_name` is a user-specified name for the quadrature device.

Refer to the NEURON C Programmer's Guide for a more detailed description of the syntax.

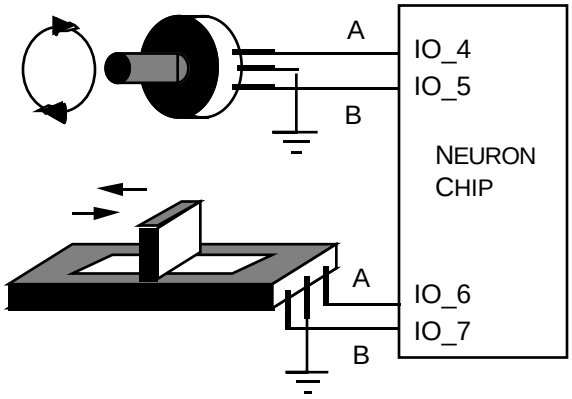


Figure 4. A typical encoder connection to the NEURON CHIP.

The following is an example for the above syntax:

```
#pragma enable_io_pullups
IO_4 input quadrature dial;
long dial_data;

when (io_update_occurs(dial))  {
     dial_data = input value;
...  //dial_data represents the change in offset of
     //the shaft encoder since last input.
}
```

The `pragma` compiler declaration enables the NEURON CHIP's on-chip pull-up resistors. This reduces the task of using a quadrature encoder to merely connecting the two encoder outputs to the two NEURON CHIP inputs (declared in your program) -- no additional components are required

Quadrature encoders normally provide a third pin designated as the common. In order to use the NEURON CHIP's built-in pull-up resistors, this pin must be connected to ground. Connecting the common pin of the encoder to +5 volts and

using external pull-down resistors would also work and would cause the relative direction of the encoder to be reversed (A and B signals are inverted).

Another way of reversing the relative direction of the encoder is to invert the sign (take the two's complement) of the number returned by the `input_value` variable. This is generally the preferred way since it only requires a simple software modification and no hardware modification.

The NEURON CHIP makes use of its internal timer/counter resources, along with its built-in firmware, to decode quadrature signals. The time periods between transitions are measured by a counter which is controlled by additional logic responsible for determining the direction of count. The user can access the results of this measurement operation by using the `io_update_occurs()` function of the NEURON C.

The `io_update_occurs()` event evaluates to TRUE when the quadrature object specified (dial) has an updated value. At that point `input_value` contains the amount of change observed at the quadrature inputs (encoder position changes). This value is a signed long (16 bit) positive (negative) number representing the number of increments (decrements) at the quadrature input since the last evaluation of `io_update_occurs()`.

The number of times a quadrature input is evaluated is therefore not only dependent on the absolute speed at which the NEURON CHIP is running, but also on the size and structure of the application code. The task scheduler in the NEURON CHIP is responsible for executing all the tasks in the program including the `when (io_update_occurs())` statement mentioned above. Refer to the *NEURON C Programmer's Guide* for more information on the operation of the scheduler.

The `input_value` is in two's complement format. Therefore, the most significant bit (sign bit) represents the direction of the movement of the quadrature encoder.

The count returned through `input_value` is cumulative. That is, between two consecutive `io_update_occurs()` events, a finite movement in one direction followed by an equal amount of movement in the opposite direction would yield a total count of zero.

The NEURON CHIP firmware limits the maximum value of `input_value` to +16383 and -16384 in either direction. If the number of code changes observed at the quadrature inputs between consecutive `io_update_occurs()` events exceeds ±16K then the `input_value` for the second event will be the maximum in that direction (+16K or -16K).

The `io_changes()` event is not a good alternative for use with the quadrature input since at a constant encoder speed the same value is observed by the event and therefore no apparent "change" is observed.

With the NEURON CHIP running at 10 MHz, the input to the quadrature pins is sampled every 200 ns. Therefore, any external encoder output values at the quadrature inputs which occur faster than every 200 ns will not be recognized (Figure 3). As a consequence, the maximum frequency on either A or B inputs of the NEURON CHIP running at 10 MHz must not exceed 1.25 MHz. This should be more than enough for typical user-interface applications. The sampling rate scales at lower clock speeds.

The following NEURON C program illustrates a typical application of the quadrature input function. The variable count contains a cumulative updated count of encoder movements.

```
#pragma enable_io_pullups

//////////////////User defined parameters///////////
#define lower_limit 0         //lower count limit
#define upper_limit 999       //upper count limit
#define shaft_direction 1     //direction of count (e.g.,
0=CW,1=CCW)

////////////////// Declarations /////////////////////
IO_4 input quadrature shaft_encoder;  //shaft encoder input
signed long count;
signed long increment;

when (io_update_occurs(shaft_encoder){

   if (shaft_direction) count += input_value;
     else count -= input_value;     //inc/dec based on user
                                    //definition
   if (count<lower_limit) count=lower_limit;
   if (count>upper_limit) count=upper_limit;  //take care of
                                              //overflow
               //and underflow due to fast shaft rotation
}
when (reset)  {
       count = 0;
}
```

Encoder Sources

The following is a partial list of some of the manufacturers of quadrature encoders:

Bourns Inc. , Resistive components Group
(714) 781-5050
Mechanical encoders. Shaft angle.

Clarostat Manufacturing Co.
(800)-872-0042
Mechanical and optical encoders. Shaft angle.

Litton Encoders
(818)341-6161
Mechanical, optical, and magnetic encoders. shaft angle.

Photoswitch Div., Allen-Bradley Co.
(617)466-8000
linear and shaft encoders. mechanical and optical encoders.

U.S Digital Corp.
(213)594-0094
Optical shaft and linear encoders.

LONWORKS® Installation Overview

January 1995 **LONWORKS Engineering Bulletin**

Introduction

LONWORKS technology offers a powerful means for implementing a wide variety of distributed systems that perform sensing, monitoring, and control. During installation, individual intelligent nodes are assembled into an interoperating network. This installation process, along with the on-going maintenance of the system, is referred to as *network management*[1]. As a solution designed from the ground up for networked control applications, network management has been a key consideration in the design of each LONWORKS tool and component. This engineering bulletin explains the tasks required to install a LONWORKS network and provides a brief summary of the various ways to accomplish these tasks.

The LONWORKS Installation Philosophy

Unlike approaches which take off-the-shelf control components and try to paste on networking, LONWORKS technology was designed to solve networked control problems. The implication of this is that networking, and all of the requirements it implies, are not an added-on afterthought; they are an integral part of the LonTalk® protocol, the Neuron® Chip, and all of the development and network services tools. This brings many benefits to both developers and their customers.

To the developer, this built-in support means a simplified development process. Since network support is included in every aspect of the technology, creating networked systems becomes easier. Rather than having to reinvent the wheel, the developer can draw on the native support of the technology. This integrated support for networked systems also means that it is exactly the same in every LONWORKS node, unlike a paper specification subject to multiple interpretations and implementations. For manufacturers, this means that the projects they build at different sites or at different times can be made to work together seamlessly. To end-users, it means that products from multiple suppliers can be integrated into new systems that leverage the strengths of the individual pieces.

In a LONWORKS system, hardware design, software design, and network design are all independent tasks. This means that a node's function can be specified and

[1] Network management is not the same as network monitoring or network control. Network management is the process of installing, configuring, and maintaining the nodes in a network. A network management tool does not participate in the exchange of application and network variable messages, so it does not need to be present for the network to operate. For systems using a network controller or a network monitor, these devices are usually active participants in the on-going operation of the system and must be present for the system to operate. A single device may combine network installation, maintenance, monitoring, and control, or separate devices may be used.

programmed without concern about the specifics of the network or networks in which it will be used. This also has several major benefits.

First, it reduces development effort and cost. Nodes can become generic building blocks that can be used in multiple applications to accomplish different tasks. For example, a generic motion sensor node could be used to monitor parts on an assembly line or as a room occupancy sensor, without any change to the application code or node hardware. The tasks the node performs in any given situation are determined by the way in which it has been connected to other nodes in the network.

Second, the separation of the various design tasks enables systems of unprecedented flexibility and interoperability. Because network parameters are independent of node application code and physical network attachment, new nodes can be added and connections between nodes can be changed dynamically to logically 'rewire' a network and redefine network behavior without the cost and delay associated with running new wires. Additionally, once installed, the nodes in one LONWORKS network can become components in any other LONWORKS network to provide increased functionality without increased cost. For example, the motion detector nodes used in a lighting system may be connected to an alarm system programmed to sound an alarm (or transmit a message to a law enforcement agency) whenever motion is detected in certain rooms during certain hours of the day.

Finally, LONWORKS technology provides a very flexible environment, with many ways to install nodes and to tune network parameters based on application needs. This flexibility makes a LONWORKS network suitable as a replacement for a wiring harness or for a master/slave control system; it also allows you to build peer-to-peer control systems. However, it is not necessary to use all the power of LONWORKS technology to enjoy many of its benefits. The task of designing LONWORKS networks becomes one of picking and choosing the options that are of value in a given application and presenting them in a way that is convenient to the end user or installer.

Requirements for Control Network Installation and Maintenance

Network Installation and Configuration

To understand what is required to install nodes on a control network, it helps to look at the types of control systems that control networks replace.

Most conventional control systems use wiring harnesses, point-to-point wires, or a central controller wired to dumb sensors and actuators. When these devices are installed, the wiring between the devices serves two purposes. It physically interconnects the devices and it determines which control signals go to which device. Once attached to the wire, the behavior and interaction among the devices is fully defined. It is also fixed, unless the wiring is modified.

With control networks, intelligent devices, called *nodes*, are connected to their physical media (for example, twisted-pair wire or a power line circuit) as they would be in conventional control systems. Unlike a conventional control system where each node has a dedicated, hard-wired connection to the nodes it communicates with, the nodes in a control network share the same communications media. Figure 1 shows the wiring difference between a conventional control system and a LONWORKS network.

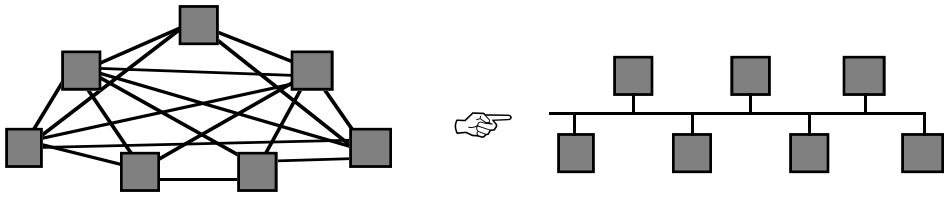


Figure 1. Wire Reduction in a LONWORKS Network

Of course, since the nodes share the same communications media, physically attaching nodes to the network is not enough for the nodes to communicate. The physical attachment only provides a path for nodes to send and receive messages; it does not tell the nodes with whom they should share data. Nodes in a control network need to be given network configuration information, such as network addresses, to enable them to understand with whom they should share data. Specifying this addition configuration information is required when installing any control network.

With a LONWORKS network, this configuration information is defined and loaded by a network installation tool. Each nodes' configuration information can be defined and loaded at manufacture time, in which case the LonBuilder Developer's Workbench is used as the installation tool. Or, each nodes' configuration information can be defined after the nodes are physically attached in the field, by an installation tool that sends LonTalk network management messages to each node over the network. In either case, network messages can also be used to adjust and redefine the control system's behavior at any time, requiring no changes to the physical wiring or to the nodes' software or hardware.

Network Maintenance and Repair

Where network installation and configuration are the one-time tasks of setting up a control network, network maintenance and repair are the on-going tasks of changing the network over time. Network maintenance includes tasks such as adding or removing nodes or connections to change the system's behavior. One benefit of LONWORKS networks is that since a node's network configuration information is separate from its application, new nodes can be added and connections between nodes can be changed dynamically to logically "rewire" the network and redefine the network's behavior without the cost and delay associated with running new wires. Any node's behavior can be modified, over the network, at any time; there is no need to change application code or move physical wires.

Network maintenance also includes loading new application software into nodes. A benefit of LONWORKS networks is that if a node's application is stored in writeable memory, a new application can be loaded into the node over the network to fix software problems or to add new features.

Network repair is the process of detecting and replacing failed nodes. Detecting a failed node means being able to detect nodes that have failed at an application level (for example, an actuator that no longer opens or closes due to a faulty motor) and nodes that have failed at a network level (for example, a node that has become detached from the network). As is discussed below , since LONWORKS nodes communicate with one another using logical rather than physical addresses, failed nodes can be quickly and easily replaced without effecting any other nodes in the systems.

The ABC's of Installation (Addressing, Binding, Configuring)

Customizing a generic node to give it a unique network personality involves specifying and loading certain pieces of configuration information. The tasks for managing this configuration information are address assignment, binding, and configuration. How and when the configuration information is provided depends upon your installation needs and will be discussed in further detail in a later section. In each scenario, however, Echelon provides products that offer a set of interoperable network installation, maintenance, configuration, monitoring, and control services to installation tool developers and network installers to easily accomplish the required tasks.

A —Address Assignment

Nodes communicate with each other by sending messages. Much like a postal address identifies a house to the mail carrier, a node's network address identifies it to LonTalk messages. A node's network address consists of three components — the domain to which the node belongs, the subnet to which the node belongs within the domain, and the node's ID within the subnet. A node can have up to two network addresses, one for each domain of which it is a member[2]. The logical address uniquely identifies the node in the network. The responsible for assigning each node a unique network address belongs to the installation tool.

There are many advantages to using logical network addresses, rather than physical serial numbers, to communicate with a node. First, logical addressing provides

---

[2] In addition to its two network addresses, a node can also be addressed by its Neuron ID or by using group addresses. A node can be a member of up to 15 different groups. Group addresses are assigned during the binding process by an installation tool.

This gives a total of five different ways to address a node — by Neuron ID (independent of domain), by domain wide broadcast, by subnet wide broadcast within a domain, by domain / subnet / node ID, and by domain / group ID. See the LonTalk Protocol Engineering Bulletin for more details.

flexibility in sending messages by allowing a single message to be addressed to more than one node. Another important benefit that logical addressing provides is the ability to share physical media with other systems. Some media, for example power line circuits and radio frequency transmissions, are open. That is, other systems may be using the same media without your knowledge and you cannot, and often times do not want to, prevent it. Other times, to reduce the total cost of a group of systems, you may want them to all share the same pair of twisted pair wires. The Neuron Chip uses its network addresses to filter out and discard messages from other systems that it is not interested in.

Another benefit of logical addresses is the ease of network maintenance they provide. When replacing a damaged node, a new node is given the same network personality (the same logical address and connection information) as the old, damaged node. Since the network addresses are unchanged, the exchange of physical devices is transparent to the rest of the network; none of the other nodes need to be informed of the repair.

Logical network addresses also increase the efficiency and performance of the message delivery system. For example, logical addresses give LONWORKS routers the ability to segment network traffic, increasing system capacity.

The process of address assignment is nothing more than pairing a *physical* node with a *logical* network address. This association is made at installation time using the unique Neuron ID of the Neuron Chip in the node. Each Neuron Chip is given a unique Neuron ID at manufacture time which, like a serial number, differentiates it from all the other Neuron Chips in the world. There are three methods for extracting a Neuron Chip's unique Neuron ID and pairing it with a network address: service pin, find and wink, and manual entry. Each of these options is discussed below. LONWORKS installation tools can use any or all of the methods. For some installation scenarios, addresses can be assigned by the LonBuilder Developer's Workbench and loaded into the device at the time of manufacture; other installation scenarios will use one of the methods discussed below. The right choice depends on the application and the type of end-user interaction you want to support.

Service Pin

Each Neuron Chip has a pin known as the service pin. When this pin is grounded, the Neuron Chip sends a broadcast message containing its Neuron ID and program ID. The method used to ground the service pin varies from node to node. Examples of mechanical methods include grounding via a push button or using a magnetic reed switch. By attaching one of the node's I/O pins to the service pin, the service pin can also be put under software control as long as the node is configured. For example, the node can ground the pin when the node is moved or when a predefined series of I/O occurs.

The requirements of your application determine how you use the service pin. For example, if you need to have the installation technician positively identify a device before installing it, you can attach the service pin to a push button. When the installation technician wants to install a node, the installation tool could give instructions to press the node's service pin button. If you don't need to know which device you are installing, you can connect the service pin to a circuit that causes it to ground when the node is attached to the network. In this case, the installation tool would need to be on line, "listening" for service pin messages. This type of installation, known as plug-and-play or automatic installation is discussed later in this bulletin.

Find and Wink

When it is impractical to reach a node to press its service pin button (for example, if the node is behind a wall or in a false ceiling), the find and wink installation method is useful. With this method, the network installation tool searches the network for unconfigured nodes. If more than one uninstalled node is found, the installation tool can use the LonTalk wink network management message to differentiate among the nodes. Based on the application code within the node, when a node receives a wink message, it responds in a way that can be easily detected by the person doing installation. For example, lights can blink, alarms can ring, motors can turn, and so on. The person doing the installation can then associate physical application nodes responding to a particular wink message as the one currently being installed.

Manual entry

The final method for pairing a physical node with a logical network address is to manually supply the node's Neuron ID. The method used to enter the Neuron ID varies by installation scenario. For example, the installer could be required to type in the Neuron ID of each device. Alternatively, each node could be manufactured with a bar code that contains the Neuron ID of the Neuron Chip on the node. As each node is installed, the bar code could be peeled off and placed on a set of plans next to another bar code defining the location. Manually entering the data could be as easy as running a bar code reader over the coded location on the plan (representing the node's physical location) followed by running the reader over the bar code containing the Neuron ID of the node at that location.

B — Binding

Physically attaching nodes to the transmission medium and assigning them addresses does not specify how they communicate and share information with one another. In a LONWORKS network, all interoperable communication between

nodes is done using network variables[3]. The types, functions, and number of network variables in each node are determined by the application code within the node. Network variables make it easy to develop networked control applications by automating all of the low-level and tedious work of building, sending, and responding to messages.

Network variable updates are sent using implicit addressing[4], in which the Neuron Chip firmware builds and sends network variable update messages using data contained in tables in its EEPROM. Binding is the process of setting up these tables. A binder is the part of an installation tool that does binding. The implicit addressing established during binding is called a connection[5]. The installation tool is responsible for allocating the network resources used by connections and for binding.

Unlike conventional messaging approaches, because of the use of implicit addressing, the binding process enables LONWORKS systems to be installed and modified efficiently and economically. Since the Neuron Chip, transparently to the application, builds messages and generates addressing information, the node's behavior can be modified, over the network, at any time; there is no need to change application code or move physical wires.

C —Configuration

Configuration is the process of tuning a node for a particular network. This includes setting both network-related and application-related parameters. Network-related parameters include the node's use of LonTalk priority, the node's use of

---

[3] Nodes can also communicate with one another through using low-level message tags. See the *Neuron C Programmers Guide* for more details.

[4] Nodes have the option of constructing, individually, network variable update messages and assigning addresses to them. This process is known as explicit addressing.

[5] There are two pieces of information that the Neuron Chip uses to process and qualify a message as an incoming network variable update. The first is the address to which the message is addressed. The node only processes the message if the message is addressed to the node. If the node determines that the message is addressed to it, the node looks at the message to see if the *network variable selector* in the message matches a network variable selector on the node.

Network variable selectors are 14-bit numbers assigned by the installation tool during the binding process to identify connected network variables. Nodes may use different names to refer to a network variable, or network variables may be located at different offsets within each node's memory. All network variables in a connection must have the same network variable selector value. Also, each network variable can have only one network variable selector. Any network variable selector can occur only one or two times in any node – once for an output network variable and once for an input. This lets the node uniquely identify each network variable. This does not mean that network variable selectors need to be unique in the network; any network variable selector can be used multiple times in the same network. As long as the nodes using the selector do not have any connections in common, the selector is unambiguous to all of the nodes.

LonTalk authentication services, and the node's location string. Application-related parameters include data such as set points, calibration values, and linearization tables. For application configuration data, nodes can use either configuration network variables or standard configuration parameters (SCPTs). SCPTs are stored in a file on the node and are read and set by the installation tool using the LonTalk file transfer protocol.

The configuration task also involves configuring network infrastructure components such as LONWORKS routers. While a LONWORKS network can be built using only a single channel, most networks will grow to use multiple channels and multiple media connected by LONWORKS routers. For multi-channel or multi-media networks, additional configuration is required to set up the router forwarding tables based on the network's topology.

Constructing a network out of many channels has several advantages. Each router physically isolates the channels it connects, protecting each side from failures on the other. Routers can also extend systems to cover a greater distance or contain more nodes than a single channel can physically support. Routers can also selectively filter and forward packets to increase system capacity. Finally, the ability to build systems from multiple media gives system designers the opportunity to pick the right media based on cost and performance requirements for each subsection of their application.

Installation Scenarios

The steps that a person follows to install a network are called the *installation scenario*. The installation scenario determines the "look and feel" of the network as viewed by the person responsible for network installation. One of the most important tasks of the developer creating a LONWORKS application is to define and implement the installation scenario for their system. The best scenario for any given network depends on many factors, such as the skill level of the installer, the amount of flexibility desired, and the requirements of the end-user. In all cases, however, as much of the installation process as possible should be automated. Automation eliminates user errors and makes installation faster and easier.

The list below describes some common installation scenarios, categorized by where and by whom the ABCs of installation are done.

- **Pre-installed systems**. These are a collection of nodes that ship from the factory configured to work with one another. This is the least flexible installation scenario. Pre-installed systems are generally limited to single-vendor systems and are typically used for cases where LONWORKS is being used as a wiring harness replacement within a machine. Pre-installed systems are created using the LonBuilder Developer's Workbench as the installation tool using the "export configured" option so that, at manufacture time, when each node is programmed with its application code, it is also programmed with its network personality. See the *LonBuilder User's Guide* for more details.

- **Self-installed nodes**. Self-installed nodes are a small step up from pre-configured systems. Self-installed nodes are actually pre-configured nodes that provide a user-interface, such as a dip-switch or thumb wheel, to allow the user to adjust some of their configuration parameters. Like pre-configured systems, self-installed nodes are created using the LonBuilder Developer's Workbench as the installation tool using the "export configured" option so that, at manufacture time, when each node is programmed with its application code, it is also programmed with its network personality. The node's application program uses functions provided by Neuron C to read the user-interface hardware and modify its configuration information[6]. See the *LonBuilder User's Guide* and the *Neuron C Programmer's Guide* for more details.
- **Automatic installation**, In the automatic installation scenario, to the end-user, the system seems to "magically" configure itself. In reality, automatic installation is accomplished by an embedded installation tool that is either part of a control panel or hidden in a "black box" that automates all installation tasks. To be able to automate installation and eliminate or minimize end-user interaction, the embedded tool's application program needs to be very knowledgeable about the system in which it resides. That is, the tool needs to supply the application knowledge so that the end-user does not have to. For this reason, automatic installation is most commonly used in single vendor systems or in systems that are dedicated to a single function.
- **Engineered system**. In the engineered system installation scenario (also sometimes referred to as *predefined components installation*), installation is a two step process of definition followed by commissioning. In the definition stage, all of the system configuration information is defined off-site, without a physical network present, and loaded into an installation tool that will later be embedded into the network. In the commissioning stage, the tool (that now contains all the network configuration information in its database) is brought on site, attached to the network, and the configuration information is loaded into the nodes. The advantage of this installation scenario is that since most of the time-consuming data entry and processing is done off-site, the network installation on-site is very quick, easy, and error free. This scenario is often used when installing systems that require preplanning or that are built in response to a bid.
- **Ad-hoc systems**. In the ad-hoc sytem installation scenario (also sometimes referred to as *connect-as-you-go installation*), installation is a one-step process.

---

[6] While it is possible to set any and all of the node's installation information using the self-installation functions of Neuron C, it is usually not practical or economical. Since the end-user is responsible for setting installation information for each self-installed node — in effect acting as an installation tool, as the amount of information they need to set increases, so does the amount of training and understanding required. In most cases, automatic installation proves to be a more cost-effective and easier-to-use installation scenario that self-installed nodes.

In this scenario the installation tool loads the network configuration information into each node as the installer defines nodes and connections. The installation tool's user interface determines the amount of information and sequencing of the information required. This is different from the engineered system scenario in that information is incrementally loaded. It is different from the automatic installation scenario in that the installer is making decisions about the network configuration instead of (or with the assistance of) the tool. The goal of connect-as-you-go installation is to integrate all installation activities into one step. This type of scenario has the advantage of offering the most flexibility by letting the installer make decisions on-site

When looking at these scenarios, it is important to remember that although there are many choices and variations available, selecting one scenario does not lock you into that choice forever. The LonTalk protocol has been designed to make all of these installation scenarios compatible. Systems installed with one of the less flexible scenarios can migrate at a later date to a field installation tool, without having to change node application code or hardware. In this way, LONWORKS technology can extend a product's useful life by allowing it to adapt to the growing and changing needs of the end-user.

The remainder of this engineering bulletin focuses on the use and construction of installation tools. These installation tools install systems using any of the last three installation scenarios discussed above. With these scenarios, OEMs can realize all of the cost savings possible through LONWORKS technology. Installed system cost is decreased through the replacement of physical wires with logical wires, reducing system installation time and lowering both labor and materials cost. Maintenance costs are dramatically reduced by allowing moves and changes to be done at any time — without requiring the network to be rewired. Device parameters can also be over the network, lowering device costs by simplifying hardware and lowering inventory costs by eliminating the need to stock multiple variations of a device.

Installation Tool Architecture

What an installation tool physically looks like depends on the application. Also, in most applications, the installation tool is usually used not only for installation and maintenance but is used as an application node in the network for system-level monitoring and control. In some cases, the installation "tool" might be a special mode of a smart thermostat. In this case, the user interface might be an LCD panel and a few push buttons. In other cases, the installation tool might be a PC running the Windows operating system. In this case the installation tool might offer a very advanced and easy-to-use graphical interface. In still other cases, the installation tool might not have any user-interface at all — it might be embedded in the network as a "black box" that automates network installation tasks.

In all cases, however, the architecture of the tool is the same. The architecture of a generic installation tool is shown in figure 2.

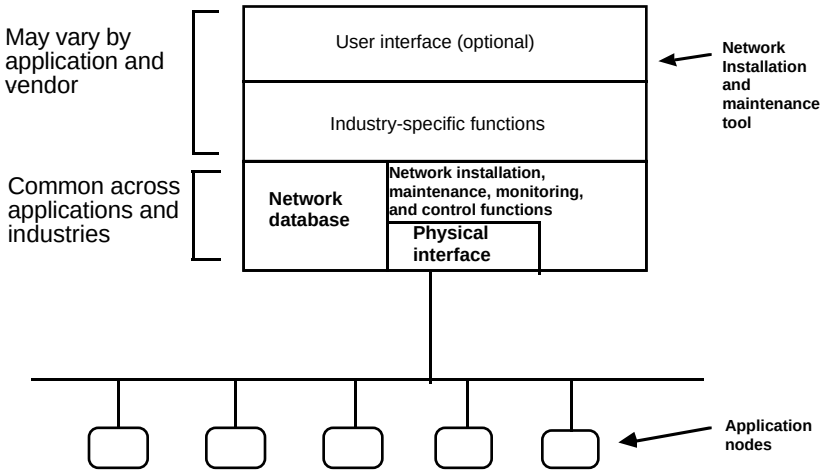


Figure 2. Network Installation Tool Architecture

Each installation tool contains the following components:

- **A physical interface**. The physical interface provides the connection between the installation tool and the LONWORKS network.
- **A network database**. The network database is use to allocate and track network resources. An installation tool must contain a network database so that it can ensure that resources are allocated correctly and efficiently and so that damaged nodes can always be replaced.
- **Network installation. maintenance, monitoring, and control functions**. These algorithms define the rules that the tool follows to assign network addresses, create logical connections, set network timers, and allocate other network resources. These algorithms use the network database to access network information and the physical interface to send and receive messages.
- **Industry-specific functions**. These algorithms, built upon the base installation. maintenance, monitoring, and control functions, customize the tool for a particular application. It is these functions that give the tool "industry smarts" and make it easy to use.
- **A user interface** (optional). If user input is required as part of the installation process, it is done from the user interface. This interface could be anything from a graphic display to a simple push button.

The last two items are unique to each industry or manufacturer. It is through the user interface and industry-specific functions that manufacturers can differentiate their product from other products while still maintaining interoperability.

The first three items in this list are common across all industries. To eliminate the need for developers to devote time and energy to the common items, Echelon provides physical interface components and network services toolkits to free developers to focus on their user interface and industry-specific functions.

Off-the-shelf Installation Tool Options

Echelon offers the LonManager LonMaker™ Installation Tool for manufacturers looking for a turn-key approach to network installation. The LonMaker software works with a companion tool called the Profiler to simplify the field technician's job so that the technician can install a control network without having to understand networking. The Profiler is designed for a specifying engineer and the LonMaker software for a field technician. Detailed tasks requiring an understanding of LONWORKS technology are moved away from the technician using the LonMaker software and to the specifying engineer using the Profiler.

The Profiler, which runs on any IBM PC/AT compatible computer, creates customized "parts catalogs" tailored to a manufacturer's products. The parts catalog describes the types of nodes the manufacture makes and how they interact with each other. For example, the network components and control schemes (the "parts catalog") of an egg packing plant's control system are very different from those of a building's HVAC control system. The parts catalog allows the installation process to be customized to a specific set of network components, simplifying installation while maintaining flexibility.

The LonMaker software, which runs on any IBM PC compatible — including palmtops, uses a parts catalog created by the Profiler as a palette of objects that the installer can select from to build and install networks in the field. The information in the parts catalog lets the LonMaker software simplify and automate many common installation tasks, speeding network installation, minimizing installation errors, and reducing training requirements. Figure 3 shows how the LonMaker software fits the generic network installation tool architecture described earlier.

LonMaker Software

- User interface
- Vendor-specific knowledge
- Network database
- Network installation. maintenance, monitoring, and control functions

PCLTA, SLTA, or LTS-10

- Physical interface

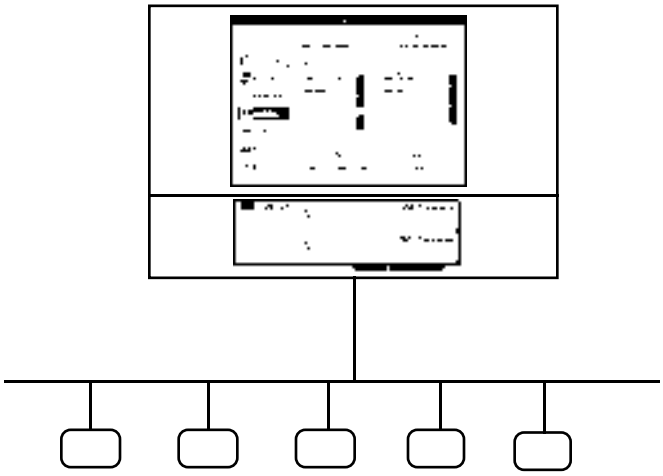


Figure 3. Off-the-shelf installation tool using a DOS host

There are also a number of installation, maintenance, monitoring and control tools available from third parties based on the LonManager NSS-10 Module and the

LonManager API. See the *LONWORKS Resource Directory* for more information on these other tools.

Building Your Own Installation Tools

For most manufacturers, off-the-shelf tools such as the LonMaker software provide a good combination of ease-of-use, customizability, flexibility, and power. There are, however, occasions where you will want to create your own installation tool. For example, you may have custom requirements for the user interface or you may have application-specific algorithms that will allow you to reduce installation time and training requirements. To simply development of LONWORKS installation, configuration, maintenance, monitoring, and control tools, Echelon provides the LonManager NSS-10 Module, the LonManager API for DOS, and the LonManager API for Windows.

Using the LonManager NSS-10 Module

The LonManager NSS-10 Network Services Server (NSS-10) module is an intelligent peripheral device that lets any microcontroller, microprocessor, or computer host perform a broad range of LONWORKS network services. These services include both network management (network installation, configuration, maintenance, and repair) and system-wide monitoring and control.

Typically the NSS-10 module is an integral part of the network it manages, either as a black box that automates installation tasks, or as part of a control panel. When used as an automated network installer, the NSS-10 module makes it possible to develop networks that install themselves with little or no end-user action. When included in a control panel, the NSS-10 module makes it possible to deliver a single device that the end-user can use to set and observe both the network and the application configuration of the system.

One of the main functions of the NSS-10 module is to give the host processor an easy way to accomplish network management operations that are actually quite complex. It does this by making network management operations available to the host application through a collection of *services*. To accomplish any given operation, the host application invokes a service on the NSS-10 module.

A single service invocation to the NSS-10 module usually results in multiple LonTalk messages. When the host invokes a service, for example to connect a set of network variables, the NSS-10 module expands the request into the required network actions — allocating network resources; building, sending, and processing network messages; performing error checking and recovery — and then returns the result of the request to the host. By invoking the appropriate service the host

application can quickly accomplish each network management task with a minimum of overhead. The NSS-10 module manages the node and network resources for the host application and sends and processes any required LonTalk messages.

Wherever possible, the NSS-10 module automates network-related tasks. For example, the NSS-10 module automatically discovers the presence of newly attached nodes on the network — without the host application having to do anything. Where automation is not possible, simplification is provided. For example, the NSS-10 module uses connection descriptions to allow the host application to specify all the attributes of a connection with a single parameter. By adding application knowledge, for example the set of nodes that it may encounter and the types of configurations that make sense, the host application can further automate tasks and simplify or eliminate the user interface. The NSS-10 module keeps a database of the network configuration. This offloads the resources and complexity associated with database maintenance from the host.

The NSS-10 module makes these network management services available to the host application:

- Discovery of new nodes as they are physically attached to the network
- Network deconfiguration
- Node commissioning
- Receiving service pin messages
- Importing node self-documentation and self-identification information
- Importing node external interface files
- Copying configuration network variable values from one node to another
- Installing, removing, and replacing nodes
- Connecting and disconnecting network variables and message tags
- Loading application images into nodes
- Querying and setting node properties, such as locations, priority slots, self documentation, and network variable attributes
- Resetting nodes
- Winking nodes
- Testing nodes

To perform system-wide monitoring and control functions, the NSS-10 host application needs to know the logical addressing information of all the nodes in the system. As the NSS-10 module assigns network addresses and connection information, it keeps a record in its on-board database. Using the network configuration information in the NSS-10 module's on-board database, the host

application can read or write any network variable on any node and send explicit messages to any node without the need for a network variable or message tag connection between the host and the node.

Once the host application has the required addressing information, it treats the NSS-10 module as a standard LONWORKS network interface. The NSS-10 network driver protocol is compatible with the protocol used by the drivers for the PCLTA PC LonTalk Adapter, the SLTA/2 and LTS-10 serial LonTalk adapters, and the LonBuilder Microprocessor Interface Program (MIP).

The host can also implement network variables and message tags on itself and use the network variable and message tag binding services of the NSS-10 module to connect itself to other application nodes. After the connection is complete, the host receives event driven updates from the nodes just like any other LONWORKS node.

Figure 4 shows how the NSS-10 module fits the generic network installation tool architecture described earlier.

Any host

- Industry-specific functions
- User interface (optional)

LonManager NSS-10 Module

- Network database
- Physical interface
- Network installation. maintenance, monitoring, and control functions

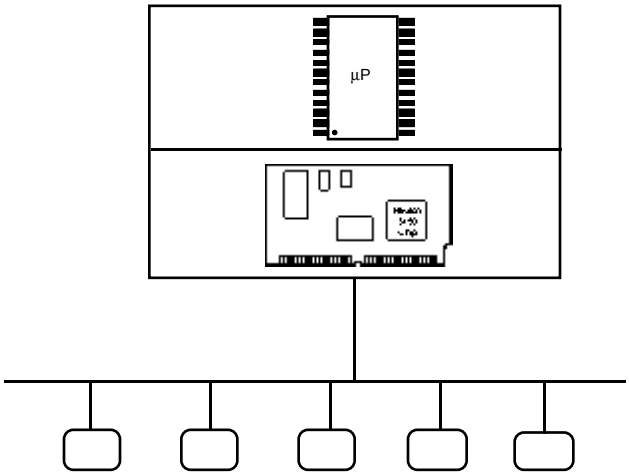


Figure 4. Custom installation tool using any host

Using the LonManager API

The LonManager Applications Programming Interface (API) for DOS and Windows is a tool for programmers (provided as DOS object libraries or as Windows dynamic link libraries). It provides developers with a comprehensive library of almost 200 network and application services, freeing them to focus on application development.

With the LonManager API, operations that would ordinarily require the development of thousands of lines of code are replaced with single function calls — saving months of design and years of development and test. For each LONWORKS installation and configuration task (such as node installation, node replacement, network variable and message tag binding, application code downloading, and

router configuration) the LonManager API provides highly integrated and easy-to-use functions. These functions perform error checking; build, send, and process all required network messages; calculate and optimize LonTalk timers as appropriate, and manage the relationships between data items, creating and removing linked database records as needed to keep the network and database synchronized.

The LonManager API can be used to create applications that interact with a wide range of networks — from tens of nodes to tens of thousands of nodes. To simply this interaction, the LonManager API includes an integrated database that reflects the addressing and configuration information of every node in the network. The database and its associated management functions greatly reduces the development effort required to create network installation, maintenance, monitoring, and control applications.

Figure 5 shows how the LonManager API fits the generic network installation tool architecture described earlier.

PC Application

- Industry-specific functions
- User interface (optional)

LonManager API

- Network database
- Network installation. maintenance, monitoring, and control functions

PCLTA, SLTA, or LTS-10

- Physical interface

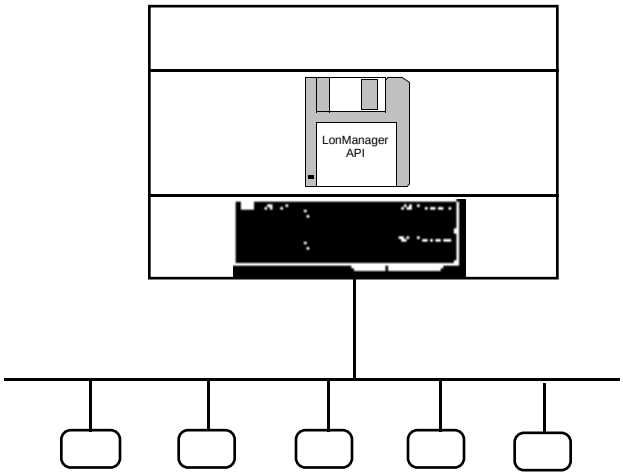


Figure 5. Custom installation tool using a DOS or Windows 3.1 host

Summary

The flexibility of LONWORKS technology has lead to its adoption in a huge range of applications with greatly varying installation scenarios and requirements. In all cases, installation in a control network is the process of setting the address, binding, and configuration information that turns a generic application node into an integrated part of an intelligent network. However, where and how the information is set varies greatly with each application, and in each case Echelon provides tools to simplify the development and deployment of LONWORKS systems.

For closed systems, all of the installation information can be defined and loaded at the time of device manufacture. For small systems with limited flexibility, the majority of the installation information can be defined at the time of manufacture and the remaining pieces set in the field using simple hardware at each node. For maximum flexibility and growth, an installation tool can be used in the field to set all of the required information. An installation tool can extend the useful life of pre-installed systems and self-installed nodes by incorporating these nodes into the system it is maintaining to adapt to the changing and growing needs of the end-user.

Enhanced Media Access Control with LonTalk® Protocol

January 1995 **LONWORKS® Engineering Bulletin**

Introduction

This note provides an introduction to the LonTalk media access control (MAC) sublayer. The MAC sublayer is part of the Data Layer of the OSI Reference Model. Many different MAC algorithms exist in networks today. One family of these algorithms is called CSMA (Carrier Sense Multiple Access). The MAC algorithm used by the LonTalk protocol belongs to the CSMA family. An explanation of the CSMA algorithm used by the LonTalk protocol is presented below. Further, its features are compared with some of the other members of the CSMA family.

Background on CSMA Family

The CSMA family of media access control algorithms requires a node to establish that the medium is idle before it begins to transmit. However, each algorithm behaves differently once the idle state is detected. This results in very different network performance results under conditions of heavy data traffic.

Some CSMA algorithms use discrete intervals of time called slots, or randomizing slots, to access the medium. By limiting access to the medium by a given node to specific time slots, slotted media access greatly reduces the probability of two packets colliding. Slotted media access is used by p-persistent CSMA, and Echelon's LonTalk CSMA protocol.

Overview of the LonTalk MAC Sublayer

Echelon's LonTalk protocol uses a new CSMA MAC algorithm called *predictive] persistent* CSMA. The LonTalk protocol retains the benefits of CSMA but overcomes its shortcomings for control applications. Existing media access control algorithms such as IEEE 802.2, 802.3, 802.4, and 802.5 do not meet all the LonTalk requirements for multiple communication media, sustained performance during heavy loads, and support for large networks.

As in p-persistent CSMA, all LONWORKS nodes randomize their access to the medium. This avoids the otherwise inevitable collision that results when two or more nodes are waiting for the network to go idle so that they can send a packet. If they wait for the same duration after backoff and before retry, repeated collisions will result. Randomizing the access delay reduces collisions. In the LonTalk protocol, nodes randomize over a minimum of 16 different levels of delay called randomizing slots. Thus the average delay in an idle network is eight slot widths.

Echelon Part Number 005-0001-01 REV C

In p-persistent CSMA when a node has a message to send, it does so in a given randomizing slot with probability p. However, the LonTalk protocol carries the added improvement that p is dynamically adjusted based upon network load. When the network is idle, all nodes randomize over only 16 slots. When the estimated network load increases, nodes may randomize a large number of slots. The number of slots increases by a factor of n, where the range of n is from 1 to 63. Echelon calls n the estimated channel backlog. It represents the number of nodes with a packet to send during the next packet cycle.

This method of estimating the backlog and dynamically adjusting the media access allows the LonTalk protocol to have just a few randomizing slots during periods of light load, while having the benefit of many randomizing slots during periods of heavy load. Thus, media access delays are minimized during periods of light load, and collisions are minimized during periods of heavy load.

As noted earlier, the ability to adjust the number of randomizing slots depends on the ability to estimate the channel backlog. In the LonTalk protocol, a transmitting node includes information in the packet on the number of acknowledgements expected as a result of sending that packet. All the nodes that receive the packet increment the estimated channel backlog by that amount. Likewise, the estimated channel backlog is decremented by 1 at the end of each packet cycle. The estimated channel backlog is never decremented below 1. Since LonTalk packets are typically acknowledged, 50% or more of the channel backlog is predictable at any time.

Benefits of LonTalk MAC Sublayer

LONWORKS systems allow thousands of nodes and multiple media on a single network. Because of the characteristics of different communication media and the potential need to cover large distances, LONWORKS networks must be able to support low data rates. Time-slotted MAC sublayer s are not appropriate for low data rates because they are suited to high data rates and small numbers of nodes. Thus, a time-slotted MAC sublayer was not chosen for the LonTalk protocol.

The multiple-media support provided by the LonTalk protocol also rules out a token-ring approach which only works on media with the propagation characteristics of wire. In a token-ring network, the token passes in an orderly fashion around the ring. This cannot work on either powerline or RF networks because all stations receive the token simultaneously. Additionally, the cost of incremental hardware to recover the token when it is lost, and to rapidly acknowledge the token, makes the hardware for this approach more expensive to implement.

A token bus architecture solves the problem of sequential passing of the token by including addressing information in the token. However, at low data rates, the process of circulating the token can result in considerable token latency. Since a node cannot transmit without first possessing the token, this latency adversely

affects response time. Additionally, token bus systems must reconfigure themselves each time a new device either becomes active or drops out. This overhead to reconfigure is a problem for all token bus networks. Since reconfiguration brings the network down for its duration, battery-powered nodes whose normal operation is to wake up, send some messages, and power down, would cause the token bus system to suffer frequent reconfigurations. Battery-powered nodes are required for applications needing RF or IR communication, for security applications, and for fire/life safety applications, to name a few.

The CSMA family of MAC sublayers does not require a ring topology, synchronization or reconfiguration, and does permit nodes to drop out and rejoin the network transparently. Additionally, it supports many nodes and is inexpensive to implement in hardware. Unfortunately, CSMA/CD (for example IEEE 802.3) behaves poorly during periods of overload, so it is generally not used for control applications. P-persistent CSMA works very well for small values of p at the expense of additional delay during relatively idle periods. The LonTalk MAC sublayer has the advantages of p-persistent CSMA without the disadvantages of additional delay during low traffic, or significantly reduced throughput under conditions of high traffic.

In summary, the LonTalk MAC sublayer specifically overcomes the shortcomings of existing MAC sublayers in the following areas: multiple-media communication, low data rates, sustained performance during conditions of heavy network traffic, and large networks.

The LonTalk Predictive P-Persistent CSMA Protocol

As mentioned earlier, Echelon's LonTalk protocol falls under the category of predictive p-persistent CSMA. It is predictive because each node dynamically predicts how many other nodes have a packet to send at any given time. This prediction influences the number of randomizing slots between each packet. The higher the prediction, the more slots there are for the nodes to randomize over. Increasing the number of slots reduces the probability of a collision. The predictive algorithm is based upon the fact that most LonTalk packets are acknowledged. The number of acknowledgements that a given packet generates is encoded into the packet. Each node on the channel receives the packet and adds the number of acknowledgements to the channel backlog. If none of the packets is acknowledged, the predictive part of the algorithm does not dynamically expand the number of randomizing slots with an increase in load. Using exclusively unacknowledged services causes the LonTalk protocol to behave like a p-persistent CSMA where $p = 0.0625$. However, p-persistent CSMA is still significantly better than IEEE 802.3 under conditions of heavy network traffic.

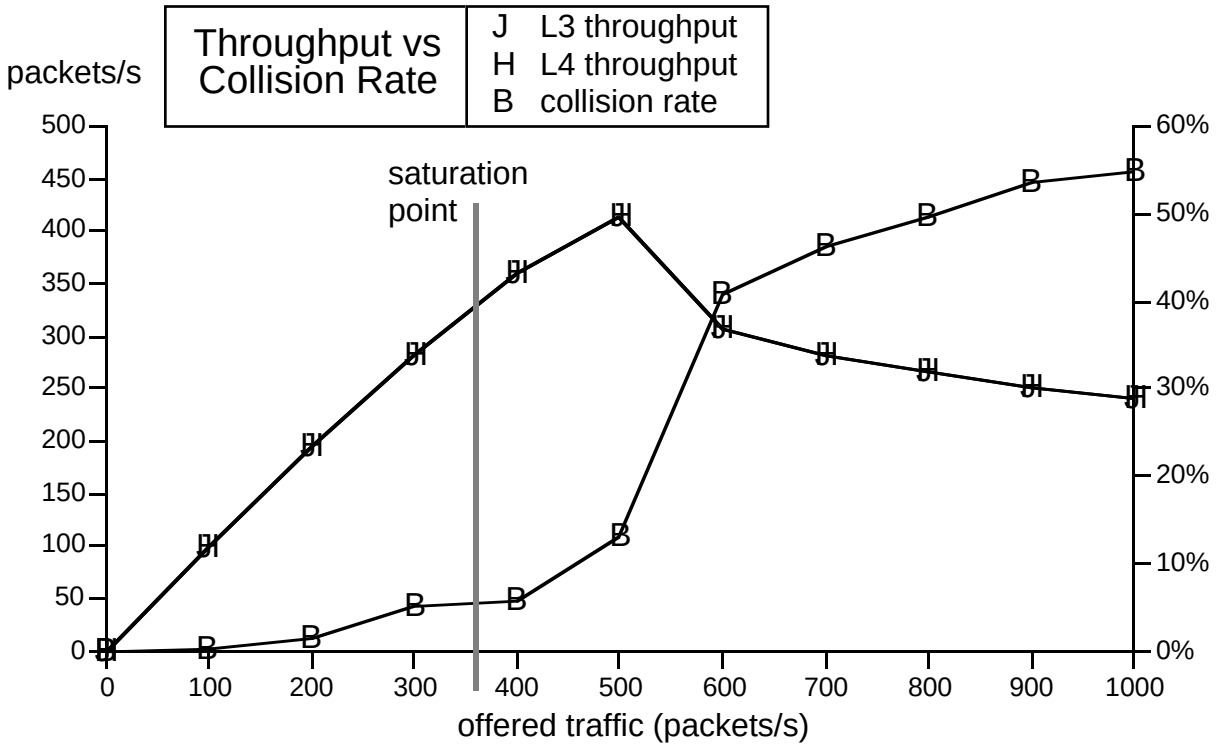


Figure 1: Effective network throughput versus offered traffic on a LONWORKS network using unacknowledged services. Note that in this mode, LonTalk behaves like conventional, p-persistent CSMA.

The results of two experiments used to illustrate this point are shown in figures 1 and 2. The graphs shown in these figures illustrate results achieved from a test bed of 36 nodes. Thirty-four of the nodes acted as traffic generators. The other two acted as test nodes, repeatedly sending messages to each other at different traffic levels. Both graphs show the amount of actual traffic that got through on the network versus the offered traffic, or the number of packets attempted for transmission.

Figure 1 shows this data using exclusively *unacknowledged* packets. In this mode, the network behavior is similar to a 0.0625-persistent CSMA algorithm. Note how the network throughput rises rapidly with offered traffic and plateaus at 400 packets/sec. The network throughput starts to fall off due to excessive collisions when the network is driven well beyond saturation.

To demonstrate the effectiveness of the predictive algorithm, the experiment was re-run with only one change. In this case, messages used *acknowledged* services. Figure 2 shows the data derived from this experiment. Note that network throughput does not continue to degrade with an increase in offered traffic past the saturation point. Additionally, it is important to note that the collision rate levels

off rather than continuing to increase with offered traffic. This active management of the collision rate is what makes the LonTalk protocol's predictive p-persistent CSMA algorithm superior to other CSMA algorithms.

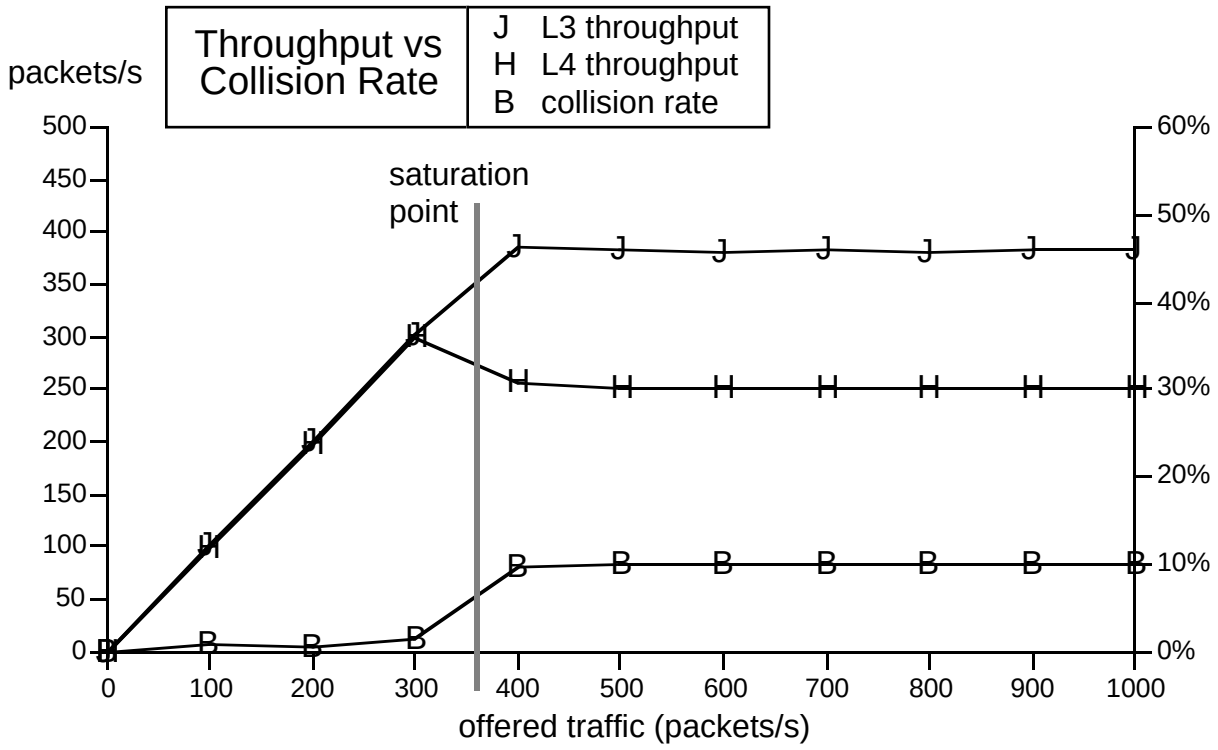


Figure 2: Effective network throughput versus offered traffic on a LONWORKS network using acknowledged services. Note that in this mode, predictive p-persistent CSMA allows active management of the collision rate and prevents throughput from degrading past the saturation point.

Conclusion

The LonTalk protocol uses a new CSMA MAC algorithm called predictive CSMA. To avoid collisions, all LONWORKS nodes randomize their access to the communication medium using time slots. The LonTalk protocol dynamically adjusts the number of randomizing time slots by predicting the channel backlog. By actively managing the collision rate, the LonTalk protocol provides a superior MAC sublayer for multiple-media communication, low data rates, sustained performance during heavy loads, and large networks.

Optimizing LONTALK Response Time

August 1991 **LONWORKS™ Engineering Bulletin**

Introduction

Several tools are available to the LONWORKS network designer to optimize response time. The LONTALK™ protocol supports a variety of optional messaging services and collision detect hardware. The features of these services and their effect on LON® response time are discussed.

Media Access Delay & Offered Traffic

When a node on a network tries to send a message, it must first wait for the medium to be idle1. The time delay between when a node queues the packet to send and the time that it is actually sent on the network plays a role in the response time of that message. This delay is known as the media access delay. As the offered traffic (the total number of packets per second offered for transmission by all the nodes on a channel) on a given channel of a network increases, the media access delay increases. In networks loaded to near-capacity, when many nodes are trying to send messages, this delay can be significant. In designing a network to meet specified response times, the worst-case offered traffic must be considered and designed for.

In order to assess the impact of media access delay, we need to first consider the amount of time a typical message spends on the media. For example, consider a network design where the average packet size is 112 bits (14 bytes) and the average number of bits between each packet is 48. Also, suppose that the channel runs at 78 Kbps. Each packet cycle takes 160 (= 112 + 48) bit times on the channel. Thus, the maximum rate at which packets can be sent on this channel is 487 packets (=78000 / 160) per second, and each packet cycle requires 2 ms.

Now consider a node on this network that has a maximum response time requirement of 50 ms. In order to design this network to meet this response time requirement, we must set a bound on the offered traffic to limit the media access delays. At 78 Kbps with 160 bit packet cycles it takes this network 50 milliseconds to send 25 packets. If we further assume that the packets on the channel may need to be sent at any time, including all at once, then, in order to always meet the requirement of 50 milliseconds without using any special priority features, the channel should never have an offered traffic greater than 25 packets per second. As this example illustrates, the response time requirements of nodes on a network

1 Details are described in the LONWORKS Engineering Bulletin entitled *Enhanced Media Access Control with Echelon's LONTALK Protocol.*

must be used to design the network and ensure that the offered traffic does not cause unacceptable media access delays

Optional Priority

The LONTALK protocol supports an optional priority messaging service that enables intelligent nodes to obtain optimal response times. When a node tries to access the communication medium, priority messages are given earlier access than non-priority messages. In the LONTALK protocol, the effectiveness of priority depends on all nodes preparing to send a message being synchronized on the end of the previous packet. When the sending nodes are synchronized, then the priority time slots remain uncontested. Priority assignment has two limitations: (i) only a limited number of nodes may be assigned priority; and (ii) reserving time slots for nodes of different priorities pre-allocates bandwidth. For Echelon's 78 Kbps twisted-pair transceiver, each priority slot is 8 bit times wide. Using the 1.25 Mbps twisted-pair transceiver the priority slots are 21 bit times wide. In effect, each packet length is increased by this amount thus using up bandwidth.

Collision Detection

A collision is defined as the event when two or more nodes access the communication medium at almost the same time, resulting in mutual interference among their electrical signals on the transmission medium. Packets involved in a collision cannot be successfully received, which results in a delay in response time.

The media access algorithm used by the LONTALK protocol minimizes the number of potential collisions by randomizing access to the medium. Details are described in the Echelon Engineering Bulletin entitled *Enhanced Media Access Control with Echelon's LONTALK Protocol.* The LONTALK protocol also encodes an estimate of the channel backlog in each packet so that the number of randomizing slots can be dynamically adjusted depending on network traffic. Randomizing slots reduces the probability of collisions, however it does not completely eliminate them. Further, as network load increases, the probability of collisions increases.

Collision detection hardware at a transmitting node informs the transmitting node shortly after a collision occurs about the need to retry. The NEURON® CHIP firmware is configured to detect a collision at the end of the packet preamble and the end of a packet transmission. In the absence of such hardware, the transmitting node only learns about the unsuccessful transmission from a lack of an explicit or implicit positive acknowledgement from the receiving node, which may take much longer. Thus, collision detection is very useful for nodes requiring fast response time.

Implementing collision detection requires media-specific hardware solutions. Echelon has designed the hardware to implement collision detection on 78 Kbps and 1.25 Mbps twisted-pair transceivers.

Network and Application Input Buffers

In a LON system, every node connected to a channel receives every packet transmitted on the channel and checks whether the packet is addressed to it. Thus the speed at which a NEURON CHIP discards packets not addressed to it determines the upper limit for the number of packets per second on a channel. If packets arrive at a node faster than the node can process the packets and free up the input buffers, packets will be lost. This has no effect if the packet was not addressed to the node in the first place. However, if the lost packet was addressed to the node, response time is affected.

Once a packet has been received in a network input buffer, its destination address is examined to see if the packet is addressed to the node. If it is, then the packet is copied into an application input buffer, and the network input buffer is freed. If no application input buffer is available (because the application is not processing its packets fast enough), then the packet is discarded and no acknowledgement is sent.

Output Buffers

The number of output buffers becomes a factor in response time when a node can generate outgoing packets faster than the network can take them. What happens in this case, is that several outgoing packets back up in the output queue. The NEURON CHIP is quite capable of generating over 300 packets per second. At lower data rates or during network saturation, this level of packet generation can easily overwhelm the ability of the network to accept packets. For this reason, it is always important to have an idea of the maximum number of messages your application may generate per second as this dictates the network data rate required.

Messaging Services

The LONTALK protocol offers four basic types of message service:

- acknowledged
- request/response
- unacknowledged repeated
- unacknowledged

These are described in the LONTALK Protocol Application Note. There are a number of tradeoffs relating to network efficiency and response time in selecting a message service. The effect of the message service timers on response time is discussed.

Transaction Timer

When the acknowledged message service is used, the transaction timer in the NEURON CHIP controls the time allowed for an acknowledgement to arrive. By

adjusting the transaction timer, the time delay before messages are retried (due to lack of acknowledgement) can be optimized, hence improving response time. A retry is initiated when the transaction timer on a node expires without an acknowledgement being received. The retry count controls the number of times that a node will retry to send a message.

The network processor in the NEURON CHIP is responsible for checking for the acknowledgement. To keep the number of retries to a minimum, this timer must be set high enough to accommodate the round trip delay of sending a message and getting an acknowledgement back. On a network running at 78 Kbps or 1.25 Mbps, where the source and destination(s) are all on the same channel, the transaction timer can be set to a low value such as 64 ms or 98 ms. In the case where the packet has multiple destinations, the longest path must be used to calculate the timer value. A node will not try to initiate retransmission until its transaction timer expires.

Repeat Timer

The unacknowledged repeated message service is used when a large number of nodes must be addressed and you want to ensure that the message gets through. With a large group, an acknowledged message would cause too much traffic. The repeat timer can generally be set much lower than the transaction timer. It specifies the interval between repeats of the initial packet. The initial packet is repeated as many times as is specified by the repeat count, from 1 to 15 times. This timer should be long enough for the receiving node(s) to overcome any short term buffer shortages.

Receive Transaction Timer

This timer is used for detecting duplicate messages. Upon receipt of a message addressed to a LONWORKS node, the protocol firmware attempts to allocate a receive transaction buffer if the packet represents a new transaction. Failure to allocate a new transaction record when it is needed causes the incoming packet to be discarded, and not to be acknowledged.

A new transaction is defined as one for which there is no transaction record containing the source address of the packet and a matching transaction ID. When this timer expires, the transaction record is deleted, and any new packet from the same source address will cause a new transaction to be created. If this timer is set for too short a time, such an additional packet (a retry) with the same source address and the same transaction ID would not be detected as a duplicate, but rather it would be acted upon again. On the other hand, if this timer is set for too long a time, transaction records tend to be allocated for a long time which could exhaust the supply of available records. This, in turn, would cause the node to discard packets because it cannot allocate a receive transaction record.

Application Response Time

The application response time indicates the time it takes an application in one node to see a change in status, and form a packet to send, plus the time for the application in the receiving node to process the packet and perform the control action.

The application in a LONWORKS node is programmed in NEURON C. The application scheduler is event-driven. The application response time is influenced by the the number of `when` clauses, the complexity of the conditional expressions within the `when` statement, and the execution times of the `when` clauses. In addition, the number of software timers declared, the number of address table entries, the number of network variables and message tags all affect response time. For most programs, the application response time can be ignored. If, however, your application program is near the maximum for any of these items, some measurements should be done.

Conclusion

The LONTALK protocol provides a variety of tools and optional services to optimize LON response time. The response time requirements of nodes on the network must be considered when selecting the type of message service to use: priority, acknowledged, unacknowledged/repeated, and also when deciding whether collision detection hardware should be included in the node design.

Scanning a Keypad with the NEURON® CHIP

LONWORKS™ Engineering Bulletin

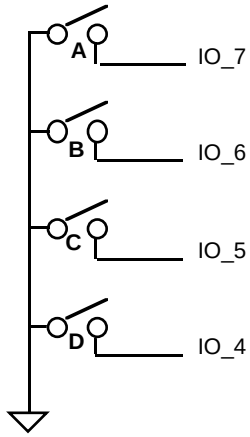
Introduction

This engineering bulletin describes how the Echelon NEURON CHIP can be used to scan a simple 16-key switch matrix to provide a numeric and/or special-function keyboard without the use of a keyboard encoder.

Depending on the number of keys to be scanned and the number of free I/O pins, different solutions are possible.

Scanning up to Eleven Keys

For up to four keys, the simplest way is to connect normally open switches to any of the pins IO_4, IO_5, IO_6, or IO_7 of the NEURON CHIP. These pins have on-chip pull-up resistors that should be enabled by the NEURON C program.



Sample software to drive these switches is shown below. After a switch is pressed, the software waits for some time (10 msec in this example) to allow the switch bounces to subside. The appropriate delay depends on the mechanical construction of the switch and how long it takes to settle in the closed or open position.

Echelon Part Number 005-0004-01

```
#pragma enable_io_pullups
IO_4 input bit io_key_A;
IO_5 input bit io_key_B;
IO_6 input bit io_key_C;
IO_7 input bit io_key_D;

when( io_changes( io_key_A ) to 0 ) {     // wait till key is pressed
      delay( 400 );                  // wait 10 milliseconds to debounce
      if( io_in( io_key_A ) == 0 ) {      // key is really pressed
            .......     // handle key press for key A here
      }
}
.....  similarly for keys B, C and D.
```

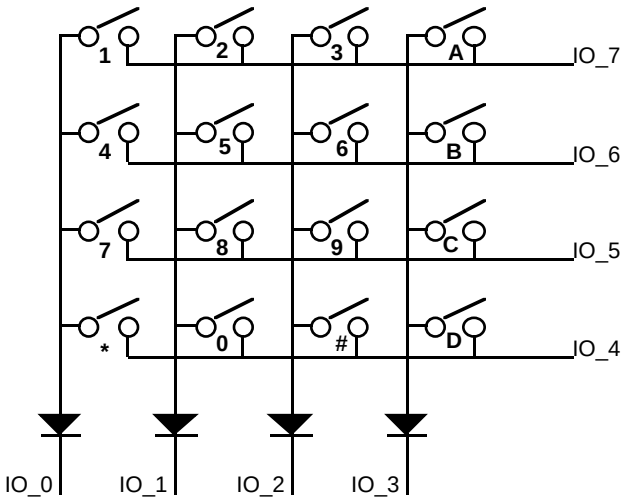
Since each key independently controls a single input pin, this kind of circuit can handle the case where multiple keys are pressed at the same time.

If a keypad has more than four keys (up to 11), the system designer can use the other pins (IO_0 through IO_10) of the NEURON CHIP in the same way. Note that for pins IO_0 through IO_3 and IO_8 through IO_10, there are no on-chip pull-ups. These should be provided with external resistors.

Scanning up to Thirty Keys

But using all the pins in this way may be wasteful if there are other I/O functions that need to be connected to the NEURON CHIP. A more economical way is to connect the keys in a rectangular matrix, using some of the pins to drive the rows of the matrix, and some of the pins to sense the columns. This engineering bulletin describes scanning a 4 X 4 matrix of keys (16 keys). A maximum of 30 keys can be scanned in this way, using all 11 pins to scan the keys in a 6 X 5 matrix.

The NEURON CHIP has 11 general-purpose I/O pins that can be used in several I/O modes. One of these modes is `nibble` mode, where groups of four I/O pins can be read or written together. Below, pins IO_0, IO_1, IO_2 and IO_3 are used as outputs to drive the columns of the switch matrix, and pins IO_4, IO_5, IO_6 and IO_7 as inputs to sense the rows of the switch matrix.



The NEURON CHIP has on-chip pull-up resistors on pins IO_4, IO_5, IO_6 and IO_7, which are enabled by the software statement `#pragma enable_io_pullups`. When none of the keys are pressed, pins IO_4 through IO_7 are therefore pulled up to logic level 1. In the declaration statements, the software configures these four pins as inputs, and pins IO_0, IO_1, IO_2 and IO_3 as outputs. External diodes are connected to these output pins to avoid logic clashes in the event that more than one key in the same row is pressed at the same time.

When none of the keys are pressed, the software writes logic 0's to the four output pins IO_0 through IO_3. When a key is pressed, one of the input pins IO_4 through IO_7 is pulled low, and the NEURON CHIP firmware detects the transition and generates an `io_update_occurs` event, thus activating a task to handle the event. In this example, this task first waits for 10 ms to ensure that any key bounce has subsided. In order to determine which key has been pressed, the software then pulls the output pins IO_0 through IO_3 low one at a time, leaving the other three output pins high, meanwhile reading the input pins each time it does this. If a low input is detected on a pin, this corresponds to the row in which the key is to be found. The row and column number are then used as lookup keys into a two-dimensional array to determine the key code for the key that was pressed.

For example, assume that no key is currently pressed. The output pins are 0000, and the input pins are 1111. When the 9 key is pressed, the pin IO_5 is pulled low, causing the input to change to 1101. The software then scans the keyboard by outputting the following values to the output pins:

Column	Output Written	Input Read
0	1 1 1 0	1 1 1 1
1	1 1 0 1	1 1 1 1

2 1 0 1 1 1 1 0 1 --> Bit 1 is 0,
so key pressed is column 2, row 1

The software updates a network variable nv_keypad_char with an ASCII character representing the key that has been pressed. In this example, the characters used are 0 - 9, * and #, corresponding to the layout of a 12-key telephone keypad. The other four keys are coded as A, B, C, and D. When the key is released, the network variable is updated with the ASCII NUL character, which represents the absence of a key press. (The NUL character is different from the '0' character, which is a valid character for this keyboard.) The output network variable is of a standard type - SNVT_char_ascii - which allows this node to interoperate with other nodes that accept this type of data from the network.

Code Example for Sixteen Keys

```
//=========================================================================
//
// File:    keypad.nc
// Created: 5/1/91
// Purpose: Scan 16-key keypad
//
//=========================================================================

//-------------------------------------------------------------------------
// Network Variable Objects
//-------------------------------------------------------------------------

network output SNVT_char_ascii nv_keypad_char;

//-------------------------------------------------------------------------
// I/O Objects
//-------------------------------------------------------------------------

#pragma enable_io_pullups

IO_0 output nibble io_keypad_columns = 0;     // pull down all lines
IO_4  input nibble io_keypad_rows;

//-------------------------------------------------------------------------
// Global Variables
//-------------------------------------------------------------------------

const char digit_table[ 4 ][ 4 ] = { "*0#D", "789C", "456B", "123A" };
                                             // map key codes to ASCII
```

```
//------------------------------------------------------------------------
// IO task
//------------------------------------------------------------------------

when( io_changes( io_keypad_rows ) ) {
    int row, col, row_val;

    delay( 400 );                               // 10 msec debounce
    for( col = 0; col < 4; col++ ) {
        io_out( io_keypad_columns, ~( 1 << col ) );
                                                // pull down one line
        row_val = ~io_in( io_keypad_rows );  // read rows
        for( row = 0; row < 4; row++ ) {     // find which one is zero
            if( row_val & ( 1 << row ) ) {
                nv_keypad_char = digit_table[ row ][ col ];
                               // propagate keypad character to network
                goto done;
            }
        }
    }
    nv_keypad_char = '\0';      // propagate NUL character
done:
    io_out( io_keypad_columns, 0 );       // pull down all lines again
}
```

Driving a Seven Segment Display with the Neuron® Chip

January 1995 **LONWORKS® Engineering Bulletin**

Introduction

This engineering bulletin describes how the Neuron Chip can be used to drive a seven-segment display controller chip, the Motorola MC14489. This device is used in Echelon's Gizmo2 and Motorola's Gizmo 3 multi-function I/O devices. The MC14489 can control up to five LED digits, each consisting of seven segments and a decimal point. No external current limiting resistors or drive transistors are required. The chip has a Serial Peripheral Interface (SPI), allowing for easy connection to the Neuron Chip's Neurowire port. This port can drive devices conforming to Motorola's SPI device interface and National Semiconductor's Microwire™ device interface. The engineering bulletin also presents software drivers written in the Neuron C programming language that display decimal numbers from binary data.

Schematic

The MC14489 can be connected to the Neuron 3150® or 3120xx® Chips as indicated in the following schematic. The Neuron Chip's Neurowire port in master mode uses pin `IO_8` as the clock pin, and `IO_9` as the serial output data pin. In this case, pin `IO_2` is used as the enable pin for the MC14489 display controller, but any of pins `IO_0` through `IO_7` could have been chosen, with the appropriate modification to the driver software.

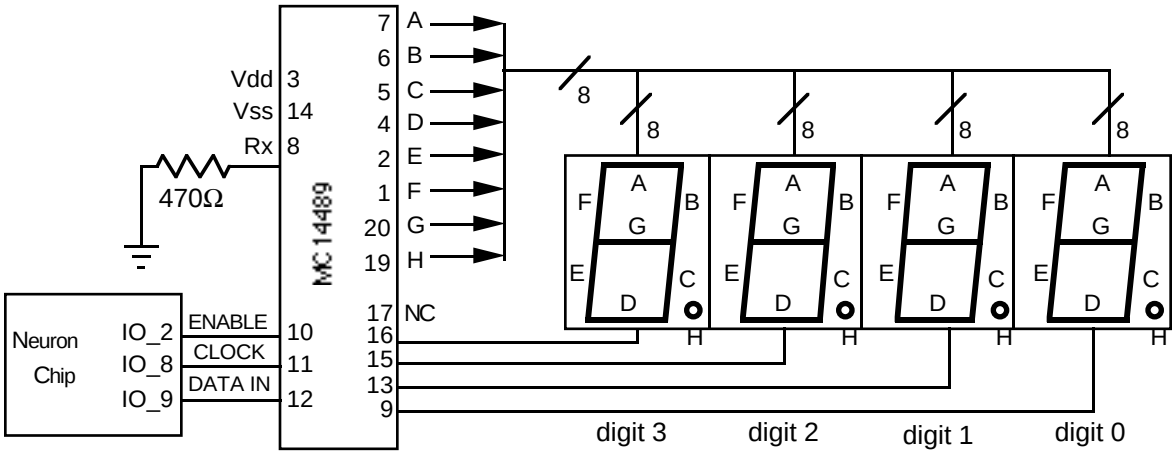


Figure 1. Seven-segment LEDs controlled by the Neuron Chip

Echelon Part Number 005-0014-01 REV C

The MC14489 is connected to four common-cathode seven-segment LED display devices. These are available from most manufacturers of opto-electronic devices, such as General Instruments, Hewlett-Packard, Industrial Electronic Engineers and William J. Purdy. The value of the current-limiting resistor connected between the Rx pin and ground depends on the application. If more than five digits are desired, several MC14489 devices may be connected in a cascade configuration, with the serial data being shifted out of one device into the next. See the Motorola MC14489 data sheet for more details. Other SPI or Microwire devices may be connected at the same time to the Neuron Chip's Neurowire port, provided each device has its own Enable pin.

Programming

The MC14489 has two write-only device registers controlled by the software on the Neuron Chip. The eight-bit configuration register shown in figure 2 contains bits that affect the decoding of the data in the 24-bit display register.

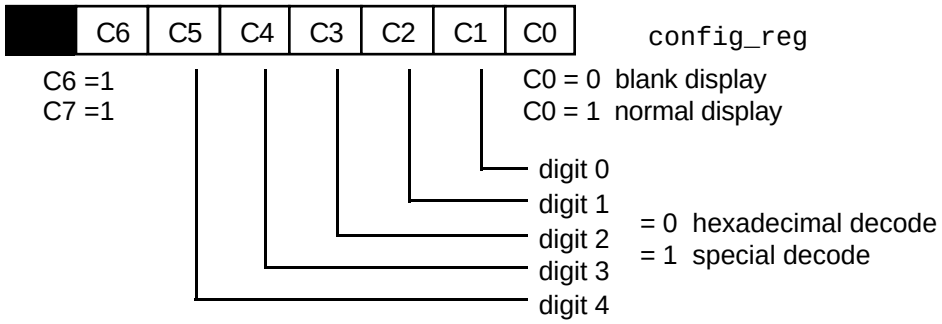
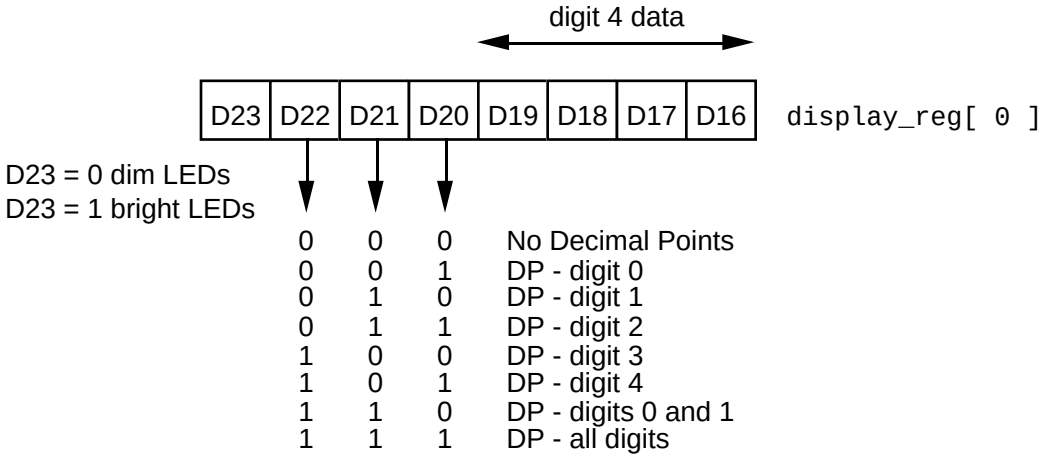


Figure 2. MC 14489 configuration register

The display register contains bits that define the display pattern of the LED digits and the decimal points as defined in figure 3.



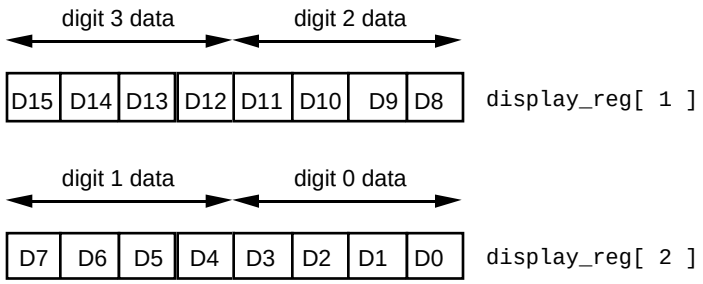


Figure 3. MC14489 display register

For the purposes of this application, there are two modes in which the data can be displayed. In hexadecimal mode, the four bits of data displayed in each digit are decoded as the hexadecimal digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. In special mode, certain other characters such as a space and a minus sign can be displayed. The table in figure 4 shows the patterns displayed in the two modes for all possible values of the data in the digit.

digit data	hex decode	special decode
0	0	
1	1	-
2	2	H
3	3	h
4	4	J
5	5	L
6	6	n
7	7	o
8	8	P
9	9	r
A	A	U
B	b	u
C	C	y
D	d	–
E	E	=
F	F	°

Figure 4. MC14489 display decoding

The software uses the Neurowire (SPI) function of the Neuron C programming language to write data to these two registers. When the application program issues an `io_out()` function call to the Neurowire device, the system software activates the chip select pin (in this case `IO_2`), and then clocks the data out on pin `IO_9`, using pin `IO_8` for the clock. The default rate of this serial data clock is 20kbps when the Neuron Chip input clock is 10MHz. When eight bits of data are clocked into the MC14489, these data bits are written to the configuration register. When 24 bits of

data are clocked in, they are written to the display register. The software drivers presented in listings 1 and 2 always write both the configuration and the display register sequentially.

Software

Two listings are presented here. The first listing is for a simple decimal display function for positive numbers. Leading zeroes are not suppressed, so that, for example, the number 123 is displayed as 0123. There is also no error check for numbers that are out of range of the display. The built-in Neuron C library function `bin2bcd()` performs the actual data conversion.

The function `DspDisplayNumber(number, dpDigit)` displays unsigned decimal numbers with a decimal point to the right of the specified digit. Note that the special values `NO_DP` and `ALL_DPS` may be used as the digit number to illuminate none or all, respectively, of the decimal points.

Listing 1. Seven-segment display driver for positive numbers

```
///////////////// SEVEN SEGMENT DISPLAY DRIVER  /////////////////

// This Neuron C #include file contains code to drive the Motorola MC14489
// seven-segment display controller chip, interfaced to the Neuron Chip
// using Neurowire master output mode.

// Pin IO_8 is the Neurowire clock, pin IO_9 is the serial output data
// Pin IO_2 is the chip select (may be modified).

// The function display_number( ) displays unsigned numbers
// with leading zeroes

////////////////////// DECLARATIONS ////////////////////////////////

IO_8 neurowire master select(IO_2) ioSevenSeg;
IO_2 output bit io7SegSelect = 1;
#pragma ignore_notused io7SegSelect

struct bcd dspDataReg;           // 24 bits for 7-seg display reg
unsigned dspCfgReg;             //  8 bits for 7-seg  config reg

////////////////////////// DISPLAY DECIMAL NUMBER //////////////////

void DspDisplayNumber(unsigned long number, int dpDigit) {
    dsoCfgReg = 0xc1;                        // Decimal decode all digits
    bin2bcd( number, &dspDataReg );          // Convert binary to decimal
    dspDataReg.d1 = 0x80 + dpDigit + 1;      // Set MS nibble for dec. pt.
    io_out (ioSevenSeg, &dspCfgReg, 8 );     // Update device registers
    io_out( ioSevenSeg, &dspDataReg, 24 );
}

#define NO_DP   -1
#define ALL_DPS  6
```

The second listing is for a more user-friendly decimal display. It handles positive, negative and out-of-range numbers, it suppresses leading zeroes where appropriate, and it allows the caller to specify the rightmost digit position. There is also a function to display strings consisting of the characters shown in Fig. 4, and a function to display temperatures values.

Before using this code, make sure that the number of digits in your display is specified correctly by the `NUM_DIGITS` parameter. For the display in the Gizmo 2 Multifunction I/O device, `NUM_DIGITS` should be set to 4. For the display in the Gizmo 3, set `NUM_DIGITS` to 5.

The software functions are divided into three groups; low-level functions, display image update functions, and high-level functions.

Low-level functions

The first group of functions provides low-level access to the display controller chip.

The function `DspClearImage()` clears a RAM copy of the configuration and display registers (the variables `dspCfgReg` and `dspDataReg)` to a state that displays all blank characters. It does this by setting all digits to special decode mode, and writing the data for the blank character to all digits.

The function `DspUpdateDisplay()` uses the Neurowire I/O device to write the contents of the RAM copy of the configuration and display registers to the actual MC14489 device registers. The Neurowire device is full-duplex, so that the `io_out()` operation which updates the hardware registers in the MC14889 also causes data to be shifted in from pin `IO_10` and stored in the RAM variables. Therefore a local copy of these variables is used for the `io_out()` operation, so that `DspUpdateDisplay()` may be called repeatedly without having to refresh the RAM copy of the variables.

Display image update functions

The second group of functions are routines that write into the RAM copy of the configuration and display registers. They do not update the hardware device registers.

The function `DspInsertData(int digitNumber, int data, boolean isNumeric)` writes the data nibble into the specified digit position in the RAM copy of the display register. Digits are numbered from right to left, with digit 0 being the rightmost (units) digit. If `isNumeric` is `TRUE`, numeric display decode is enabled for that digit.

The function `DspInsertDecimal(int digitNumber, int number)` writes the specified decimal (0 - 9) into the specified digit position in the RAM copy of the display register.

The function `DspInsertDecimal2(int rightDigit, unsigned number)` writes a two-digit decimal number (00 - 99) into the specified digit positions in the RAM copy of the display register.

The function `DspInsertDP(int digitNumber)` writes the appropriate bit in the RAM copy of the display register to illuminate the decimal point to the right of the specified digit.

The function `DspInsertMinus(iny digitNumber)` writes the appropriate values in the RAM copy of the display register to illuminate the minus sign (segment G) in the specified digit.

The function `DspInsertChar(int digitNumber, char ch)` writes the data value for an ASCII character in the RAM copy of the display register. If the ASCII character does not appear in Fig. 4, nothing is written, leaving the display blank for that digit.

The function `DspInsertNumber(long number, int dpDigit, int rightDigit)` updates the RAM copy of the display register to display a signed decimal number. The function illuminates a decimal point to the right of the specified digit. If the `dpDigit` parameter is `NO_DP`, no decimal point is illuminated. If the `dpDigit` parameter is `ALL_DPS`, all decimal points will be illuminated. The caller also specifies the right-most digit position of the displayed number. Display data to the right of this position are unchanged. If the number to be displayed does not fit in the specified field, all digits are set to the minus character.

High-level functions

The third group of functions forms complete display images and updates the display hardware.

The function `DspDisplayBlanks()` clears the display.

The function `DspDisplayString(const char * pString, int dpDigit)` causes the first 4 or 5 ASCII characters in the specified string to be displayed. Four characters are displayed for a Gizmo 2, and five characters are displayed for a Gizmo 3. The letters available in upper case are "A, C, E, F, H, I, J, L, O, P, S, U, Y, Z", and in lower case "b, c, d, h, l, n, o, r, u, y". Digits 0-9 are displayed, as well as the space ' ', degree '°', minus '-', and equals '=' special characters. If other letters, or more elegant letters are desired, an alphanumeric display should be used instead of a seven-segment display.

The function `DspDisplayNumber(long number, int dpDigit, int rightDigit)` displays positive or negative decimal numbers with a decimal point to the right of the specified digit, with suppression of leading zeroes. The special values `NO_DP` and `ALL_DPS` may be used as the decimal point digit number to illuminate none or all, respectively, of the decimal points. The parameter `rightDigit` indicates the digit position for the least significant digit of the displayed number. Numbers that do not fit into the specified field are displayed as all minus characters '-----'.

Table 1 shows some examples of the display produced by different input values with a four-digit display.

```
DspDisplayNumber(123, 0 ,0)          =>      1 2 3.
DspDisplayNumber(123, 1 ,0)          =>      1 2.3
DspDisplayNumber(123, 2 ,0)          =>      1.2 3
DspDisplayNumber(123, 3 ,0)          =>    0.1 2 3
DspDisplayNumber(123, NO_DP ,0)      =>      1 2 3
DspDisplayNumber(123, ALL_DPS ,0)    =>     .1.2.3.

DspDisplayNumber(-123, 0 ,0)         =>    - 1 2 3.
DspDisplayNumber(-123, 1 ,0)         =>    - 1 2.3
DspDisplayNumber(-123, 2 ,0)         =>    - 1.2 3
DspDisplayNumber(-123, 3 ,0)         =>    -.1 2 3
DspDisplayNumber(-123, NO_DP ,0)     =>    - 1 2 3
DspDisplayNumber(-123, ALL_DPS ,0)   =>    -.1.2.3.
```

Table 1. Display produced by various input parameters to the DspDisplayNumber function.

The function `DspDisplayTemp(SNVT_temp temp, boolean dspFahrenheit)` displays temperature values in either Celsius or Fahrenheit, with one decimal place. For more details on the Standard Network Variable Type `SNVT_temp`, see the *SNVT Master List and Programmer's Guide.* As an example, `DspDisplayTemp(2940, FALSE)` displays '`20.0C`', and `DspDisplayTemp(2940, TRUE)` displays '`68.0F`'.

Listing 2. General-purpose seven-segment display driver

```
// DISPLAY.H -- Display handler for Gizmo 2 and Gizmo 3 LED displays
//
// Date last modified: 29-Dec-94
//
// Driver for the Motorola MC14489 seven-segment display controller chip
// on the Gizmo 2 and Gizmo 3
//
////////////////////////////// Include Files ///////////////////////////////

#include <stdlib.h>

////////////////////////////// I/O Objects ////////////////////////////////

IO_8 neurowire master select(IO_2) ioSevenSeg;
IO_2 output bit io7SegSelect = 1;    // Initially unselected
#pragma ignore_notused io7SegSelect

////////////////////////////// Constants ///////////////////////////////

//  The constant NUM_DIGITS depends on the display.
//  Gizmo 2 has 4 digits, Gizmo 3 has 5 digits.
//  The rightmost digit is numbered 0.   The default device is Gizmo 3

//  #define NUM_DIGITS 4    // uncomment this line for Gizmo 2

#ifndef NUM_DIGITS
#define NUM_DIGITS  5       // for Gizmo 3
#endif

//  The following constants may be used as the dpDigit argument for
//  DspDisplayNumber( ) and DspDisplayString( )

#define NO_DP   -1    // display number without  decimal point
#define ALL_DPS -2    // display number with all decimal points

////////////////////////////// Global Variables ////////////////////////////

unsigned dspCfgReg;         //  8 bits for 7-seg  config reg
unsigned dspDataReg[3];     // 24 bits for 7-seg display reg

////////////////////////////// Functions ///////////////////////////////

//  DspUpdateDisplay( ) -- update device hardware registers

void DspUpdateDisplay(void) {
    static unsigned cfgRegCopy;
    static unsigned dataRegCopy[3];

    cfgRegCopy = dspCfgReg;
    memcpy(dataRegCopy, dspDataReg, 3);     // copy images
    io_out(ioSevenSeg, &cfgRegCopy, 8);     // shift out
    io_out(ioSevenSeg, dataRegCopy, 24);
}
```

```
////////////////////////////////////////////////////////////////////////

//  DspClearImage( ) -- clear image of device registers to all blanks

void DspClearImage(void) {
    dspDataReg[0] = 0x80;   // max brightness
    dspDataReg[1] = 0;      // blanks on all digits
    dspDataReg[2] = 0;
    dspCfgReg = 0xFF;       // special decode on banks 1-5, normal mode
}

////////////////////////////////////////////////////////////////////////

//  DspInsertData( ) -- insert a nibble (0-F) in specified digit

void DspInsertData(int digitNumber, int data, boolean isNumeric) {
    dspDataReg[2 - digitNumber / 2] |=              // insert nibble
        (digitNumber & 1) ? (data << 4) : data;
    if (isNumeric)
        dspCfgReg &= ~(1 << (digitNumber + 1));     // set numeric decode
}

////////////////////////////////////////////////////////////////////////

//  DspInsertDecimal( ) -- insert a decimal number (0-9) in specified digit

void DspInsertDecimal(int digitNumber, int number) {
    DspInsertData(digitNumber, number, TRUE);
}

////////////////////////////////////////////////////////////////////

//  DspInsertDecimal2( ) - insert a two-digit decimal number (00-99)

void DspInsertDecimal2(int rightDigit, unsigned number) {
    DspInsertDecimal(rightDigit++, number % 10);
    DspInsertDecimal(rightDigit, number / 10);
}

#pragma ignore_notused DspInsertDecimal2

////////////////////////////////////////////////////////////////////////

//  DspInsertDP( ) -- insert a decimal point to right of specified digit

void DspInsertDP(int digitNumber) {
    dspDataReg[0] |= (digitNumber + 1) << 4;        // set DP bit
}

////////////////////////////////////////////////////////////////////

//  DspInsertMinus( ) -- insert a minus sign in specified digit

void DspInsertMinus(int digitNumber) {
    DspInsertData(digitNumber, 0xD, FALSE);
}
```

```
////////////////////////////////////////////////////////////

//  DspInsertChar( ) -- insert a character from the Gizmo character set

void DspInsertChar(int digitNumber, char ch) {
    // Table of ASCII characters that may be displayed in 7 segments

    typedef struct {
         unsigned      charCode  : 4;
         unsigned      isNumeric : 1;
         char          asciiChar;
    } charTable;       // Gizmo character set table

    static const charTable CHAR_TABLE[ ] = {
    // Sorted in ASCII collating sequence.  '%' is displayed as a degree sign

        { 0x0, FALSE, ' ' }, { 0xF, FALSE, '%' }, { 0xD, FALSE, '-' },
        { 0x0, TRUE,  '0' }, { 0x1, TRUE,  '1' }, { 0x2, TRUE,  '2' },
        { 0x3, TRUE,  '3' }, { 0x4, TRUE,  '4' }, { 0x5, TRUE,  '5' },
        { 0x6, TRUE,  '6' }, { 0x7, TRUE,  '7' }, { 0x8, TRUE,  '8' },
        { 0x9, TRUE,  '9' }, { 0xE, FALSE, '=' },
        { 0xA, TRUE,  'A' }, { 0xC, TRUE,  'C' }, { 0xE, TRUE,  'E' },
        { 0xF, TRUE,  'F' }, { 0x2, FALSE, 'H' }, { 0x1, TRUE,  'I' },
        { 0x4, FALSE, 'J' }, { 0x5, FALSE, 'L' }, { 0x0, TRUE,  'O' },
        { 0x8, FALSE, 'P' }, { 0x5, TRUE,  'S' }, { 0xA, FALSE, 'U' },
        { 0xC, FALSE, 'Y' }, { 0x2, TRUE,  'Z' }, { 0xB, TRUE,  'b' },
        { 0x1, FALSE, 'c' }, { 0xD, TRUE,  'd' }, { 0x3, FALSE, 'h' },
        { 0x1, TRUE,  'l' }, { 0x6, FALSE, 'n' }, { 0x7, FALSE, 'o' },
        { 0x9, FALSE, 'r' }, { 0xB, FALSE, 'u' }, { 0xC, FALSE, 'y' }
    };
    const charTable *pLow, *pHigh, *pHalf;
    char tableChar;

    pLow = CHAR_TABLE;              // set up for binary search
    pHigh = CHAR_TABLE + sizeof(CHAR_TABLE) / sizeof(charTable);
    while (pHigh > pLow + 1) {
        pHalf = pLow + (pHigh - pLow) / 2;  // probe the mid-point
        tableChar = pHalf->asciiChar;
        if (ch == tableChar) {
            DspInsertData(digitNumber, pHalf->charCode, pHalf->isNumeric);
            return;
        }
        if (ch > tableChar) pLow = pHalf;
        else pHigh = pHalf;
    }
}

////////////////////////////////////////////////////////////////////////

// DspInsertNumber( ) -- insert signed decimal number

void DspInsertNumber(long number, int dpDigit, int rightDigit) {
    static const struct rangeTable {      // min and max displayable numbers
        long   minNum;
        long   maxNum;
    } RANGE_TABLE[] = {
```

```
        {  0, 9 },          // 1 digit
        { -9, 99 },         // 2 digits
        { -99, 999 },       // 3 digits
        { -999, 9999 },     // 4 digits
        { -9999, 32767 }    // 5 digits
    };
    const struct rangeTable *pRange;

    DspInsertDP(dpDigit);       // display decimal point

    pRange = &RANGE_TABLE[NUM_DIGITS - 1 - rightDigit];
    if (number > pRange->maxNum || number < pRange->minNum) {
        while (rightDigit < NUM_DIGITS)    // display "-----" for overrange
            DspInsertMinus(rightDigit++);
        return;
    }

    dpDigit = max(dpDigit, rightDigit);
    if (number < 0) {
        DspInsertMinus(NUM_DIGITS - 1);   // leading minus sign
        number = - number;      // get absolute value
        if (dpDigit == NUM_DIGITS - 1) dpDigit--;    // allow -.xxxx format
    }

    while (number || rightDigit <= dpDigit) {
                // convert binary to decimal with leading zero suppress
        DspInsertDecimal(rightDigit++, (int)(number % 10));
        number /= 10;
    }
}

////////////////////////////////////////////////////////////////////////

//  DspDisplayNumber( ) -- display signed decimal number

void DspDisplayNumber(long number, int dpDigit, int rightDigit) {
    DspClearImage();       // clear image of display registers
    DspInsertNumber(number, dpDigit, rightDigit);   // convert number
    DspUpdateDisplay() ;   // update hardware
}

#pragma ignore_notused DspDisplayNumber

////////////////////////////////////////////////////////////

//  DspDisplayTemp( ) -- Display a temperature in Fahrenheit or Celsius
//                       For Gizmo 2, range is  -9.9 to  99.9 degrees
//                       For Gizmo 3, range is -99.9 to 999.9 degrees

void DspDisplayTemp(SNVT_temp temp, boolean dspFahrenheit) {
    long value;
```

```
    value = temp - 2740;       // tenths of degrees C
    if (dspFahrenheit)         // display in tenths of degrees F
        value = value * 9 / 5 + 320;
    DspClearImage();
    DspInsertNumber(value, 2, 1);      // xxx.xX
    DspInsertData(0, dspFahrenheit ? 0xF : 0xC, TRUE);   // display F or C
    DspUpdateDisplay();
}

#pragma ignore_notused DspDisplayTemp

////////////////////////////////////////////////////////////

//  DspDisplayString( ) -- display first NUM_DIGITS characters of a string

void DspDisplayString(const char *pString, int dpDigit) {
    int digitNumber;

    DspClearImage();        // all non-displayable chars show as blank
    for (digitNumber = NUM_DIGITS - 1; digitNumber >= 0; digitNumber--)
        DspInsertChar(digitNumber, *pString++);

    DspInsertDP(dpDigit);  // display decimal point
    DspUpdateDisplay();    // update hardware
}

#pragma ignore_notused DspDisplayString

////////////////////////////////////////////////////////////////////////

//  DspDisplayBlanks( ) -- display all blanks

void DspDisplayBlanks(void) {
    DspClearImage();
    DspUpdateDisplay();
}

#pragma ignore_notused DspDisplayBlanks
```

Analog-to-Digital Conversion with the Neuron® Chip

January 1995 **LONWORKS® Engineering Bulletin**

This document describes some of the more popular analog to digital (A/D) conversion schemes available for use with the Neuron Chip. Included is the description and an example of the Neuron Chip's built-in Dualslope I/O object. The intent is not to provide an exhaustive survey of all conversion techniques but to provide an application-oriented reference for the user of the Neuron Chip.

For more specific information on A/D conversion techniques discussed in this application note, and on A/D conversion in general, refer to the list of references at the end of this document.

This document is divided into two major sections. The first section deals with traditional A/D conversion techniques and the specific attributes of each. Other aspects of analog data acquisition such as sample-and-hold circuits, analog multiplexers, and voltage references are also discussed in order to provide a foundation for what follows.

The second section of this engineering bulletin addresses the previously discussed A/D schemes in light of the capabilities and features of the Neuron Chip. Network performance issues relating to A/D conversion in a distributed control application are also covered, in order to make the developer aware of some of the common pitfalls associated with such systems.

The A/D design examples discussed include some off-the-shelf ICs in addition to circuits using more simplified external logic along with the Neuron Chip's I/O function blocks. As an aid to the designer, a sampling of different off-the-shelf A/D conversion ICs along with their features and attributes is included.

The Neuron Chip provides several I/O models which greatly simplify the task of interfacing to off-the-shelf A/D chips. These I/O model include the neurowire, bitshift, and I^2C I/O models. Refer to the *Neuron Chip Data Book* for more information on these I/O models

Background

Figure 1 shows the typical functional blocks found in most analog data acquisition designs. Different functional blocks may have varying complexities depending on the application. For example, processing low-level analog signals at high speed (conversion rate), might require more robust signal conditioning and high-speed conversion circuitry.

The majority of this engineering bulletin deals with the internals of the A/D block shown in figure 1. Some attention is given to the other functional blocks only to

Echelon Part Number 005-0019-01 REV C

provide the reader with a working knowledge by which to support the A/D block design.

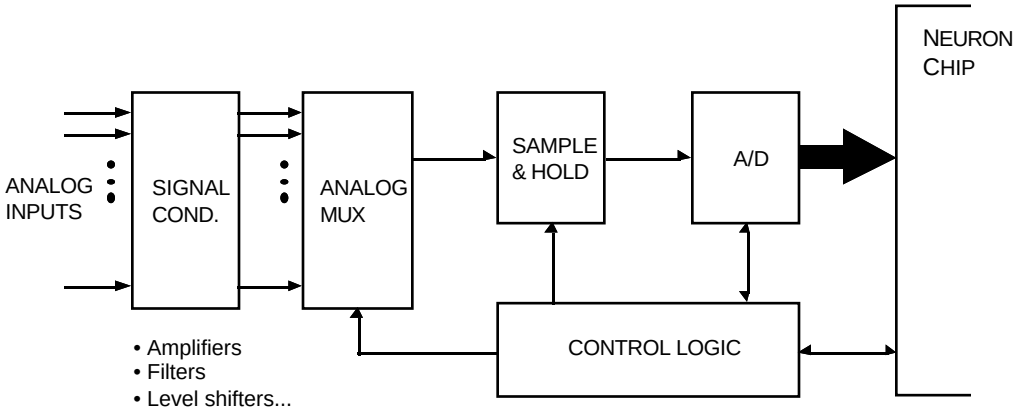


Figure 1. A typical analog data acquisition sub-system usually consists of several of the functional blocks shown.

Analog-to-digital conversion techniques can be divided into two distinct classes: *direct* and *indirect*.

In the *direct* conversion technique, the analog signal is continuously compared to the output of a D/A converter. The D/A converter's input is changed based on the results of the comparison, until the D/A converter's output matches the analog input signal. The input to the D/A converter is then the desired digital output.

Figure 2 illustrates the basic building blocks of a direct converter. The various direct conversion schemes differ in the way the input of the D/A converter is changed, namely the control process that uses the comparator output to generate the next digital input for the D/A converter.

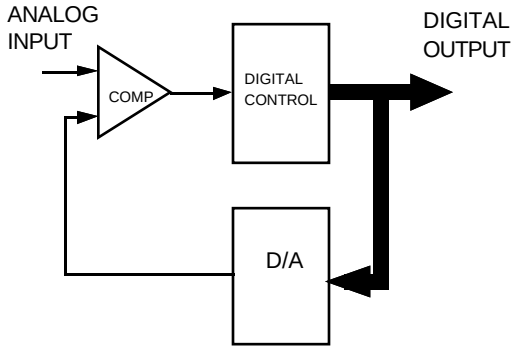


Figure 2. Basic building blocks for direct conversion.

With *indirect* conversion, the analog signal is converted into a time or frequency domain signal that is then measured by the digital logic and converted to a binary value.

The following A/D conversion schemes are discussed in the following sections:

- Dual-Slope integrating converter
- Counting converter
- Successive approximation converter
- Voltage-to-frequency converter

The actual implementation of these techniques can vary significantly from one system to another depending on the requirements of the system. Such factors as conversion speed, accuracy, resolution, linearity and cost all affect the overall design of a particular A/D conversion sub-system.

Dual-Slope Integrating Converters

A very popular form of the indirect converter, the dual-slope integrating converter incorporates an analog integrator. The input analog signal is first integrated over a fixed period of time, T1. The integrator's input is then switched to a reference voltage, V_{ref1}, where it is allowed to integrate down while a digital counter is incrementing. The counter is stopped when the integrator's output reaches a preset voltage, V_{ref2} (T2 period). The contents of the counter then represent the converted digital value. Figure 3 shows the basic building blocks of a dual-slope integrating converter.

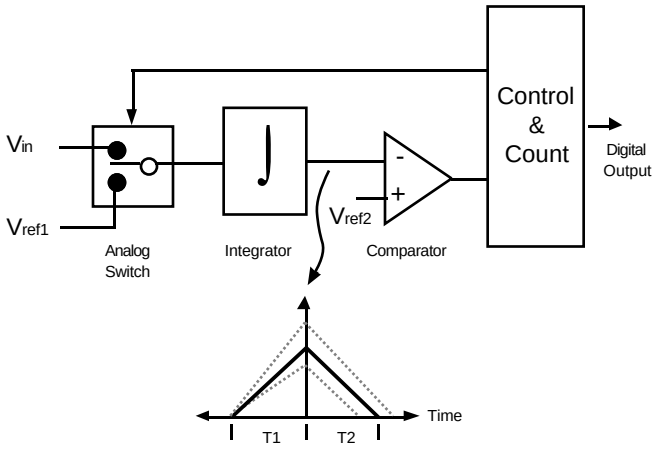


Figure 3. Basic building blocks for integrating converter

The dual-slope integrating converter has several important advantages. Because of the complementary nature of the two opposing slopes, the conversion accuracy is independent of the accuracies of both the clock frequency and the integrator capacitor. Also, due to the inherent nature of the built-in integrator, the output digital value represents an averaging of the analog input signal over the integration period. This makes the converter highly immune to input noise.

A disadvantage of the dual-slope integration is that the required integration period places a restriction on the maximum conversion speed. Dual-slope integrating converters are thus generally used for slower application such as environmental monitoring of temperature or humidity.

The Neuron Chip includes a dual-slope integrating converter as one of its built-in I/O objects called dualslope input. The Neuron Chip essentially replaces the control and counter block shown in figure 3. An example of the use of the dualslope input object will be presented later.

Counting Converters

This is the simplest form of the direct conversion technique. The implementation flexibility of this scheme allows for wide design variations, from an entirely hardware-based design to a combined hardware-software implementation. This allows the use of already available resources within the Neuron Chipin order to reduce overall cost and complexity.

There are two basic types of counting converters: stair-step and tracking. In the stair-step scheme, the count and control functional block shown in figure 2 is an N-bit up-counter whose counting operation is controlled by the comparator output.

At the start of the conversion cycle, the counter simply starts counting up from zero until the output of the D/A converter equals the input analog signal. At that point, the counter stops incrementing. The N-bit binary number in the counter represents the digital output corresponding to the analog input signal. The counter is then reset and the entire sequence is repeated for another conversion.

The conversion speed of this technique is relatively low due to the inherently slow counting process required. For an N-bit converter, the worst-case (full-scale) conversion time is 2^N clock cycles.

The counting operation can also be performed by software. Although logically equivalent to the all-hardware implementation, this scheme allows the use of already available resources (microprocessor or microcontroller) in order to reduce the amount of external logic. The use of software also permits design modification (e.g., changes in the conversion algorithm) at a much lower cost.

The basic structure for a software-oriented counting-type A/D converter is shown in figure 4.

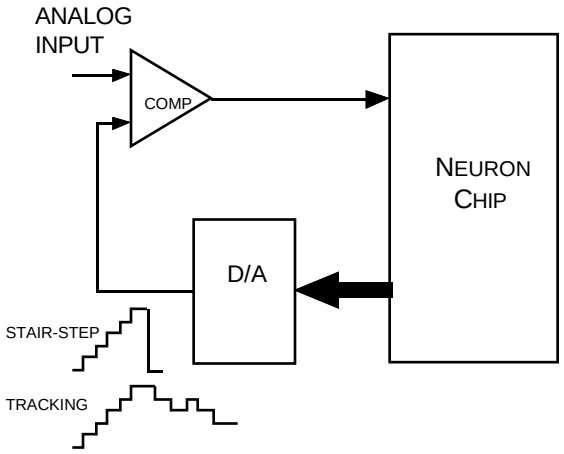


Figure 4. Software-driven counting-type A/D converter. The processor replaces the digital control block shown in Figure. 2.

The average conversion speed of the stair-step counter converter may be increased by using an up-down counter. In a tracking or servo-type converter, the counter is allowed to count in both directions and track the input analog voltage. This translates to shorter conversion time for small changes in the analog input. For large changes, however, the conversion time approaches that of the original stair-step converter. The tracking counter converter can also be implemented in software (figure 3) by modifying the software to behave accordingly.

Counting converters (stair-step or tracking) are suitable in applications where conversion speed is of small concern relative to the cost and simplicity of the design.

Successive Approximation Converters

This direct-type conversion scheme is similar to the counting type A/D converter in all but the count and control section, as shown in figure 5.

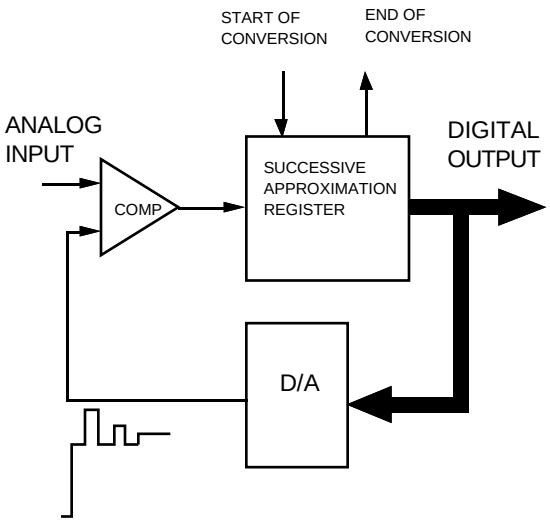


Figure 5. Basic building blocks for a successive approximation-type A/D converter.

The successive approximation register bypasses the long and slow process of incrementing the count sequentially. This is accomplished by toggling each of the N bits of the digital output one at a time, starting with the MSB, while monitoring the output of the comparator. With this technique, only N cycles are needed for a conversion.

The successive approximation converter has gained popularity due to its relatively balanced attributes (speed, complexity and cost). A number of off-the-shelf converters on the market today are of the successive approximation type. Some of the following design examples will use such ICs.

The circuit shown in figure 4 can also be used to implement a successive approximation converter. The only change is a software modification that implements the successive approximation algorithm instead of the counting algorithm in the microprocessor or microcontroller.

Voltage-to-Frequency Converters

This indirect conversion technique transforms the analog signal into the frequency domain. The frequency is then measured with a separate circuit or by a software routine. A block diagram is shown in figure 6.

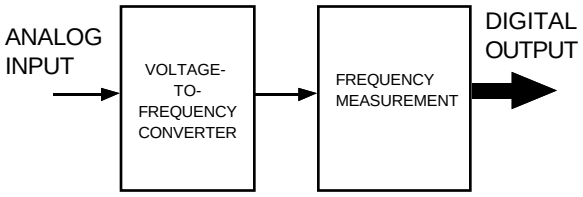


Figure 6. Voltage-to-frequency conversion.

The interface between the voltage-to-frequency (V/F) converter block and the frequency measurement block is a two-wire connection that carries a waveform with a frequency proportional to the amplitude of the input analog signal.

The V/F conversion technique is therefore useful in cases where the analog sensor is physically distanced from the rest of the system. In such a setup, the V/F converter is situated next to the source of the analog signal. A connecting pair of wires carries the frequency output to the main system for further processing. Note that the analog signal no longer has to travel for a long distance, thus avoiding signal degradation and interference.

A variation on the V/F method is the pulse-width conversion method. In this technique, the amplitude of the analog signal determines the output pulse width of a one-shot circuit. The pulse width, as with the frequency in the V/F case, is then measured and appropriately converted to a digital number.

As with any other conversion scheme, there is a balanced tradeoff between speed of conversion and resolution. A unique characteristic of the V/F conversion technique, however, is the high degree of control the designer has over this balance.

By changing the conversion scale of the V/F converter block shown in figure 6, for example, one can easily increase or decrease the resolution while decreasing or increasing the conversion speed.

The Neuron Chip, with its built-in timer/counter, easily lends itself to the above time measurements. The input objects than can be used for time/period measurements are ontime, period, and pulsecount, all having 16-bit resolution.

Other A/D Conversion Techniques

There are a number of other analog conversion techniques that we have not discussed here. This should by no means reflect on their relative importance in the A/D conversion arena. The scope of the conversion techniques discussed has been limited to the ones that directly relate to the Neuron Chip interface and capitalize on its features.

Conversion techniques such as flash, half-flash, and delta-sigma are addressed in many texts, some of which are noted in the reference section at the end of this engineering bulletin.

Sample/Track and Hold

In the previous conversion techniques, a basic assumption was implied about the rate of change of the input analog signal.

Generally, input signal changes of greater than 1/2 LSB during any of the conversion processes results in a misleading digital output with the exception of the V/F converter and dual-slope integrating converter which inherently averages the input signal over the integration period. Therefore, it is assumed that the input is

constant, or that the change is so small that it is not recognized by the A/D converter (e.g., outdoor light intensity or temperature).

For a time-varying analog signal, there is a need for circuitry that stabilizes the input to the A/D converter while the conversion process is taking place. The simplest approach is to use a low-pass filter that discriminates against higher frequencies (slew rates) that fall outside the converter's sampling rate. Another option would be to use a sample-and-hold (S/H) circuit.

A S/H block, as shown in figure 1, takes a snap-shot of the input signal when instructed to do so, and holds that value until the next sample.

A track-and-hold (T/H) circuit, on the other hand, spends most of its time tracking the input voltage and switches to the hold mode for brief periods of time when instructed to do so.

A wide selection of S/H and T/H ICs exist on the market that enable the designer to bypass the design of this functional block in all but the most demanding cases. Most S/H ICs are also capable of performing in the T/H mode.

Analog Multiplexers

Analog multiplexers, as shown in figure 1, permit the use of only a single A/D converter (and its associated S/H) for acquiring analog signals from multiple sources. They perform in the analog domain what digital multiplexers do in the digital domain.

An analog multiplexer has the advantage of fast switching time without the 'bounce' typically associated with mechanical relays. However, since semiconductor switches are used in these devices, the ON resistance might affect the measurement in some design situations.

As with the S/H case, a variety of off-the-shelf analog multiplexer ICs are available on the market to aid the designer.

Voltage References

In any A/D conversion design, there is a need for a point of reference against which the input signal must be compared.

When measuring physical quantities, the analog input is either an absolute signal (e.g., voltage or current) or it is in the form of a varying ratio. In the case of an absolute signal, the converter must be able to measure absolute levels of input.

For example, a thermocouple, which has a 0 to 0.25 volt output range, must be used with an A/D converter that can recognize an absolute voltage of 0.125 as the half-scale value. This is accomplished by providing an accurate reference to the A/D converter.

In the case of a varying ratio, absolute accuracy is not important. This is referred to as ratiometric measurement since the measured analog signal is dependent on a ratio provided by the transducer and not the voltage driving the transducer.

Both referencing schemes are used today as they each address a different need. An absolute referencing technique must be used where the analog sensing device produces an absolute voltage. Many such sensors exist today. Semiconductor transducers that measure such physical quantities as temperature, force and acceleration produce an absolute voltage relative to the measured quantity.

Use of the absolute referencing design, however, requires an accurate and stable reference voltage circuit which might be a disadvantage in terms of cost and complexity. The ratiometric approach eliminates this need while providing highly accurate measurements. Since both the measurement transducer and the A/D converter use the same reference voltage, voltage changes are observed by both and are therefore cancelled out.

The use of this scheme requires, however, a transducer capable of providing a variable impedance or ratio as an output, much like a center-tapped potentiometer.

Design Examples

The following is a series of actual design examples using the Neuron Chip. All circuits and their accompanying software have been tested to validate their functionality.

Several off-the-shelf ICs have been used with the designs in order to provide a good sampling of the available options. In addition, an A/D converter circuit discussed uses the internal functions of the Neuron Chip with minimal additional logic to accomplish its task.

Using an A/D converter IC with a parallel interface requires more I/O pins than a serial interface. In addition, the interface is further complicated by any converter with a resolution greater than eight bits. For this reason, the off-the-shelf A/D converter ICs used in the following examples are serial.

In the following circuits, as with any A/D circuit in general, careful attention must be given to potential noise and error sources such as ground loops, offset voltages and currents, and layout techniques. This is especially important with higher resolution converters.

Example 1

As mentioned before, the Neuron Chip supports a dual-slope integrating converter through its dualslope input object. One of the internal timer/counters of the Neuron Chip is used to control and measure the integration process. This permits a conversion resolution of up to 16 bits at a integration period of 13.11ms (10MHz input clock). Faster conversion rates can be attained at the expense of bit resolution.

Figure 7 is a typical converter circuit using the dualslope object.

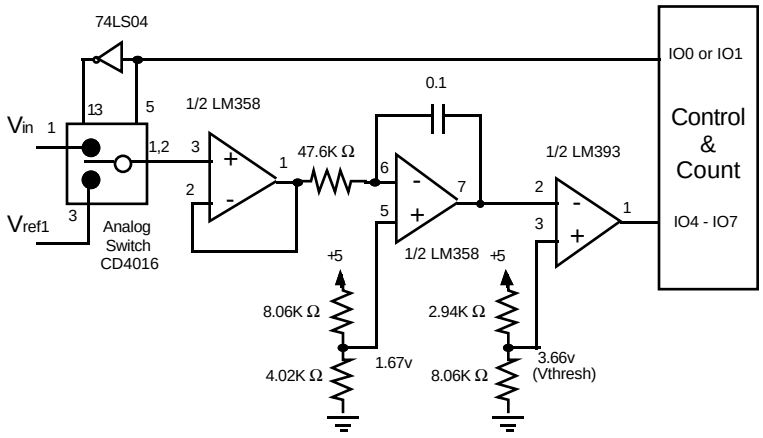


Figure 7. An example of a dual slope integrating A/D using the Neuron Chip's built in dualslope input object

For the dualslope input object the Neuron Chip's timer counter is set up as both an input and output device. The output signal typically is used to select the input source for the integrator: either the unknown analog input voltage, or a known reference voltage. The input signal to the timer counter is typically the output of a comparator which is comparing the integrator output to a reference voltage or zero voltage reference.

Declaring the dualslope input object will configure a timer counter to perform two important functions: count only when the input signal is active, and latch the count on the falling edge of the input signal. The maximum value for either of the integration periods can be controlled by the `clock (n)` keyword in this object declaration. See the *Neuron C Reference Guide* for the ranges provided by the various clock settings. The `invert` keyword may also be used here if the input signal is active low rather than active high.

The conversion process must be started by the application, through the use of the `io_in_request()` function. The end of the conversion is detected by the firmware by monitoring both the timer/counter control output and input signals. Therefore the application does not need to wait around during the conversion process, its completion is detected as an event through an `io_update_occurs()` `when` clause.

Figure 8 illustrates the relationships between the timer/counter input and output signals, and the integrator output. Three different input voltage integration slopes are shown for comparison.

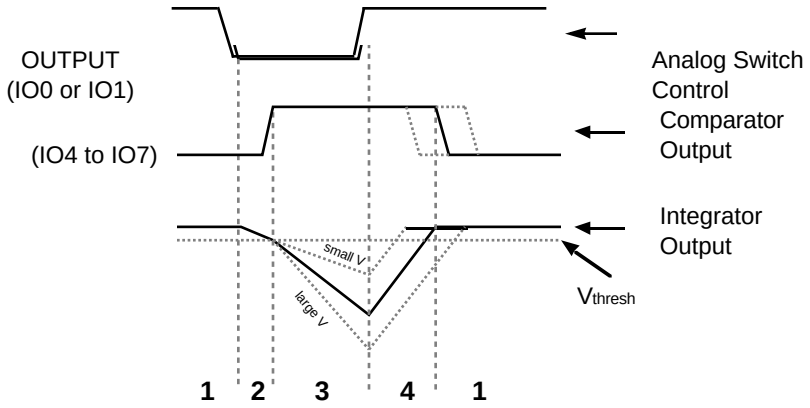


Figure 8. Various states of the dualslope converter

State 1 *is* the idle state. The timer/counter is not counting since the input signal from the comparator is low. The integrator input is selected to the reference voltage.

State 2 entered by calling the `io_in_request()` function for the dualslope input object. This step performs two tasks: it loads the timer/counter with the value (defined by the `control_value` passed by the `io_in_request()` call) that will produce the first integration period, which is typically a constant period, and, it activates the timer/counter output signal, which in turn selects the V_{in} (unknown analog input) signal as the input to the integrator. Note that until the integrator crosses the comparator threshold, which in turn activates the timer/counter input signal, the timer/counter is not counting. This ensures that the first integration period starts at this threshold point rather than at the start of State 2.

State 3 when the timer/counter input signal goes active and runs for the length of the first (fixed) integration period.

State 4 when the timer/counter reaches terminal count. At this point the timer/counter output toggles, selecting the V_{ref1} input as the input to the integrator. This also re-loads the timer/counter with the value defined by `control_value`. This means that the timer/counter will count from this value rather than zero, and implies that the second integration period will not be longer than the first. This is usually the case with this type of converter. State 4 ends when the integrator output crosses the comparator threshold in the other direction. The timer/counter input signal goes inactive, the current count is latched, and the timer/counter stops counting as State 1 is entered. At this point the timer/counter latch is read and stored into the `input_value` variable, and may also be retrieved through an `io_in()` function call. This value reflects the second integration period (State 4 period), which reflects the unknown V_{in} value as compared to the V_{ref1} value.

The value stored in `input_value` will always be biased by the two's complement of the value passed to the `io_in_request()`, and will lie between this value and the overflow value (0xFFFF).

In this example a 5MHz node will control a dual slope converter with a 24ms integration period. The conversion will occur every 500ms. The dedicated timer counter is used with a clock (0) setting which yields a 400ns resolution at 5MHz. IO_4 is the input signal to the timer/counter, and IO_1 is the output signal from the timer/counter. An `io_in_request()` count value of 60,000 will produce a 24ms integration period and provide a resolution of just under 16 bits.

Note that, in addition to decreasing the resolution, the integration period could also be reduced by reducing the overall voltage span of the integrator output. This can be accomplished by reducing the level of the comparator reference voltage. This however can lower the system signal-to-noise ratio and must be considered carefully.

An application timer will be set up to start the process periodically. Another application timer is used to detect an underrange condition. This condition can occur if the second integration period goes on forever due to an underrange input voltage.

```
// A/D conversion using the dualslope input
// of the Neuron Chip.
// Perform a measurement every 500ms

mtimer repeating go_time;
mtimer fail_time;
unsigned long raw_ds;
IO_4 input dualslope ded clock (0) dsad_1;

when(reset) {
   go_time = 500;               // Convert every 500ms.
}

//Conversion is started by this event

when(timer_expires(go_time)) {   io_in_request(dsad_1, 60000UL);
   fail_time = 80;      // Underrange if 80ms passes.
}
//End of conversion is detected by this event

when(io_update_occurs(dsad_1)) {
```

```
   raw_ds = input_value + 60000UL;
   fail_time = 0;    // Stop the timer.
   // raw_ds may now be scaled and used.
}
//Error condition is handled by this event
when(timer_expires(fail_time)) {
   // This is an error condition usually caused by an input voltage
   // that is too low - the comparator output never went active.
}
```

Example 2

Figure 9 illustrates a typical connection of the Neuron Chip to the Motorola MC145053 single-chip A/D converter IC for performing ratiometric measurements.

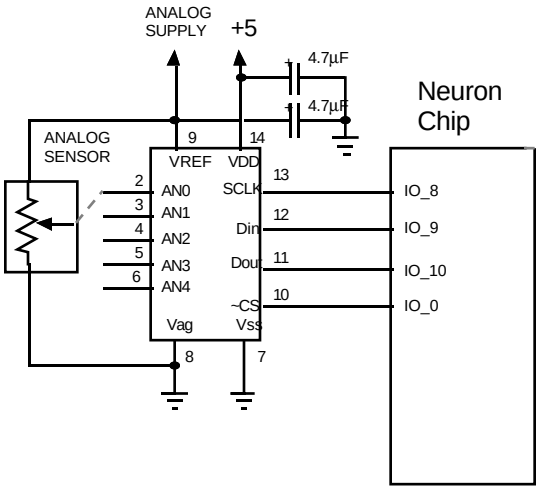


Figure 9. An example of an off-the-shelf A/D solution with the Neuron Chip using the Motorola MC145053.

This is an example of a 10-bit, multi-channel, full-duplex, serial A/D IC. The address information is clocked into the IC at the same time the converted digital output is clocked out.

Using the Neuron Chip's neurowire I/O object, also a full-duplex interface, we can simultaneously shift out the new address and shift in the converted data from the previous address.

An internal S/H function is provided by the MC145053. In addition, three test addresses are available for verification of the operation of the chip. Refer to the MC145053 data sheet for specific feature and timing information.

The Motorola MC145051 is identical to the MC145053 except that it provides 11 analog input channels in a 20-pin package.

The following is an example of typical Neuron C code for use with the circuit of figure 9.

```
// A/D conversion function for the
// Motorola MC145053 (and MC145051).
// The function analog_to_digital(analog_addr) returns a
// 10-bit number corresponding to the analog input with
// address of analog_addr from previous call.

//////////////////I/O declarations////////////////////
IO_8 neurowire master select (IO_1) ADC_IO;        //A/D chip interface
IO_1 output bit ADC_CS = 1;                        //running at 20Kbaud

//////////////////Function declaration///////////////
unsigned long analog_to_digital(unsigned long analog_addr); //function
                                                            //protocol
//////////////////Function definition////////////////
unsigned long analog_to_digital(unsigned long analog_addr) {
static unsigned long adc_info;      //raw data
unsigned long digital_out;          //returned digital data
   adc_info = analog_addr<<12;      //shift new addr out, and
   io_in(ADC_IO, &adc_info, 10);    //get converted data for previous address
   digital_out = ((adc_info>>6) & 0x3fc) | (adc_info & 0x03);
                                    //shuffle bits for correct format
   return digital_out;
}
```

The neurowire interface is operating at 20kbps (default). Alternatively, it could be configured for operation at 10 or 1kbps, if using an A/D converter IC that cannot work at such a high speed.

The select pin for the neurowire is chosen to be the IO_1 pin in this example and must be given an initial value of logic 1, as required by the IC.

The address that is clocked into the chip must reside in the four least significant bits of `analog_addr`. Also, the data clocked out of the IC, `adc_info`, needs to be shifted due to the fact that the MSB is shifted out first, in blocks of eight bits. For more

information on the operation of the neurowire I/O object refer to the *Neuron C Reference Guide*.

The conversion rate is dependent on the neurowire bit rate selected, the particular mode the MC145053 is used in, and the actual program itself. Using the above program (20kbps, 16-bit mode), a complete conversion can be performed in about 1ms.

Example 3

Figure 10 is an example of multiple A/D ICs interfacing to the Neuron Chip. The Linear Technology LTC1095 is a multi-channel, 10-bit successive approximation, half-duplex serial A/D IC with additional features such as S/H, voltage reference, single-ended or differential analog input modes, and unipolar or bipolar operation, all conveniently integrated into a single IC.

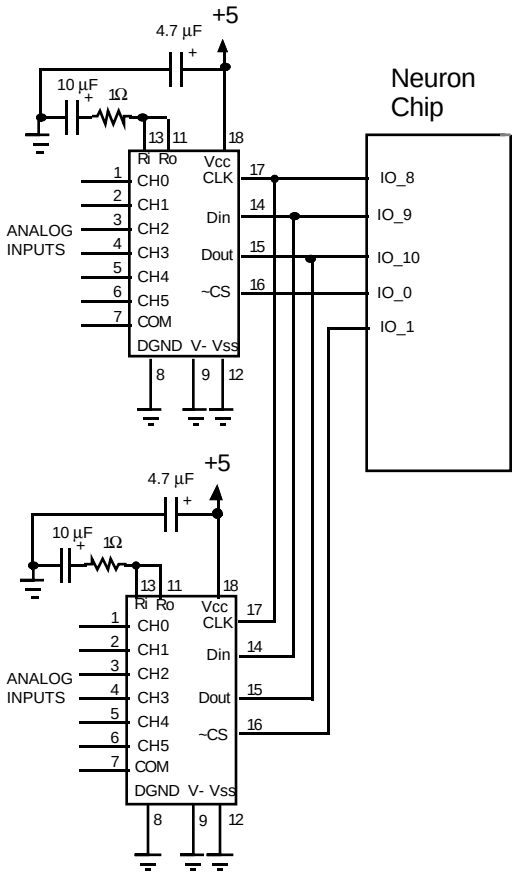


Figure 10. Multiple A/D interface using Linear Technology's LTC1095.

The circuit in figure 10 uses the absolute referencing technique discussed earlier. The internal, 5 volt, reference voltage generator of the IC provides an accurate

source for scaling the input analog voltages. In the unipolar mode of operation, this translates to an input range of 0 to 5 volts. In the bipolar mode, the range is increased to -5 to 5 volts. Both modes assume a direct connect of the LTC1095 reference output pin to the reference input pin.

The reference input of the LTC1095 can be connected to other reference voltages to accommodate any input signal range. The availability of an input reference pin permits ratiometric referencing.

In addition to the analog channel address, the LTC1095 requires other configuration information. The format of the 7-bit configuration word is as follows:

START	SGL/ ~DIFF	ODD/ ~SIGN	SEL 1	SEL 0	UNI	MSBF

Each of the above fields permits the user to configure the different features of the IC for different applications. The above configuration bits are defined as follows:

START - Initiates data transfer

SGL/~DIFF - Inputs referenced to COM, or differential pairs

ODD/~SIGN - Select the single-ended input, or polarity of differential pair

SEL0, SEL1 - Channel selection

UNI - Unipolar or bipolar operation

MSBF - MSB out on data output from the converter

Additional information on the timing and functionality is contained in the LTC1095 data sheet.

Since this IC uses a half-duplex communication protocol, the address clock-in and the data clock-out cycles do not occur simultaneously. This implies that the data can be accessed immediately after the address has been clocked into the IC.

The address clock-in and data clock-out are part of a single operation, meaning that the Chip Select (~CS) line must be held active during the full sequence.

A neurowire I/O function call normally activates the designated select line at the beginning of the operation and deactivates it at the end. Therefore, unlike the previous example, we cannot use the full-duplex neurowire function.

The bitshift I/O object is used in conjunction with the neurowire I/O object in this example. The address is clocked-out to the LTC1095 first using the bitshift function. This is immediately followed by a neurowire function call that clocks-in the converted data from the previous address in to the Neuron Chip. The neurowire is used, therefore, as a half-duplex interface.

The ~CS line is held active (low) throughout this entire sequence of events by overlaying the select line of the neurowire with a bit output object. This allows for separate control of that line by both the neurowire object and the application program.

The following is a Neuron C program that can be used with the circuit of figure 10.

```
// A/D conversion function
// for the Linear Technology LTC1095.
// The function analog_to_digital(analog_addr) returns a
// 10-bit number corresponding to the analog input with
// address of analog_addr, from the ADC designated by adc_num.

//////////////////I/O declarations////////////////////
IO_0 output bit ADC_CS_1 = 1;
IO_1 output bit ADC_CS_2 = 1;
IO_8 neurowire master select (IO_0) ADC_IO_1;   //A/D chip interface
IO_8 neurowire master select (IO_1) ADC_IO_2;   //A/D chip interface
IO_8 output bitshift numbits (8) clockedge (+)  //A/D chip interface
   ADC_group_control;

//////////////////Function declaration///////////////
unsigned long analog_to_digital(int adc_num, unsigned short analog_addr);

//////////////////Function definition////////////////
unsigned long analog_to_digital(int adc_num, unsigned short analog_addr) {
static unsigned long adc_data;         //raw data from A/D chip
unsigned long digital_out;             //returned digital data
static unsigned short addr;            //analog address/config
                                       //info to A/D
   addr = analog_addr;
   if (adc_num==1)io_out(ADC_CS_1, 0); //Activate first ADC
   if (adc_num==2)io_out(ADC_CS_2, 0); //Activate second ADC
   io_out(ADC_group_control, addr);    //send addr info to A/D
   if (adc_num==1) io_in(ADC_IO_1, &adc_data, 16); //get converted data
   if (adc_num==2) io_in(ADC_IO_2, &adc_data, 16); //get converted data
   digital_out = adc_data >> 6;        //right justify
   return digital_out;
}
```

The function `analog_to_digital(adc_num, analog_addr)` returns the converted data for the specified converter, `adc_num`, and the given configuration information, `analog_addr`.

One completed A/D conversion cycle using the above program requires 8 bitshift clock cycles (15kbps), 16 neurowire clock cycles (20kbps), and the latency associated with executing the code to perform the I/O functions. For a Neuron Chip running at 10MHz, the total conversion time is approximately 1.5 ms.

It is important to follow the rules outlined in the LTC1095 data sheet for implementing a ground plane. Proper bypassing, good layout techniques, and minimization of noise on the reference input are also important for reducing errors to a minimum.

Example 4

This example makes use of the available resources within the Neuron Chip to implement a stair-step counting type A/D converter suitable for relatively low conversion rates and resolution.

The design in figure 11 is perhaps the simplest of all the examples covered in this document in terms of hardware requirements external to the Neuron Chip. The counter and its associated logic needed for this type of conversion are implemented in software using Neuron C. To simplify the circuitry even further, the D/A converter is reduced to a simple passive integrator driven by the pulsewidth output object of the Neuron Chip.

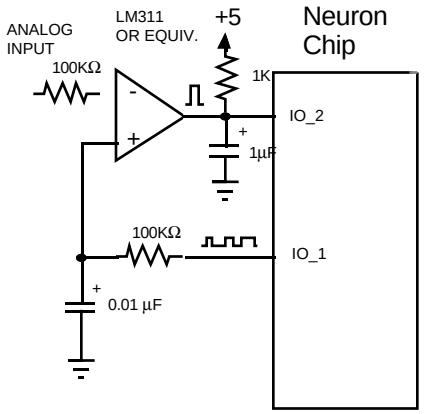


Figure 11. A counting type converter with some help from the Neuron Chip.

The pulsewidth output object of the Neuron Chip produces a continuous train of pulses. The frequency of the output is set during the declaration of the output object and is therefore set at compile-time to one of the eight available frequencies, from 152.6Hz to 19.53kHz. The duty cycle of the output, however, can be varied under program control. This provides a means by which to produce an analog voltage using a simple resistor-capacitor network.

The resolution of the pulsewidth output object can be set at either 8 or 16 bits. Therefore, it is theoretically possible to use the circuit of figure 11 for a converter with a resolution of up to 16 bits. There are, however, practical limitations in implementing a converter with such high resolution.

The highest output frequency at the 16-bit resolution is around 76Hz. At this frequency, the time constant of the RC network shown in figure 11 must be increased to accommodate the increased output period. This, in turn, must be accompanied by extra delays incorporated in the program to allow voltages to settle. This translates to longer conversion times that may be unacceptable in certain applications.

In addition, without careful attention to noise suppression and ground looping effects, such a converter circuit would produce erroneous results at higher resolutions.

Very good stability can be achieved considering the simplicity of the circuit in figure 11. This is due to the fact that the pulse-width output waveforms of the Neuron Chip are derived from a stable crystal time-base.

The resistor and capacitor values in figure 10 are chosen to provide a balance between circuit speed and ripple rejection, given a pulse-width output resolution of 8 bits. For large time constants, the Neuron Chip must wait for the capacitor to charge up to the new value for every update of the duty cycle. For small time constants, the voltage at the non-inverting input of the comparator would not be stable between duty cycle transitions. The values shown were used for a pulse-width frequency of 9.77kHz.

Careful attention must be given to proper grounding, layout, and component selections in order to reduce conversion error and increase stability. This is especially important when working with higher resolution converters. For example, an improperly biased comparator could result in missed codes at the beginning or at the end of the conversion range.

The design in figure 11 can be adapted for a tracking type counting converter. This is achieved by modifying the software to count in both directions. That is, the output of the comparator would be used by the Neuron C program to determine the direction of the next count change.

The program below is an example of typical code that can be used with the circuit of figure 11 to implement an 8-bit A/D converter.

```
// counting type ADC function using
// the Neuron Chip's internal resources
#include <control.h>

///////////////I/O Declarations//////////////////
IO_2 input bit match;                        //comparator input
IO_1 output pulsewidth clock (1) DAC_out = 0;//PWM output, 8-bit resolution

///////////////Function prototype/////////////
unsigned short analog_to_digital(void);

///////////////Function definition/////////////
unsigned short analog_to_digital(void) {
unsigned int digital_out =0;        //initialize to zero
   io_out(DAC_out, 0);              //reset output
   delay(4000);                     //wait for cap to discharge
   digital_out = 0;
   while (io_in(match)==0) {        //ramp up the DAC output
      io_out(DAC_out, digital_out);
      digital_out++;                //until it equals the input signal
      if (digital_out == 255) break;   //ceiling is 255
      watchdog_update();            //tickle watchdog timer
   }
   return digital_out;
}
```

Example 5

This example is a variation of example 4. The circuit used in the previous example (figure 11) can be used to implement another conversion technique by using a different software algorithm.

The slow conversion of the stair-step is improved in this example by using the successive approximation algorithm. The program below is an implementation for such an algorithm, written in Neuron C, that implements an 8-bit converter.

```
// Successive approximation type ADC function
// using Neuron Chip's internal resources
#include <control.h>

//////////////////////////Declarations//////////////////////
IO_2 input bit match;                              //comparator input
IO_1 output pulsewidth clock (1) DAC_out = 0;   //PWM output, 8-bit resolution

//////////////////////Function Prototype////////////////
unsigned short analog_to_digital(void);

//////////////////////Function Definition/////////////
unsigned short analog_to_digital(void) {
unsigned int digital_out;
unsigned int bit_num;
digital_out=0;
for (bit_num =0x80; bit_num!=0; bit_num >>=1) { //setup for //8 bit conv.
   io_out(DAC_out, digital_out+bit_num);        //increment output
                                                //by bit weight
   delay (1000);                                //wait for RC to settle
   if (io_in(match)==0) digital_out+=bit_num;   //set new output
   watchdog_update();
   }
   return digital_out;
}
```

Note that each conversion now takes a constant number of cycles equal to the number of bits desired in the digital output.

As previously mentioned in example 4, this program could also be implemented using the 16-bit feature of the pulsewidth output object. The same limitation, however, discussed in the previous example still applies.

As always, the circuit should be designed with close attention to biasing and grounding. Proper comparator offset is especially important as it can be the major source of error in this design.

Example 6

This example is an implementation of the voltage-to-frequency conversion technique, providing the simplest interface to the Neuron Chip compared to all previous examples.

The 555 timer is used to produce a square-wave whose frequency depends on an analog voltage. Figure 12 shows the circuit and its connection to the Neuron Chip.

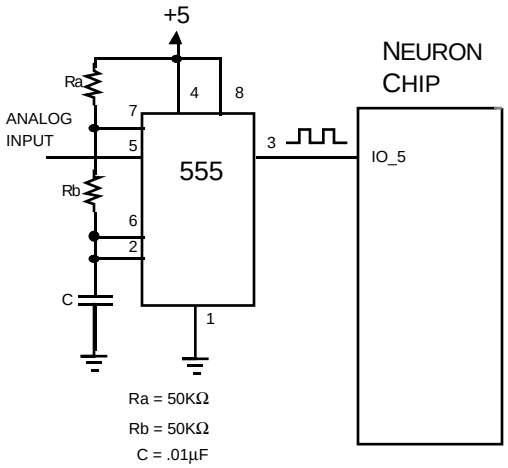


Figure 12. Voltage-to-frequency conversion using the 555 timer.

As shown in figure 12, only one pin is needed for connection of the 555 to the Neuron Chip. This design, therefore, is well suited for applications where Neuron Chip I/O resources are at a high demand.

In addition, in applications where the 555 timer is not situated near the Neuron Chip, this design provides for the minimum amount of interface wiring.

The 555 timer IC in figure 12 is wired as an oscillator with an output frequency that is dependent on the input voltage level on pin 5, in addition to Ra, Rb, and C.

The 555 used in the above configuration does not provide a linear relationship between voltage and frequency, desirable in applications requiring relative accuracy. Also, since the circuit is not a closed-loop system, it is more susceptible to output variations due to temperature, humidity, and long-term drift.

The circuit in figure 12 does however provide a simple and inexpensive solution for providing relative (non-absolute) conversions within a relatively small input voltage range.

For applications requiring higher precision and accuracy, an off-the-shelf voltage-to-frequency converter, such as LM131, may be used. This IC is specifically designed for A/D conversion applications and provides very good stability and a wide range of full scale frequencies in a small 8-pin DIP package.

Regardless of the circuit used to convert an analog voltage to frequency, the software in the Neuron Chip must be able to extract a meaningful value from the incoming waveform.

The Neuron Chip provides several ways for accomplishing this task. The ontime, period, and pulsecount input objects of the Neuron Chip may be used to convert an input square wave of a varying frequency into a digital number. All three methods provide a 16-bit resolution.

The ontime input object measures the duration of either the high or low (user selectable) section of one input waveform cycle. The conversion time is therefore dependent on the absolute frequency range and the amplitude of the input voltage. A drawback of the ontime method is the instability of the output due to cycle-to cycle variations in waveform timing.

The ontime input function can also be used in a voltage-controlled one shot conversion system, where the output pulse-width of the one shot is a function of the input voltage.

The period input object performs the same measurement as the ontime input object, except the entire cycle is measured. All attributes of the ontime object scheme apply to the period object scheme. One advantage is that, since a larger piece of the waveform is now being measured, the contribution of the errors due to cycle-to-cycle variations and input range limitations are less pronounced.

The pulsecount input object provides an alternate solution by measuring the number of input edges within a 0.8388608 second time frame. This implies a constant conversion time.

The major advantage in using the pulsecount input function block is the inherent averaging performed by the function itself. Since the actual number of pulses are counted within a given time period, the minor cycle-to-cycle time variations will be averaged over many cycles and would therefore have a much smaller impact on the overall conversion result.

It should also be noted that the pulsecount object, unlike ontime and period, provides an inverse relationship between input voltage and the digital number. An increase in the input voltage causes a decrease in the frequency which translates to a lower count value returned by the pulsecount function.

The following is an example of typical code for use with the circuit in figure 12. The pulsecount input object is used to implement a 16-bit converter.

```
// A/D conversion using a 555 timer
// as a voltage-to-frequency converter
#include <control.h>

////////////////////// I/O Declaration /////////////////
IO_5 input pulsecount analog_input;

//////////////////////Function declaration///////////////
unsigned long analog_to_digital(void);

//////////////////////Function Definition///////////////
unsigned long analog_to_digital(void) {
unsigned long digital_out;
   digital_out = io_in(analog_input);
   return digital_out;
}
```

Performance Issues

Several factors must be taken into account when dealing with real-time, asynchronous, analog signals in a system. These factors become especially important in a distributed network, where the interaction of one node with the others directly affects the network performance and therefore the overall system functionality.

In a distributed control environment, a typical node might convert an analog signal to a digital number, either for direct measurement of the level of the signal, or for calculating another parameter that depends on the analog signal.

In either case, if the resultant digital number is placed on the network for transmission to one or more nodes, overall network traffic is affected, depending on how and when the information is passed, and on the behavior of the analog signal itself.

For example, if the converted digital number is passed from one node to another each time there is a new analog value, the number of packets transferred is directly proportional to how often the sending node performs a conversion.

It is generally not desirable to allow network traffic to be controlled so closely by an external event. This is specially true in the case of an A/D converter where the digital output can change as a result of noise or quantization error.

Therefore, in order to minimize the amount of redundant network traffic and to improve overall system response time, one or more of the following techniques should be used.

1) Using polled network variables. This would cause a packet transfer only when the receiving node requests the sending node to do so.

2) Making A/D conversions less frequently. By optimizing the rate of conversion, redundant packet transmission is reduced. This would also free the application processor for other tasks.

3) Introducing hysteresis by sending updates over the network only when the change in the converted digital number is larger than a specified value. See the *LONMARK™ Application Layer Guidelines* for standard definitions for hysterisis values.

4) As a variation on (2) above, time-averaging the digital values between update intervals.

A/D Converter Sources

The following is a partial listing of serial A/D converter ICs available on the market. The ICs used in the previous examples are included.

PART NUMBER	BITS	CONV. SPEED	PKG SIZE	# OF CHN	S/H	REF	COMMENTS
Analog Devices							
AD575	10	30 µs	14 pin DIP	1	N	Y	Dual supply, unipolar or bipolar
Linear Technology							
LTC1091	10	20 µs	8-pin DIP	2	Y	N	Vcc=Vref
LTC1092	10	20 µs	8-pin DIP	1	Y	N	
LTC1094	10	20 µs	20-pin DIP	8	Y	N	Unipolar/bipolar
LTC1095	10	20 µs	18-pin DIP	6	Y	Y	Vcc=Vref
LTC1291	12	13 µs	8-pin DIP	2	Y	N	
LTC1292	12	13 µs	8-pin DIP	1	Y	N	
LTC1294	12	13 µs	20-pin DIP	8	Y	N	
National							
AD0831	8	32 µs	8-pin DIP	1	N	N	
AD0832	8	32 µs	8-pin DIP	2	N	N	
AD0834	8	32 µs	14-pin DIP	4	N	N	
AD0838	8	32 µs	20-pin DIP	8	N	N	
Motorola							
MC145051	10	88 µS	20-pin DIP	11	Y	N	
MC145053	10	88 µS	14-pin DIP	5	Y	N	
Maxim Integrated Products							
MAX170	12	5 µs	8-pin DIP	1	N	Y	Dual supp, uni/bi
MAX171	12	5 µs	8-pin DIP	1	N	Y	Opto-isolated
MAX190	12	13 µs	24-pin DIP	1	Y	Y	

References

1) Analog Devices, Analog-Digital Conversion Handbook, Prentice-Hall, 1986.
2) Analog Devices, Data Conversion Products Data Book, 1989/90.
3) Linear Technology, 1990 Linear Data Book.
4) Maxim Integrated Products, Integrated Circuits Data Book, 1990.
5) Motorola, CMOS ASIC ICs, Q4/90.
6) National Semiconductor, Data Acquisition Linear Devices Handbook, 1989.

Creating Neuron® C Applications with the Gizmo 3

January 1997 **LONWORKS® Engineering Bulletin**

Introduction

The Neuron C programming language provides 39 I/O objects that support a wide variety of I/O devices. These I/O objects range from simple bit I/O to more complex timer counter objects which precisely generate and measure various square wave signals. The I/O objects are implemented through 11 I/O pins using firmware and hardware support on the Neuron Chip.

The Gizmo 3 enables rapid development of prototype applications that demonstrate several of these I/O objects. The I/O devices supported by the Gizmo kit are used to build examples for distributed sense and control problems. Quick prototypes and feasibility studies are possible without developing any I/O hardware.

This document describes how use the Gizmo 3 and the example application programs provided with it. Multi-node application examples are included to help the user get started.

Getting Started

The Gizmo 3 is included with LonBuilder® development systems shipped since February 1997, and is included with the NodeBuilder® development tool. It is also available as part of the LonBuilder Gizmo 3 Kit (Echelon Model Number 28001), which includes the following:

- Gizmo 3 (Motorola MC143206EVK) - Contains 11 Neuron Chip compatible physical I/O devices.
- Application Interface Board - Provides access to the I/O pins of the Neuron Chip on a LonBuilder processor board through a DB25 connector. This board works with LonBuilder Neuron Emulators.
- Application I/O Interface Cable - Connects the application interface board to the Gizmo 3.
- Programming Examples - These are included with the LonBuilder and NodeBuilder tools, and are also available on the World-Wide Web at URL:

  `http://www.lonworks.echelon.com/dev_toolbox/dev_toolbox/examples/neuron_c/gizmo_ex.zip`

Echelon Part Number 005-0016-01 REV G

This archive contains sample source code to demonstrate a variety of I/O objects supported by Neuron C and the Gizmo 3. The examples are designed to support several different distributed sense and control applications.

Hardware Installation

Connect the Gizmo 3 to a LonBuilder Neuron emulator, or to the LTM-10 LonTalk Node, using the supplied ribbon cable.

Software Installation

The Gizmo 3 programming examples provide Neuron C source code for use in example applications (see last section of this document). This source code is compiled and loaded into nodes using the LonBuilder or NodeBuilder software. To install and load the example applications, follow the steps in the appropriate User's Guide.

Gizmo 3 I/O Hardware

The Gizmo 3 is used to prototype LONWORKS applications. It includes the physical I/O devices listed in table 1.

Table 1. Gizmo 3 I/O Devices

Type	***Device***	***Function***
Analog	Analog-to-Digital	4-channel 10-bit MC14053. Neurowire device using IO1 as the select line.
	Digital-to-Analog	4-channel 6-bit MC14411. Neurowire device using IO3 as the select line.
Sound	Piezo-electric transducer	Operated by square wave output from IO0
Time	Real Time Clock	MC68HC68T1. Neurowire device using IO6 as the select line.
Temperatur e	Temperature Sensor	Connected to channel 0 of the A/D converter
Switches	Left Push-button	When pressed, drives the IO7 input low.
	Right Push-button	When pressed, drives the IO3 input low.
	Quadrature Control	When rotated, drives the quadrature shaft encoder input on IO4 and IO5.
LED Indicators	IO1 Bit Output LED (Red)	Indicates the state of the IO1 output. The LED is on when IO1 is low.
	IO2 Bit Output LED (Green)	Indicates the state of the IO2 output. The LED is on when IO2 is low.
	LED Display	4-digit, 7 segment display controlled by an MC14489 Multi-Character LED Display Driver. Neurowire device using IO2 as the select line.

The block diagram of the Gizmo 3 is shown in figure 1.

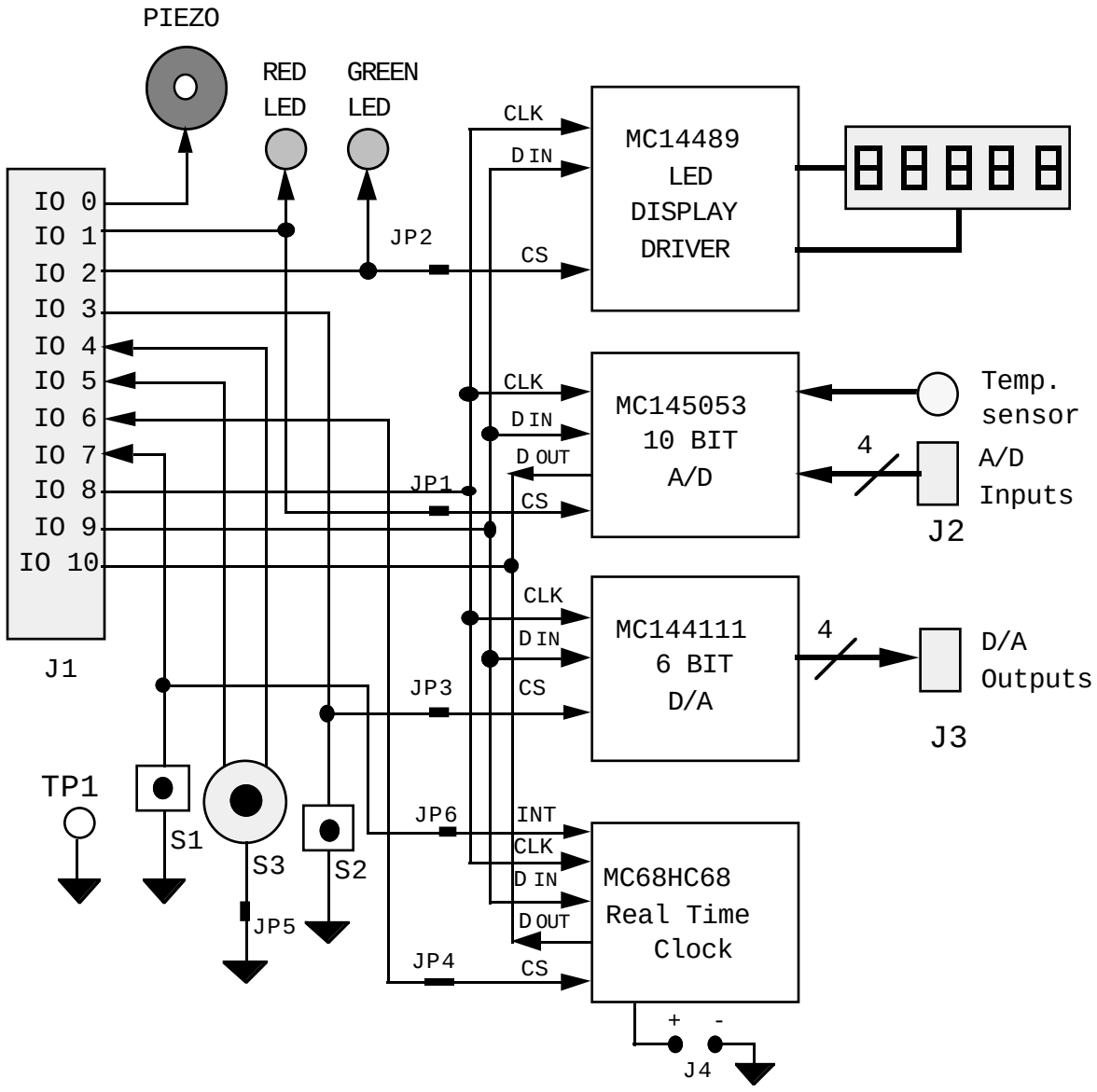


Figure 1. Gizmo 3 Block Diagram

Three of the I/O pins are used both for Neurowire device selection as well as for direct I/O devices. If the application uses the direct I/O devices, the associated Neurowire device may exhibit undesirable behavior. Jumpers are provided to disconnect the chip select pin for each of the affected Neurowire devices, as shown in table 2.

Table 2. Gizmo 3 Neurowire devices

I/O Pin	*Selected device*	*Direct I/O Function*	*Jumper*
IO1	A/D converter	Red LED	JP1
IO2	7-segment display	Green LED	JP2
IO3	D/A converter	Right push-button	JP3
IO6	Real time clock	N/A	JP4

LonBuilder Hardware

The Gizmo 3 attaches to a LonBuilder processor board using the Echelon model 27810 LonBuilder Application Interface Board. This board is fully described in the *LonBuilder Hardware Guide*. The features and function of this board are described here for convenience to the user.

The Application Interface Board provides access to the 11 I/O pins, 5 communications pins, reset pin, and service pin of the Neuron Chip. The board also provides +5V, ground, +12V, and -12V to external hardware. The Gizmo 3 uses the 11 I/O pins, +5V, and GND. Table 3 shows the pin-out for the application I/O interface cable shipped as part of the LonBuilder Gizmo 3 Kit.

Table 3. Gizmo 3 Application I/O Interface Cable Pin-out

Gizmo 3 and LTM-10 20-pin Connector Pin #	Signal Name	LonBuilder AIB DB-25 Connector Pin #
1	IO0	13
2	IO1	25
3	IO2	12
4	IO3	24
5	IO4	11
6	IO5	23
7	IO6	10
8	IO7	22
9	IO8	9
10	IO9	21
11	IO10	8
12	--	20
13	--	7
14	--	19
15	~SERVICE*	6
16	~RESET*	18
17	-12VDC*	5
18	+12VDC*	17
19	+5VDC	4
20	GND	16

*These signals are not used by the Gizmo 3.

IMPORTANT

For the Gizmo 3's quadrature shaft encoder, and left push-button devices to function properly with a LonBuilder emulator, you must **remove** the **JP2** jumper on the emulator board. However, this jumper must be **installed** to use the IO4 push-button on the emulator's front panel.

Also, the Neuron Chip's internal I/O pull-ups must be enabled. All applications using the Gizmo3 must therefore include the following Neuron C compiler directive:

```
#pragma enable_io_pullups
```

The Application Interface Board provides unbuffered access to the 11 I/O pins (IO0-I10), the 5 communications pins (CP0-4), and the `~Service` pin. Refer to the *Neuron Chip Data Book* for the electrical specification of these signals.

WARNING: These signals are unbuffered. When using the application interface board with hardware other than the Gizmo 3, the Neuron Chip may be easily damaged due to electrostatic discharge or incorrect voltage levels.

The `~Reset` signal is a buffered CMOS input. Asserting this signal low causes a hardware reset to occur on the emulator. Refer to the *Neuron Chip Data Book* for the timing specification of the reset signal.

The application interface board provides fuse protected access to the development station power supply (+5V, -12V, and +12V). These fuses are user replaceable. Replacement fuses must not exceed a 0.5A rating. External hardware should not exceed 400 mA for 5V, and 35 mA for +12V and -12V.

There are three jumpers on the application interface board. These jumpers connect `~Reset`, `~Service`, and `Clock` to the DB25 connector when installed. The silk screen references on the board document the function of each jumper. The default configuration (jumpers removed) should be used when using the Gizmo 3.

NodeBuilder Hardware

The Gizmo 3 attaches to the LTM-10 LonTalk Node using the supplied 20-conductor ribbon cable. The pin-out of this cable is shown in table 3.

Using the IO4 Push-button and the IO0 LED

Both the LonBuilder Neuron emulator and the LTM-10 LonTalk Node supplied with the NodeBuilder Development Tool include a momentary push-button controlling the Neuron Chip IO4 pin, and a green LED controlled by the IO0 pin.

All applications that wish to use the IO4 push-button on the LTM-10 LonTalk node must include the following Neuron C compiler directive:

```
#pragma enable_io_pullups
```

Some revisions of the LonBuilder Neuron emulator may require you to *omit* this directive for reliable operation of the IO4 push-button on the emulator. You must **install** the **JP2** jumper on the emulator board to use the IO4 push-button on the emulator. However, this jumper must be **removed** to use the Gizmo 3 with the emulator. You must install the emulator JP1 jumper to use the IO0 LED, but you need not remove it if you wish to use the Gizmo 3. See Chapter 2 of the *LonBuilder Hardware Guide* for more details.

For the LonBuilder Neuron emulator, the IO4 push-button and IO0 LED operate with *non-inverted* logic, so that if the push-button is pressed, the Neuron C application will read a 1, and if the Neuron C application writes a 1, the LED will light.

For the LTM-10 LonTalk node, the IO4 push-button and IO0 LED operate with *inverted* logic, so that if the push-button is pressed, the Neuron C application will read a 0, and if the Neuron C application writes a 0, the LED will light.

Programming Examples

A working knowledge of how to use the LonBuilder or NodeBuilder software to assign Neuron C applications to specific nodes is assumed. The Neuron C source files listed in table 4 are provided in the EXAMPLES directory.

Table 4. Neuron C Source Files

Source File	*Program Function*	*Hardware Required*
ACTUATOR.NC	LONMARK closed-loop discrete actuator	IO0 LED
ALARM.NC	Temperature alarm node	Gizmo 3
ANLGSNSR.NC	LONMARK open-loop analog sensor	Gizmo 3
BLINK.NC	Blink an LED	IO0 LED
DIMMER.NC	AC lamp control	Triac AC dimmer*
DISPLAY.NC	Display value of network variable	Gizmo 3
ENCODER.NC	Display dial position	Gizmo 3
LAMP.NC	Control an LED	IO0 LED
SCPT_EX.NC	LONMARK configuration parameter file example	None
SENSOR.NC	LONMARK closed-loop discrete sensor	Push-button (IO4 or Gizmo 3)
SWITCH.NC	Sense a push-button	Push-button (IO4 or Gizmo 3)
SWITCH_8.NC	LONMARK node object, eight open-loop sensor objects	Eight switches*
TEMP.NC	Monitors and displays temperature	Gizmo 3
TEMP_MON.NC	Monitor temperature with guard-point and alarm	Gizmo 3
TEST.NC	Tests Gizmo 3 I/O devices	Gizmo 3
THERMOS.NC	Thermostat device	Gizmo 3

(*) Hardware not included with LonBuilder or NodeBuilder tools

A functional specification for each example program (other than those marked with an asterisk in table 4) is provided in Appendix A. In addition, suggestions for combining the example programs to create distributed sense and control applications are described.

The Neuron C include files listed in table 5 contain drivers for the various Gizmo 3 hardware devices. These files are also provided in the EXAMPLES directory. See Appendix B for information on using these drivers in your own applications.

Table 5. Neuron C Include Files

Source File	*Functions Included*	*Hardware Required*
ANALOG.H	Sample analog inputs Update analog outputs	A/D converter D/A converter
DISPLAY.H	Numeric and alphanumeric display	Seven-segment display
GIZMO_IO.H	Miscellaneous I/O functions	Push-buttons, discrete LEDs, quadrature dial, piezo transducer
RTCLOCK.H	Maintain time and date	Real time clock
TEMPERAT.H	Temperature-related functions	Temperature sensor

Using the Examples with the LonBuilder Tool

To prepare your LonBuilder object database to run the example applications with the LonBuilder tool, follow these steps. The first program you will run is TEST.NC. This application exercises a variety of the I/O devices on the Gizmo 3.

To install this program example into a Neuron emulator:

1. Make the directory to which the examples were copied the current working directory. Create a sub-directory named PROJECT if one does not already exist.
2. Invoke the LonBuilder software using the following command: LB PROJECT
3. Using the Navigator screen in the LonBuilder system, select the App Node class of objects from the top-level menu.
4. Select the Node Spec sub-class of objects from the **App Node** window. Create a node specification for the emulator which has the Gizmo 3 attached, and enter TEST in the App Image Name field. Enter the name of the emulator (for example emulator_1) in the Target HW field, and a node name (for example test_node) in the Node Name field. Save the node specification record.
5. Return to the **App Node** window, and request an Automatic Load from the project menu. Review the functional specification that follows for this application to verify proper hardware operation.
6. Experiment with the other examples provided:

```
ACTUATOR ALARM ANLGSNSR BLINK DIMMER DISPLAY ENCODER LAMP
SCPT_EX SENSOR SWITCH SWITCH_8 TEMP TEMP_MON TEST THERMOS
```

Using the Examples with the NodeBuilder Tool

To prepare your NodeBuilder project to run the example applications with the NodeBuilder tool, follow these steps. The first program you will run is TEST.NC. This application exercises a variety of the I/O devices on the Gizmo 3.

To install this program example into the LTM-10 LonTalk Node:

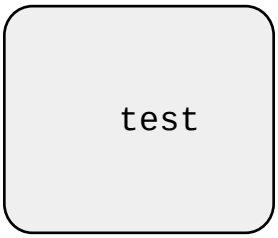
1. Make the directory to which the examples were copied the current working directory.
2. Invoke the NodeBuilder software by clicking on its icon in the Windows Program Manager
3. If a device window is already open, close it. Create a new device by clicking on the Device icon.
4. In the General tab of the Device window, enter TEST into the Application Image edit box. Make sure that the Device Template is set to LTMRAM.DTM.
5. Click on the Save button, and save the device as TEST.DEV.
6. Click on the Build and Load button to compile and load the application into the LTM-10 LonTalk Node.
7. Review the functional specification that follows for this application to verify proper hardware operation.
8. Experiment with the other examples provided.

   ```
   ACTUATOR ALARM ANLGSNSR BLINK DIMMER DISPLAY ENCODER LAMP
   SCPT_EX SENSOR SWITCH SWITCH_8 TEMP TEMP_MON TEST THERMOS
   ```

   You cannot use the NodeBuilder tool to bind nodes together in order to create the multiple node examples described below. You can use a network management tool such as LonMaker™ for this purpose.

Appendix A - Sample Applications

Program - test.nc



Input Network Variables

None

Output Network Variables

None

Input I/O Objects

IO_4..5	ioDial	quadrature
IO_7	ioLeftPB	bit
IO_3	ioRightPB	bit

Output I/O Objects

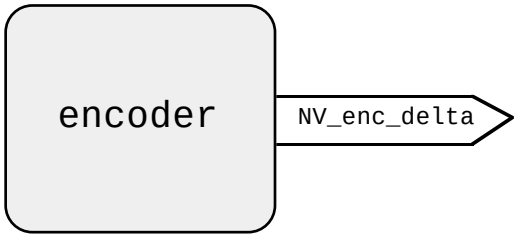
IO_0	ioPiezo	frequency
IO_1	ioRedLED	bit
IO_2	ioGreenLED	bit
IO_8..9, IO_2	ioSevenSeg	neurowire

Functional Specification

This program tests the LED, push-button, piezo-electric and quadrature devices on the Gizmo 3. The following operation is expected for a working module:

1. All segments of the display are lit when the Neuron Chip is reset.
2. The piezo transducer plays a tune when the right push-button is pressed.
3. The encoder position is displayed when changed.
4. The red LED lights up when the left push-button is pressed.
5. The green LED flickers as the display is updated.

Program - encoder.nc



Input Network Variables

None

Output Network Variables

NV_enc_delta	long

Input I/O Objects

IO_4..5	ioDial	quadrature

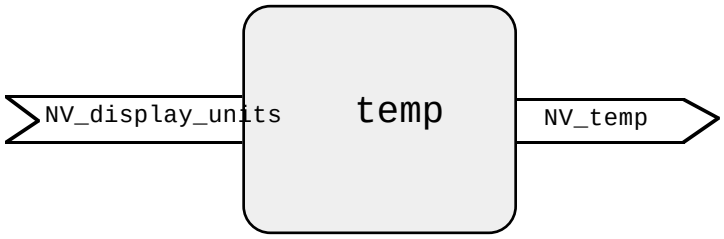
Output I/O Objects

None

Functional Specification

When the dial is turned, the node attached to the Gizmo 3 sends out a network variable, NV_enc_delta, indicating the incremental change in value of the encoder.

Program - temp.nc



Input Network Variables

NV_display_units	config	TempUnits

Output Network Variables

NV_temp	SNVT_temp

Input I/O Objects

IO_8..9, IO_1	ioA2D	neurowire master

Output I/O Objects

IO_8..9, IO_2	ioSevenSeg	neurowire master

Functional Specification

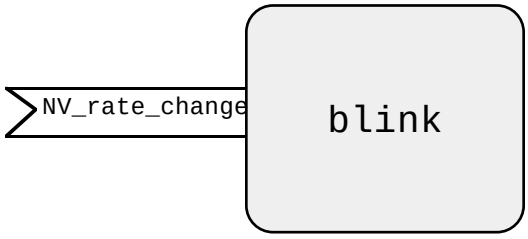
The temperature sensor is read and displayed every 0.5 seconds. If a change in temperature has occurred, the updated value is sent out as the NV_temp network variable. The NV_display_units network variable is declared as a config network variable and can therefore be modified by a network management node. The value of this variable determines whether the temperature is shown in degrees Celsius or Fahrenheit. The resolution of the value displayed is either 1 degree Fahrenheit or 0.5 degrees Celsius. Modify the default value by changing the

```
#define DEFAULT_TEMP_UNIT
```

statement in the file TEMPERAT.H. Choose either of the following:

```
#define DEFAULT_TEMP_UNIT CELSIUS
#define DEFAULT_TEMP_UNIT FAHRENHEIT
```

Program - blink.nc



Input Network Variables

NV_rate_change	long

Output Network Variables

None

Input I/O Objects

None

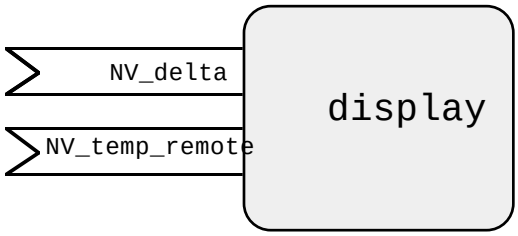
Output I/O Objects

IO_0	IO_emulator_led	oneshot

Functional Specification

The LED is flashed at a frequency changed in proportion to the integrated change in the value of the NV_rate_change input network variable. The input network variable for this application may be attached to the output network variable of the ENCODER.NC application.

Program - display.nc



Input Network Variables

NV_delta	long
NV_temp_remote	SNVT_temp

Output Network Variables

None

Input I/O Objects

None

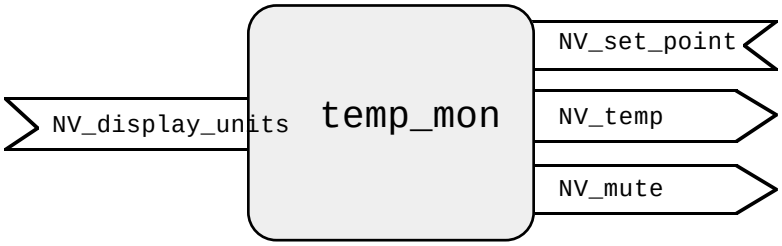
Output I/O Objects

IO_1	ioRedLED	bit
IO_8..9, IO_2	ioSevenSeg	neurowire master

Functional Specification

A slave seven-segment display that either shows temperature or an incrementing value. While showing a temperature, the red LED is lit. The input network variables to this application support connection to the output network variables of the TEMP.NC, and ENCODER.NC applications.

Program - temp_mon.nc



Input Network Variables

NV_set_point	SNVT_temp
NV_display_units	config TempUnits

Output Network Variables

NV_temp	SNVT_temp
NV_mute	SNVT_lev_disc

Input I/O Objects

IO_8..9, IO_1	ioA2D	neurowire master
IO_7	ioLeftPB	bit

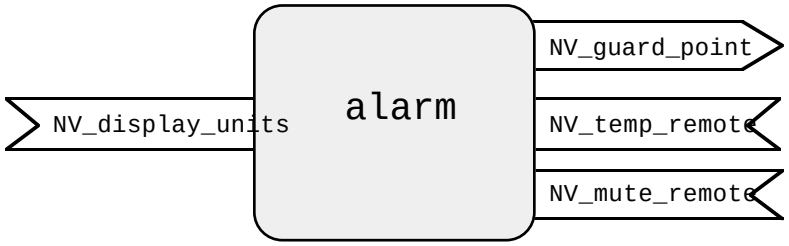
Output I/O Objects

IO_1	ioRedLED	bit
IO_8..9, IO_2	ioSevenSeg	neurowire master

Functional Specification

The current temperature of the sensor is sent out as the NV_temp network variable. The NV_set_point network variable provides a temperature setpoint. The left push button toggles the seven-segment display between displaying the setpoint and displaying the current temperature. Whenever the setpoint is displayed, the red LED is lit. The NV_display_units configuration network variable determines whether the temperature is shown in degrees Celsius or Fahrenheit.

Program - alarm.nc



Input Network Variables

NV_temp_remote	SNVT_temp
NV_mute_remote	SNVT_lev_disc
NV_display_units	config TempUnits

Output Network Variables

NV_guard_point	SNVT_temp

Input I/O Objects

IO_3	ioRightPB	bit
IO_4	ioDial	quadrature
IO_7	ioLeftPB	bit

Output I/O Objects

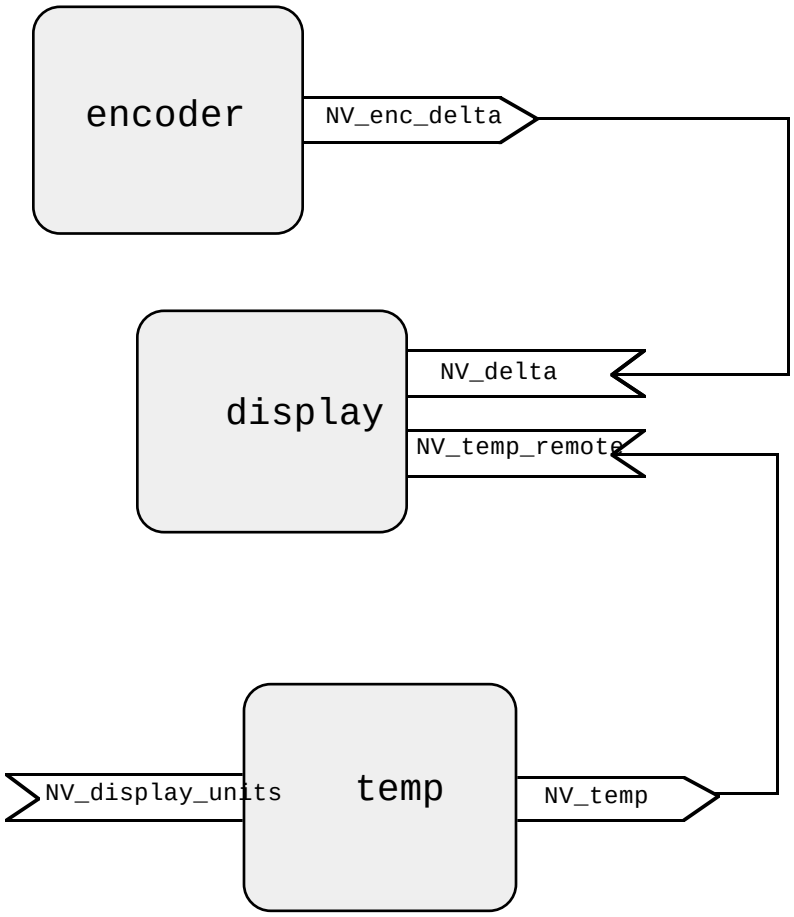
IO_0	ioPiezo	frequency
IO_1	ioRedLED	bit
IO_8..9, IO_2	ioSevenSeg	neurowire master

Functional Specification

The dial is used to set a temperature guard point which is sent out as the NV_guard_point network variable. The left push button toggles the seven-segment display between displaying the guard point and displaying the current temperature. The guard point is automatically displayed for 2 seconds when its value is updated. Whenever the guard point is displayed, the red LED is lit. The NV_temp_remote network variable is compared with the guard point temperature and the audible alarm is sounded if the temperature equals or exceeds the limits. The alarm can be muted based on the NV_mute_remote network variable. The NV_display_units

network variable determines whether the temperature is shown in degrees Celsius or Fahrenheit. The default value is specified in the file TEMPERAT.H.

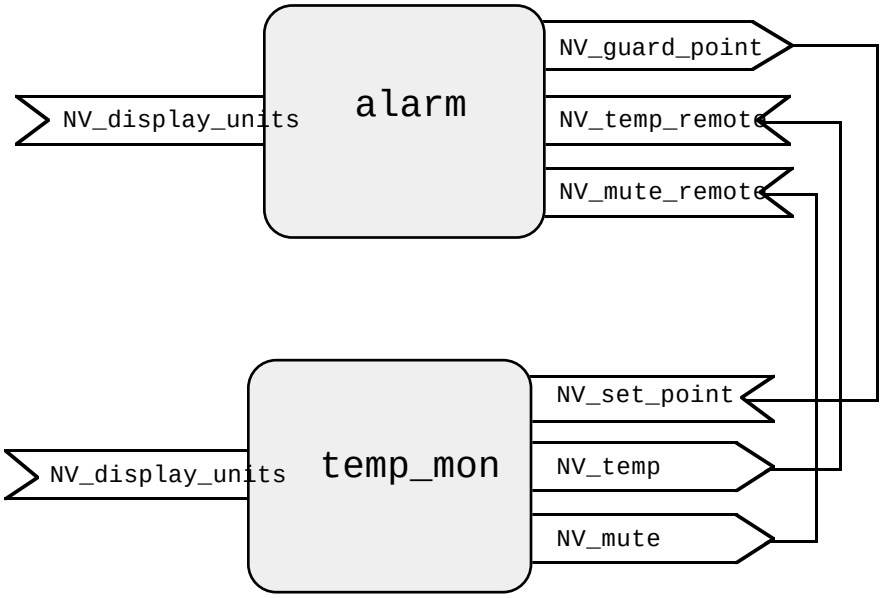
Application Example - Encoder/Temperature Display



Description

This example illustrates an encoder or temperature display. The display node can receive either a temperature value from the temperature sensor or an incremental value from the encoder for display on the seven-segment display.

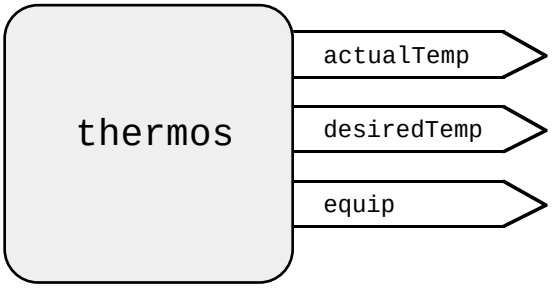
Application Example - Remote Temperature Alarm



Description

This example illustrates a remote temperature alarm. One node acts as a temperature sensor and the other node as a remote alarm. The temperature guard point is set on the alarm node. The current temperature is monitored at the sensor. Whenever its value changes, the updated value is communicated to the remote alarm. The alarm checks the current temperature against the guard point. The alarm is sounded when the guard point is exceeded and remains on until the temperature goes below the guard point. If the `NV_mute_remote` network variable changes state to `ST_OFF`, the alarm is silenced.

Program - thermos.nc



Input Network Variables

None

Output Network Variables

actualTemp	SNVT_temp
desiredTemp	SNVT_temp
equip	enum { OFF, HEATING, COOLING }

Input I/O Objects

IO_8..9, IO_1	ioA2D	neurowire master

Output I/O Objects

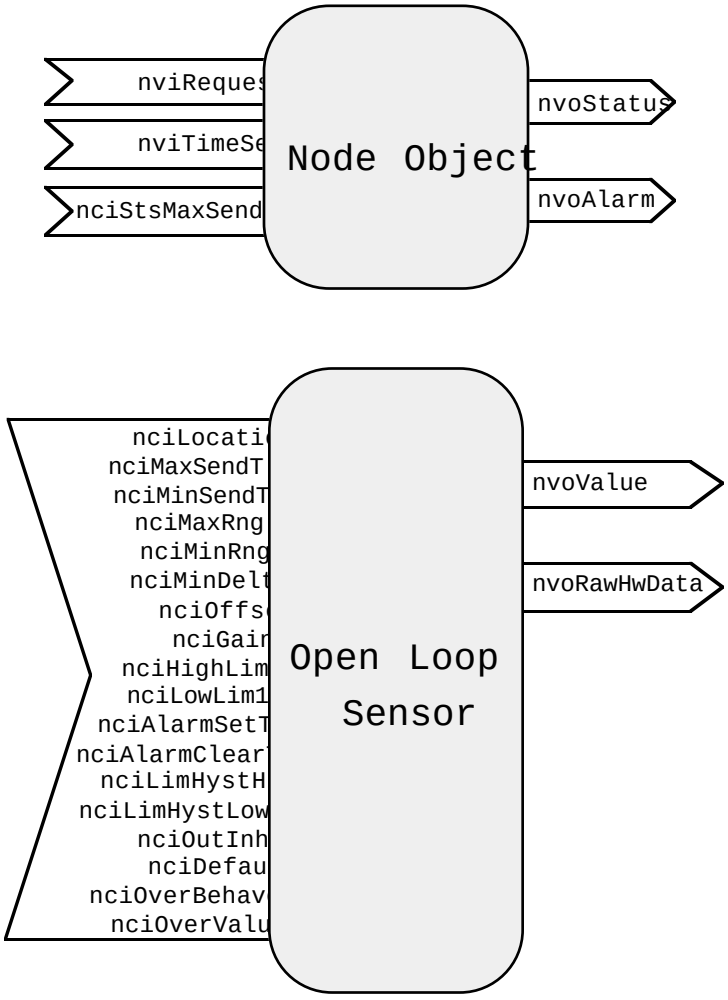
IO_1	ioRedLED	bit
IO_2	ioGreenLED	bit

Functional Specification

The current temperature of the sensor is measured every five minutes and compared to the desired temperature. If it is too hot, the cooling equipment is activated (indicated by the green LED). If it is too cold, the heating equipment is activated (indicated by the red LED). If the measured temperature is in a dead band of 1°C around the desired temperature, all equipment is inactivated. The desired temperature is set using the quadrature dial encoder. The output network variables actualTemp and desiredTemp contain the measured and desired temperature respectively. The output network variable equip contains the state of the heating and cooling equipment.

Program - anlgsnsr.nc

This example complies with the LONMARK® Application Layer Interoperability Guidelines, Revision 3.0.



Input Network Variables

nviRequest	SNVT_obj_request
nviTimeSet	SNVT_time_stamp
nciStsMaxSendT	config SNVT_elapsed_tm
nciLocation	config SNVT_str_asc
nciMaxSendT	config SNVT_elapsed_tm
nciMinSendT	config SNVT_elapsed_tm
nciMaxRng	config SNVT_temp
nciMinRng	config SNVT_temp
nciMinDelta	config SNVT_temp
nciOffset	config SNVT_temp
nciGain	config SNVT_muldiv
nciHighLim1	config SNVT_temp
nciLowLim1	config SNVT_temp
nciAlarmSetT1	config SNVT_elapsed_tm
nciAlarmClearT1	config SNVT_elapsed_tm
nciLimHystHi1	config SNVT_temp
nciLimHystLow1	config SNVT_temp
nciOutInhT	config SNVT_elapsed_tm
nciDefault	config SNVT_temp
nciOverBehave	config SNVT_override
nciOverValue	config SNVT_temp

Output Network Variables

nvoStatus	SNVT_obj_status
nvoAlarm	SNVT_alarm
nvoValue	SNVT_temp
nvoRawHwData	SNVT_count

Input I/O Objects

IO_3	ioRightPB	bit
IO_7	ioLeftPB	bit
IO_8..9, IO_1	ioA2D	neurowire master

Output I/O Objects

IO_8..9, IO_2	ioSevenSeg	neurowire master

Functional Specification

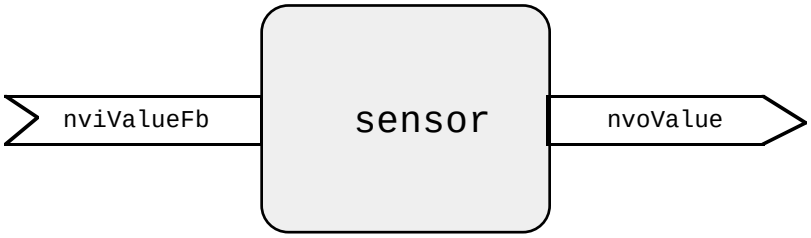
The application reads the temperature sensor using the A/D converter, calibrates the raw data, and displays the result. When the right push-button is pressed, the result is displayed in degrees Celsius. When the left push-button is pressed, the result is displayed in degrees Fahrenheit. See the *LONMARK Application Layer Interoperability Guidelines* for a complete description of the functionality of the Node Object and the Open Loop Sensor Object.

This example must be compiled with version 6.02 or later of the SNVT.TYP file.

Push-button and LED Application Examples

Program - sensor.nc

This example complies with the LONMARK® Application Layer Interoperability Guidelines, Revision 3.0.



Input Network Variables

nviValueFb	SNVT_switch

Output Network Variables

nvoValue	SNVT_switch

Input I/O Objects

IO_4	ioButton	bit
IO_7	ioLeftPB	bit
IO_3	ioRightPB	bit

Output I/O Objects

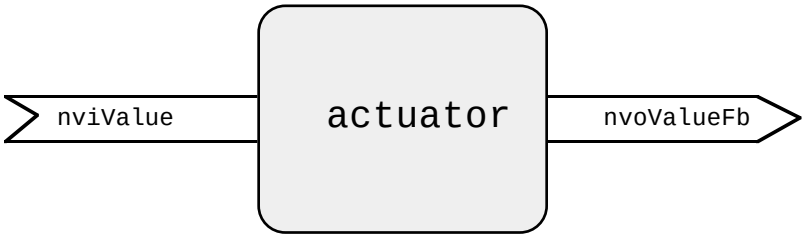
IO_0	ioLED	bit

Functional Specification

When any push-button (IO4 or Gizmo 3) is pressed, the node updates its output network variable nvoValue. If the feedback input network variable nviValueFb indicates OFF, nvoValue is set to the ON state with a full-scale level. If the feedback input indicates ON, nvoValue is set to the OFF state with a full-scale level. When the feedback input network variable nviValueFb is updated, the IO0 LED is set to the state indicated by the feedback variable. The node is implemented as a LONMARK-compliant closed loop sensor.

Program - actuator.nc

This example complies with the LONMARK® Application Layer Interoperability Guidelines, Revision 3.0.



Input Network Variables

nviValue	SNVT_switch

Output Network Variables

nvoValueFb	SNVT_switch

Input I/O Objects

None

Output I/O Objects

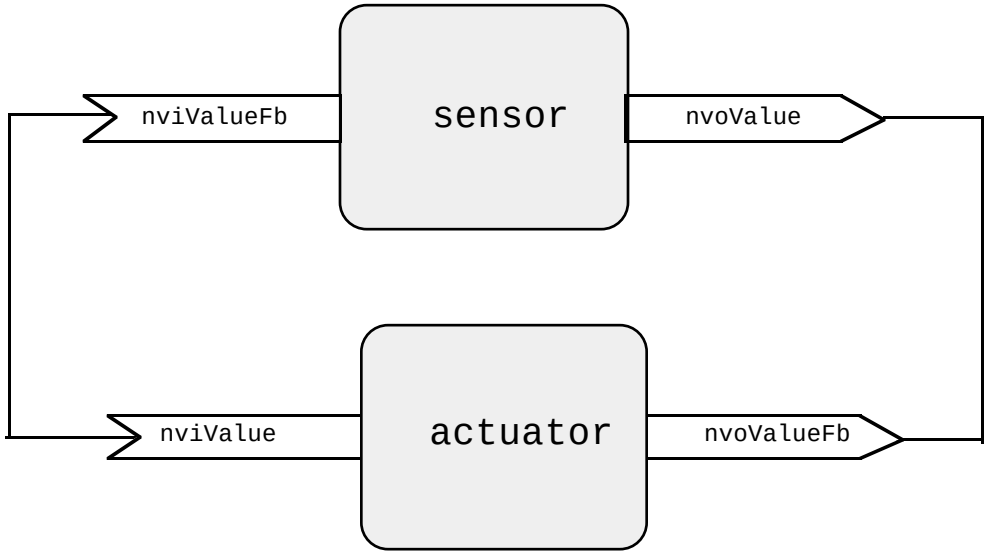
IO_0	ioLED	bit

Functional Specification

When the input network variable nviValue is updated, the IO0 LED is set to the state indicated by the input variable. The feedback output network variable nvoValueFb is updated to reflect the value of the input network variable. The node is implemented as a LONMARK-compliant closed loop actuator.

Application Example - Closed Loop Sensor and Actuator

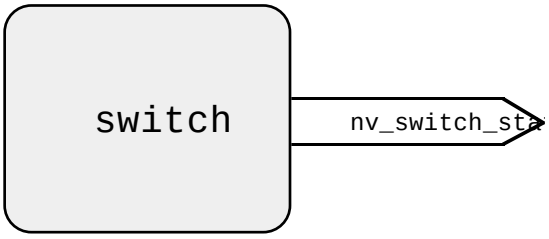
This example complies with the LONMARK® Application Layer Interoperability Guidelines, Revision 3.0.



Description

This example illustrates a closed-loop sensor and actuator system. When the push-button on the sensor node is pressed, the state of the actuator is toggled, and then fed back to the sensor node. Thus each time the push-button is pressed, the LED on the actuator alternates between off and on. If more than one sensor is used for the same actuator, the feedback keeps them all synchronized.

Program - switch.nc



Input Network Variables

None

Output Network Variables

nv_switch_state	SNVT_lev_disc

Input I/O Objects

IO_4	ioButton	bit
IO_7	ioLeftPB	bit
IO_3	ioRightPB	bit

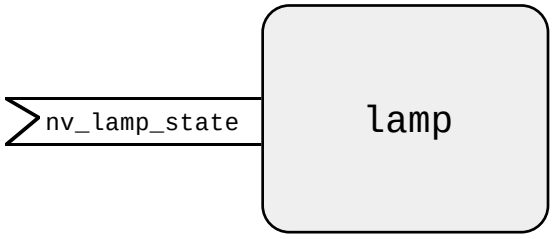
Output I/O Objects

None

Functional Specification

When any push-button (IO4 or Gizmo 3) is pressed, the node updates its output network variable nv_switch_state to the value ST_ON. When the push-button is released, the output network variable is set to ST_OFF.

Program - lamp.nc



Input Network Variables

nv_lamp_state	SNVT_lev_disc

Output Network Variables

None

Input I/O Objects

None

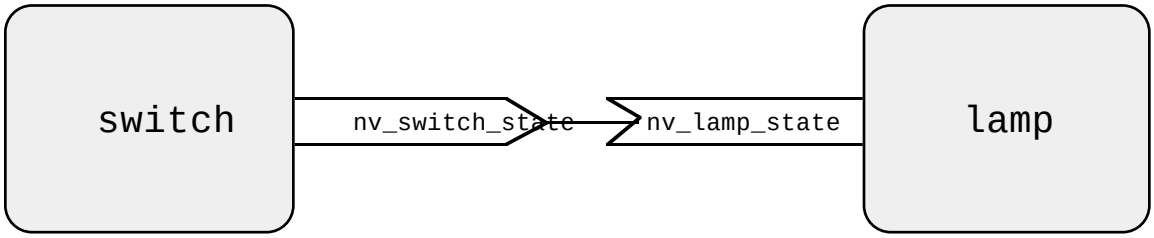
Output I/O Objects

IO_0	ioLED	bit

Functional Specification

When the input network variable nviValue is updated, the LED is set to the state indicated by the input variable.

Application Example - Switch and Lamp



Description

This example illustrates a simple switch and lamp system. When any push-button on the sensor node is pressed, the IO0 LED on the lamp node is turned on. When the push-button on the sensor node is released, the IO0 LED on the lamp node is turned off.

Appendix B - Device Driver Software

The Gizmo 3 example software comes with four Neuron C include files that contain useful definitions and functions for driving the Gizmo 3 hardware devices. This section describes how you can use these functions to simplify your own application development for the Gizmo 3 hardware. See the *Neuron Chip Data Book* and the *Neuron C Reference Guide* for more detailed descriptions of the I/O models used. Two of these device driver files may be used with the older Gizmo 2 device.

Driver File	Devices Supported
GIZMO_IO.H	Simple I/O devices. Same for both Gizmo versions.
DISPLAY.H	Seven segment LED display Gizmo 3 has 5 digits, Gizmo 2 has 4 digits

The files ANALOG.H, RTCLOCK.H, and TEMPERAT.H may only be used with the Gizmo 3.

GIZMO_IO.H

This file contains Neuron C declarations for the simpler I/O devices on the Gizmo 3.

```
IO_0 output frequency   clock(1) ioPiezo;
IO_1 output bit         ioRedLED = 1;
IO_2 output bit         ioGreenLED = 1;
IO_3 input  bit         ioRightPB;
IO_4 input  quadrature  ioDial;
IO_7 input  bit         ioLeftPB;
```

This file also contains the following definitions of symbolic literals that may be used to represent the states of the push-buttons and discrete LEDs.

```
#define     LED_OFF     1
#define     LED_ON      0
#define     PB_DOWN     0
#define     PB_UP       1
```

Examples:

```
io_out (ioPiezo, 250);     // create 5 kHz sound
io_out (ioRedLED, LED_ON); // turn on red LED
when (io_changes(ioRightPB) to PB_DOWN) // wait for button to be pressed
when (io_update_occurs(ioDial))  // wait for quad dial to be turned
```

ANALOG.H

This file contains the following I/O declarations for the A/D and D/A converters.

```
IO_8 neurowire master select(IO_1) ioA2D;
IO_1 output bit ioA2DSelect = 1;
IO_8 neurowire master select(IO_3) ioD2A;
IO_3 output bit ioD2ASelect = 1;
```

The Neuron C function definition to read the A/D converter is as follows:

```
long A2DConvert (unsigned muxAddr);
```

The A/D converter has a five-channel multiplexer, and returns a 10-bit result in the range 0 .. 1023, with a full-scale value corresponding to 5 volts on the input.

The application program calls the `A2DConvert()` function, passing in the multiplexer channel address in the range 0 .. 4. This function initiates an A/D conversion on that channel. At the same time, it reads the results of the *previous* A/D conversion, for whatever channel was specified on the *previous* call to `A2DConvert()`. The function returns the value of the latest conversion on the channel specified by the `muxAddr` parameter. If the `muxAddr` parameter is out of range, or if there has been no previous conversion on the specified channel, then the function returns the value -1.

Example:

```
value = A2DConvert(0);  // read the temperature sensor
```

The Neuron C function definition to write to the D/A converter is as follows:

```
void D2AConvert (unsigned data, unsigned muxAddr);
```

The D/A converter has four channels, and six bits of resolution. The application program calls the `D2AConvert()` function, passing in the channel address in the range 0 .. 3, and a data value in the range 0 .. 63. Full scale corresponds to 5 volts on the output.

Example:

```
D2AConvert (25, 2);  // output 1.98 volts on channel 2
```

TEMPERAT.H

This file contains definitions relating to the temperature sensor. The Neuron C function `ConvertRawTemp()` converts the raw reading from channel 0 of the A/D converter to `SNVT_temp` units. The function definition is as follows:

```
SNVT_temp ConvertRawTemp (unsigned long rawTemperature);
```

The standard network variable type SNVT_temp has a resolution of 0.1°C, and a range of -274.0°C to 6279.5°C. See the *SNVT Master List and Programmer's Guide* for more details.

Example:

```
SNVT_temp currentTemperature;
currentTemperature = ConvertRawTemp (A2DConvert (0)); // read temperature
```

This file also contains macro definitions which convert integer decimal constants in degrees Fahrenheit and degrees Celsius to `SNVT_temp` units.

Examples:

```
SNVT_temp maxTemperature = TEMP_DEG_C(85);   // initialize to 85°C
SNVT_temp minTemperature = TEMP_DEG_F(0);    // initialize to 0°F
```

RTCLOCK.H

This file contains the following I/O declarations for the real-time clock chip.

```
IO_8 neurowire master select(IO_6) ioRtc;
IO_6 output bit ioRtcSelect = 1;
```

The Neuron C function definition to write to the real-time clock chip is as follows:

```
void RTCSetTime(const SNVT_time_stamp * pTimeSet);
```

The `pTimeSet` parameter points to a structure of type SNVT_time_stamp with the following declaration:

```
typedef struct {
      unsigned long  year;
      unsigned short month;
      unsigned short day;
      unsigned short hour;
      unsigned short minute;
      unsigned short second;
} SNVT_time_stamp;
```

See the *SNVT Master List and Programmer's Guide* for more details of this structure.

The valid range for `year` is 1990 to 2089.

Example:

```
SNVT_time_stamp timeSet = { 1996, 12, 1, 20, 30, 0 }; // 8:30 pm Dec 1, 1996
RTCSetTime (&timeSet);  // set the real-time clock
```

The Neuron C function to read the real-time clock chip is as follows:

```
void RTCGetTime(SNVT_time_stamp *pTimeGet,
     SNVT_date_day *pDayOfWeek);
```

The parameter `pTimeGet` points to a structure of type `SNVT_time_stamp`. For the returned date and time. The parameter `pDayOfWeek` points to an enumeration byte for the returned day of the week, where 0 = Sunday, 1 = Monday … 6 = Saturday. The `pDayofWeek` pointer may be 0, in which case the day of the week is not returned.

Example:

```
SNVT_time_stamp Now;
SNVT_date_day Today;
RTCGetTime (&Now, &Today);    // read the real-time clock
```

DISPLAY.H

This file contains the following I/O declarations for the seven-segment LED display chip.

```
IO_8 neurowire master select(IO_2) ioSevenSeg;
IO_2 output bit io7SegSelect = 1;
```

The file contains Neuron C functions to display decimal data on the Gizmo 3 LED display, functions to display strings consisting of the characters displayable in seven segments, and a function to display temperatures values.

Before using this code, make sure that the number of digits in your display is specified correctly by the `NUM_DIGITS` parameter. For the display in the Gizmo 3, leave the `NUM_DIGITS` parameter at 5. For the display in the older Gizmo 2 device, change `NUM_DIGITS` to 4. The digits are numbered as shown in figure 2.

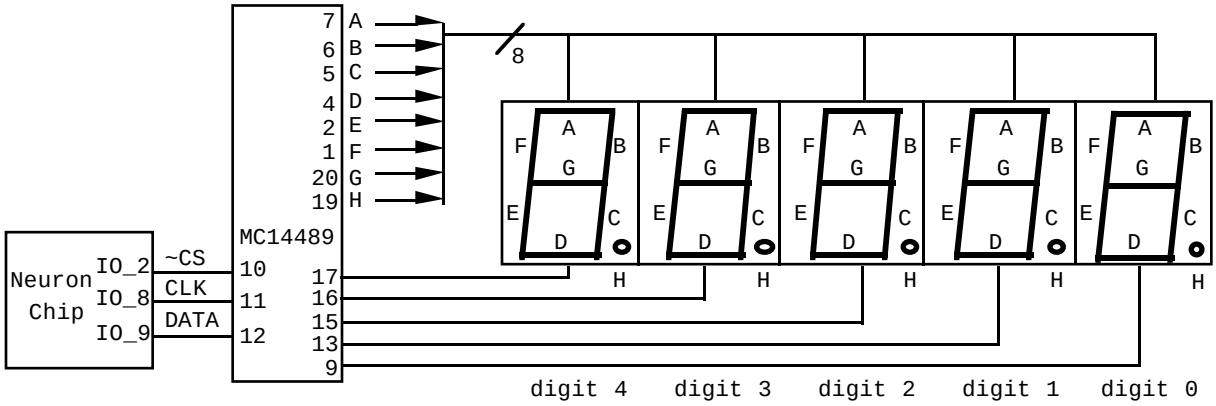


Figure 2. **Gizmo 3 Seven Segment LED Display**

The software functions are divided into three groups; low-level functions, display image update functions, and high-level functions.

Low-Level Functions

The first group of functions provides low-level access to the display controller chip.

The `DspClearImage()` function clears a RAM copy of the configuration and display registers (the variables `dspCfgReg` and `dspDataReg`) to a state that displays all blank characters. It does this by setting all digits to special decode mode, and writing the data for the blank character to all digits.

The `DspUpdateDisplay()` function uses the Neurowire I/O model to write the contents of the RAM copy of the configuration and display registers to the actual MC14489 device registers. The Neurowire device is full-duplex, so that the `io_out()` operation which updates the hardware registers in the MC14889 also causes data to be shifted in from the IO10 pin and stored in the RAM variables.

Therefore a local copy of these variables is used for the `io_out()` operation, so that `DspUpdateDisplay()` may be called repeatedly without having to refresh the RAM copy of the variables.

Display Image Update Functions

The second group of functions are routines that write into the RAM copy of the configuration and display registers. They do not update the hardware device registers.

The `DspInsertDecimal(int digitNumber, int number)` function writes the specified decimal (0 - 9) into the specified digit position in the RAM copy of the display register.

The `DspInsertDecimal2(int rightDigit, unsigned number)` function writes a two-digit decimal number (00 - 99) into the specified digit positions in the RAM copy of the display register.

The `DspInsertMinus(int digitNumber)` function writes the appropriate values in the RAM copy of the display register to illuminate the minus sign (segment G) in the specified digit.

The `DspInsertChar(int digitNumber, char ch)` function writes the data value for an ASCII character in the RAM copy of the display register. The letters available in upper case are "A, C, E, F, H, I, J, L, O, P, S, U, Y, Z", and in lower case "b, c, d, h, l, n, o, r, u, y". Digits 0-9 are displayed, as well as the space ' ', degree '°', minus '-', and equals '=' special characters. If the ASCII character is none of these characters, nothing is written, leaving the display unchanged for that digit.

The `DspInsertNumber(long number, int dpDigit, int rightDigit)` function updates the RAM copy of the display register to display a signed decimal number. The function illuminates a decimal point to the right of the specified digit. If the `dpDigit` parameter is `NO_DP`, no decimal point is illuminated. If the `dpDigit` parameter is `ALL_DPS`, all decimal points will be illuminated. The caller also specifies the right-most digit position of the displayed number. Display data to the right of this position are unchanged. If the number to be displayed does not fit in the specified field, all digits are set to the minus character.

High-Level Functions

The third group of functions forms complete display images and updates the display hardware.

The `DspDisplayBlanks()`function clears the display.

The `DspDisplayString(const char * pString, int dpDigit)` function causes the first 4 or 5 ASCII characters in the specified string to be displayed. Five characters are displayed for a Gizmo 3, and four characters are displayed for a Gizmo2. The letters available in upper case are "A, C, E, F, H, I, J, L, O, P, S, U, Y, Z",

and in lower case "b, c, d, h, l, n, o, r, u, y". Digits 0-9 are displayed, as well as the space ' ', degree '°', minus '-', and equals '=' special characters. If other letters, or more elegant letters are desired, an alphanumeric display should be used instead of a seven-segment display.

The `DspDisplayNumber(long number, int dpDigit, int rightDigit)` function displays positive or negative decimal numbers with a decimal point to the right of the specified digit, with suppression of leading zeroes. The special values `NO_DP` and `ALL_DPS` may be used as the decimal point digit number to illuminate none or all, respectively, of the decimal points. The parameter `rightDigit` indicates the digit position for the least significant digit of the displayed number. Numbers that do not fit into the specified field are displayed as all minus characters '`-----`'.

The following examples show the display produced by different input values on a five-digit display.

Examples:

```
DspDisplayNumber(1234, 0, 0)        =>     1 2 3 4.
DspDisplayNumber(1234, 1, 0)        =>     1 2 3.4
DspDisplayNumber(1234, 2, 0)        =>     1 2.3 4
DspDisplayNumber(1234, 3, 0)        =>     1.2 3 4
DspDisplayNumber(1234, 4, 0)        =>   0.1 2 3 4
DspDisplayNumber(1234, NO_DP, 0)    =>     1 2 3 4
DspDisplayNumber(1234, ALL_DPS, 0)  =>    .1.2.3.4.

DspDisplayNumber(-1234, 0, 0)       =>   - 1 2 3 4.
DspDisplayNumber(-1234, 1, 0)       =>   - 1 2 3.4
DspDisplayNumber(-1234, 2, 0)       =>   - 1 2.3 4
DspDisplayNumber(-1234, 3, 0)       =>   - 1.2 3 4
DspDisplayNumber(-1234, 4, 0)       =>   -.1 2 3 4
DspDisplayNumber(-1234, NO_DP, 0)   =>   - 1 2 3 4
DspDisplayNumber(-1234, ALL_DPS, 0) =>   -.1.2.3.4.
```

The `DspDisplayTemp(SNVT_temp temp, boolean dspFahrenheit)` function displays temperature values in either Celsius or Fahrenheit, with one decimal place. For more details on the Standard Network Variable Type `SNVT_temp`, see the *SNVT Master List and Programmer's Guide.*

Examples:

`DspDisplayTemp(2940, FALSE)` displays '`20.0C`'

`DspDisplayTemp(2940, TRUE)` displays '`68.0F`'

LONTALK™ Protocol

April 1993

LONWORKS™ Engineering Bulletin

Overview

The LONTALK Protocol is designed to support the needs of applications spanning a range of industries and requirements. To meet its broad objectives, the protocol is presented to programmers and installers as a collection of services that may be optionally invoked. Services may be chosen by the programmer and fixed at compile time. In addition, many of the service choices may be changed by an installer when a node is installed or reconfigured in a particular LONWORKS application.

The LONTALK Protocol follows the reference model for open systems interconnection (OSI) developed by the International Standard Organization (ISO). In the terminology of the ISO the LONTALK Protocol provides services at all 7 layers of the OSI reference model as shown below:

- Physical channel management (layers 1 and 2)
- Naming, addressing, and routing (layers 3 and 6)
- Reliable communications and efficient use of channel bandwidth (layers 2 and 4)
- Priority (layer 2)
- Remote actions (layer 5)
- Authentication (layers 4 and 5)
- Network management (layer 5)
- Network interface (layer 5)
- Data interpretation and foreign frame transport (layer 6)
- Application Compatibility (layer 7)

The details of these services are described in this chapter. Table 1 summarizes the OSI reference model layers and the LONTALK services provided at each layer.

Echelon Part Number 005-0017-01 REV C

Table 1 LONTALK Protocol Layering

	OSI Layer	Purpose	Services Provided
7	Application	Application Compatibility	Standard Network Variable Types
6	Presentation	Data Interpretation	Network Variables; Foreign Frame Transmission
5	Session	Remote Actions	Request-Response; Authentication; Network Management; Network Interface
4	Transport	End-to-End Reliability	Acknowledged & Unacknowledged; Unicast & Multicast Authentication; Common Ordering; Duplicate Detection
3	Network	Destination Addressing	Addressing Routers
2	Link	Media Access and Framing	Framing; Data Encoding; CRC Error Checking; Predictive CSMA; Collision Avoidance; Optional Priority & Collision Detection
1	Physical	Electrical Interconnect	Media-Specific Interfaces and Modulation Schemes (twisted pair, power line, radio frequency, coaxial cable, infrared, fiber optic)

The Physical Channel

The LONTALK protocol supports networks with segments using differing media. The media supported by the LONTALK protocol include twisted pair, power line, radio frequency, infrared, coaxial cable, and fiber optics. The specifications for each LONWORKS transceiver provides the distance, bit rates, and topologies supported.

Every LONWORKS node is physically connected to a channel. A channel is a physical transport medium for packets; a LONWORKS network is composed of one or more channels. The physical form of a channel depends on the medium. For example, a twisted pair channel is a twisted pair wire; an RF channel is a specific radio frequency; a power line channel is a contiguous section of AC power wiring.

Multiple channels are connected by routers. Routers are installation devices that connect two channels, and route packets between them. Routers can be installed to use one of four routing algorithms: configured router, learning router, bridge, or repeater. Configured routers and learning routers are a class of router known as intelligent router. See *Routers*, later in this document.

A set of channels connected by bridges or repeaters is a segment. A node sees every packet from every other node on its segment. Intelligent routers can be used to

isolate traffic within a segment to increase total system capacity and improve reliability.

The bit rate of a channel is dependent upon the medium and transceiver design. Multiple transceivers with different bit rates may be designed for a medium to allow trade-offs of distance, throughput, and node power consumption and cost.

The transaction throughput supported by a given channel is limited by several factors in addition to the channel bit rate. At low bit rates or with long packets, the packet transmission time and average media access delay form the bounds of packet throughput. At higher bit rates with short packets, the packet processing power of the NEURON CHIP limits channel performance.

Tables 2 and 3 estimate the approximate network throughput as a function of bit rate and packet size. The "peak" traffic numbers can be supported for short bursts.

See the following engineering bulletins for further information:

Engineering Bulletin	Description
LONTALK Response Time Measurements	Characterization of LONWORKS network performance
Enhanced Media Access Control with Echelon's LONTALK Protocol	Describes the media access sublayer of the LONTALK protocol
Optimizing LONTALK Response Time	Guidelines on optimizing response time

Table 2 LONTALK Protocol Channel Throughput — 12 byte packets

Bit rate (kbps)	Peak Number of Packets/Sec	Sustained Number of Packets/Sec
4.883	25	20
9.766	45	35
19.531	110	85
39.063	225	180
78.125	400	320
156.25	625	500
312.5	700	560
625.0	700	560
1,250.0	700	560

Table 3 LONTALK Protocol Channel Throughput — 64 byte packets

Bit rate (kpbs)	Peak Number of Packets/Sec	Sustained Number of Packets/Sec
4.883	7	5
9.766	13	10
19.531	25	20
39.063	50	40
78.125	100	80
156.25	200	160
312.5	340	270
625.0	500	470
1,250.0	700	560

Naming, Addressing, and Routing

A name is an identifier that uniquely identifies a single object within an object class. A name is assigned when an object is created and does not change over its lifetime. The 48-bit NEURON ID is a name for a NEURON CHIP because it uniquely distinguishes a NEURON CHIP from all other NEURON CHIPs, and does not change over the lifetime of the NEURON CHIP.

An address is an identifier that uniquely identifies an object or group of objects within an object class. Unlike a name, an address may be assigned and changed any time after an object is created.

LONTALK addresses uniquely identify the source node and destination node (or nodes) of a LONTALK packet. These addresses are also used by routers to selectively pass packets between two channels.

A NEURON ID may be used as an address. However, the NEURON ID is not used as the sole form of addressing in the LONTALK protocol because such addressing only supports one-to-one transactions (i.e., no groups), and would require excessively large routing tables to optimize network traffic. This addressing mode is used primarily during installation and configuration, since it allows communications before they have been assigned an address.

To simplify routing, the LONTALK protocol defines an hierarchical form of addressing using domain, subnet, and node addresses. This form of addressing can be used to address the entire domain, an individual subnet, or an individual node. To further facilitate the addressing of multiple dispersed nodes, the LONTALK protocol defines another class of addresses using domain and group addresses.

This also simplifies replacement of nodes in a functioning network. The replacement node is assigned the same address as the node it replaces. Thus all

references to this node from elsewhere on the network do not need to be modified, as would be the case if NEURON ID addressing were to be used.

The various address forms are described in the following sections, along with discussions on routers and address generation.

The Domain Address Component

A domain is a logical collection of nodes on one or more channels. Communications can only take place among nodes configured in a common domain; therefore, a domain forms a virtual network. Multiple domains can occupy the same channels, so domains may be used to prevent interference between nodes in different networks.

For example, two adjacent buildings using nodes with RF transceivers on the same frequency would be on the same channel. To prevent interference between the applications carried out by the nodes, the nodes in each building would be configured to belong to different domains.

The NEURON CHIP may be configured so that a LONWORKS node may belong to one or two domains. A node that is a member of two domains may be used as a gateway between the two domains. The LONTALK protocol does not support communications between domains, but an application program may be implemented to forward packets between two domains.

A domain is identified by a domain ID. The domain ID may be configured as 0, 1, 3, or 6 bytes. Six byte domain IDs can be used to ensure that the domain ID is unique: for example, using the 48-bit NEURON ID of one of the NEURON CHIPs in the domain as the domain ID ensures that no other network can have the same domain ID, since all NEURON IDs are unique. However, six byte domain IDs add six bytes of overhead to every packet. The overhead may be reduced by using a shorter domain ID. In a system where there is no possibility of interference between multiple networks, the domain ID may be configured as zero bytes. For example, LONWORKS applications using twisted-pair channels may be configured with zero-byte domains if only one application will be using the twisted pair channels. Domain IDs may be configured as 1 or 3 bytes in systems where a single administrator controls assignment of domain IDs to prevent duplicate IDs.

The domain ID can also be used for application-level purposes. For example, a domain ID could be used by service personnel as a system identifier.

The Subnet Address Component

A subnet is a logical collection of up to 127 nodes within a domain. Up to 255 subnets can be defined within a single domain. All nodes in a subnet must be on the same segment. Subnets cannot cross intelligent routers.

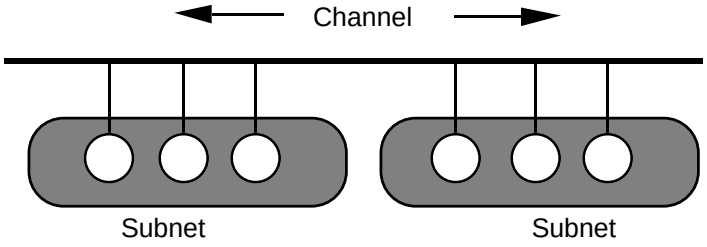


Figure 1 Multiple subnets on a common channel

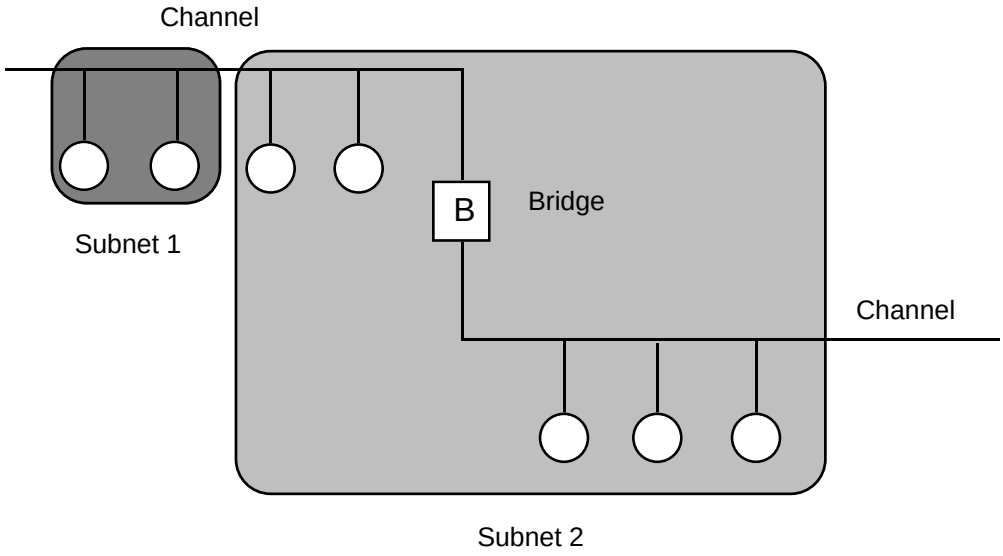


Figure 2 Multiple subnets on a common segment

If a node is configured to belong to two domains, it must be assigned to a subnet within each of the domains.

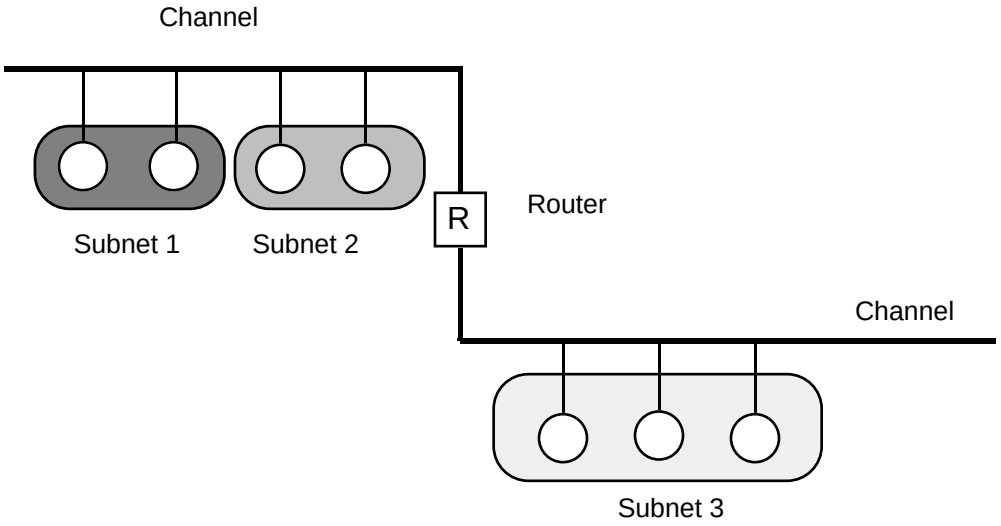


Figure 3 Subnets cannot span intelligent routers

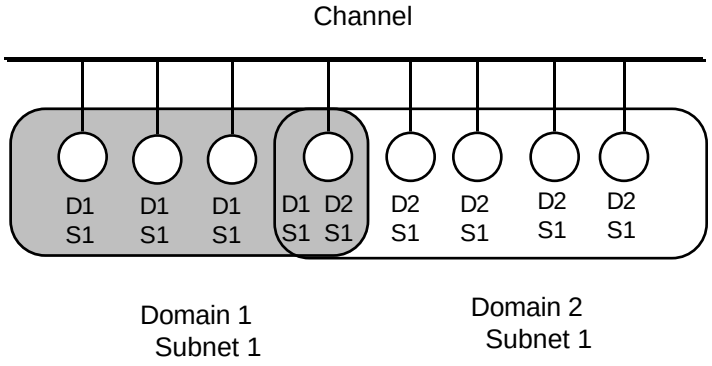


Figure 4 A node configured in 2 domains has subnet assignments for each domain

All nodes within a domain are typically configured in the same subnet except in the following cases:

- They are located on different segments with intervening intelligent routers. Since subnets cannot cross intelligent routers, the nodes must be on different subnets.
- Configuring the nodes in the same subnet would exceed the maximum number of nodes allowed in a subnet. Subnets are limited to 127 nodes. Multiple subnets may be configured on a segment to increase the capacity of the segment above 127 nodes. For example, a segment with two subnets may have up to 254 nodes; three subnets may have up to 381 nodes.

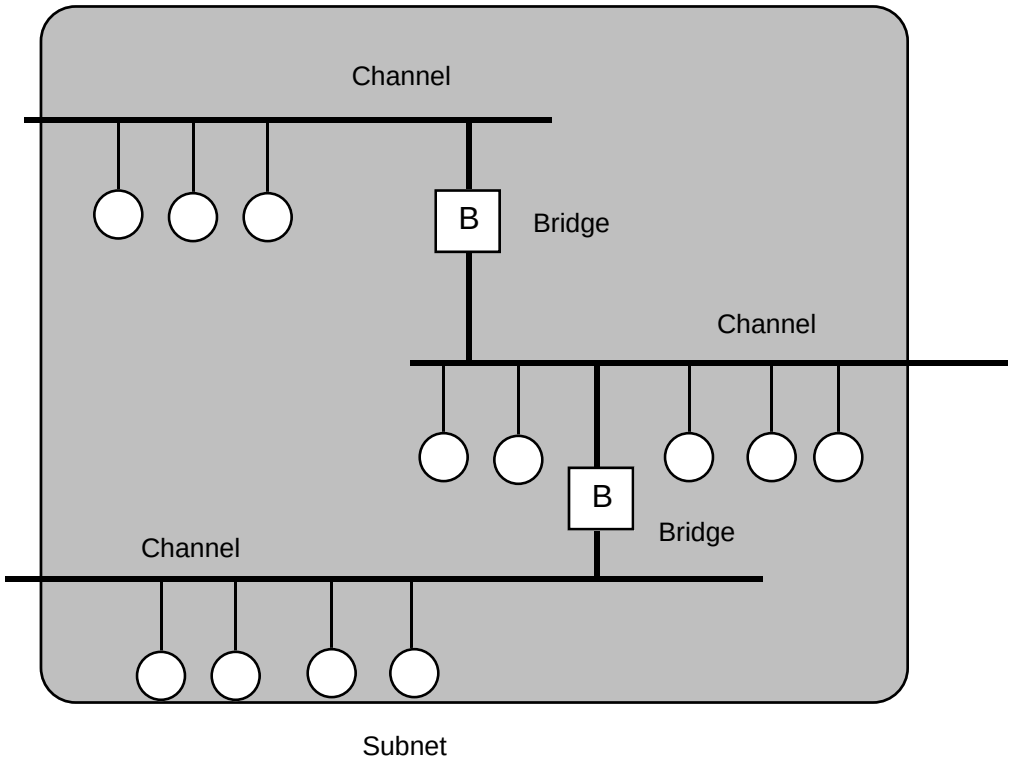


Figure 5 All nodes in one domain configured as a single subnet

The Node Address Component

Every node within a subnet is assigned a unique node number within that subnet. The node number is 7 bits, so there may be up to 127 nodes per subnet. A maximum of 32,385 nodes (255 subnets x 127 nodes per subnet) may be in a single domain.

Groups

A group is a logical collection of nodes within a domain. Unlike a subnet, however, nodes are grouped together without regard for their physical location in the domain. The NEURON CHIP allows a node to be configured to be a member of up to 15 groups.

Groups are an efficient way to use network bandwidth for one-to-many network variable and message tag connections.

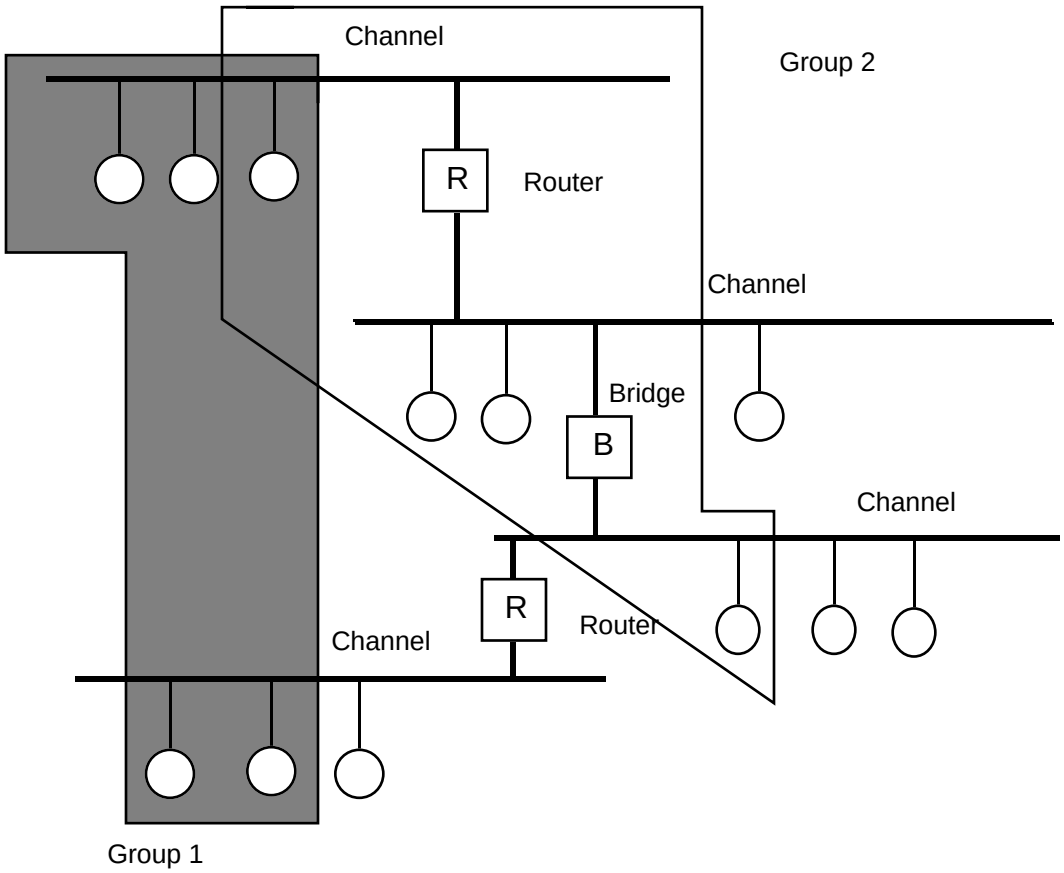


Figure 6 Group membership can span any number of channels, routers, or bridges

Groups are identified by a one byte group number, so a single domain may contain up to 256 groups.

NEURON ID

In addition to the Subnet / Node address, a node may always be addressed by its NEURON ID. The NEURON ID is 48 bits long, and is assigned when each NEURON CHIP is manufactured. This ID is guaranteed to be unique world-wide.

The *Domain/NEURON ID* addressing format is used by a network management tool in the initial configuration of nodes at installation time to assign each node to one or two domains, and to assign subnets and node numbers. Domain / NEURON ID addressing should not be used for application messages except in applications where node replacement and network capacity are not an issue, such as asset management and inventory control functions.

Addressing Formats

Nodes are addressed using one of five addressing formats. The particular addressing format used determines the number of bytes required for the source and destination address. Table 4 defines the formats and number of bytes required for each. The total address size is computed by adding the appropriate number of bytes indicated in the table to the size of the domain ID, which can range from 0 to 6 bytes depending on the configured size of the domain ID.

Table 4 LONTALK Protocol Address Formats

Address Format	Destination	Address Size (bytes)
Domain (Subnet = 0)	All nodes in the domain	3
Domain, Subnet	All nodes in the subnet	3
Domain, Subnet, Node	Specific node within a subnet	4
Domain, Group	All nodes in the group	3
Domain, NEURON-ID	Specific node	9

Network Management and Address Generation

Depending on the level of a given application, a LONWORKS network may or may not require the use of a network management tool. A network management tool is a node that has been specifically designated to perform network management functions, such as:

- Find unconfigured nodes and download network addresses
- Stop, start, and reset node applications
- Access node communication statistics
- Configure routers
- Download new application programs
- Extract the topology of a running network

In a development environment, the role of the network management tool is typically performed by the LONBUILDER™ Network Manager. The LONBUILDER Network Manager includes the tools required to define, configure, load, and control LONWORKS networks. The LONBUILDER Protocol Analyzer provides capability to monitor, collect, and display network traffic and performance statistics.

In a production system, the role of the network management tool is typically performed by the LONMANAGER NetMaker installation tool, using a customized installation profile defined by the LONMANAGER NetProfiler software.

LONBUILDER and NetMaker create the connections between nodes and assign all groups automatically, in a manner as to optimize network traffic.

Routers

Routers are installation devices that connect two channels and route packets between them. Routers can be installed to use one of four routing algorithms:

- **Repeater**. A repeater is the simplest form of router, simply forwarding all packets between the two channels. Using a repeater, a subnet can exist across multiple channels.
- **Bridge**. A bridge simply forwards all packets which match its domains between the two channels. Using a bridge, a subnet can exist across multiple channels.
- **Learning Router**. A learning router monitors the network traffic and learns the network topology at the domain/subnet level. The learning router then uses its knowledge to selectively route packets between channels. Learning routers cannot learn group topology, so all packets using group addressing are forwarded.
- **Configured Router**. Like a learning router, a configured router selectively routes packets between channels by consulting internal routing tables. Unlike a learning router, the contents of the internal routing tables are defined by a network management tool.

  A network management tool can optimize network traffic by defining routing tables for both subnet and group address routing.

Configured routers and learning routers are a class of routers known as learning routers.

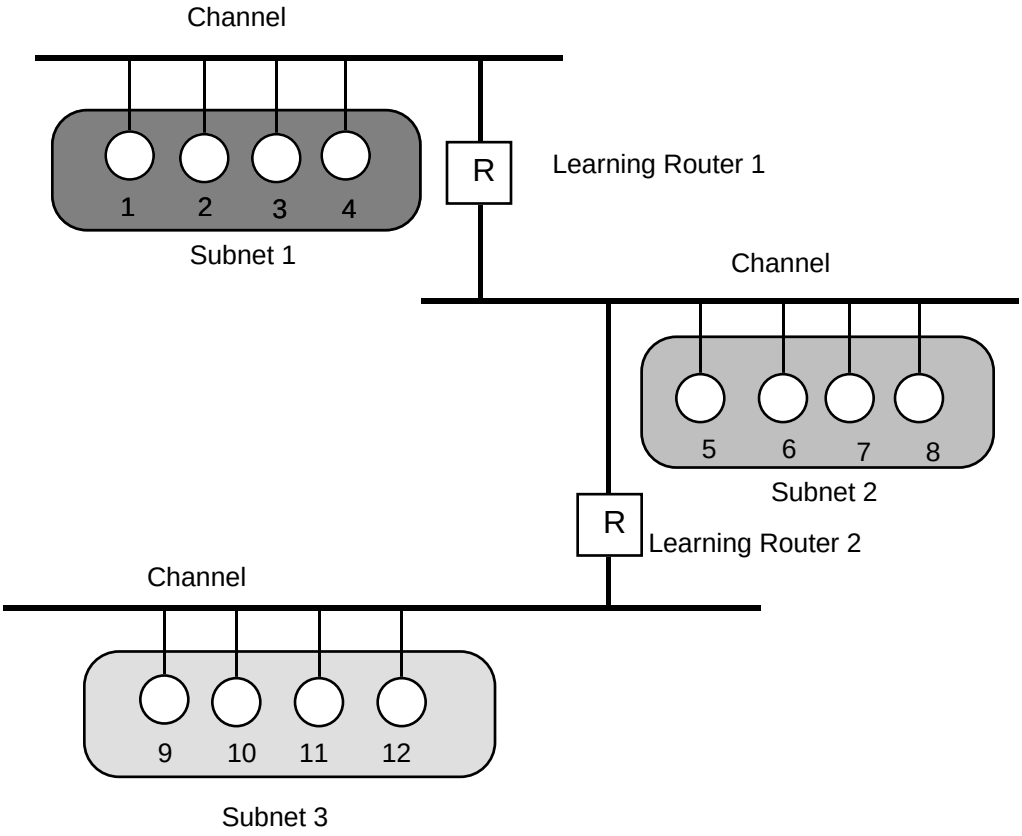


Figure 7 Learning Routers

Choosing Between Learning and Configured Routers

Initially, each learning router sets its internal routing tables to indicate that all subnets could lie on either side of the router. Referring to figure 7, suppose that node 6 generates a message bound for node 2. Learning router 1 initially picks up the message. Examining the source subnet field of the message, the learning router notes in its internal routing tables that subnet 2 lies below it. The router then compares the source and destination subnet IDs, since they are different, the message is passed on.

Meanwhile, learning router 2 has also passed on the message, making an appropriate notation in its internal routing tables regarding the location of subnet 2.

Suppose now that node 2 generates an acknowledgement. This acknowledgement is picked up by learning router 1, that now notes the location of subnet 1. Learning router 1 examines its internal routing tables, and, upon discovering that subnet 2 lies below, passes the message on. When the message appears on subnet 2, it is noted by both node 6 (the destination node), and learning router 2, who does not pass it on but merely notes that subnet 1, like subnet 2, lies somewhere above.

Learning router 2 will not learn of the existence or location of subnet 3 until a message is originated from there.

When choosing between learning and configured routers, the following should be taken into account:

- The initial flood of traffic that occurs while a learning router is learning the network topology may cause congestion problems.
- The network topology may have inadvertent "loops," common in power line and RF networks, that can cause a learning router to develop an inaccurate network image.
- A learning router is always learning, and will update its internal routing tables to follow changes in network topology.
- The internal routing tables in a learning router do not have to be explicitly programmed.
- Learning routers do not learn about groups but configured routers can be configured to selectively forward group addressed packets.

Subnets cannot cross intelligent routers. While bridges and repeaters allow subnets to span multiple channels, the two sides of an intelligent router must belong to separate subnets. The fact that intelligent routers are selective about the packets they forward to each channel can be used to increase the total capacity of a system in terms of nodes and connections. In general, it is always a good idea to segment traffic among "communities of interest" if possible.

Communications Services: Efficiency, Response Time, Security, and Reliability

There are a number of tradeoffs between network efficiency, response time, security, and reliability: using acknowledged service is most reliable, but uses greater network bandwidth than unacknowledged or unacknowledged repeated service for large groups; prioritizing packets will ensure that those packets will be sent in a timely fashion, to the detriment of others; adding authenticated service to designated transactions adds a level of security, but requires twice the number of packets to complete a transaction than non-authenticated transactions.

Selecting Message Services for Reliability and Efficiency

The LONTALK protocol offers four basic types of message service:

The most reliable service is *acknowledged*, or end-to-end acknowledged service, where a message is sent to a node or group of nodes and individual acknowledgements are expected from each receiver. If an acknowledgement is not received from all destinations, the sender times out and re-tries the transaction. The number of re-tries and the time-out are both selectable (see *LONTALK Protocol Timers*, later in this document). The acknowledgements are generated by the

network CPU without intervention of the application. Transaction IDs are used to keep track of messages and acknowledgements so that the application does not receive duplicate messages.

An equally reliable service is *request/response*, where a message is sent to a node or group of nodes and individual responses are expected from each receiver. The incoming message is processed by the application on the receiving side before a response is generated. The same retry and time-out options are available as with acknowledged service. Responses may include data, so that this service is particularly suitable for remote procedure call, or client/server applications.

The next most reliable is *repeated*, or unacknowledged repeated service, where a message is sent to a node or group of nodes multiple times, and no response is expected. This service is typically used when broadcasting to large groups of nodes, in which the traffic generated by all the responses would overload the network.

The least reliable is *unacknowledged*, where a message is sent once to a node or group of nodes and no response is expected. This is typically used when the highest performance is required, network bandwidth is limited, and the application is not sensitive to the loss of a message.

Collision Detection

The LONTALK protocol uses a unique collision avoidance algorithm which has the property that under conditions of overload, the channel can still carry close to its maximum capacity, rather than have its throughput degrade due to excess collisions.

When using a communications medium that supports hardware collision detection (twisted pair, for example), the LONTALK protocol can optionally cancel transmission of a packet as soon as a collision is detected by the transceiver. This allows the node to immediately retransmit any packet that has been damaged by a collision. Without collision detection, the node would have to wait the duration of the retry time to notice that no acknowledgement was received — at which time it would retransmit the packet, assuming acknowledged or request/response service. For unacknowledged service, an undetected collision means that the packet is not received and no retry is attempted.

Priority

The LONTALK protocol optionally offers a priority mechanism to improve the response time of critical packets. The protocol permits the user to allocate priority time slots on a channel, dedicated to priority nodes. A network management tool that assigns priority slots to individual nodes can ensure that one and only one node is assigned to a particular priority slot on the channel. Each priority time slot on a channel adds a minimum of two bit times to the transmission of every message. The amount of overhead will vary based upon the bit rate, oscillator accuracy, and transceiver requirements. For example, using a TP/XF-1250 1.25 Mbps

twisted pair transceiver with all nodes on the channel having an oscillator accuracy of 0.2% or better, each priority slot is 30 bit times wide. Because there is no contention for the media during the priority portion of a packet cycle, nodes configured with priority have better response time than non-priority nodes. The combination of priority and collision detection allows for bounded response time.

The priority slot assigned to a node applies to all priority packets sent from that node. One, all, or some of the packets sent from a node may be marked as using priority service. The priority designation within a node is made on a per network variable or per message tag basis, and may be set at compile time. In the case of network variables, the priority designation can optionally be changed during or after installation.

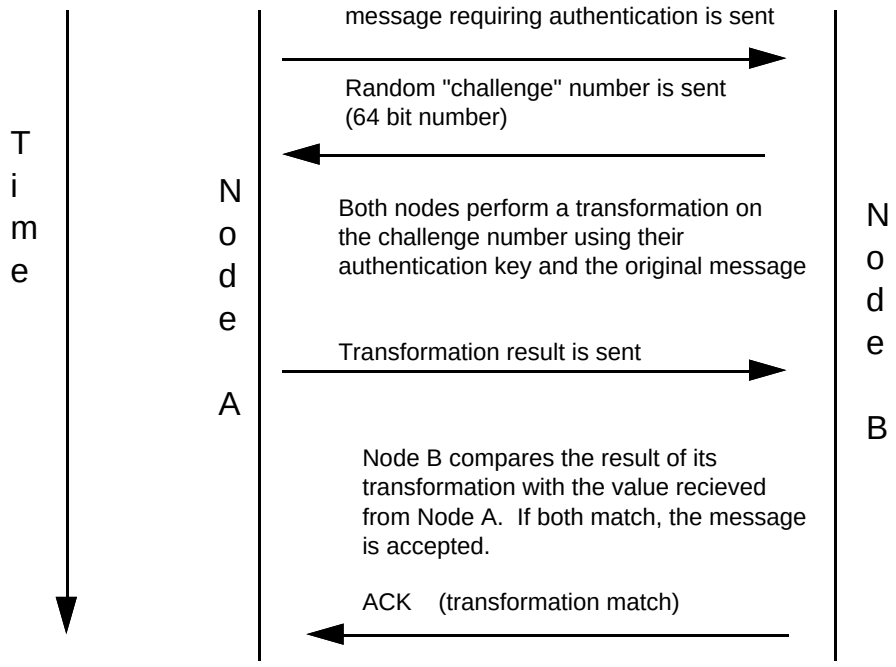
Lower priority numbers indicate higher levels of priority: a priority packet from a node with priority slot #2 will be transmitted before a priority packet from a node with priority slot #4. Setting a node's priority to 0 indicates that none of its packets will be transmitted in a priority slot, regardless of the message service assignment made at node compilation or installation time. Slot #1 is reserved for a network management tool, to ensure that no application can render a channel incapable of interruption by a network management tool. Slots 2 through 127 (depending on the medium, and the number of slots allocated on the channel) are then available for prioritized packets from designated nodes.

When a priority packet is generated within a node, it travels out of the node on the priority queue, ahead of any pending non-priority packets buffered for transmission. Similarly, when a priority packet reaches a router, it goes to the head of the router queue (behind any other queued priority packets) and is forwarded to the far channel using the router's priority slot if one has been configured.

Authentication

The LONTALK protocol supports authenticated messages; the receivers of an authenticated message determine if the sender is authorized to send that message. This can prevent unauthorized access to nodes and their applications. The use of authentication is configured individually for each network variable. Network management transactions may also be optionally authenticated.

Authentication is implemented by distributing 48-bit keys to the nodes at installation time, with the sender and receiver of an authenticated message both possessing the same key. When an authenticated message is sent, the receiver challenges the sender to provide authentication, each time using a different random challenge. The sender then uses the authentication key and the data from the original packet to perform a transformation on the challenge, and responds. The receiver compares the reply to the challenge with its own transformation on the challenge. If the transformations match, the transaction succeeds. The transformation used is designed so that is is extremely difficult to deduce what the key is, even if the challenge and response are both known.



LONTALK Protocol Timers

There are several timers which need to be properly set in order for the LONTALK protocol to function efficiently. These include:

- Transaction Timer
- Repeat Timer
- Group Receive Timer
- Non-group Receive Timer
- Free-buffer Wait Timer

These times are collectively known as the Layer 4 Timers, and are typically automatically configured by a network management tool. LONBUILDER and NetMaker automatically calculate and configure these timers.

The following section details, for each of the four messaging services, how packets flow through the NEURON CHIP and where the timers come into play.

Packet Flow through the NEURON CHIP

There are four different messaging services in the LONTALK protocol: Unacknowledged, Acknowledged, Repeated, and Request/Response. Usually, these services can be used with any addressing mode: domain-wide broadcast, unicast,

multicast, or NEURON ID. There are two exceptions and caveats: first, when performing a broadcast request/response, the application will receive only the first response; all others will be discarded by the network CPU. Second, broadcast acknowledged transactions complete once a single acknowledgment is received.

Unacknowledged: When this service is used, the only timer that is involved is the free buffer wait timer. This timer determines the maximum length of time the node will wait for a free buffer when sending a message. This timer can be deactivated (the node will wait forever) by setting the timer value to zero. If it is set to another number, n, then the node will wait between 2n and 2n + 1 seconds. For example, if the configured number n is set to 2, then the node will wait for a free buffer for between 4 and 5 seconds. If a buffer is not obtained before the timer expires, the node assumes a fatal error and resets.

Acknowledged: Acknowledged service also uses the free buffer wait timer, but additional timers are necessary. Some additional timers are also involved:

1. The *transaction timer* determines how long the node waits for an acknowledgement before retrying. The value of the transaction timer used by the node is taken from the transmitting node's address table entry for the destination address of the packet being sent. The transaction timer is individually configurable by destination address in the address table. If the node does not receive an acknowledgement before the transaction timer expires, it will retry, sending the same packet again (along with an indication of which nodes did acknowledge, in the case of group addressing). This retry process will continue until the *retry count* has been exhausted or until all acknowledgements have been received. The retry count is configurable from 0 to 15 by address table entry.

   Note that a packet may go through routers to reach its final destinations. The transaction timer should be just long enough so that a packet can reach the "furthest" destination and the acknowledgement from this destination can be received before the transaction timer expires. If the transaction timer is too short, excess retries will be generated; if too long, the time for a transaction to complete will increase on average.

   LONBUILDER and NetMaker automatically calculate the transaction time defaults based upon topology, bit rate, and the NEURON CHIP input clock frequency.

2. When a packet arrives at its final destinations, the receiving nodes look at the packet's source address and transaction ID. If no "receive transactions" are active with this source address/transaction ID pair, a new receive transaction record is created. If no receive transaction record exists because the node has used them all up with active transactions, the incoming message is lost. Assuming that the receiving node can allocate a receive transaction record, it starts a *receive timer*. It chooses which receive timer to use based upon the address mode that the transmitter used. If the transmitter used group addressing, there exists an address table entry for that group, and the *group receive timer* value is taken

from that entry in the address table. If any other addressing mode is used, the node uses its *non-group receive timer* value.

When the receive timer expires, the transaction record is deleted, and any new transmission having the same transaction ID from the same source address will be treated as a new transaction. Therefore, this timer must be greater than the greatest product of retry count and transaction timer that can be received from the transmitter. A good rule of thumb for setting this timer is *((retry count + 2) * transaction timer)*.

If the receive transaction timer is too long, then it is likely that the node will run out of memory for receive transaction buffers. If it is too short, then the node may mistake legitimate retries for new transactions, causing duplicate messages to mistakenly be passed on to the application for processing. A good rule of thumb is to keep the retry count low (say, 4) and design networks with as few end-to-end hops as possible; this will keep the transaction timer short. As an example, for a received message that originates from a node with a retry count of 4 and a transaction timer of 200 milliseconds, use the rule-of-thumb above to arrive at ((4 + 2) * 200), or 1200 milliseconds. A shorter value could result in retries being interpreted as new transactions; a longer value could result in a node's running out of receive transaction buffers and losing incoming messages.

Repeated: This service follows essentially the same message flow as acknowledged service, with some exceptions. In the address table of the transmitter there is a separate timer known as the *repeat timer*. This timer specifies how frequently the message is repeated when using repeated service. This time can be shorter than the transaction timer, because no acknowledgement is expected (no time for the acknowledgement need be allotted) when these messages are sent. Transaction IDs and duplicate detection are in effect for these transactions. The transmitter sends the message *n* times at intervals of *m* milliseconds, where *n* is the retry count and *m* is the repeat timer value.

Request/Response: The message flow for this service is identical to acknowledged service, except that the application sends a response in lieu of an acknowledgement. The fact that the application program adds extra processing time to the generation of the acknowledgement should be taken into account when setting the transaction and receive timers.

Network Management Services

The LONTALK protocol provides network management services for installing and configuring nodes, downloading software, and diagnosing the network. Message types in the network management and diagnostic class are summarized in table 5.

Table 5. Network Management Messages

Message	From → To	Action	Comments
Query ID*	Net Mgr →Broadcast	report node's unique 48-bit NEURON ID	Used to get NEURON IDs of nodes that are configured or unconfigured, or have a matching node type
Response to Query*	Net Mgr →Node— or — Net Mgr →Broadcast	set node's "response to Query ID" state	used during installation and data base recovery
Update Domain	Net Mgr →Node	assign a node to a domain	propagates encryption key in the clear
Leave Domain	Net Mgr →Node	remove a node from a domain	used to uninstall a node
Security	Net Mgr →Node	add an increment to the current encryption key to form a new key	does not send encryption key directly, for security
Modify Address Table	Net Mgr →Node	change an address table entry on the node	sets address and timer info. No check performed for duplicate addresses or groups.
Report Address	Net Mgr →Node	report an entry in the address table	response includes address and timer info
Report Net Variable	Net Mgr →Node	report an entry in the network variable configuration table	response includes NVID, priority, direction, service, security
Update Address Data	Net Mgr →Node	update a group entry in the address table	updates group size, timers, retry count
Report Domain	Net Mgr →Node	reports domain info	response includes encryption key
Modify Net Variable	Net Mgr →Node	add or modify entries in the network variable configuration table	sets priority, direction, NVID, service, security
Set Node Mode	Net Mgr →Node	puts the node's application in off-line or on-line state, or resets the node	result should be verified with status request. Typically used to suspend application during EEPROM downloading.
Read Memory	Net Mgr →Node	read any memory location in the node	application code, NV Fixed table, and Event table can be read protected

Write Memory	Net Mgr →Node	write any memory location in the node (subject to write permission)	can be used to download an application into a node. Can restart the NEURON CHIP after writing. Confirm restart with read memory request. A single write should not cross an EEPROM memory boundary.
Checksum Recalculate	Net Mgr →Node	recompute EEPROM checksum	checksum computed over network and application images
Wink	Net Mgr →Node	cause node to execute wink clause in application program	used to identify a node installed on a network but not yet configured
Memory Refresh	Net Mgr →Node	refresh memory	used to extend data retention time of EEPROM
SNVT Fetch	Net Mgr →Node	retrieve SNVT information	gets SNVT information from a node where the NEURON CHIP is a communication chip attached to a mircoprocessor containing the node's SNVT information
Network Variable Value Fetch	Net Mgr →Node	retrieve variable information	used to poll network variables by NV index
Status*	Net Mgr →Node	retrieve network error statistics accumulators	response contains: #transaction errors, #transaction timeouts, #times no receive transaction memory was available, #lost messages, #missed messages, cause of most recent reset, node state, ROM version, most recent error, and model number.
Clear Status	Net Mgr →Node	clear network error statistics accumulators	clears statistics info, reset cause, error log
Proxy Command*	Net Mgr →Node A'Node B	request node to deliver command to another node	Net Mgr sends command for node A to send a request to node B. Node B sends response to A, which in turn responds to Net Mgr.
Retrieve Transceiver Status	Net Mgr →Node	retrieve transceiver status registers	response consists of the contents of all 7 status registers in associated transceiver
Set Router Mode	Net Mgr →Router	set router temporary bridge status and initialize routing tables	used for router installation

Group or Subnet Table Clear	Net Mgr →Router	clear all entries in either forwarding table	used for router installation
Group or Subnet Table Download	Net Mgr →Router	configure a forwarding table	used for router installation
Group Forward	Net Mgr →Router	sets the forwarding flag for a specified group	used for router installation
Subnet Forward	Net Mgr →Router	sets the forwarding flag for a specified subnet	used for router installation
Group No Forward	Net Mgr →Router	clears the forwarding flag for a specified group	used for router installation
Subnet No Forward	Net Mgr →Router	clears the forwarding flag for a specified subnet	used for router installation
Group or Subnet Table Report	Net Mgr →Router	reports current settings of either forwarding table	used for router installation
Router Status	Net Mgr →Router	retrieve routing algorithm	used for router installation
Far Side Escape Code	Net Mgr →Router	pass network management message to far side of router	used for installation of the far side of a router
Send Modem String	Node →SLTA 2	send modem commands	used by an application to control a remote modem
Modem Status Readback	Node →SLTA 2	test the state of the modem RS-232 signals	used by an application to control a remote modem
Modem Response Query	Node →SLTA 2	retrieve the last response from the modem	used by an application to control a remote modem
Connection Status Query	Node →SLTA 2	poll status of modem connection	used by an application to control a remote modem
Install Directory Entry	Node →SLTA 2	install a dial-out string in the SLTA dialing directory	used by an application to control a remote modem
Dial From Directory	Node →SLTA 2	dial-out using a dialing directory entry	used by an application to control a remote modem
Hang Up	Node →SLTA 2	hang up the modem	used by an application to control a remote modem
Install Password	Node →SLTA 2	set the dial-in password	used by an application to control a remote modem
Install Start-Up Configuration String	Node →SLTA 2	install a modem configuration string	used by an application to control a remote modem

Install Dial Prefix	Node →SLTA 2	install a modem dialing prefix	used by an application to control a remote modem
Install Hangup String	Node →SLTA 2	install a modem hang-up string	used by an application to control a remote modem
Configure Modem	Node →SLTA 2	send modem the configuration string	used by an application to control a remote modem
Product Query	Node →SLTA 2	poll SLTA product type	used by an application to control a remote modem

A system may be configured so that the network management messages listed above (except those marked with an *) are subject to authentication protection independent of application-level messages. This means that only authorized network manager tools may request these functions.

The Modify Address Table and Modify Net Variable messages may be used to dynamically connect network variables and message tags. This is used during installation and reconfiguration to establish the addressing information needed to route messages and network variable updates between nodes.

Network Interface

The LONTALK protocol includes an optional network interface protocol that can be used to support LONWORKS applications running on any host processor. A host processor may be any microcontroller, microprocessor, or computer. The host processor manages layers 6 and 7 of the LONTALK protocol and uses a LONWORKS network interface to manage layers 1 through 5. The LONTALK network interface protocol defines the format of packets exchanged between the network interface and the host.

Different network interface protocols are defined for each type of network interface. The network interface may be a turn-key device such as the Echelon Serial LONTALK Adapter (SLTA), or may be a custom device based on the LONBUILDER Microprocessor Interface Program (MIP). The MIP extends the NEURON CHIP firmware to transform the NEURON CHIP into a communications processor that can be used to create a LONWORKS network interface.

A host application running on the host processor communicates with the network interface through a network driver. The network driver manages buffer allocation, buffer transfers to and from the network interface, and isolates the host application from any differences in the network interface link layer protocol. The LONTALK network driver protocol defines a standard message format between the host application and network driver.

Nodes using a host processor are known as host-based nodes. Nodes that run their applications entirely on the NEURON CHIP are known as NEURON CHIP-hosted nodes.

Data Interpretation

The LONTALK protocol employs a data oriented application protocol. In this approach, application data items such as temperatures, pressures, states, text strings, and other data items are exchanged between nodes in standard engineering and other predefined units. Commands are encapsulated within the application programs of the receiver nodes rather than being sent over the network. In this way, the same engineering value can be sent to multiple nodes which each have a different application program for that data item.

Data interpretation is performed by the NEURON CHIP firmware for NEURON CHIP-hosted nodes, and is performed by the host processor for host-based nodes.

Network Variables

The data items in the presentation layer of the LONTALK protocol are called *network variables*. Network variables can be any single data item or data structure. Each network variable has a data type declared by the application program.

For NEURON CHIP-hosted nodes, network variables are declared much like local C variables, except that the `network` keyword is used to make the variable available to any other node on the network. When output network variables change via assignment operations within the application program, the NEURON CHIP firmware automatically propagates the new value over the network using LONTALK protocol services. Network variables are transmitted as LONTALK messages, but the NEURON CHIP firmware automatically handles the buffer management, message initialization, message parsing, and error handling.

For host-based nodes, the host processor manages the layer 6 processing and translates network variable updates to and from LONTALK messages stored in application buffers. These buffers are transferred to and from the network interface using the network interface protocol.

Explicit Messages

Applications requiring a different data interpretation model than network variables can send and receive *explicit messages*. Explicit messages use the messaging services of the LONTALK protocol with minimal data interpretation. Each explicit message contains a message code that the application can use to determine the type of interpretation to be used on the contents of the message.

For NEURON CHIP-hosted nodes, explicit messages are transmitted by assigning the message code and message contents to a special output object used for transmitting explicit messages. Explicit messages are received in another special input object that contains the message code and contents.

For host-based nodes, the layer 6 processing on the host processor copies explicit messages to and from application buffers that are exchanged with the network interface.

Foreign Frame Transmission

A special range of message codes is reserved for foreign frame transmission. Up to 229 bytes of data may be embedded in a message packet and transmitted like any other message. The LONTALK protocol applies no special processing to foreign frames - they are treated as a simple array of bytes. The application program may interpret the data in any way it wishes.

Foreign frame messages are sent and received by both NEURON CHIP-hosted nodes and host-based nodes using the same techniques as explicit messages, except that a different range of message codes is used.

Application Compatibility

Application compatibility is facilitated through the use of Standard Network Variable Types, or SNVTs. The list of SNVTs includes nearly 100 types and covers a very wide range of applications. The definition of a SNVT includes units, a range, and a resolution. Using the appropriate network management commands, a LONWORKS node can extract the SNVT information (ID # and optional text string) from any other node. Currently defined SNVTS are listed in the *SNVT Master List and Programmer's Guide*, which is part of the *LONWORKS Interoperability Packet*.

Protocol Services and Parameters

LONTALK protocol services may be selected by a node's application program or by a network management tool. In general, a network management tool may override services selected by the application program; however, in some instances it may not be overridden.

During development, the LONBUILDER Network Manager performs the role of the network management tool that will ultimately be used to install nodes in the final application. Services selected and configured by the LONBUILDER Network Manager at node development (or manufacturing) time may be different from the services selected and configured by the network management tool in the final application. For example, the NetMaker installation tool may override services selected by LONBUILDER.

Table 6 Setable Network Image Parameters

Parameter	When/where initialized	Basis for Configuration	Changeable when node is installed?	Compile-time option to prevent field-override of initial setting?
Channel Bit rate	Compilation or installation	Per Node	Yes	No
Domain ID	Installation	Per Domain	Yes	No
Subnet/Node Address	Installation	Per Domain	Yes	No
Group Address(es)	Installation	Per Node	Yes	No
NEURON ID*	Manufacture	Per Node	No	No

Acknowledged Service - Explicit Messages	Compilation	Per Network Variable or Explicit Message	No	No
Acknowledged Service - Network Variables	Compilation or Installation	Per Network Variable or Explicit Message	Yes	Yes
Retry Count	Installation	Per Network Variable or Explicit Message	Yes	No
Authenticated Service - Explicit Messages	Compilation	Per Network Variable or Explicit Message	No	No
Authenticated Service - Network Variables	Compilation or Installation	Per Network Variable or Explicit Message	Yes	Yes
Parameter	When/where initialized	Basis for Configuration	Changeable when node is installed	Compile-time option to prevent field-override of initial setting
Authentication Key	Compilation or Installation	Per Domain	Yes	No
Number of Priority Slots	Installation	Per Node	Yes	No
Priority Service - Explicit Messages	Compilation	Per Network Variable or Explicit Message	No	No
Priority Service - Network Variables	Compilation or Installation	Per Network Variable or Explicit Message	Yes	Yes
Network Variable Types	Compilation	Per Network Variable or Explicit Message	No	No

* Fixed at the time the NEURON CHIP is manufactured. Cannot be modified at any time.

Limits and Bounds

LONTALK Domain IDs may be 0, 1, 3 , or 6 bytes in length. All nodes in a common domain must have identical Domain IDs of the same length. Within each LONTALK Domain, the following limits apply:

LONTALK Limits

- A maximum of 255 subnets
- A maximum of 127 nodes per subnet
- A maximum of 256 groups
- A maximum of 64 nodes per group (acknowledged services only -- there is no node limit on groups using unacknowledged services)
- A maximum of 32,385 nodes
- A node has one Subnet and one Node address per Domain to which it belongs
- Group membership must be in a Domain to which the node belongs
- A node may have up to 4096 bindable network variables defined.

LONWORKS Node Limits

- A node based on a NEURON 3120™ or NEURON 3150™ CHIP may belong to a maximum of 2 Domains
- A node may have a single outgoing transaction in progress at a time
- A node may have a singe authenticated transaction in progress at a time

- A NEURON CHIP-hosted node may have up to 62 bindable network variables defined (this limit can be increased to the LONTALK protocol limit of 4096 by using a LONWORKS network interface)
- A node based on a NEURON 3120 or NEURON 3150 CHIP may be a member of up to 15 groups
- All nodes must be able to receive a 60-byte layer 2 frame
- All nodes must be able to send a 32-byte layer 2 frame

NEURON CHIP Network Image Road Map

The following items are among the contents of the EEPROM in every NEURON CHIP:

NEURON ID	6 bytes, set at chip manufacture time, cannot be changed. This is the address used in installation and network management. This ID is unique worldwide.
Mfg Data	2 bytes, set at chip manufacture time, cannot be changed. Identifies model number; e.g., Motorola 3120.
Node Type	8 bytes, set at application compile time. In non-certified nodes, contains the NEURON C program name. In certified nodes, this field contains the manufacturer's ID, the node type and sub type, etc. See the *NEURON C Programmer's Guide* for details of the bit fields.
Node Address	One or two structures, each containing a domain (0, 8, 24, or 48 bits long), an authentication key (48 bits), a node number (8 bits), and a subnet number (8 bits). There are two of these if the node can be a member of two domains.
Address Table	(see the following section)
Location ID	2 byte channel number, 6 byte location string. Set at installation time to correspond to a physical location represented on a building plan, etc.
NV Config	Table with an entry for each network variable. For each network variable: priority bit, direction (input or output), network variable ID (assigned by binder), protocol service to use, authenticated bit, and address table index.
NV fixed	Table with an entry for each network variable describing the compile time attributes: synchronized, SI-SNVT/SD Data index, NV length in bytes, NV address.
Comm Data	The number of priority slots on this channel, this node's priority slot, bit rate, and transceiver parameters.
Mode Table	Node's configuration; e.g., how many domains, how many address table entries, how many application buffers, how many

	network buffers, number of receive transaction structures, number of address table entries, number of network variables.
SI-SNVT/SD	(Self-Identifying Standard Network Variable Type with optional Self-Documentation) SNVT index if SNVT's are used, and optional self documentation data about the node. Variable length structure with optional text or other data for each NV that uses a SNVT.

The Address Table

The size of the address table of a node is configurable by the application writer. The default specifies 15 separate network addresses in the address table. The address table is used to get the address to send a message, and, in the case of group addressing, is used to determine if the message received is from a group where the node is a member. For each address table entry, the first byte contains the type of address table entry that follows. There are three types: group, subnet/node, and broadcast.

The group structure has entries for the group size (how many acknowledgements to expect), which domain to use, what this node's group member number is, (to identify an acknowledgement as coming from this node), a transmit timer, a repeat timer, a retry count, a receive timer, and the group ID.

The subnet/node structure has entries for the domain, destination node number, destination subnet number, a repeat timer, a retry count, and a receive timer.

The broadcast structure has entries for the domain and destination subnet (0 implies all subnets), a repeat timer, a retry count, and a receive timer.

A Hybrid System for Fast Synchronized Response

January 1994 **LONWORKS™ Engineering Bulletin**

Introduction

Network control technology is increasingly being designed into the next generation of high-speed machines. This type of machine currently contains a wiring harness emanating from a high-speed central controller. The harness includes both power and individual signal lines for each sensor and actuator within the machine.

Two key objectives for the next generation of these machines are in conflict. The first objective is to replace the wiring harness with a communications network to reduce weight, assembly cost and processing demands on the central controller. The conflicting objective is to run the machine ever more quickly so that it can do more work in the same amount of time.

Unfortunately, because all communications networks share the available bandwidth among the devices, they cannot communicate information as quickly as hard-wired solutions. Thus system designers who must multiplex the wiring *and* increase the speed of the machine are faced with a new set of design issues. In particular, the parallel processing capability inherent in a network of intelligent nodes must be exploited if the wiring harness is to be reduced while the speed of the machine is increased.

Exploiting this parallelism requires that a new set of error-conditions be evaluated and handled. In the hard-wired system, the central controller is a single point of failure which means that the system is either operational or not. A network introduces the problem of dealing with partial failure, that is, some of the system is operating correctly and some portion is not. In networked machine control, those portions of the system which are correctly operating must typically sense those which have failed very quickly so that corrective action may be taken before the partial failure leads to more serious consequences.

Multiplexed Buses

One approach to the problem is to take what is, in effect, a computer bus and multiplex it up onto a pair of wires. An example of this approach is CAN, a bus which proposes to replace the wiring harness in automobiles with a single pair of wires. The CAN system supports peer-to-peer communication with OSI Data Link layer acknowledgements implemented in hardware. The acknowledgements come in the form of levels on the bus immediately following a command packet. The bus is biased such that the sender can get a positive ACK or NAK from the recipients of the command.

Echelon Part Number 005-0043-01 REV B

In CAN, if the command was destined for multiple nodes, an ACK means acknowledgement from all of the *operational* nodes which received the command. If some, but not all, of the destination nodes are out of service at the time that the command is sent, the sender may still receive an ACK. Thus, the low-level acknowledgement does not identify any malfunctioning nodes and instead assumes that they should be ignored. This is one way to handle partial failure, but not always the best way. Additionally, since the acknowledgement is at the Data Link layer, some subsequent error could cause the command to be lost before a receiver node acted on the packet, and the sender would not be notified of the failure. If higher-level acknowledgement is required, CAN requires that it be provided by the application developer.

LonTalk™ Protocol

Rather than multiplex a computer bus onto the communications wires, the LonTalk protocol has taken conventional network design as its starting point. This means that true end-to-end acknowledgements are supported rather than just Data Link layer acknowledgements, and multi-channel networks are supported to accommodate the need for large numbers of nodes which serve a wide area. This is a conventional networking approach, not a multiplexed bus approach. Conventional networks require more processing of the packets than do the multiplexed bus alternatives. End-to-end acknowledgement, duplicate detection, automatic retrying, and a true layer of network addressing to support multiple channels all add additional overhead and processing time to each message.

Because the LonTalk protocol is implemented in custom integrated circuits designed for this purpose, its performance is very good compared to other conventional networks. End-to-end acknowledged transactions can routinely be completed in under 15 milliseconds on a single channel, and the overhead through a router to a second channel only increases latency by about 5 milliseconds. Application developers can achieve even better results using unacknowledged messaging where application-to-application response times under 8 milliseconds can be attained on a single channel.

In many cases, the typical LonTalk protocol response time is sufficient, but for some transactions in some applications it is not. For example, after setting up an automated operation it may be necessary to open several valves within a millisecond, or it may be necessary to set the time across many nodes to within one millisecond of accuracy so that event sequencing may be accomplished. There are even cases where a predefined sequence of events needs to be controlled across multiple nodes to a very tight timing specification. In all of these cases it would be preferable to bypass the protocol overhead and send a simple signal to other nodes which, because the context was set previously with a LonTalk packet, could be responded to and acknowledged very quickly. The remainder of this engineering bulletin discusses how this can be accomplished.

A Hybrid System — Adding a Multiplexed Bus Architecture to LonTalk

If both the features of a multiplexed bus and a true network are needed, the system designer is faced with three alternatives: 1) pick a multiplexed bus and add the missing network features at the application layer; 2) pick a multiplexed bus and then do protocol conversion to a conventional network to connect multiple machines together; or 3) find a way to bring some of the capabilities of a multiplexed bus to a true network. Since designing robust communications protocols is notoriously difficult, alternative 1 will consume significant development time and resources. Protocol conversions are routinely done in control networks, but they tend to add cost and delay to the system, so it would be desirable to avoid them.

To implement alternative 3, a means to provide a signal to initiate a control function and a method for all the participants to acknowledge the accomplishment of the control function is needed. This method must be capable of much higher performance than simply traversing the protocol stack and the network. One way to achieve this is to demultiplex the network by running extra signals to each node. Of course, by running individual control wires for each control action to each node, the system degenerates into a hard-wired one.

Instead of that approach, the signal wires can be shared among the nodes, much as the communications medium is shared among the nodes. In this way, the number of signal wires can be reduced to only two regardless of the number of nodes in the system. Access to the wires is controlled via the LonTalk protocol. As an example, consider the problem of updating time in a distributed system. In figure 1, the node on the left is the master time keeper. Its task is to update all the nodes simultaneously such that they are synchronized to within a millisecond. This is not possible with the LonTalk protocol alone because there is no guarantee that all of the nodes will process the message at the application layer at exactly the same time.

In figure 1, two I/O lines are connected to each node. The ACK line is connected to 5 volts through a weak pull-up resistor. The other line, CLOCK, is also weakly biased high. To update time, the master sends a LonTalk packet to all the nodes telling them, in effect, that at the clock pulse the time will be X. Upon successfully completing of this acknowledged transaction, the master configures its I/O pin connected to the clock line as an output driven high, and configures its I/O pin connected to the ACK line as an input. When all the other nodes receive the time update packet, they configure their I/O pins such that the CLOCK line is an input and the ACK line is an output driven low.

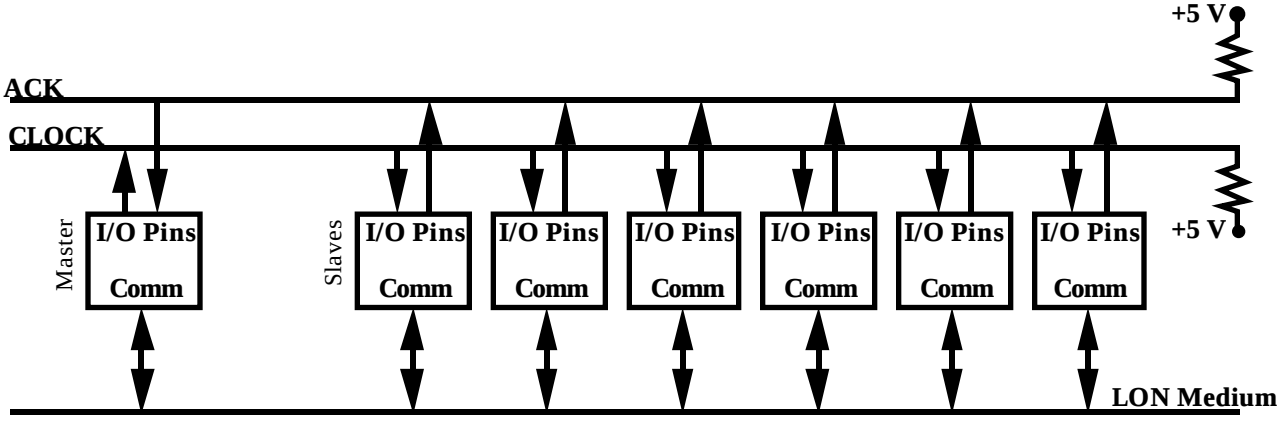


Figure 1 Ready For An Event

All of the slave nodes then enter a tight loop waiting for the clock line to change state. When the master detects that the time has occurred for the update, it toggles the value of the clock line. The slaves then update their time and turn around their ACK I/O pin as an input. This tri-states their driver. Once all of the nodes have done this, the master (and all of the slaves) see the ACK line go high because the pull-up resistors take effect when all of the drivers are tri-stated. Thus the master knows when all of the nodes have acted on the time update. Since the Neuron® C scheduler is not used to detect the I/O pin changes, the timing will be very accurate.

To determine the exact time that it takes a program to circulate around the loop, it is necessary to obtain an assembly listing from the LonBuilder™ Developer's Workbench. Find which instructions are used and then look in Chapter 3 of the April 1993 version of the *Neuron 3150 Chip and Neuron 3120 Chip Data Book* to determine how many clock cycles are required by each instruction. Add to this the time it takes to check the I/O pin, as documented in section 8.2.1 of the data book. This time will not vary, since the Neuron Chip has no interrupts, so the worst case skew is the time around the loop. Sample code with assembly listing and timings is provided in the appendix of this document.

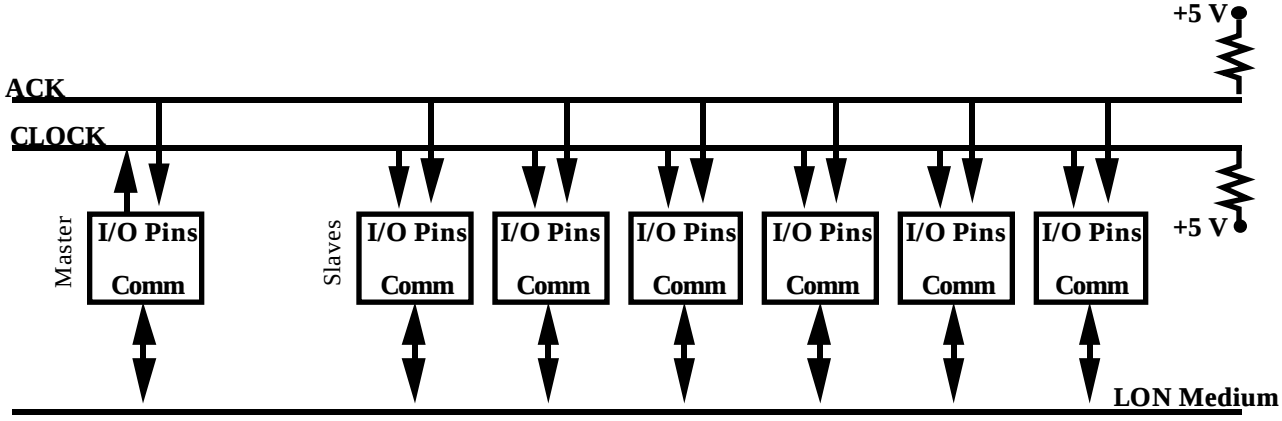


Figure 2 Acknowledging the Update

Although the I/O pins can be directly connected together in the way shown in the diagram, this is only advisable when the nodes are very close together and share the same power supply as they might on a single circuit board or on cards within the same card cage. The IO_0 through IO_3 pins should be used for this application, since they can sink 20 mA, but they are standard TTL inputs, not line drivers. When greater distance is required or ESD is a concern, the I/O lines should be connected to more standard line drivers for more protection and a higher current source driver. The distance limit will depend upon the maximum speed of the transitions on the bus and characteristics of the driver hardware.

Generalizing the Approach

Although our first example was for the timing of a single event, the algorithm can be generalized to a transition from state to state in a series of state machines. Suppose each node has several state machines that it can execute. Based upon operator command or some other external event, a specific task is selected for the machine to perform. The node which receives the operator input or the external event then uses the LonTalk protocol to instruct the other nodes to prepare to execute the selected state machine. The nodes follow the algorithm, as above, for a single event, but instead of returning to normal operation after the CLOCK line toggles and they send an acknowledgement, they return to their tight loop waiting for the next clock toggle. In this way all of the nodes advance to their next state in unison as the master toggles the CLOCK line. As before, once the specific task is complete, any of the other nodes can be the master for the next task.

In Figure 3 below, the timing relationship between the clock and the ACK lines is shown for two full cycles through a multi-state machine. Note that between each state advance, the CLOCK line returns to its initial state as does the ACK line. This serves to resynchronize the master and the slaves between each state change so that

the CLOCK and ACK lines are biased with the correct polarity for the subsequent state transitions and acknowledgements.

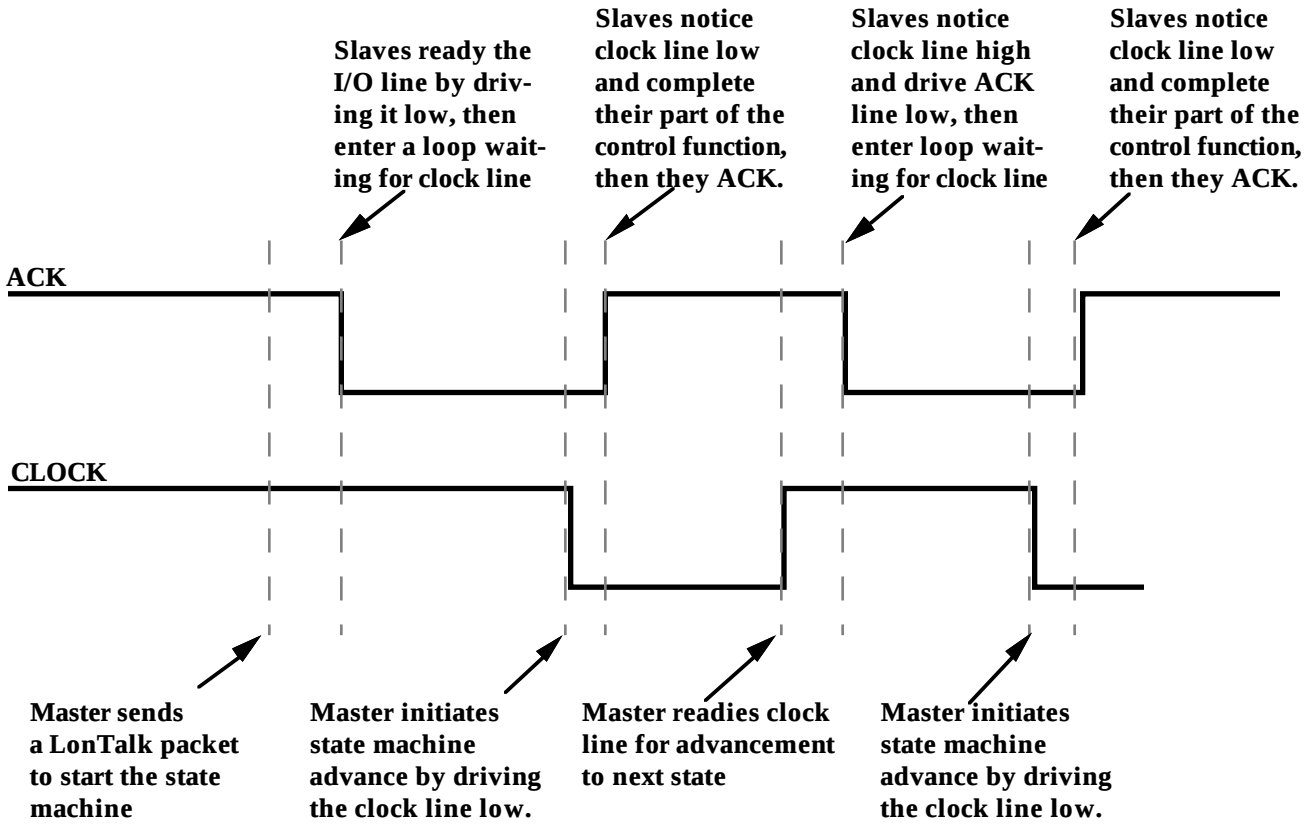


Figure 3 Timing on the CLOCK and ACK lines

When using this approach for controlling the transitions between multiple states, it may be necessary to check for other events while operating within the tight timing loop. This can be accomplished by calling the firmware routine `post_events()` followed by whatever checks are required by the application, i.e., network variable updates, timer expirations, or changes in additional I/O pins . Care should be used in defining the interactions and the timing relationships between the nodes since each application program will have more states and be more complex. Note that `post_events()` performs a number of functions (see Chapter 5 of the *Neuron C Programmer's Guide*,), so it may be necessary to experimentally measure its worst-case timing on each of the system nodes.

Conclusion

Very fast and deterministic application-to-application response times can be achieved on a hybrid system combining the LonTalk protocol with a multiplexed

bus. In applications requiring a true network, this approach represents a better choice than either attempting to put higher-level network features into a multiplexed bus or implementing a complete peer-to-peer network. Additionally, the acknowledgements on the ACK line are application-to-application acknowledgements rather than just hardware-level acknowledgements. The master and all of the slaves can determine that the control function really occurred, or have rapid notification that it did not. Application layer acknowledgements instead of Data Link layer acknowledgements make it possible to detect and respond to partial network failures, increasing the robustness and reliability of the network. Using the algorithm defined in this engineering bulletin, developers will obtain benefits of a multiple-sourced custom IC for their application, and meet the very short response time criterion for demanding machine control applications.

Appendix

A simple version of the algorithm described can easily be prototyped on a LonBuilder development system with three or more nodes. The nodes in this example use IO_2 for the CLOCK line and IO_5 for ACK. In real implementations, IO_0 through IO_3 pins should be used; IO_5 was used to allow use of the built-in pullup resistor for rapid prototyping. In addition, IO_0 is used to simulate a counter circuit reset line which resets the counter to zero and starts it running. In reality, IO_0 is connected to an LED and the reset pulse lengthened to provide an activity indicator. The code for master and slave nodes follows, as does the associated assembly code fragment and timing analysis.

```
/*
 *  Name:  msynch.nc
 *
 *  Purpose:  Master portion of the very fast response time
 *      program.  This program uses a combination of LonTalk
 *      messages and a multiplexed bus protocol to
 *      synchronize Neuron actions within a millisecond of
 *      each other.
 *
 *  Description:  This program uses a network variable
 *      message to tell the connected slave nodes when to
 *      enter "clock synch mode".  When acknowledgements are
 *      received from all slaves, the master knows that they
 *      are all waiting for the synch signal and that they
 *      have driven the ACK line low.  The master drives the
 *      bus CLOCK line low--the synch signal.  Each slave
 *      then synchs its clock and changes its ACK pin
 *      direction around to input, causing the ACK line to go
 *      high when all slaves have ACKnowledged.
 *
 */

#include <control.h>
```

```
#include "synch.h"


/* synch_clock means that the slave nodes
 * should enter clock synch mode, watching
 * the CLOCK line.
 */
network output boolean  nvo_synch_clock = FALSE;
boolean  updating_time = FALSE;

/* Benchmark statistics */
long     synch_errors = 0;
long     max_loopct = 0,
         min_loopct = 200,
         num_synchs = 0;


/* Declare the CLOCK and ACK bus lines on I/O pins 2 and 5.
 * WARNING:  When implementing a synchronized application
 *           like this for a real product, always use lines
 *           IO_0 through IO_3 for CLOCK and ACK lines.
 *           This demo uses IO_5 in order to use the built-in
 *           pullup resistors, but is not suitable for
 *         greater distances or larger numbers of nodes.
 *         This was tested with three nodes and a total
 *         wire length of three feet on the I/O lines.
 */
IO_2 output bit  CLOCK;
IO_5 input  bit  ACK;


/* Periodically resynching counters throughout the system */
stimer repeating  Synch_Timer;
```

```
/* For delaying a few milliseconds to let NV message reach
 * application processors of all nodes.
 */
mtimer  Wait_for_Apps;


when (reset)
{
    /* Synch counters every X seconds starting at reset */
    Synch_Timer = SYNCH_TIMERS_PERIOD;

    /* Drop the clock line to inactive state */
    io_out(CLOCK, CLOCK_NO_SYNCH);
}

/*
 * Time to synchronize clocks again.  Start it off with
 * a network variable message.
 */
when (timer_expires(Synch_Timer))
{
    /* Tell all slaves to start watching CLOCK */
    nvo_synch_clock = TRUE;
}

/*
 *  When all nodes have verified that they are watching
 *  CLOCK, go into bus emulation mode.
 */
when (nv_update_succeeds(nvo_synch_clock))
{
    Wait_for_Apps = WAIT_FOR_SLAVES_TIME;
    if (WAIT_FOR_SLAVES_TIME == 0)
    {
```

```
        updating_time = TRUE;
    }
}

/*
 *  Could not get acknowledgment from all nodes.
 *  Give a synch signal to the ones that are
 *  waiting to get them out of wait state.
 */
when (nv_update_fails(nvo_synch_clock))
{
    ++synch_errors;
    io_out(CLOCK, CLOCK_SYNCH);
    delay(100);
    io_out(CLOCK, CLOCK_NO_SYNCH);
}

/*
 * Wait for all slaves to percolate NV message
 * to their Application processors and get into
 * the wait loop.
 */
when (timer_expires(Wait_for_Apps))
{
    updating_time = TRUE;
}

/*
 *  When updating the time, wait for the ACK line
 *  to go high after all nodes have seen the
 *  CLOCK line signal.
 */
priority when(updating_time)
{
```

```
  unsigned int  loopct;
  boolean  synch_failed;

  synch_failed = TRUE;

  /* Give the synch signal and wait for ACK line to
   * indicate acknowledgment from all slaves.
   */
  io_out(CLOCK, CLOCK_SYNCH);
  for (loopct = 1; loopct < MAX_MASTER_ITERATIONS;
loopct++)
  {
      if (io_in(ACK) == ACK_SYNCH)
      {
          synch_failed = FALSE;
          break;
      }
  }

  /* Benchmark statistics */
  if (loopct > max_loopct)  max_loopct = loopct;
  if (loopct < min_loopct)  min_loopct = loopct;
  ++num_synchs;
  if (synch_failed)
  {
      ++synch_errors;
  }

  /* Drop clock line back to normal */
  io_out(CLOCK, CLOCK_NO_SYNCH);

  /* Get out of the synch mode */
  updating_time = FALSE;
```

```
}


/*
 *  Name:  synch.h
 *
 *  Description:  Definitions for the Fast Synchronized
 *      Response Engineering Bulletin example program.
 *
 */

/* Enable I/O 4 through 7 pullups For demo only!!!! */
#pragma enable_io_pullups

/* Defined values for I/O lines, CLOCK and ACK */
#define CLOCK_NO_SYNCH          1
#define CLOCK_SYNCH             0
#define ACK_SYNCH               1
#define ACK_WAIT                0

/* How often does master re-synch timers on slaves? (seconds) */
#define SYNCH_TIMERS_PERIOD     3

/* How long to pulse "counter reset line".  This is
 * actually an LED on the LonBuilder emulators, and
 * this value will pulse it for about half a second.
 */
#define COUNTER_RESET_PULSE     65000

/* How many loop iterations to wait for synchronization
 * of the nodes.  The best value for these should be
 * determined experimentally for your application.  Be
 * sure to update the watchdog timer if necessary.
 */
```

```
#define MAX_MASTER_ITERATIONS          255
#define MAX_SLAVE_ITERATIONS           10000

/* The amount of time to wait for all of the slave nodes
 * to percolate the NV update up to their application and
 * get into the wait state.  This is application-dependent,
 * equal to the the slave application's greatest when clause
 * processing time, plus the scheduler latency, plus the
 * entry time for the NV update when clause, minus the amount
 * of time for the ACK to make its way back over the network.
 * (milliseconds)
 */
#define WAIT_FOR_SLAVES_TIME    0



/*
 *  Name:  ssynch.nc
 *
 *  Purpose:  Slave portion of the very fast response time
 *      program.  This program uses a combination of LonTalk
 *      messages and a multiplexed bus protocol to
 *      synchronize Neuron actions within a millisecond of
 *      each other.
 *
 *  Description:  Each slave gets a network variable message
 *      to tell it when to enter "clock synch mode".  It
 *      should then set the ACK line direction to output,
 *      drive it low, then wait for the master node to drive
 *      the CLOCK line low.  When this CLOCK synch signal
 *      arrives, reset the external timer counter by pulsing
 *      its I/O line, and change the ACK line direction to
 *      input to allow it to go high when all of the attached
 *      slaves have done this.
 *
```

```
 */

#include <control.h>
#include "synch.h"



/* synch_clock means that slave should go into
 * clock synchronization procedure.
 */
network input boolean  nvi_synch_clock;

/* Debugging information */
long  synch_errors = 0;
long  max_loopct = 0,
      min_loopct = 200,
      num_synchs = 0;


/* Declare the CLOCK and ACK bus lines on I/O pins 2 and 5.
 * WARNING:  When implementing a synchronized application
 *           like this for a real product, always use lines
 *           IO_0 through IO_3 for CLOCK and ACK lines.
 *           This demo uses IO_5 in order to use the built-in
 *           pullup resistors, but is not suitable for
 *           greater distances or larger numbers of nodes.
 *           This was tested with three nodes and a total
 *           wire length of three feet on the I/O lines.
 */
IO_2 input  bit  CLOCK;
IO_5 input  bit  ACK_IN;
IO_5 output bit  ACK_OUT;
```

```
/* Declare the synchronized counter in I/O pin 0 */
IO_0 output oneshot clock(5)  COUNTER_RESET;




/*
 *  Set the ACK line to the no-ACK state.
 */
when (reset)
{
    io_set_direction(ACK_OUT, IO_DIR_OUT);
    io_out(ACK_OUT, ACK_WAIT);
}

/*
 *  When requested via NV update, go into bus emulation mode.
 */
priority when (nv_update_occurs(nvi_synch_clock))
{
    boolean  synch_failed;
    long  loopct;

    synch_failed = TRUE;

    /* Bring the ACK line low to set up for acknowledgment */
    io_set_direction(ACK_OUT, IO_DIR_OUT);
    io_out(ACK_OUT, ACK_WAIT);

    /* Sit and wait for the CLOCK synch signal from master */
    for (loopct = 1; loopct < MAX_SLAVE_ITERATIONS; loopct++)
    {
        if (io_in(CLOCK) == CLOCK_SYNCH)
        {
```

```
            synch_failed = FALSE;
            break;
        }

        /* update watchdog timer every 256 iterations */
        if ((loopct && 0xff) == 0)
        {
            watchdog_update();
        }
    }

    /* Benchmark statistics */
    if (loopct > max_loopct)  max_loopct = loopct;
    if (loopct < min_loopct)  min_loopct = loopct;
    ++num_synchs;

    if (!synch_failed)
    {
        /* Pulse the counter--reset to 0 and start running */
        io_out(COUNTER_RESET, COUNTER_RESET_PULSE);

        /* Acknowledge by tri-stating ACK line */
        io_set_direction(ACK_IN, IO_DIR_IN);
    }
    else
    {
        ++synch_errors;
    }

}
```

The following assembly code fragment is from `msynch.nl`, the assembly file for `msynch.nc`. The `when` clauses are clearly labeled in the assembly listing. The reset clause is handled specially and positioned after all of the other `when` clauses in the

assembly listing. The important timing parameter is the master node time between dropping the CLOCK line low and seeing the ACK line go high to indicate acknowledgment from all nodes. This timing is somewhat greater than the largest synchronization skew among all of the nodes. The assembly fragment begins with a call to `_bit_output_lo1` corresponding to a call to `io_out()` in the source code.

```
offst code      instr                cycles
0005  0000*      CALL _bit_output_lo1    6
0007  81         PUSHS   #1              4
0008  E5         DROP NEXT               2

0009  B5 00FF    PUSHD   #0FF            6
000C  80         PUSHS   #0              4
000D  BE         PUSH [DSP][-2]          5
000E  0000*      CALL _less16s           6
0010  4B         SBRZ WHEN5+1C           3
0011  85         PUSHS   #5              4
0012  0000*      CALL _bit_input         6
0014  3F         DEC                     2
0015  63         SBRNZ   WHEN5+19        3
0016  80         PUSHS   #0              4
0017  FF         POP  [DSP][-1]          5
0018  23         SBR  WHEN5+1C           1
0019  3E         INC                     2
001A  71 ED       BR   WHEN5+9            2
```

During several thousand runs, the timing loop in the master node was executed a maximum of 4 times. Instruction counting from the CLOCK `io_out()` return to the ACK `io_in()` (corresponding to `_bit_input` in assembly) return gives the maximum difference in the synchronization time between the three nodes.

The return from procedure `_bit_output_lo1`, as documented in the *Bit Output* section of chapter 8 of the *Neuron 3120 Chip and Neuron 3150 Chip Data Book*, takes 5 µs at 10 MHz. In the *Bit Input* section of the same document chapter, note that the procedure `_bit_input` takes 41 µs from call, including CALL instruction, to sampling and 41.15 µs to return for IO_5.

The undocumented procedure `_less16s` takes 132 cycles to execute when the exit condition is not true, and this procedure will be executed 4 times, for 4 * 132 = 528 cycles. The rest of the cycles in the code path totals 139 cycles. Total I/O overhead is 5 µs for the return from `io_out()` and (4 * 82.15) - 41.15 = 287.45 µs for `io_in()`. We subtract the time for the final return from `io_in()`, since it is simply reporting that the I/O change has been detected.

There are a total of 667 cycles plus 292.45 µs in the code path. The formula given in chapter 3 of the *Neuron 3120 Chip and Neuron 3150 Chip Data Book* for CPU instruction times is "Instruction Time (µs) = #cycles * 6 / Input Clock (MHz)", which converts 667 cycles to (667 * 6)/10 = 400.2 µs. Thus, the total time to synch the bus network is less than 692.65 µs.

Note: A better response time may be achieved by tightening the wait loops. If a loop forever condition is used instead of a 16-bit signed compare, the loops will be much smaller and the sync time will therefore be smaller. Nodes will then have to rely on the watchdog timer reset to break out of a failed synchronization attempt.

EIA-232C Serial Interfacing with the Neuron® Chip

January 1995 **LONWORKS® Engineering Bulletin**

Introduction

The Neuron Chip is a programmable device that includes a rich variety of input/output capabilities. The Neuron Chip's firmware can configure the 11 I/O pins of the processor in more than 30 different modes supported by software drivers. Application programs running on the processor can then access this I/O functionality through simple calls to the I/O driver functions. The Neuron Chip is also a device that can communicate with other Neuron Chips over a variety of networking media, such as twisted-pair wiring and power line, using the LonTalk® protocol. It is thus an ideal device to implement applications for control networks.

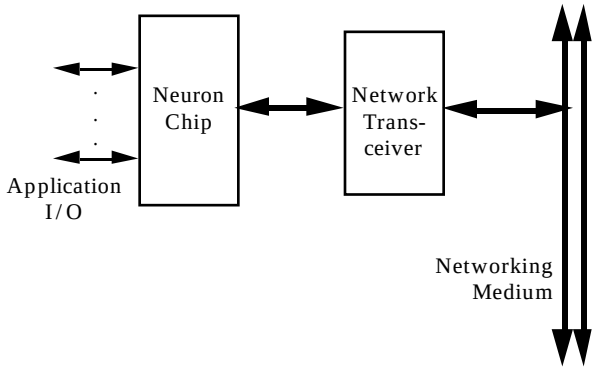


Figure 1. Architecture of a Neuron Chip-based node.

A LONWORKS® network uses a multi-drop concurrent-access architecture, so that multiple nodes can communicate in a peer-to-peer fashion. The RS-485 standard supports such a multi-drop architecture, allowing any node to communicate with any other node on the network, with up to 32 nodes per physical channel. On the other hand, the EIA-232C standard (formerly known as RS-232C) is a point-to-point architecture, which allows only two devices to communicate with each other. The standard was originally designed for communications between Data Terminal Equipment (DTE) and Data Communications Equipment (DCE) such as modems. However, it has been widely applied in recent years to other point-to-point communications needs.

This engineering bulletin describes implementation of an EIA-232C asynchronous serial interface that enables a Neuron Chip to communicate with another device that employs the EIA-232C standard. This could be, for example, a PLC, machine

Echelon Part Number 005-0008-01 REV D

controller, CRT terminal, printer, card reader, or modem. The same Neuron Chip can also be a part of a network communicating with the LonTalk protocol, forming a gateway between an EIA-232C link and a LONWORKS network.

The Neuron Chip does not contain any UART hardware; instead, the serial I/O interface is implemented in firmware, moving data a bit at a time using the Neuron Chip's application CPU. If bit rates higher than 4,800 bps or serial data buffering greater than one bit are required, then an external hardware UART should be used. Input buffering is especially important. With a software UART, the Neuron Chip's application CPU must be waiting at an input statement when the start bit of the asynchronous character arrives. At 4,800 bps, a bit time is approximately 200 µs, which is below the typical event latency for a Neuron C program. However, if hardware buffering is used, the required latency is much greater. For example, if the UART has a 16-character input FIFO, messages of up to 16 characters in length may be received at any supported bit rate without loss of data, and the Neuron C program may read this data at its leisure.

The PSG/2 and PSG-10 Programmable Serial Gateways are suitable for handling high speed asynchronous data streams. They include a Neuron 3150 Chip with a 5MHz or 10MHz input clock, a socket for a PROM to contain system and application firmware, and a 16550-compatible UART. This UART provides sixteen characters of serial input and output data buffering, and bit rates of up to 115,200 bps. A low-level software library for creating serial gateway applications is included with LonBuilder® and NodeBuilder™ development systems. For more details on the Programmable Serial Gateway, see the *Serial LonTalk Adapter and Serial Gateway User's Guide.*

For applications where a serial interface to a host computer is required, the SLTA/2 or LTS-10 Serial LonTalk Adapters may be used. These are pre-programmed serial interface devices based on the same hardware as the Programmable Serial Gateway devices. The on-board firmware communicates with a driver on the host computer, and supports all network management functions, as well as remote access via modems. The Serial LonTalk Adapter firmware moves the LonTalk application layer to the host computer, giving it the capability to send and receive network variables and to implement LONMARK objects. Source code for DOS and UNIX drivers is included. For more details, see the *Serial LonTalk Adapter and Serial Gateway User's Guide.*

This engineering bulletin only describes the use of the Neuron C serial I/O object, which is implemented as a software UART.

Asynchronous Data Format

The Neuron Chip receives and transmits serial data using eight-bit character frames, with one start bit and one stop bit. The EIA-232C standard defines two voltage levels.

The negative voltage corresponds to logic level 1 and the positive voltage to logic level 0. When the line is idle, it is at logic level 1. A character frame begins with a start bit, which holds the line at 0 for one bit time. Then the eight data bits are transmitted with the least significant bit first. Finally, the line returns to 1 for at least one bit time, forming the stop bit. A new character frame may start at any time after the end of the stop bit. This asynchronous data format is commonly used with the EIA-232C standard interface, although strictly speaking, it is not a part of the standard. It is termed asynchronous since it is not necessary to share a clock between the transmitting and receiving devices. Both devices can use independent local clocks running at the same nominal frequency. Actual synchronization is on a character-by-character basis using the start and stop bits.

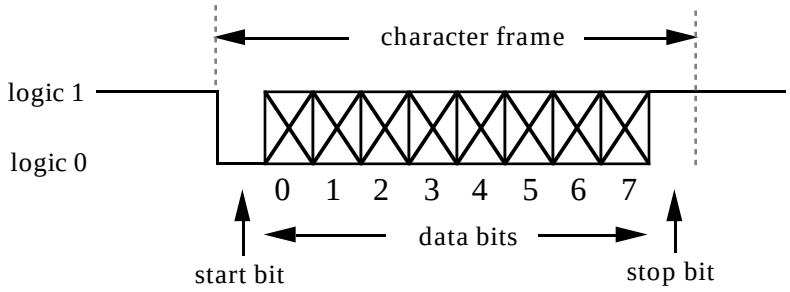


Figure 2. Asynchronous data format.

Hardware Considerations

The Neuron Chip supports this asynchronous serial data format using the serial I/O object. The serial output object is implemented on IO10 pin and the serial input device on the IO8 pin . The nine remaining I/O pins can be used for other I/O objects. The I/O pins have TTL input levels and standard CMOS output levels. Devices such as the Motorola MC145407 may be used to convert these levels to and from EIA-232C voltage levels. Figure 3 shows a typical schematic for a bi-directional EIA-232C interface for a Neuron Chip configured as Data Terminal Equipment (DTE). The interface chip chosen is a 5-volt-only driver/receiver that uses an on-chip charge pump to generate the EIA-232C voltage levels with the help of four external capacitors.

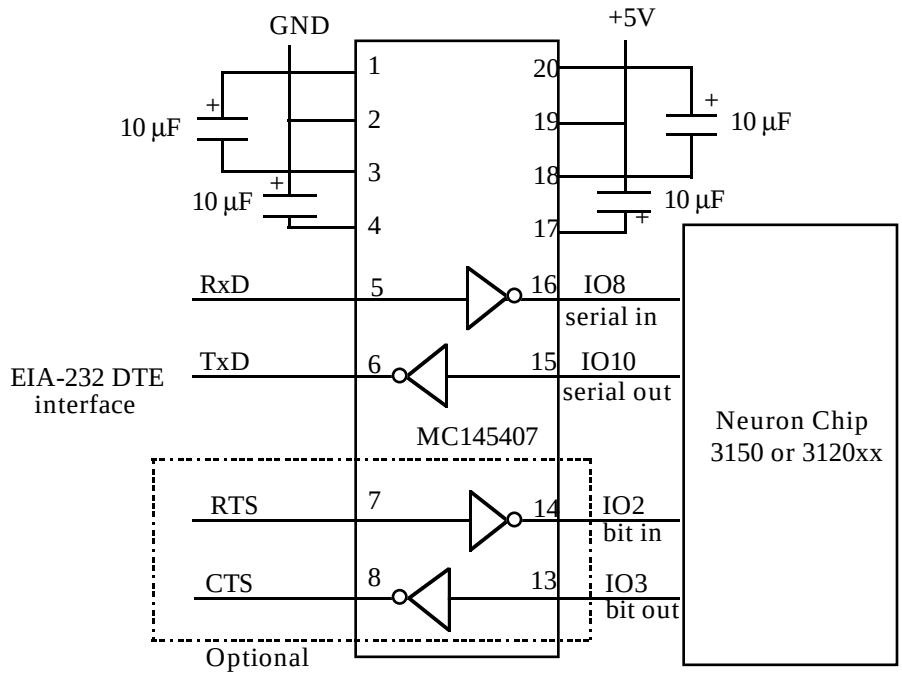


Figure 3. Typical EIA-232C interface circuit.

If additional modem control lines such as CDET, DSR, DTR, CTS, and RTS are required, then any of the other Neuron Chip I/O lines may be configured as bit input or output lines under control of the application software. Not all of these signals are active high. The application developer should check which sense is valid and handle each one of them appropriately. For full details on the modem control lines, see the Electronics Industries Association EIA-232C Standard document.

Software Considerations

The designer uses statements in the Neuron C programming language to declare and activate serial I/O objects. This provides a high-level interface that frees the application designer from considerations of bit timing, data framing, and character assembly and disassembly.

Serial Output

To declare a serial output object, a statement of the form should be used:

```
IO_10 output serial baud (constant) io_object_name;
```

For example, the following statement declares the device named `CRT_screen` as a serial output object operating at 1200bps:

```
IO_10 output serial baud(1200) CRT_screen;
```

To output data to the serial device, use a statement of the form:

```
io_out(io_object_name, buffer_pointer, character_count);
```

For example, the following statement sends the twelve ASCII characters "Hello, world" in serial format on the IO10pin:

```
io_out(CRT_screen, "Hello, world", 12);
```

The parameters to the `io_out()` function call for a serial output object are:

`io_object_name`	A name declared as a serial output object
`buffer_pointer`	A parameter of type `(const char *)` pointing to an array of characters
`character_count`	A parameter of type `(unsigned int)`

The `io_out()` function call suspends execution of the application program until all the characters have been transmitted on the output pin. For example, transmitting 120 characters at 600bps will suspend the application for 120 * (1 + 8 + 1) / 600 = 2 seconds. During serial I/O, the network and media access CPUs on the Neuron Chip continue to execute the network protocol, but the application processor does not execute other tasks to handle any generated events. Application designers should take this into account.

The allowable values for the constant expression used to specify the bit rate are 600, 1200, 2400, and 4800, which refer to the serial bit rate at a Neuron Chip input clock rate of 10 MHz. If other input clock rates are used, refer to Table 1.1.

Input Clock (MHz)	**baud(600)**	**baud(1200)**	**baud(2400)**	**baud(4800)**
10.0	600	1,200	2,400	4,800
5.0	300	600	1,200	2,400
2.5	150	300	600	1,200
1.25	75	150	300	600
0.625	37.5	75	150	300

Table 1. Serial bit rates for different Neuron Chip input clock rates.

Serial Input

To declare a serial input object, a statement of the form should be used:

```
IO_8 input serial baud(constant) IOobjectName;
```

For example, the following statement declares the device named `keyboard` as a serial input object, operating at 600 bps:

```
IO_8 input serial baud(600) IOkeyboard;
```

To input data from the serial device, a statement of the form should be used:

```
num_chars_received =
        io_in(IOobject_name, pBuffer, maxCharacterCount);
```

For example, the following statement causes the Neuron Chip to wait for up to twelve serial characters to be received in serial format on the IO_8 pin:

```
numCharsReceived = io_in(IOkeyboard, inputBuffer, 12);
```

The parameters to the `io_in()` function call for a serial input device are:

IOobjectName	A name declared as a serial input object
pBuffer	A parameter of type `(char *)` pointing to an array of characters to accept the received data
maxCharacterCount	A parameter of type `(unsigned int)`

The `io_in()` function returns a value of type `(unsigned int)`, indicating the number of characters actually received. Suitable declarations for the above example are:

```
char inputBuffer[12];
unsigned numCharsReceived;
```

Note that in the C language, an object of type `(char[])` is automatically promoted to type `(char *)` in the context of a function call. It is the designer's responsibility to ensure that the input buffer for serial input is large enough to contain the number of characters requested. If the `io_in()` call specifies a maximum character count that exceeds the size of the input buffer, unpredictable behavior will result.

The `io_in()` function suspends application processing until it is complete. This occurs at the first of the following:

1. The number of characters specified in the `maxCharacterCount` parameter of the function call have all been received,
2. The line has been continuously at the idle level for 20 character times, for example 166.7 msec at 1200 bps, or
3. A framing error has occurred – an expected start bit or stop bit has the wrong polarity.

In all cases, the returned value of the function is the number of characters actually received and stored in the buffer.

The Neuron Chip's application CPU is suspended during execution of the serial `io_in()` function. This means that it cannot process other I/O events, timer expirations, network variable updates, or incoming messages while it is waiting for

input characters on the serial port. If no input occurs for 20 character times, then execution proceeds after the `io_in()` call. The application can simply tickle the watchdog timer to avoid resetting the node, and re-enter the serial input call. This has the advantage that the application is unlikely to miss any input characters, because it is almost always in the tight loop waiting for input. For example:

```
#include <control.h>                                  // define 'watchdog_update'
IO_8 input serial baud( 2400 ) IOcharIn;              // serial input device
char inputChar;                                       // place to store received
                                                      // character
unsigned numChars;
when( serial input is desired ) {
   do {                                               // tight loop until character
                                                      //is received
      numChars = io_in(IOcharIn,&inputChar, 1);       // read one character
      watchdog_update( );                             // avoid watchdog timeouts
   } while( numChars == 0 );                          // if error or time_out,
                                                      //try again
   process_char( inputChar );                         // process this character
}
```

However, the node is frequently required to process other input events besides the serial character input. In this case, the application processor must periodically return to the event scheduler to handle the other events. This has the disadvantage that the application processor cannot handle serial input while it is processing those events. There is a chance that some characters will be missed, depending on the relative time taken to process these other events, and how fast the application processor can return to the `io_in()` call. For example, to process other events every time the `io_in()` call returns, the following code can be used:

```
IO_8 input serial baud( 2400 ) IOcharIn;              // serial input device
char inputChar;                                       // place to store received
                                                      // character
unsigned numChars;

when (1) {                                            // do this every pass through
                                                      // the scheduler
   numChars = io_in(IOcharIn, &inputChar, 1 );        // read one character
   if (numChars != 0) ProcessChar(inputChar);
                                                      // process this input
                                                      // character
}                                                     // end of task, return to
                                                      // scheduler to handle other
                                                      // events
```

```
when (other events).....                    // process NV updates,
                                            //messages etc.
```

If the `io_in()` call is successful, and a character has been received, the `io_in()` call will return to the calling task approximately at the center of the stop bit. The Neuron C application must re-enter the `io_in()` call before the beginning of the next start bit. If the characters are being transmitted back-to-back with one stop bit, that means that the application has approximately half of a bit time to process the character. At 1200bps, that is about 400µs. If more time is desired, it will be advantageous to have the sending device use two or more stop bits, since this will extend the idle period before the start bit of the next character. The best solution may very well be to spend most of the time waiting for serial input to occur, and occasionally returning to the scheduler to check for network and timer events.

Synchronizing with an External Serial Device

Because the serial input object is implemented using software-driven serial I/O, there are a few restrictions in the use of serial input:

- The application program must be waiting at an `io_in()` function call when the start bit of the character is received. If the call to `io_in()` is made after the beginning of the start bit of a character, an immediate framing error is likely. If the call to `io_in()` is made after the end of a character, then that character will be totally missed.
- If the start bit of the first character is delayed more than 20 character times after calling the `io_in()` function, then the call will time out and no characters will be returned. A time-out will also occur if there is a pause of more than 20 character times between the end of one character and the beginning of the next. In this case, the call will return the characters received up until the time-out.

Solutions to these limitations depend on the device that is generating the input characters. If the input device is not an operator typing at a keyboard, but rather another processor, then some form of handshaking can be implemented. For example, the sending device can indicate its desire to transmit by raising a request to send (RTS) signal connected to a bit input of the Neuron Chip. The Neuron Chip detects the request to send, and activates a bit output that indicates to the sending device that the Neuron Chip is ready to receive data (a clear-to-send, or CTS signal). The Neuron Chip then immediately enters the `io_in()` call for the serial input object. The external device, when it receives the CTS indication, waits until the Neuron Chip is in the `io_in()` call, and then transmits its characters. If the number of characters transmitted is fixed in advance, then the `io_in()` call can specify this number of characters. Alternatively, it can wait 20 character times for the time-out, or the input device can generate a *break* condition on the line at the end of the data,

causing a framing error and termination of input. Or, the Neuron Chip can read a single character at a time, and look for a terminating character such as a carriage return. The following example assumes that the input line is sent without pauses greater than 20 character times between the characters, and that a carriage return is always sent at the end of the line, which is never longer than 120 characters.

```
IO_3 output bit CTS;             // clear to send output
IO_2 input bit RTS;              // request to send input
IO_8 input serial RXD;           // serial data input
char inputBuf[120];              // RAM buffer for input line
char * pBbuf;                    // pointer into buffer

when (io_changes(RTS) to 1) {    // wait for request to send
   pBuf = inputBuf;              // initialize buffer pointer
   io_out(CTS, 1);               // raise clear to send
   do {
      (void)io_in(RXD, pBuf, 1); // get one character into buffer
   } while (*pBbuf++ != '\r');   // keep going if not CR
   io_out(CTS, 0);               // drop clear to send
   ProcessBuffer();              // handle input line
}
```

Using Serial I/O to Interface with a CRT Terminal

If the device generating characters is an operator typing at a keyboard, then there are user interface issues to be considered. Normally, an operator will type with unpredictable pauses between characters. These pauses will probably exceed the 20 character times time-out limit. The serial `io_in()` function does not check for any input terminators such as the ASCII carriage return character, which is conventionally used to indicate the end of user input. It also does not echo input characters, nor does it recognize any input line editing characters such as back-spaces, as would be expected by a user typing at an interactive terminal device.

If this kind of functionality is required, it must be implemented by the application code calling `io_in()` and `io_out()` to perform the basic I/O. The following code shows how some of these interactive terminal capabilities might be implemented. The function `GetLine()` reads characters one at a time from the serial input device, processing some control characters and echoing printable characters to the serial output device. A null-terminated string is returned in `inputBuf`.

Following the function `GetLine()` is a demonstration task that calls `GetLine()` to read a line from the keyboard and then echoes it back to the screen.

```
// Sample EIA-232 driver for an interactive serial ASCII keyboard device

#include <control.h>
IO_8   input serial baud(2400) IOcharIn;  // serial input device
IO_10 output serial baud(2400) IOcharOut; // serial output device

char inputBuf[120];                       // input buffer for GetLine

unsigned GetLine(void) {
      // function waits for input line, returns number of characters received

   char * pInputBuf;    // next place to store received character
   char * pEndOfBuf;    // last byte in buffer
   char   thisChar;     // character being processed
   unsigned numChars;

   pInputBuf = inputBuf;            // initialize buffer pointers
   pEndOfBuf = inputBuf + sizeof( inputBuf ) - 1;

   while (pInputBuf < pEndOfBuf) {  // don't read past end of buffer

      do {
         numChars = io_in(IOcharIn, pInputBuf, 1); // read one character
         watchdog_update();            // avoid watchdog timeouts
       } while (numChars == 0);        // if error or time_out, try again

      thisChar = *pInputBuf & 0x7F;    // discard any parity bit
      if (thisChar == '\r') break;     // exit loop if a carriage return

      if ((thisChar == '\b') || (thisChar == 0x7F)) {
                                       // backspace or rubout
         if (pInputBuf > inputBuf) {   // buffer is not empty
            pInputBuf--;               // forget last character
            io_out(IOcharOut, "\b \b", 3);   // erase character on screen
         }
         continue;
      }
      if (thisChar < ' ') continue; // ignore other control characters
      *pInputBuf = thisChar;        // store this character
      io_out(IOcharOut, pInputBuf++, 1);
                                    // echo this character, increment pointer
   }  // end while

   io_out(IOcharOut, "\r\n", 2);    // send CR LF at end of line
   *pInputBuf = '\0';               // null terminate string
   return (unsigned)(pInputBuf - inputBuf);
}                                   // return number of characters

// Demonstration task for the GetLine() function
unsigned numChars;


when (TRUE) {                 // do forever
   numChars = GetLine();      // read a line
   if (numChars != 0) {
      io_out(IOcharOut, output_buf, numChars);  // echo line
      io_out(IOcharOut, "\r\n", 2);             // send CR LF
   }
}
}
```

LONWORKS™ 78kbps Self-Healing Ring Architecture

August 1993 **LONWORKS Marketing Bulletin**

Introduction

Echelon's TP/XF-78 Twisted Pair Control Module and TPT/XF-78 Twisted Pair Transceiver Module provide a simple, cost effective method of adding LONWORKS technology to machine controls, process controls, building management systems, fire and security devices, and general purpose control systems. Twisted pair wiring is an excellent medium for LONWORKS control networks because it is reliable, inexpensive, supports high speed communications, and simple to install and repair in the event of faults.

The actual performance of the network under different fault conditions will vary according to the method by which the TP/XF-78 Control Module and TPT/XF-78 Transceiver Module are connected to the twisted pair wire. Two wiring methods are typically employed: bus and loop. This document describes the benefits and limitations of each of these two wiring methods.

By way of introduction, the TP/XF-78 Control Module consists of a miniature circuit card containing a Neuron® 3150 Chip, EPROM socket, transformer isolated 78kbps differential Manchester communication transceiver, and connectors for power, I/O, and the two wire network data bus. The TP/XF-78 Control Module requires only +5VDC, an EPROM, and user-supplied application electronics to form a complete node.

The TPT/XF-78 Transceiver Module consists of a module containing a transformer isolated 78kbps differential Manchester communication line interface and connectors for power, the Neuron Chip communication port lines, and the two wire network data bus. The TPT/XF-78 Transceiver Module is a complete network transceiver, and aside from requiring +5VDC and a Neuron Chip node, is otherwise self-contained.

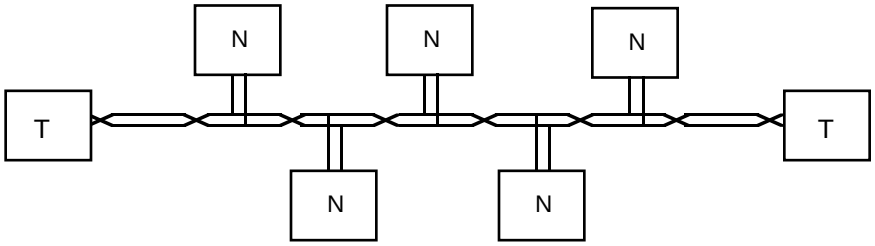
Bus Wiring

A LONWORKS bus topology entails wiring nodes in parallel along the length of a twisted wire pair, and terminating each end of the bus with the proper termination network (figure 1). This method of wiring is suitable for applications in which the following assumptions apply:

1 The typical fault mode is an open circuit;

Echelon Part Number 005-0032-01 REV C

2 It is not necessary for all nodes to communicate with one another in the event of a circuit fault.

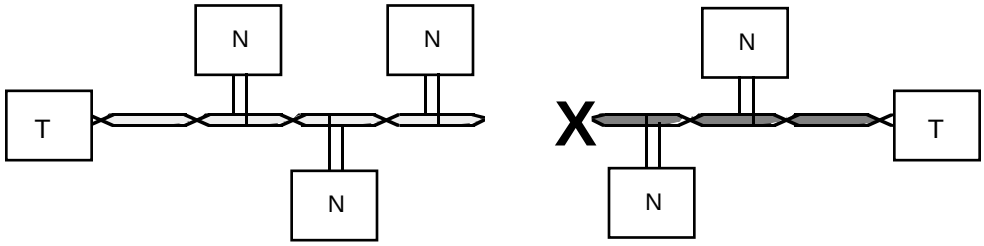


T = Termination Network
N = Neuron Chip Node

Figure 1 Typical 78kbps Bus Wiring

In normal operation, all of the nodes can receive packets from, and transmit packets to, any other node via the twisted pair wire. The termination network suppresses signal ringing and insures stable network performance over the specified wire distance and number of nodes.

In the event of an open circuit fault, the twisted pair bus will be split into two pieces (figure 2). In this case, packets will be unable to reach every node in the network because the open circuit will block packet transmissions across the fault. Only those nodes that are directly connected on each side of the open fault will be able to communicate with one another, assuming the absence of excessive reflections and ringing due to the loss of both network terminations.



T = Termination Network
N = Neuron Chip Node

Figure 2 Typical 78kbps Bus Wiring with Open Fault

There are situations in which it is desirable to maintain network communications in the event of a circuit fault. In these cases, it is important that the fault condition be identified and that the system take some corrective action to automatically restore network communications with all of the nodes. This situation can be handled using loop wiring.

Self-Healing Ring Wiring

A LONWORKS loop topology entails wiring nodes in parallel along the length of a 200 meter twisted wire pair, and terminating each end of the bus at an Intelligent Switch that includes suitable termination networks, two Neuron Chips, and a relay bus switch (figure 3). In normal operation of the loop the Intelligent Switch is open, and the network communications flow in a manner identical to the bus topology. In the event of an open fault, however, an application program causes the Intelligent Switch to close, effectively reconnecting the network to nodes on both sides of the open fault. Communications continue with all nodes in spite of the open fault condition.

The role of the Intelligent Switch is critical to the detection and correction of an open fault condition. Since the Intelligent Switch is normally open and the loop is not completed, an open fault on the wiring will be detected as a communication failure with those nodes that become isolated from the rest of the network. This fault condition can then be reported as a wiring failure to maintenance personnel. It is only after the detection of this fault condition that the Intelligent Switch closes, healing the loop and re-establishing communications with all nodes.

If the Intelligent Switch were absent and the network wired as a loop, an open fault condition would not be detected. This is because the network signals would still reach all of the nodes without interruption by traversing around both sides of the loop and avoiding the open fault. The open fault would therefore go undetected, and maintenance personnel would not be notified of the problem. If a second open fault subsequently occurred, communications would be lost with all of the nodes between the two open faults, and automatic self-healing would not be possible.

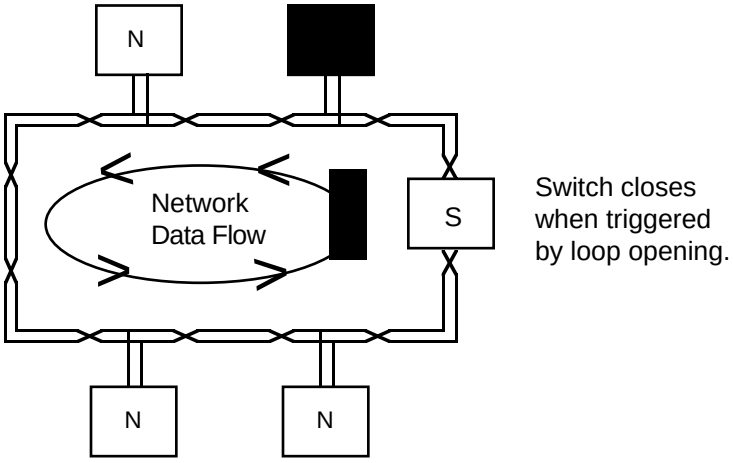
The loop/Intelligent Switch method of wiring is suitable for applications in which the following assumptions apply:

1 The typical fault mode is an open circuit;

2 It is necessary for all nodes to communicate with one another at all times, even in the event of a circuit fault.

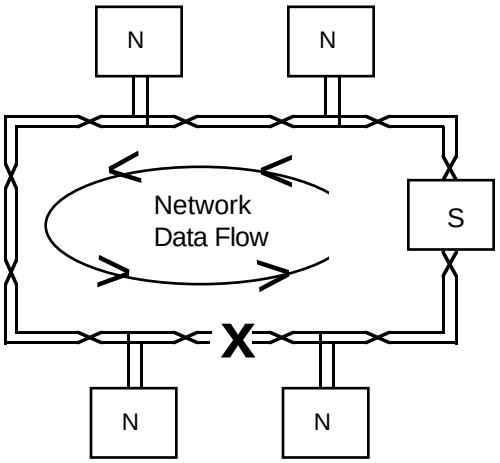
In normal operation, all of the nodes can receive packets from, and transmit packets to, every other node via the twisted pair wire. In the event of an open circuit fault, the twisted pair bus will be split into two pieces (figure 4). This condition causes an application program to trigger the Intelligent Switch, causing the loop to be closed and restoring communications to all of the nodes.

One of the advantages of the loop architecture is that it will automatically restore communication in the event of an open circuit condition. This self-healing design is especially important in applications where every node must be able to communicate with every other node at all times. Typical applications include fault alarm networks, machine controls, fire and security alarms, and process controls.



N = Neuron Chip Node
S = Switch & Termination Node (Closes Loop When Activated)

Figure 3 Typical 78kbps Loop Wiring



N = Neuron Chip Node
S = Switch & Termination Node (Closes Loop When Activated)

Figure 4 Typical 78kbps Loop Wiring with Open Fault

The Intelligent Switch itself is made of the following components shown in the Intelligent Switch box in figure 5. The operation of the Intelligent Switch in the event of an open fault proceeds according to the following steps:

1 Master Neuron Chip opens the Intelligent Switch.

2 Master Neuron Chip sends a packet to the slave Neuron Chip. Since the Intelligent Switch is open, this packet must traverse the entire twisted pair loop in order to reach the slave NEURON CHIP.

3 If the slave Neuron Chip acknowledges the packet, return to step 2. Otherwise proceed to step 4.

4 The master Neuron Chip sequentially sends query status messages to all of the nodes on the loop. This requires that the master has a table of node addresses for the channel in EEPROM memory.

5 The master Neuron Chip pulses the I/O line labelled "Test."

6 The slave Neuron Chip executes the operation described in step 4 (sending packets to all of the nodes on the loop, reading the locations out of EEPROM.)

7 The slave Neuron Chip pulses the I/O line labelled "Complete."

8 The master Neuron Chip closes the Intelligent Switch and sends a packet to the slave Neuron Chip to request the responses received by the slave Neuron Chip under step 6.

9 The master Neuron Chip compares its list of responses with the list received from the slave Neuron Chip and reports a cable break between locations A and B.

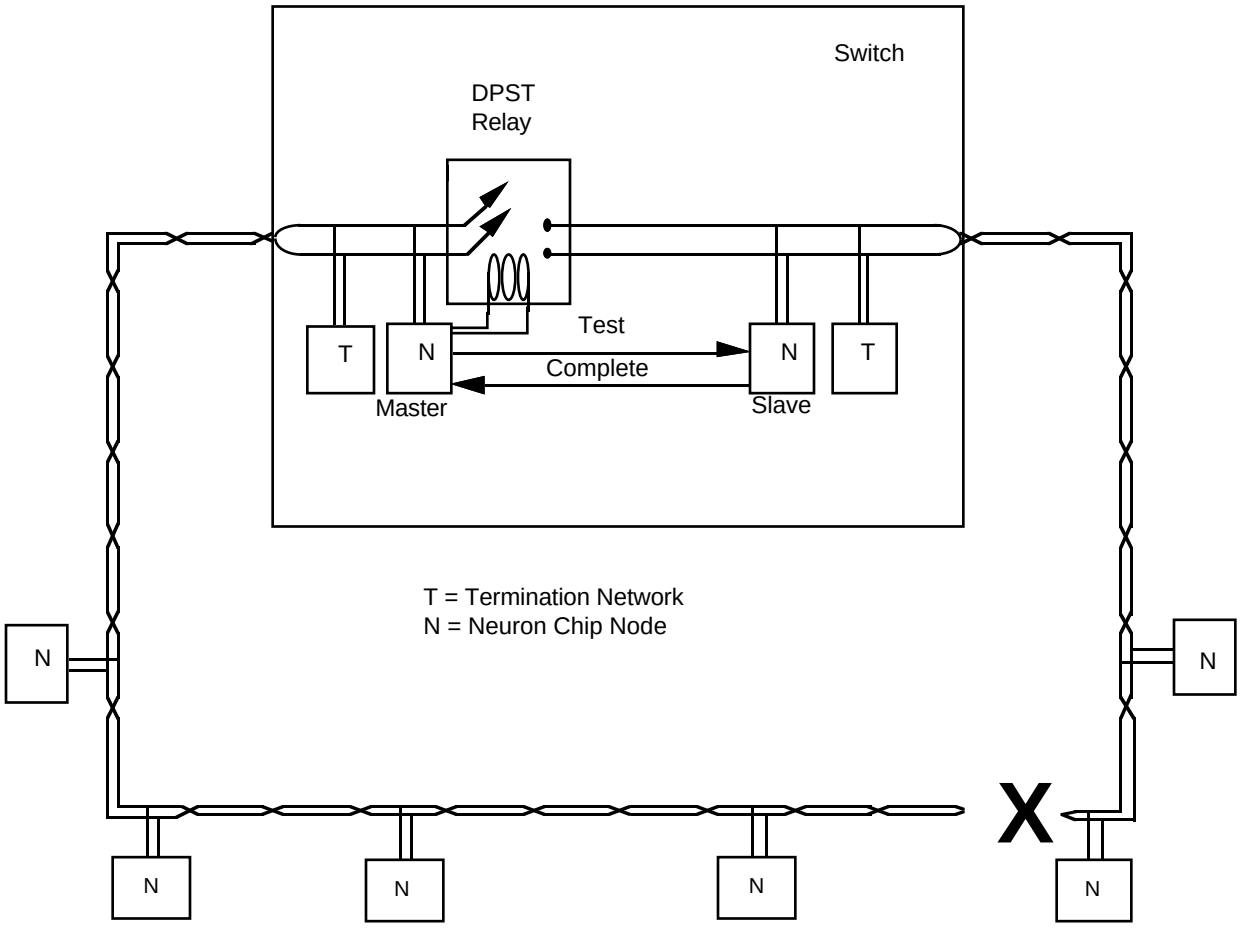


Figure 5 Intelligent Switch Block Diagram

LONWORKS® Custom Node Development

January 1995 **LONWORKS Engineering Bulletin**

Introduction

This engineering bulletin describes the steps needed to design and test a LONWORKS custom node. By definition, a custom node includes a Neuron® Chip and contains all the components necessary to function as a LONWORKS node. This bulletin describes custom nodes that run their entire application program on the Neuron Chip; see the LONWORKS *Host Application Programmer's Guide* for a description of custom nodes based on other host processors.

A custom node may be intended for shipment to customers or intended for use as a prototype to analyze the node's behavior in its working environment. In both cases, the node is based on custom hardware remote from the LonBuilder® or NodeBuilder™ hardware connected using an appropriate transceiver, or implemented with the NodeBuilder LTM-10 node.

Before moving to a custom node, the Neuron C application program for the node, along with its associated I/O hardware, should be tested using a LonBuilder Developer's Workbench or NodeBuilder.

Although not residing within the LonBuilder Development Station, a custom node's interaction with other nodes (emulators or custom nodes) may be closely monitored using the LonBuilder or LonManager® Protocol Analyzer. In addition, the LonBuilder Network Manager or LonMaker Installation Tool may be used to perform network management functions including application code downloading, binding, node testing, and reconfiguration.

Both LonBuilder and NodeBuilder permit browsing of network variables of a node. This feature allow the user to easily verify a node's input and output network variables' behavior.

Echelon's LONWORKS transceivers and control modules provide a straightforward and quick means to move from the development environment to a custom node using off-the-shelf-parts. Restrictions on form factor, special transceiver requirements, and the need for different memory configurations may require the development of a custom node without use of the control modules. This bulletin describes how to build a custom node whether or not you are using LONWORKS control modules.

In addition, a list of common pitfalls is included at the end of this document. This list includes hints, advice, and assorted bits of wisdom collected from experiences in developing custom nodes.

Echelon Part Number 005-0024-01 REV C

Related Documentation

The following is a list of documents available from Echelon that provide additional information related to custom nodes:

- *Neuron Chip Data Book.* Describes the memory, I/O, and communications ports of the Neuron Chip, together with the support circuitry required for clock, reset, and service functions. Pin assignment, pad layout, and electrical and environmental specifications are also provided.
- *Neuron 3150 Chip External Memory Interface* engineering bulletin. Describes the interface and timing requirements for the memory bus of the Neuron 3150 Chip. Describes interfaces to PROM, RAM, EEPROM, flash memory, and NVRAM.
- *TPT/XF-78 Twisted Pair Transceiver Module Data Sheet.* Describes a 78kbps twisted-pair transceiver module which can be used as a drop-in transceiver solution for any custom node.
- *TPT/XF-1250 Twisted Pair Transceiver Module Data Sheet.* Describes a 1.25Mbps twisted-pair transceiver module which can be used as a drop-in transceiver solution for any custom node.
- *LPT-10 Link Power Twisted Pair Transceiver Module Data Sheet*. Describes a free-topology 78kbps twisted-pair transceiver module which permits the data and power delivery on the same pair of wires.
- *FTT-10 Free Topology Twisted Pair Transceiver Module Data Sheet*. Describes a free-topology 78kbps twisted-pair transceiver module which can be used as a drop-in transceiver solution for any custom node.
- *PLT-10 Power Line Transceiver Module Data Sheet*. Describes a 10kbps spread spectrum power line transceiver which can be used as the core module for developing a power line communication interface for a custom node.
- *PLT-20 Power Line Transceiver Module Data Sheet*. Describes a 5kbps narrow-band power line transceiver which can be used as the core module for developing a power line communication interface for a custom node.
- *PLT-30 Power Line Transceiver Module Data Sheet.* Describes a 2kbps spread spectrum power line transceiver which can be used as the core module for developing a power line communication interface for a custom node.
- *LonBuilder Hardware Guide.* Describes the process of software and hardware prototyping of custom nodes using LonBuilder. Chapter 2 describes the hardware products that can be used for prototyping and developing custom nodes. These products include the LonBuilder I/O Evaluation Board, the LonBuilder Extender Card, and the LonBuilder Application Interface Kit. The LonBuilder Application Interface Kit includes the LONWORKS Module Application Interface which is used to debug custom nodes based on LONWORKS control modules.

- *LonBuilder User's Guide.* Describes the process of software and hardware prototyping of custom nodes using LonBuilder. Chapter 7 includes a detailed discussion of custom node development.
- *NodeBuilder User's Guide.* Describes the process of software and hardware prototyping of custom nodes using NodeBuilder.
- *LTM-10 User's Guide.* Describes how the NodeBuilder LTM-10 hardware can be used to develop and test custom node hardware.
- *Neuron C Programmer's Guide.* Provides an overview of the Neuron C programming language.
- *Neuron C Reference Guide.* Provides reference information on the functions and features available within Neuron C.
- *LONWORKS Host Application Programmer's Guide.* Describes the process of developing application programs that run on host processors other than the Neuron Chip.
- *LONWORKS Twisted Pair Control Module User's Guide.* Describes the interface requirements for custom nodes using LONWORKS Control Modules.

Bringing Up a Custom Node

This section discusses the steps required to bring up a custom node. The actual procedure applies to both Neuron 3150 and 3120xx Chip-based custom nodes unless otherwise noted. To create a custom node, a developer must:

- Create the node's application program
- Specify the node's hardware
- Program the node's memory
- Install the node on a network
- Load the node
- Test the node

Figure 1 summarizes the steps required to bring up Neuron 3150 and 3120xx Chip-based LONWORKS custom nodes using the LonBuilder Development Station.

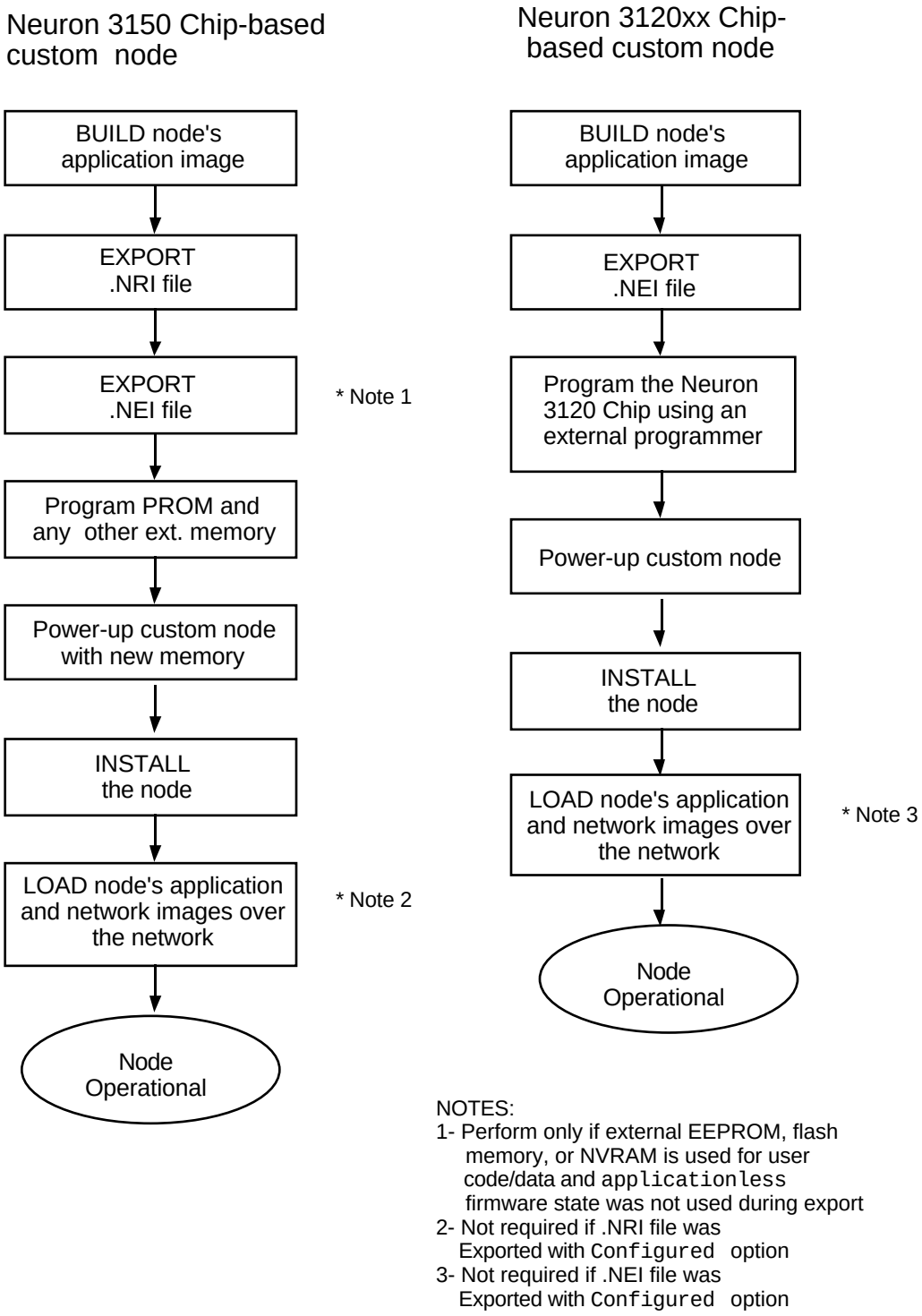


Figure 1 Steps for bringing up a LONWORKS custom node with a LonBuilder Development Station

Figure 2 summarizes the steps requires to do the same with NodeBuilder.

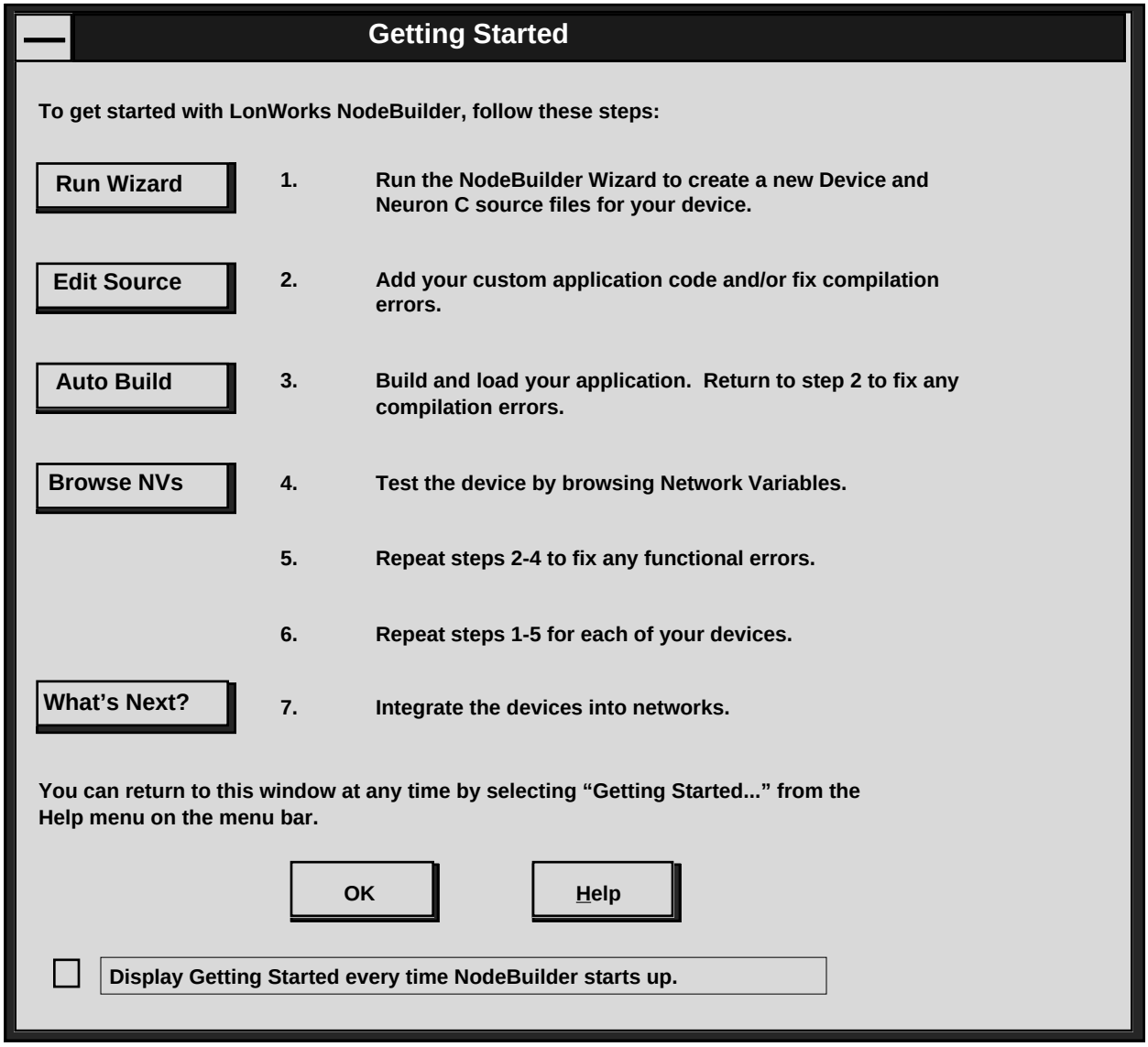


Figure 2 The Getting Started Screen from the NodeBuilder Help Menu

Creating an Application Program

The first step in creating a custom node is to create the application program for the node. The application program should be tested and debugged initially on a LonBuilder Neuron Emulator or NodeBuilder LTM-10 node. This allows the application program to be debugged independently of the hardware design. The *LonBuilder User's Guide* and the *NodeBuilder User's Guide* describe the process of developing and testing an application program.

Specifying the Node's Hardware

A custom node is a LONWORKS node implemented by a LonBuilder or a NodeBuilder user. A custom node can be programmed, implemented, and tested with the LonBuilder Developer's Workbench or NodeBuilder as long as it meets the specifications in the following section. Figure 3 below shows the basic components of a LONWORKS custom node. The components inside the dashed box may be replaced by a LONWORKS Control Module.

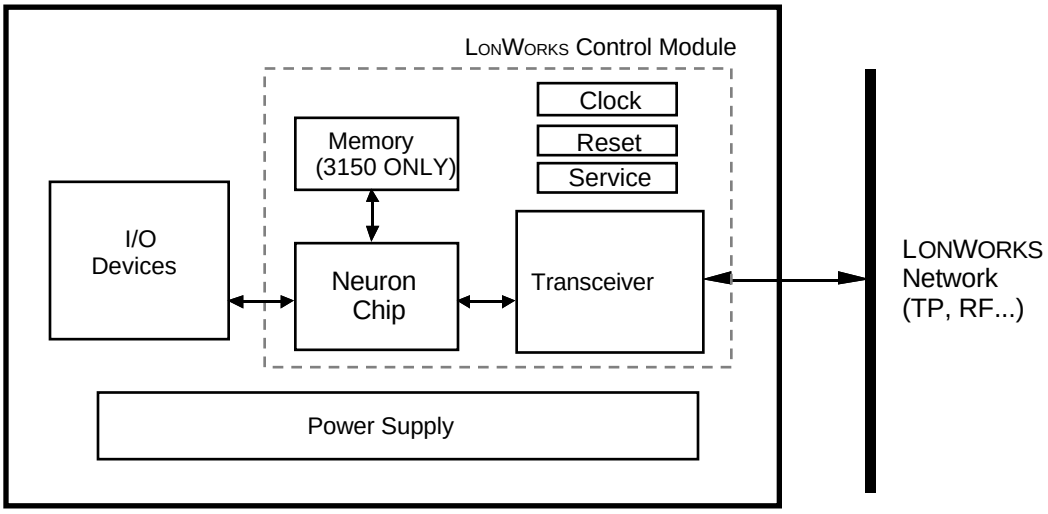


Figure 3 A LONWORKS custom node

Note that external memory is not needed for a node based on a Neuron 3120xx Chip. The following is a description of each of the components needed in a LONWORKS node.

- **Neuron Chip**

  A Neuron 3120, 3120E1, 3120E2, or 3150 Chip. The Neuron Chip provides the control, communications, and I/O for the node. The Neuron Chip may be the only processor on a custom node, or the Neuron Chip may be used as an interface to another microprocessor or microcontroller as described in the *LONWORKS Host Application Programmer's Guide.* The Neuron Chip is programmed as described in the next section.

- **Off-Chip Memory**

  Custom nodes implemented with the Neuron 3120, 3120E1, or 3120E2 Chip use only the memory on the Neuron Chip and do not have off-chip memory. Custom nodes implemented with the Neuron 3150 Chip must have at least

16Kbytes of off-chip ROM or flash memory for the Neuron Chip firmware, and may have additional ROM, EEPROM, flash memory, and RAM as or described below. The off-chip memory is programmed as described in the next section:

- **ROM** 0 or 16 - 68Kbytes (in multiples of 256 bytes) of contiguous memory starting at 0. It may be any type of ROM, PROM, EPROM, or NVRAM. However, the memory must be non-volatile and the custom node developer must program the memory as described in the next section.
- **FLASH** 0 or 21.5 - 42Kbytes (in multiples of 256 bytes) of contiguous flash memory located after the ROM block, if any, but before the RAM block, if any. If there is no ROM on the node, flash memory must start at address 0000 and there must be at least 21.5Kbytes. Flash memory is automatically protected by the Neuron Chip firmware.
- **EEPROM** 0 - 42Kbytes (in multiples of 256 bytes) of contiguous EEPROM located after the ROM block and before the RAM block, if any, but not necessarily contiguous with either. The write time may be configured from 0 to 255 ms. Circuitry should be included for write protecting the EEPROM during power down/power up, if necessary. This can also be other non-volatile memory, such as battery-backed RAM.
- **NVRAM** 0 - 58Kbytes (in multiples of 256 bytes) of contiguous non-volatile RAM located after the ROM block but before the RAM block, if any. NVRAM may be used for the Neuron Chip firmware. Memory used for the Neuron Chip firmware may be write protected, but the custom node developer is responsible for controlling the write protection circuitry.
- **RAM** 0 - 42Kbytes (in multiples of 256 bytes) of contiguous read/write memory.
- **MEMORY MAPPED I/O** 0 - 42Kbytes (in multiples of 256 bytes) of address space dedicated to memory mapped I/O.

For details on interfacing memory to a Neuron Chip 3150, see the *Neuron Chip 3150 External Memory Interface Engineering Bulletin.*

- **Transceiver**

A transceiver is required to interface the custom node with a LONWORKS network. The transceiver must meet the specifications for the communications port interface. Transceiver modules for various media are available from Echelon and other vendors. Echelon provides the following transceivers: TPT/XF-78 and TPT/XF-1250 twisted pair transceivers, LPT-10 link power transceiver, FTT-10 free topology transceiver, PLT-10, PLT-20, and PLT-30 power line transceivers.

- **Clock**

  An input clock for the Neuron Chip. The clock must run at one of the standard input clock rates: 10MHz, 5MHz, 2.5MHz, 1.25MHz, and 625kHz. Refer to the *Clock* section of the *Neuron Chip Data Book* for more information on the clock circuitry.

- **Reset**

  A power-on reset circuit is required to reset a custom node on power up. A reset button is optional, but is useful for resetting a custom node at times other than power up. An LVI (Low Voltage Interrupt) device should be connected to the Neuron Chip's ~Reset pin in order to ensure that the Neuron Chip is not running below a minimum power supply voltage. Refer to the *Reset* section of the *Neuron Chip Data Book* for more information on the reset circuit requirements.

- **Service Button and Service LED**

  A service button is used to install a custom node on a network. When a custom node is installed in a development network, the LonBuilder or NodeBuilder software requests that the user press the service button which causes the Neuron Chip firmware to send a service message. The network manager uses the service message to install the custom node. A service LED is optional, but is highly recommended as it is useful for diagnosing the state of the Neuron Chip firmware: on indicates that the node is applicationless (does not have an application loaded), blinking indicates that the node has an application but has not yet been configured, off indicates that the node has an application and has been configured. Refer to the *Service* section of the *Neuron Chip Data Book* for more information on the service circuitry.

- **I/O Interface**

  An interface between the eleven Neuron Chip I/O pins and external I/O devices. The interface and devices are dependent upon the custom node application.

Programming the Node's Memory

Once the node's application program has been debugged and the custom node hardware has been built, the next step is to program the node's memory. The task of programming memory involves the *export* of the appropriate files. Custom nodes contain memory within the Neuron Chip, and may contain off-chip memory when using the Neuron 3150 Chip. When a node is mass produced, the Neuron Chip, external ROM or flash memory may be programmed as part of the manufacturing process. During development much of the on-chip and off-chip memory may be programmed over the network by the LonBuilder Network Manager or NodeBuilder, with the following two exceptions:

- **Communications Parameters**

The communications parameters are contained in the Neuron Chip's EEPROM and are a portion of a node's application image. The communications parameters define the type of transceiver connected to the Neuron Chip and are used by the Neuron Chip firmware for interfacing with the transceiver. Once a node is installed on a development network, the communications parameters can be changed over the network by LonBuilder or NodeBuilder. However, for a node to be installed on a development network and communicate with the development tool, the node's communications parameters must match the node's transceiver. The initial default communications parameters on a new Neuron Chip are for a 1.25 Mbps twisted-pair or direct-connect transceiver. Therefore, nodes implemented with a 1.25Mbps twisted-pair or direct-connect transceiver do not require their communications parameters to be changed.

For nodes based on the Neuron 3150 Chip, the ROM or flash image created by the LonBuilder or NodeBuilder software defines the initial communications parameters that are copied to internal EPROM when the boot ID in EEPROM does not match the boot ID in the ROM or flash image. If the communications parameters are modified over the network, the original defaults in the ROM or flash image are not used unless the ROM or flash memory is changed. For nodes based on the Neuron 3120xx Chip, the communications parameters must be programmed by using an external Neuron Chip programmer. NodeBuilder does not update communications parameters other than priority.

- **ROM or Flash**

A custom node's ROM cannot be programmed over the network and therefore must be programmed prior to powering-up the node. This is also true of flash memory that takes the place of ROM and contains the system image. The ROM always contains at least the Neuron Chip firmware. On a Neuron 3150 Chip with off-chip memory, the ROM or flash memory can also contain all or part of the application image as described under *Off-Chip Memory* in Chapter 6 of the *Neuron C Programmer's Guide*. The Neuron 3120xx Chip's ROM is pre-programmed and does not require programming by the user. See *Building Custom Nodes* in Chapter 7 of the *LonBuilder User's Guide* for instructions on how to create ROM or flash files that can be used with a PROM programmer to create the ROM or flash chips for a Neuron 3150 chip-based custom node. Refer to *Exporting a LONWORKS Device* and *Building a Custom Node* topics under the NodeBuilder online help for more information.

Installing the Node on a Network

A newly developed custom node will initially be installed on a development network that includes a LonBuilder Development Station or a NodeBuilder. This allows the LonBuilder network management, protocol analysis, and network variable browsing tools and the NodeBuilder's debugging and network variable browsing tools to be used to test the node and its interaction with other nodes.

When developing multiple nodes with LonBuilder, the developer should focus on one node at a time when making the transition from an emulator-based network to a custom node-based network, so that potential problems may be isolated.

Installation of a custom node in a network involves the identification of the node by a network management tool using the node's Neuron ID. LonBuilder and NodeBuilder get this ID from the service pin message issued when the node's service pin is activated. This means that the node must be able to communicate over the network, which implies that it must have a working transceiver and appropriately configured communication parameters. These parameters are set during the programming step. In addition, any routers between the node and the network manager must be installed and loaded. See *Installing Application Node Target Hardware* in Chapter 6, and *Installing Custom Nodes* in Chapter 7, of the *LonBuilder User's Guide*. Refer to the *Building and Loading a LONWORKS Device* topic under the NodeBuilder online Help for more information.

To move an application program from a LonBuilder Neuron Emulator to a custom node, the hardware properties specification must match those of the custom node hardware. Hardware properties are described under *Defining Properties* in Chapter 6 of the *LonBuilder User's Guide*. After this is checked, three modifications may be required from the LonBuilder software for installing the custom node:

- Create a new target hardware object with a hardware type of *Custom Node*. This step is described under *Defining Target Hardware* in Chapter 6 of the *LonBuilder User's Guide*.
- If the media characteristics for the custom node are different from those used when the node was debugged on a LonBuilder Neuron Emulator, creating a new channel object as described under the *Defining Channels* in Chapter 10 of the *LonBuilder User's Guide*. Ensure that a communications path exists from the network manager and protocol analyzer to this channel, either by moving the network manager and protocol analyzer to the new channel, or by installing and loading a router to the new channel. Debugging a transceiver is simplified by first testing it on a LonBuilder Neuron Emulator, in which case the channel definitions will be the same.
- Change the definition of the node specification to select the new target hardware and channel objects. This step is described under *Defining Node Specifications* sections in Chapter 6 of the *LonBuilder User's Guide*.

Loading the Node

Loading a custom node is required if the node has been created using either the `applicationless` or the `unconfigured` state, or if you wish to make changes to the application or configuration without reprogramming the node's memory. A node created in the `configured` state does not need to be loaded for it to be fully functional. See *Loading Built Nodes and Routers* in Chapter 7 of the *LonBuilder User's Guide*. Refer to the *Building and Loading a LONWORKS Device* topic under the NodeBuilder online Help for more information.

For a Neuron 3150 Chip-based node, the loading process involves updating application image parts that reside in external flash memory, EEPROM, and NVRAM, in addition to the network image. This includes the internal EEPROM and any external memory declared as EEPROM that may exist on the node. Application code allocated to RAM by the user will also be updated. For a Neuron 3120xx Chip-based node, the loading process only updates the on-chip EEPROM.

Reloading a Neuron 3150 Chip-based custom node with application changes will require the programming step to be repeated if any application code will reside in ROM. This is because changes to the program linkage may invalidate the code already there. If you are going to change the remainder of the application over the network, you only need to program the ROM. If you want the node to come up in the `configured` or `unconfigured` state, you must also reprogram any portion of the application image in external flash memory, EEPROM, or RAM. You should load the node after you have installed the new ROM.

If you want to be able to download new application programs completely over the network, limit the ROM memory map to 16 Kbytes so that all of the application program is loaded into the on-chip EEPROM. Alternatively, you could use flash memory for the application program which could also contain the Neuron Chip firmware.

Changing the communication parameters may cause the node to stop functioning if the new parameters do not match the transceiver attached to the Neuron Chip.

Testing the Node

The LonBuilder Application Interface Kit (AIK), available from Echelon, is specifically designed to help in the debugging process of custom nodes. Two adapters are included for debugging control module-based devices or custom Neuron-based devices. The control module adapter replaces the LONWORKS control module in the custom node to allow connection to a LonBuilder emulator for both I/O and transceiver testing.

The Neuron Chip adapter is designed specifically for use on custom nodes which are not based on the LONWORKS control module family of products. This is accomplished through a surface mount adapter which essentially replaces the Neuron Chip on the target board with the necessary hardware and interface logic for proper connection to a LonBuilder emulator. I/O and transceiver testing can then be accomplished in much the same way as with the application interface kit.

See the *LonBuilder Hardware Guide* for information on the application interface kit.

Assuming no hardware problems, once the custom node has been properly installed and loaded, the application on that node should start running.

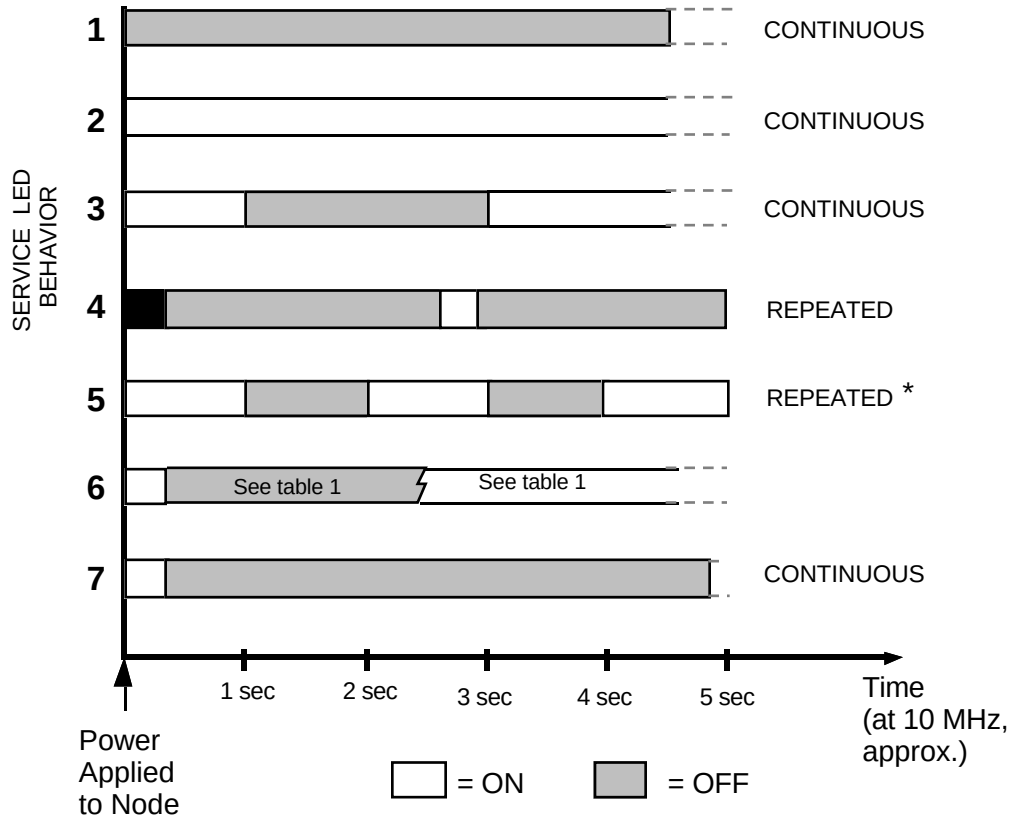
In addition to the application I/O on the node, the condition of the service LED may be observed for clues about the actual state of the node. The behavior of the service

LED, specifically the duration of ON and OFF states, varies depending on the operating state of the node.

Figure 3 illustrates the different service LED behaviors alternatives for any Neuron Chip-hosted node. These are the most common behaviors, others are possible since the state of the service LED is under firmware control and can be affected by both hardware and software anomalies. The state of the service LED can also be affected by the application program using the `active_service_led` built-in variable.

Table 1 describes each of the behaviors shown in Figure 4 under different contexts. Again, this list is not exhaustive and therefore does not provide explanations for every possible service LED behavior.

Behavior 7 represents that of a fully configured node during normal operation.



*Does not scale with the Neuron Chip clock

Figure 4 Possible service LED behaviors showing different duty cycles

Table 1 Explanation of the service LED behaviors shown in Figure 3

BEHAVIOR	CONTEXT	LIKELY EXPLANATION

1	Power-up of a Neuron 3120xx Chip-based node, or a Neuron 3150 Chip-based node with any PROM	Bad node hardware
2	Power-up of a Neuron 3120xx Chip-based node, or a Neuron 3150 Chip-based node with any PROM	Bad node hardware
3	Power-up/Reset	Node is applicationless May be caused by the Neuron Chip firmware when a mismatch occurs on application checksums This behavior is normal if the application was exported to come up `applicationless`
4	Anytime	Watchdog timer resets occurring Possible corrupt EEPROM For a Neuron 3150 Chip-based node, use a newly programmed PROM, or EEBLANK and follow bring-up procedure If you are using an LTM-10 node, you can make the Neuron Chip applicationless by holding down the service button while pressing the rest button. Hold the service button down until the reset LED briefly flashes.
5	Anytime	Node is unconfigured but has an application. Proceed with loading the node.

6	Using EEBLANK on a Neuron 3150 Chip-based custom node	The OFF duration is approximately 10 seconds (10MHz Neuron Chip). This duration scales proportional to the system clock. After this OFF time the service LED should turn ON and stay ON, indicating the completion of the blanking process
6	First power up with a new PROM on a Neuron 3150 Chip-based custom node. `Applicationless` firmware state exported	The OFF duration is approximately 1 second. Service LED should then turn ON and stay on indicating an applicationless state
6	First power up with a new PROM on a Neuron 3150 Chip-based custom node. `Unconfigured` firmware state exported	The OFF duration is 1-15 seconds depending on the application size and system clock. Service LED should then begin flashing as in behavior 5 shown in figure 3, indicating an unconfigured state
6	First power up with a new PROM on a Neuron 3150 Chip-based custom node. `Configured` firmware state exported	The OFF duration is indefinite (1-15 seconds to load internal EEPROM; stays OFF indicating configured state.)
7	Anytime	Node in configured and running normally

As a first step in establishing communication with a newly powered-up (previously blank) custom node, the service pin may be used. Activating the service pin on any node running the Neuron Chip firmware, regardless of the state of the application and network configuration for that node, causes a broadcast message containing the Neuron ID and the program ID for that Neuron Chip to be sent on the network. The LonBuilder Protocol Analyzer or the LonManager Protocol Analyzer can be used to monitor the network for any such messages.

Activating the service pin on any node can serve as a simple test; it should cause the protocol analyzer status LED to blink, indicating the presence of a packet on the network. The protocol analyzer can be used to log the service pin message as described in Chapter 12, *Monitoring Development Networks*, of the *LonBuilder User's Guide*. The logged service pin message indicates that the node hardware and the network connection for that node are operating normally.

It is possible for the protocol analyzer to receive a packet but not have it logged. This may indicate a communication parameter mismatch between the protocol analyzer and the node sending the packet.

Once communication has been established with a node and the node has been installed, the *Test* and *Wink* commands may be issued from the LonBuilder network manager or NodeBuilder to further test the hardware and communication integrity of the custom node. The network manager also includes a network variable browser that can be used to send network variable updates to the custom node and receive network variable updates from the node. The browser can therefore be used to validate that a custom node is functioning correctly. Use of the LonBuilder network manager is described in Chapter 13, *Testing Development Networks*, of the *LonBuilder User's Guide.* Refer to the *Testing a LONWORKS Device* topic of the NodeBuilder online help for more information.

Once all the nodes have been installed on the development network and tested as custom nodes, they may be disconnected from the development network and moved to a production network. Or, the debugged node image may be used to produce the additional custom nodes. See *Exporting and Importing* in Chapter 7 of the *LonBuilder User's Guide*. Refer to *Exporting a LONWORKS Device* topic of the NodeBuilder online help for more information.

Common Pitfalls

This section lists common pitfalls along with hints and suggestions to use during custom node development and bring-up.

• *Installation of custom node fails.* The network manager and the custom node may not have matching transceiver properties and/or hardware. This problem may also be caused if there is no communication path between the network manager and the custom node.

• *Protocol Analyzer not receiving or losing packets.* The protocol analyzer and the custom node do not have matching transceiver properties and/or hardware. This problem may also be caused if there is no path between the protocol analyzer and the custom node. Also make certain that proper termination exists if using a twisted-pair network.

• *Loading errors while trying to load a custom node in a highly active network.* Heavy network traffic on the network could cause a node to miss LonBuilder loading messages. In such a case, the network statistics screen will show a very low network bandwidth utilization figure. In addition, the *test* command should indicate many missed messages for that node. A solution is to take the node(s) causing the heavy traffic offline before trying to load the node.

• *A custom node stops working after a reload.* The communication parameters were changed (LonBuilder only) and then installed into the Neuron Chip of the custom node. If the new parameters have taken effect and if they require the

custom node to have a new transceiver, the node will no longer communicate with the network. For a Neuron 3150 Chip-based custom node, use EEBLANK (see Chapter 7, *Programming Custom Nodes*, in the *LonBuilder User's Guide)* and proceed with bring-up as shown in Figure 1 or 2. For a Neuron 3120xx Chip-based custom node, change the node's transceiver so that it corresponds to the newly installed communication parameters.

• *Neuron 3150 Chip-based node hangs during application/configuration update procedure.* The EEPROM data inside the Neuron Chip has been corrupted. Use EEBLANK and proceed with bring-up as shown in Figure 1 or 2. Or else, program a new PROM with the `unconfigured` or `configured` options.

• *Service LED exhibits behavior not shown in Figure 3.* Bad node hardware. Check for possible mis-wirings and PCB faults.

The SNVT Master List and Programmer's Guide

May 1997

Overview

Standard Network Variable Types (SNVTs) facilitate interoperability by providing a well-defined interface for communication between nodes made by different manufacturers. A node may be installed in a network and logically connected to other nodes via network variables as long as the data types match. A list of all available SNVTs and details of their definitions is provided in this document. For further explanation of SNVTs please refer to the *Neuron® C Programmer's Guide* and the *LONMARK® Application Layer Interoperability Guidelines.*

The Master SNVT List

The Master SNVT list shown in Table 1 provides information on the definition of all available SNVTs. SNVTs are expressed as either fixed point or floating point numbers, enumeration lists or structures.

The representation for all fixed point numeric SNVTs is as follows:

SNVT Range			Type Definition
0	..	65535	unsigned long
-32768	..	32767	signed long
0	..	255	unsigned short
-128	..	127	signed short

The representation for the floating point SNVTs is ANSI/IEEE 754 floating point: 1 sign bit, 8 exponent bits, and 23 mantissa bits, for a total of 32 bits. For all the floating point SNVTs, the range is approximately -1E38 to +1E38 units. Floating point objects may be declared as local variables and as network variables, and can be communicated across the network. These data types are compatible with the Neuron C Extended Arithmetic type `float_type`. For more details on the use of floating point, see the *Neuron C Reference Guide*. All units are Système Internationale (SI) except those shown in *italics.*

The Master SNVT List will be updated periodically as new SNVTs are added. Refer to the *LONMARK Application Layer Interoperability Guidelines* Appendix A for the procedure to follow for proposing new SNVTs.

The definitions of SNVTs are made available to various software tools through a set of special files contained in the PKZIP archive `SNVT_Vn.ZIP`, where *n* is the version number (`SNVT_V8.ZIP` for this release). This file may be accessed on the World Wide Web at `http://www.lonworks.echelon.com`. Log in or register for the Developer's Toolbox and select Updates.

Echelon Part Number 005-0027-01 REV K

The SNVT Master List and Programmer's Guide (this document) may be downloaded in Adobe Acrobat format from the World Wide Web at `http://www.lonmark.org`.

SNVT files should be installed in the following directories, depending on which software products are installed on your PC. All directory names are default, and should be modified appropriately if you have installed the products in other directories. The directory names shown are examples only, these names may be different if the "destination directory" is changed from the default at the time of installation. If you re-install any of these products from the original installation disks, the SNVT files may be overwritten, and you should re-install the latest version SNVT files again.

Software	File Name	Install Directory
LonBuilder®		
SNVT type file	SNVT.TYP	\LB\TYPES
SNVT include files	SNVT_*.H	\LB\INCLUDE
Profiler		
SNVT type file	SNVT.TYP	\LNP
LonMaker™		
SNVT type file	SNVT.TYP	\LNM
SNVT enum file	SNVT.ENM	\LNM
LonManager® API for DOS		
SNVT type file	SNVT.TYP	\LM_API\DB
SNVT enum file	SNVT.ENM	\LM_API\DB
LonManager API for Windows		
SNVT type file	SNVT.TYP	\LM_WIN\DB
SNVT enum file	SNVT.ENM	\LM_WIN\DB
LNS for Windows		
SNVT type file	STANDARD.TYP	\LONWORKS\TYPES
SNVT enum file	STANDARD.ENM	\LONWORKS\TYPES
All Other Products		
SNVT type file	SNVT.TYP	\LONWORKS\TYPES
SNVT enum file	SNVT.ENM	\LONWORKS\TYPES
SNVT format file	SNVT.FMT	\LONWORKS\TYPES
SNVT include files	SNVT_*.H	\LONWORKS\NEURONC\INCLUDE

For NodeBuilder®, the special format file `NB_SNVT.FMT` should be installed in the `\LONWORKS\NB` directory. Note that the `\LONWORKS` directory may be named `\ECHELON` on some systems. Before installing `SNVT.FMT` or `NB_SNVT.FMT`, delete all files with the `.ACH` and `.LS` extensions in the same directory.

This version of the Master SNVT List corresponds to version 8 of the SNVT files. The following table shows the number of SNVTs defined in this and earlier versions of the SNVT files. The DOS utility program SNVTVER.EXE will display the version of the SNVT.TYP file in the current working directory.

SNVT.TYP file version	STANDARD.TYP file version	SNVT IDs defined
4	-	1 - 101
5	-	1 - 113
6	-	1 - 114
-	7	1 - 114
8	8	1 - 122

New SNVTs are added in numerical order. The SNVTs added for version 8 are as follows:

SNVT ID	SNVT name
115	SNVT_scene
116	SNVT_scene_cfg
117	SNVT_setting
118	SNVT_evap_state
119	SNVT_therm_mode
120	SNVT_defr_mode
121	SNVT_defr_term
122	SNVT_defr_state

Table 1
Master SNVT List

Measurement	Name	Range (Resolution)	SNVT #
Address, Neuron Chip	SNVT_address	0x4000 .. 0xF1FF (hexadecimal)	114
Alarm state	SNVT_alarm	see Structures below	88
Angular velocity	SNVT_angle_vel	-3,276.8 .. 3,276.7 radians / sec (0.1 radians / sec)	4
	SNVT_angle_vel_f	-1E38 .. 1E38 radians / sec	50
	SNVT_rpm[3]	*0 .. 65,534 revolutions/minute (1 rpm)*	102
Area	SNVT_area[3]	0 .. 13.1068 m^2 (200 mm^2)	110
Character	SNVT_char_ascii	0 .. 255	7
Char string	SNVT_str_asc	see Structures below	36
	SNVT_str_int	see Structures below	37
Color	SNVT_color	see Structures below	70
Concentration	SNVT_ppm	0 .. 65,535 parts per million (1 ppm)	29
	SNVT_ppm_f	0 .. 1E38 ppm	58
Count, event	SNVT_count	0 .. 65,535 counts (1 count)	8
	SNVT_count_f	0 .. 1E38 counts	51
Count, incremental	SNVT_count_inc	-32,768 .. 32,767 counts (1 count)	9
	SNVT_count_inc_f	-1E38 .. 1E38 counts	52
Currency	SNVT_currency	see Structures below	89
Current	SNVT_amp	-3,276.8 .. 3,276.7 amps (0.1 A)	1
	SNVT_amp_f	-1E38 .. 1E38 A	48
	SNVT_amp_mil	-3,276.8 .. 3,276.7 mA (0.1 mA)	2
Date	SNVT_date_cal	Use SNVT_timestamp instead	10
Day of week	SNVT_date_day	see Enum Lists below	11
Defrost mode	SNVT_defr_mode	see Enum Lists below	120
Defrost termination	SNVT_defr_term	see Enum Lists below	121
Defrost progress	SNVT_defr_state	see Enum Lists below	122
Density	SNVT_density	0 .. 32,767.5 kg / m^3 (0.5 kg / m^3)	100
	SNVT_density_f	0 .. 1E38 kg / m^3	101
Emergency mode, HVAC	SNVT_hvac_emerg	see Enum Lists below	103
Energy, elec	*SNVT_elec_kwh*	*0 .. 65,535 kilowatt-hour (1 kWH)*	13
	SNVT_elec_whr	*0 .. 6,553.5 watt-hours (0.1 WHR)*	14
	SNVT_elec_whr_f	*0 .. 1E38 watt-hour*	68
Energy, thermal	*SNVT_btu_f*	*0 .. 1E38 btu*	67
	SNVT_btu_kilo	*0 .. 65,535 kilo btu*	5
	SNVT_btu_mega	*0 .. 65,535 mega btu*	6
Evaporator state	SNVT_evap_state	see Enum Lists below	118
File position	SNVT_file_pos	see Structures below	90
File request	SNVT_file_req	see Structures below	73
File status	SNVT_file_status	see Structures below	74
Flow	SNVT_flow[3]	0 .. 65,534 liters / sec (1 l / sec)	15
	SNVT_flow_f	-1E38 .. 1E38 l / sec	53
	SNVT_flow_mil	0 .. 65,535 milliliters / sec (1 ml / sec)	16

Table 1
Master SNVT List
continued

Measurement	Name	Range (Resolution)	SNVT #
Frequency	SNVT_freq_f	0 .. 1E38 Hertz	75
	SNVT_freq_hz	0 .. 6553.5 Hz (0.1 Hz)	76
	SNVT_freq_kilohz	0 .. 6553.5 kHz (0.1 kHz)	77
	SNVT_freq_milhz	0 .. 6.5535 Hz (0.0001 Hz)	78
Gain	SNVT_muldiv	see Structures below	91
Grammage	SNVT_grammage	0 .. 6,553.5 gsm (0.1 gsm)	71
	SNVT_grammage_f	0 .. 1E38 gsm	72
HVAC mode	SNVT_hvac_mode[2]	see Enum Lists below	108
HVAC override	SNVT_hvac_overid[2]	see Structures below	111
HVAC status	SNVT_hvac_status[2]	see Structures below	112
Humidity	SNVT_lev_percent	See Level, percent below	81
Illumination	SNVT_lux	0 ..65,535 lux (1 lux)	79
Installation source	SNVT_config_src	see Enum Lists below	69
Length	SNVT_length	0 .. 6,553.5 meters (0.1 m)	17
	SNVT_length_f	0 .. 1E38 meters	54
	SNVT_length_kilo	0 .. 6,533.5 km (0.1 km)	18
	SNVT_length_micr	0 .. 6,553.5 µm (0.1 µm)	19
	SNVT_length_mil	0 .. 6,533.5 mm (0.1 mm)	20
Level, continuous	SNVT_lev_cont	0 .. 100 % (0.5%)	21
	SNVT_lev_cont_f	0 .. 100 %	55
Level, discrete	SNVT_lev_disc	see Enum Lists below	22
Level, percent	SNVT_lev_percent[4]	-163.84% .. 163.83% (0.005% or 50 ppm)	81
Magnetic cards	SNVT_magcard	see Structures below	86
	SNVT_ISO_7811	Use SNVT_magcard instead	80
Mass	SNVT_mass	0 .. 6,553.5 grams (0.1 g)	23
	SNVT_mass_f	0 .. 1E38 g	56
	SNVT_mass_kilo	0 .. 6,553.5 kg (0.1 kg)	24
	SNVT_mass_mega	0 .. 6,553.5 metric tons (0.1 tonne)	25
	SNVT_mass_mil	0 .. 6,553.5 milligrams (0.1 mg)	26
Multiplier	SNVT_multiplier	0 .. 32.7675 (0.0005)	82
Object request	SNVT_obj_request	see Structures below	92
Object status	SNVT_obj_status	see Structures below	93
Occupancy	SNVT_occupancy	see Enum Lists below	109
Override	SNVT_override	see Enum Lists below	97
Phase / rotation	SNVT_angle	0 .. 65.535 radians (0.001 radians)	3
	SNVT_angle_deg[4]	*-359.98 .. +360.00 degrees (0.02 degrees)*	104
	SNVT_angle_f	-1E38 .. 1E38 radians	49
Phone state	SNVT_telcom	see Enum Lists below	38
Power	SNVT_power	0 .. 6,553.5 watts (0.1 W)	27
	SNVT_power_f	-1E38 .. 1E38 watts	57
	SNVT_power_kilo	0 .. 6,553.5 kW (0.1 kW)	28
Power factor	SNVT_pwr_fact	-1.0 .. 1.0 (0.00005)	98
	SNVT_pwr_fact_f	-1.0 .. 1.0	99
Preset	SNVT_preset	Use SNVT_scene(_cfg) instead	94

Table 1
Master SNVT List
continued

Measurement	Name	Range (Resolution)	SNVT #
Pressure - gauge	SNVT_press	-3,276.8 .. 3,276.7 kilopascals (0.1 kPa)	30
Pressure - absolute	SNVT_press_f	0 .. 1E38 pascals	59
Pressure - gauge	SNVT_press_p[4]	-32,768 .. 32,766 pascals (1 Pa)	113
Resistance	SNVT_res	0 .. 6,553.5 ohms (0.1 ohms)	31
	SNVT_res_f	-1E38 .. 1E38 ohms	60
	SNVT_res_kilo	0 .. 6,553.5 kohms (0.1 kohms)	32
Scene control	SNVT_setting	see Structures below	115
Scene setting	SNVT_scene	see Structures below	
Scene configuration	SNVT_scene_cfg	see Structures below	116
Sound level	SNVT_sound_db	-327.68 .. 327.67 decibels (0.01 dB)	33
	SNVT_sound_db_f	-1E38 .. 1E38 dBspl	61
Speed	SNVT_speed	0 .. 6,553.5 meters / sec (0.1 m / s)	34
	SNVT_speed_f	-1E38 .. 1E38 m / s	62
	SNVT_speed_mil	0 .. 65.535 m / s (0.001 m / s)	35
State	SNVT_state	see Structures below	83
Switch	SNVT_switch	see Structures below	95
Temperature	SNVT_temp [1]	-274 .. 6,279.5 ^{o}C (0.1 ^{o}C)	39
	SNVT_temp_p[2, 4]	-273.17 .. +327.66 ^{o}C (0.01 ^{o}C)	105
	SNVT_temp_f	-273.17 .. 1E38 ^{o}C	63
Temperature setpts	SNVT_temp_setpt	see Structures below	106
Thermostat mode	SNVT_therm_mode	see Enum Lists below	119
Time of day	SNVT_date_time	Use SNVT_timestamp instead	12
Time - elapsed	SNVT_time_f	0 .. 1E38 sec	64
	SNVT_elapsed_tm	See Structures below	87
	SNVT_time_sec[3]	0.0 .. 6553.4 sec (0.1 sec)	107
	SNVT_time_passed	Use SNVT_elapsed_tm instead	40
Time stamp	SNVT_time_stamp	see Structures below	84
Translation table	SNVT_trans_table	see Structures below	96
Volume	SNVT_vol	0 .. 6,553.5 liters (0.1 l)	41
	SNVT_vol_f	0 .. 1E38 l	65
	SNVT_vol_kilo	0 .. 6,553.5 kiloliters (0.1 kl)	42
	SNVT_vol_mil	0 .. 6,553.5 milliliters (0.1 ml)	43
Voltage	SNVT_volt	-3,276.8 .. 3,276.7 volts (0.1 V)	44
	SNVT_volt_dbmv	-327.68 .. 327.67 dB µv (0.01 db µv dc)	45
	SNVT_volt_f	-1E38 .. 1E38 volts	66
	SNVT_volt_kilo	-3,276.8 .. 3,276.7 kilovolts (0.1 kV)	46
	SNVT_volt_mil	-3,276.8 .. 3,276.7 millivolts (0.1 mV)	47
Zero and Span	SNVT_zerospan	see Structures below	85

[1] SNVT_temp represents tenths of a degree Celsius above -274 ^{o}C. To get SNVT_temp units define a constant: *C_to_K* equal to 2740 which is added to temperature expressed in tenths of degrees C.

[2] To be used for heating, ventilation and air conditioning applications.

[3] The value 0xFFFF represents invalid data.

4 The value 0x7FFF represents invalid data.

SNVT Structures

SNVTs identified as structures in the Master SNVT List are described in this section. A definition of each SNVT structure is provided below and information on specific fields is summarized in Table 2.

The structure definition used by *SNVT_alarm* is:

```
typedef struct {
   char               location[ LOCATION_LEN ];
   unsigned long      object_id;
   alarm_type_t       alarm_type;
   priority_level_t   priority_level;
   unsigned long      index_to_SNVT;
   unsigned short     value[ 4 ];
   unsigned long      year;
   unsigned short     month;
   unsigned short     day;
   unsigned short     hour;
   unsigned short     minute;
   unsigned short     second;
   unsigned long      millisecond;
   unsigned short     alarm_limit[ 4 ];
} SNVT_alarm;
```

The length of SNVT_alarm is 29 bytes. SNVT_alarm is used as the fundamental output network variable within the Node Object to report alarm status. The fields year through second in SNVT_alarm are compatible with SNVT_timestamp.

The structure definition used by *SNVT_color* is:

```
typedef struct {
   unsigned long  L_star;
     signed long  a_star;
     signed long  b_star;
} SNVT_color;
```

The structure definition used by *SNVT_currency* is:

```
typedef struct {
    currency_t      currency;
    signed short    power_of_10;
    unsigned short  value[ 4 ];
} SNVT_currency;
```

SNVT_currency represents a monetary value in a specified currency. The value field is a 32-bit signed value compatible with the Neuron C Extended Arithmetic type s32_type. Positive values correspond to credits, negative values to debits. The power_of_10 field scales the value field, so that for example, $1.23 is represented as:

```
{ CU_UNITED_STATES_DOLLAR, -2, { 0, 0, 0, 123 } }
```

The structure definition used by *SNVT_elapsed_tm* is:

```
typedef struct {
   unsigned long      day;
   unsigned short     hour;
   unsigned short     minute;
   unsigned short     second;
   unsigned long      millisecond;
} SNVT_elapsed_tm;
```

The structure definition used by *SNVT_file_pos* is:

```
typedef struct {
   unsigned short     rw_ptr[ 4 ];
   unsigned long      rw_length;
} SNVT_file_pos;
```

Network variables of type SNVT_file_pos are used to control the position of the read / write pointer in a file used for random access, as well as to specify the length of the next file transfer. The rw_ptr field is a 32-bit value compatible with the Neuron C Extended Arithmetic type s32_type. For more details, see the *File Transfer* LONWORKS Engineering Bulletin (005-0025-01).

The structure definition used by *SNVT_file_req* is:

```
typedef struct {
   file_request      request;
   unsigned long     index;
   unsigned long     receive_timeout;
   union {
      struct {
          unsigned   type;      // 1 for subnet/node
          unsigned   domain     : 1;
          unsigned   node       : 7;
          unsigned              : 4;
          unsigned   retry      : 4;
          unsigned              : 4;
          unsigned   tx_timer   : 4;
          unsigned   subnet;
      } sn;
      struct {
          unsigned   type       : 1;   // 1 for group
          unsigned   size       : 7;
          unsigned   domain     : 1;
          unsigned              : 7;
          unsigned              : 4;
          unsigned   retry      : 4;
          unsigned              : 4;
          unsigned   tx_timer   : 4;
          unsigned   group;
      } gp;
   } dest_address;
   signed short      auth_on;
   signed short      prio_on;
} SNVT_file_req;
```

The `sn` and `gp` structures are compatible with the `snode_struct` and `group_struct` structures defined in `ADDRDEFS.H`. For more details, see the *File Transfer* LONWORKS Engineering Bulletin (005-0025-01).

The structure definition used by *SNVT_file_status* is:

```
typedef struct {
   file_status           status;
   unsigned long         number_of_files;
   unsigned long         selected_file;
   union {
      struct {
          char           file_info[ 16 ];
          unsigned short size[ 4 ];
          unsigned long  type;
      } descriptor;
      struct {
          unsigned short domain_id[ 6 ];
          unsigned short domain_length;
          unsigned short subnet;
          unsigned short node;
      } address;
   } adr;
} SNVT_file_status;
```

For more details, see the *File Transfer* LONWORKS Engineering Bulletin (005-0025-01).

The structure definition used by *SNVT_hvac_overid* is:

```
typedef struct {
     hvac_overid_t     state;
     SNVT_lev_percent  percent;
     SNVT_flow         flow;
} SNVT_hvac_overid;
```

The structure definition used by *SNVT_hvac_status* is:

```
typedef struct {
     SNVT_hvac_mode    mode;
     SNVT_lev_percent  heat_output_primary;
     SNVT_lev_percent  heat_output_secondary;
     SNVT_lev_percent  cool_output;
     SNVT_lev_percent  econ_output;
     SNVT_lev_percent  fan_output;
     boolean           in_alarm;
} SNVT_hvac_status;
```

The structure definition used by *SNVT_magcard* is:

```
typedef struct {
     unsigned digit1   : 4;
     unsigned digit2   : 4;
     .........
     unsigned digit40  : 4;
} SNVT_magcard;
```

A card reader conforming to ISO 7811 will read standard financial transaction cards (credit cards and ATM cards). ISO 7811 is similar to the credit-card account numbering system given in ANSI Standard X4.13-1971. This data type is compatible with the `magcard` input object in Neuron C. See the *Neuron C Reference Guide* for more details. The start sentinel (0x0B) is always present in `digit1`. The null value for `SNVT_magcard` is defined as a start sentinel in `digit1`, and an end sentinel (0x0F) in `digit2`. Parity and LRC fields are not included in the structure.

The structure definition used by *SNVT_muldiv* is:

```
typedef struct {
     unsigned long multiplier;
     unsigned long divisor;
} SNVT_muldiv;
```

A configuration network variable of type `SNVT_muldiv` may be used as a gain factor for fixed point sensor objects. It is designed for use with the Neuron C `muldiv()` function, which provides a 16X16 unsigned multiplication with a 32-bit intermediate result, followed by a 32/16 unsigned division with a 16 bit end result. For more details, see the *Neuron C Reference Guide*.

The structure definition used by *SNVT_obj_request* is:

```
typedef struct {
    unsigned long       object_id;
    object_request_t    object_request;
} SNVT_obj_request;
```

`SNVT_obj_request` allows an object to be placed in one of several functional modes. For more details, see the *LONMARK Application Layer Interoperability Guidelines* (078-0120-01).

The structure definition used by *SNVT_obj_status* is:

```
typedef struct {
   unsigned long object_id;
   unsigned  invalid_id             : 1;
   unsigned  invalid_request        : 1;
   unsigned  disabled               : 1;
   unsigned  out_of_limits          : 1;
   unsigned  open_circuit           : 1;
   unsigned  out_of_service         : 1;
   unsigned  mechanical fault       : 1;
   unsigned  feedback_failure       : 1;
   unsigned  over_range             : 1;
   unsigned  under_range            : 1;
   unsigned  electrical_fault       : 1;
   unsigned  unable_to_measure      : 1;
   unsigned  comm_failure           : 1;
   unsigned  fail_self_test         : 1;
   unsigned  self_test_in_progress  : 1;
   unsigned  locked_out             : 1;
   unsigned  manual_control         : 1;
   unsigned  in_alarm               : 1;
   unsigned  in_override            : 1;
   unsigned  report_mask            : 1;
   unsigned  programming_mode       : 1;
   unsigned  programming_fail       : 1;
   unsinged  alarm_notify_disabled  : 1;
   unsigned  reserved1              : 1;
   unsigned  reserved2              : 8;
} SNVT_obj_status;
```

Network variables of type SNVT_obj_status are used to indicate the status of the various objects within a node. For more details, see the *LONMARK Application Layer Interoperability Guidelines* (078-0120-01).

The structure definition used by *SNVT_preset* is:

```
typedef struct {
   learn_mode_t       learn;
   unsigned long      selector;
   unsigned short     value[ 4 ];
   unsigned long      day;
   unsigned short     hour;
   unsigned short     minute;
   unsigned short     second;
   unsigned long      millisecond;
} SNVT_preset;
```

Network variables of type SNVT_preset are used to allow an Actuator Object to adopt one of several programmable values and ramp rates in addition to the normal control mode. The time-related fields in SNVT_preset are compatible with SNVT_elapsed_tm. For more details, see the *LONMARK Application Layer Interoperability Guidelines* (078-0120-01).

The structure definition used by *SNVT_scene* is:

```
typedef struct {
    scene_t         function;
    unsigned short  scene_number;
} SNVT_scene;
```

Network variables of type SNVT_scene are used to tell an actuator object to save its current setting as a scene, or recall a previously saved scene.

The structure definition used by *SNVT_scene_cfg* is:

```
typedef struct {
   scene_config_t    function;
   unsigned short    scene_number;
   unsigned short    setting;
    signed long      rotation;
   unsigned long     fade_time;
   unsigned long     delay_time;
    unsigned short   priority;
} SNVT_scene_cfg;
```

Network variables of type SNVT_scene_cfg are used to tell an actuator object to save a specified setting as a scene, report the scene data for a specified scene, and manage scene storage space.

The structure definition used by *SNVT_setting* is:

```
typedef struct {
      setting_t         function;
      unsigned short    setting;
      signed long       rotation;
} SNVT_setting;
```

For more details, see the LONMARK Functional Profiles: *3050-10 Constant Light Controller, 3060-10 Light Sensor, 3070-10 Occupancy Sensor*.

The structure definition used by *SNVT_state* is:

```
typedef struct {
   unsigned bit0  : 1;
   unsigned bit1  : 1;
    .....
   unsigned bit15 : 1;
} SNVT_state;
```

SNVT_state can be used to communicate the state of a set of 16 boolean values. Each bit indicates the state of the boolean, with the following interpretations:

0	1
off	on
inactive	active
disabled	enabled
low	high
false	true
normal	alarm

The structure definition used by *SNVT_str_asc* is:

```
typedef struct {
   unsigned char ascii[ 31 ];  /* 0..30 chars */
} SNVT_str_asc;           /* + NUL terminator */
```

The structure definition used by *SNVT_str_int* is:

```
typedef struct {
   unsigned short char_set;
   unsigned long wide_char[ 15 ]; /* 0..14 chars */  }
SNVT_str_int;              /* + NUL terminator */
```

The structure definition used by *SNVT_switch* is:

```
typedef struct {
   unsigned short     value;
   unsigned short     state;
} SNVT_switch;
```

Often a differentiation is needed between load value and state for an actuator. SNVT_switch caters for this by separating load value from state. The value field is used to control the load's value i.e. position, speed or setting, the state field being used to control whether the load is on or off (enabled or disabled).

SNVT_switch should be used for communicating state with discrete devices as well as level with continuous devices.

When used as the output of a discrete sensor device, the OFF state is represented by a SNVT_switch network variable with state = FALSE and value = 0. The other discrete states are represented by state = TRUE and value > 0. When used as the output of a two-state sensor device, the ON state is represented by state = TRUE and value = 200 (meaning 100% of full scale).

When used as the input of a two-state discrete actuator, a SNVT_switch network variable with state = TRUE will be interpreted as the ON state if value > 0, and as the OFF state if value = 0. Additionally, a SNVT_switch input network variable with state = FALSE should be interpreted as the OFF state, whether or not value = 0.

A state value of 0xFF indicates the switch value is undefined.

The structure definition used by *SNVT_temp_setpt* is:

```
typedef struct {
     signed long      occupied_cool;
     signed long      standby_cool;
     signed long      unoccupied_cool;
     signed long      occupied_heat;
     signed long      standby_heat;
     signed long      unoccupied_heat;
} SNVT_temp_setpt;
```

Each of the fields in this structure represents a temperature in units compatible with SNVT_temp_p. A value of 0x7FFF is used to represent unavailable data for that field.

The structure definition used by *SNVT_time_stamp* is:

```
typedef struct {
   unsigned long      year;
   unsigned short     month;
   unsigned short     day;
   unsigned short     hour;
   unsigned short     minute;
   unsigned short     second;
} SNVT_time_stamp;
```

The structure definition used by *SNVT_trans_table* is:

```
typedef struct {
   struct {
        unsigned   sign          : 1;
        unsigned   MS_exponent   : 7;
        unsigned   LS_exponent   : 1;
        unsigned   MS_mantissa   : 7;
        unsigned   long LS_mantissa;
   } point[ 7 ];
   unsigned   interp_pts_0_to_1 : 2;
   unsigned   interp_pts_1_to_2 : 2;
   unsigned   interp_pts_2_to_3 : 2;
   unsigned   interp_pts_3_to_4 : 2;
   unsigned   interp_pts_4_to_5 : 2;
   unsigned   interp_pts_5_to_6 : 2;
   unsigned   interp_pts_6_to_0 : 2;
} SNVT_trans_table;
```

A translation table is defined by two of sets network variables of type `SNVT_trans_table`; one for the X axis and one for the Y axis. The seven-element array `point` contains a single axis of translation values, represented as IEEE754 single precision floating point values compatible with the Neuron C Extended Arithmetic type `float_type`. The point values in the network variable for the X axis must be monotonically increasing. The fields `interp_pts_`*`m`*`_to_`*`n`* specify the type of interpolation to be used between the indicated pair of points. The values in these fields are of the enumeration type `interp_t`, and may be `IP_LINEAR` or `IP_CUBIC_SPLINE`.

If more than one pair of network variables of type SNVT_trans_table are present in an object, the field interp_pts_6_to_0 specifies the type of interpolation to be used between point 6 of this table and point 0 of the subsequent table (in order of X point value).

The structure definition used by *SNVT_zerospan* is:

```
typedef struct {
     signed long zero;
   unsigned long span;
} SNVT_zerospan;
```

A network variable of type SNVT_zerospan may be used to represent a linear transformation on fixed-point data. The data is multiplied by a span factor, and then the zero term is added.

Table 2
SNVT structures

Name	Units	Range	Notes
SNVT_alarm	Alarm status		
	8-bit bytes unsigned 16 bits alarm_type_t priority_level_t unsigned 16 bits 32 bit value year month day hour minute second millisecond 32 bit limit	6 bytes 0..65,535 see Enum Lists see Enum Lists 1.. #SNVTs Specific to SNVT 0 .. 3,000 (years AD) 0 .. 12 0 .. 31 0 .. 23 0 .. 59 0 .. 59 0 .. 999 Specific to SNVT	location of node ID of object within node SNVT ID of value type value, 4 bytes or fewer 0 means year not specified 0 means month not specified 0 means day not specified alarm limit, 4 bytes or fewer
SNVT_color	L*,a*,b*		CIELAB coordinate system
	L* a* b*	0.0 .. 100.0 -200.0 .. +200.0 -200.0 .. +200.0	resolution 0.1 units resolution 0.1 units resolution 0.1 units
SNVT_currency	International money		
	currency_t decimal factor signed 32 bits	see Enum Lists -128..+127 -2,147,483,648 .. +2,147,483,647	country currency code power of 10 credit positive, debit negative
SNVT_elapsed_tm	elapsed time		
	day hour minute second millisecond	0 .. 65,534 0 .. 23 0 .. 59 0 .. 59 0 .. 999	65,535 means null elapsed time
SNVT_file_pos	file random access		
	rw_ptr rw_length	0 .. 2^{32}-1 0 .. 65,535	read / write pointer transfer length in bytes
SNVT_file_req	file system request		
	file_request index receive_timeout dest_address auth_on prio_on	see Enum Lists 0 .. 65,535 0 .. 65,535 union 0 .. 1 0 .. 1	function code index of file in directory milliseconds address of receiver(s) use authentication protocol use priority messaging

Table 2
SNVT structures
continued

Name	Units	Range	Notes
SNVT_file_status	file system status		
	file_status number_of_files selected_file descriptor address	see Enum Lists 0 .. 65,535 0 .. 65,535 structure structure	status code size of directory index of selected file directory entry for lookup initiator address for open
SNVT_hvac_overid	HVAC output override		
	hvac_overid_t percent flow	see Enum Lists -163.83 .. +163.83% 0 .. 65,534 l/sec	override mode position or flow override value flow override value
SNVT_hvac_status	HVAC status		
	mode heat_output_primary heat_output_secondary cool_output econ_output fan_output in_alarm	see Enum Lists } -163.83 .. } +163.83% } (percentage } of full } scale) 0 .. 1	SNVT_hvac_mode primary heat output secondary heat output cooling output economizer output fan output 1 means unit is in alarm
SNVT_muldiv	Gain factor	0 .. 65,535	
	multiplier divisor	0 .. 65,535 1 .. 65,535	
SNVT_obj_request			
	unsigned 16 bits object_request_t	0..65,535 see Enum Lists	ID of object within node

Table 2
SNVT structures
continued

Name	Units	Range	Notes
SNVT_obj_status			
	unsigned 16 bits	0..65,535	ID of object within node
	invalid_id	0..1	1 means requested ID is not implemented in this node
	invalid_request	0..1	1 means request for unimplemented function
	disabled	0..1	1 means object disabled
	out_of_limits	0..1	1 means object exceeded alarm limits
	open_circuit	0..1	1 means open circuit detected
	out_of_service	0..1	1 means object not functional
	mechanical_fault	0..1	1 means mechanical fault detected
	feedback_failure	0..1	1 means feedback signal not received
	over_range	0..1	1 means max range exceeded
	under_range	0..1	1 means min range exceeded
	electrical_fault	0..1	1 means electrical fault detected
	unable_to_measure	0..1	1 means I/O line failure
	comm_failure	0..1	1 means network communications failure
	fail_self_test	0..1	1 means self-test failed
	self_test_in_progress	0..1	1 means self-test in progress
	locked_out	0..1	1 means node is on-line, but actuator movement is prevented
	manual_control	0..1	1 means actuator under local control
	in_alarm	0..1	1 means object is in alarm
	in_override	0..1	1 means object is overridden
	report_mask	0..1	1 means status is event mask
	programming_mode	0..1	1 means object is in program mode
	programming_fail	0..1	1 means object programming has failed
	alarm_notify_disabled	0..1	1 means object alarms disabled
SNVT_scene	scene control		
	function	see Enum Lists	RECALL, LEARN
	scene number	1 .. 255	
SNVT_scene_cfg	scene configuration		
	function	see Enum Lists	SAVE, CLEAR, REPORT, SIZE, FREE
	scene number	1 .. 255	
	8 bit setting	0 .. 100%	setting as percentage of full scale, resolution 0.5%
	16 bit rotation	-359.98…. +360.00	Rotational angle, resolution 0.02 degrees
	fade time	degrees 0.0 .. 6553.4 sec	fade time
	delay time	0.0 .. 6553.4 sec	delay time
	8 bit priority	1.0 00… 255	00 is highest priority

Table 2
SNVT structures
continued

Name	Units	Range	Notes
SNVT_setting	setting control		
	Function 8 bit setting 16 bit rotation	See Enum Lists 0 .. 100% -359.98..+360.00 degrees	OFF, ON, DOWN, UP, STOP, STATE setting as percentage of full scale, resolution 0.5% rotational angle, resolution 0.02 degrees
SNVT_str_asc	ASCII Characters		30 characters
	8 bit chars terminator	30 char 0x00	zero-length string with terminator represents null
SNVT_str_int	Int'l Char Set		14 'wide' characters
	char set code 16 bit chars terminator	0 .. 255 14 char 0x0000	tbd zero-length string with terminator represents null
SNVT_switch			
	8 bit value state	0 .. 100% 0 .. 1, 0xFF	setting as percentage of full scale, resolution 0.5% 0 means off, 1 means on, 0xFF means undefined
SNVT_temp_setpt			
	occupied_cool standby_cool unoccupied_cool occupied_heat standby_heat unoccupied_heat	-273.17 .. 327.66 °C	resolution 0.01 °C 0x7FFF means undefined
SNVT_time_stamp			
	year month day hour minute second	0 .. 3,000 (years AD) 0 .. 12 0 .. 31 0 .. 23 0 .. 59 0 .. 59	0 means year not specified. 65,535 represents null date 0 means month not specified 0 means day not specified
SNVT_trans_table			
	7 axis points type of interpol'n	-1.E38..+1.E38 see Enum Lists	IEEE754 floating point numbers linear or cubic spline
SNVT_zerospan	offset, multiplier		
	zero offset span multiplier	-163.84% .. +163.835% 0 .. 32.7675	resolution 0.005% or 50 ppm resolution 0.0005

Enumeration Lists

All SNVTs that are identified as enumeration lists have `enum` type definition files that can be optionally included. The following declaration must be included in the application code to use the SNVT enum type definition files.

```
#include "typedef file"
```

Table 3
Enumeration Lists

Details of the enumeration lists are described in Table 3.

Name	Enum Definition	Notes
SNVT_alarm	(alarm_type field)	typedef file: SNVT_AL.H typedef name: alarm_type_t
	0 AL_NO_CONDITION 1 AL_ALM_CONDITION 2 AL_TOT_SVC_ALM_1 3 AL_TOT_SVC_ALM_2 4 AL_TOT_SVC_ALM_3 5 AL_LOW_LMT_CLR_1 6 AL_LOW_LMT_CLR_2 7 AL_HIGH_LMT_CLR_1 8 AL_HIGH_LMT_CLR_2 9 AL_LOW_LMT_ALM_1 10 AL_LOW_LMT_ALM_2 11 AL_HIGH_LMT_ALM_1 12 AL_HIGH_LMT_ALM_2 AL_NUL	No alarm condition present Unspecified alarm condition present Total/service interval alarm 1 Total/service interval alarm 2 Total/service interval alarm 3 Alarm low limit alarm clear 1 Alarm low limit alarm clear 2 Alarm high limit alarm clear 1 Alarm high limit alarm clear 2 Alarm low limit alarm 1 Alarm low limit alarm 2 Alarm high limit alarm 1 Alarm high limit alarm 2 0xFF
SNVT_alarm	(priority_level field)	typedef file: SNVT_PR.H typedef name: priority_level_t
	0 PR_LEVEL_0 1 PR_LEVEL_1 2 PR_LEVEL_2 3 PR_LEVEL_3 PR_NUL	lowest alarm priority level highest alarm priority level 0xFF
SNVT_config_src		typedef file: SNVT_CFG.H typedef name: config_source_t
	0 CFG_LOCAL 1 CFG_EXTERNAL CFG_NUL	Node will use self installation functions to set its own network image Node's network image will be set by an outside source 0xFF

Table 3
Enumeration Lists
continued

Name	Enum Definition	Notes
SNVT_currency	(currency field) For a complete list of currencies, see the typedef file	typedef file: SNVT_CU.H typedef name: currency_t
	1 CU_AUSTRALIA_DOLLAR 6 CU_BRITAIN_POUND 7 CU_CANADA_DOLLAR 14 CU_EUROPEAN_CURRENCY_UNIT 16 CU_FRANCE_FRANC 17 CU_GERMANY_MARK 26 CU_JAPAN_YEN 54 CU_UNITED_STATES_DOLLAR CU_NUL	 0xFF
SNVT_defr_mode	Type of defrost	typedef file: SNVT_DFM.H typedef name: defrost_mode_t
	0 DF_MODE_AMBIENT 1 DF_MODE_FORCED 2 DF_MODE_SYNC DF_NUL	No forced heating required Start up after defrost ignored Synchronized 0xFF
SNVT_defr_state	Defrost progress indicator	typedef file: SNVT_DFS.H typedef name: defrost_state_t
	0 DF_STANDBY 1 DF_PUMPDOWN 2 DF_DEFROST 3 DF_DRAINDOWN 4 DF_INJECT_DLY DF_NUL	 0xFF
SNVT_defr_term	Defrost termination	typedef file: SNVT_DFT.H typedef name: defrost_term_t
	0 DF_TERM_TEMP 1 DF_TERM_TIME 2 DF_TERM_FIRST 3 DF_TERM_LAST DF_NUL	Terminate on temperature Terminate on time Terminate on first occuring Terminate on last occurring 0xFF
SNVT_date_day		typedef file: SNVT_DT.H typedef name: days_of_week_t
	0 DAY_SUN 1 DAY_MON 2 DAY_TUE 3 DAY_WED 4 DAY_THU 5 DAY_FRI 6 DAY_SAT DAY_NUL	 0xFF

Table 3
Enumeration Lists
continued

Name	Enum Definition	Notes
SNVT_evap_state		typedef file: SNVT_EVP.H typedef name: evap_t
	0 EVAP_NO_COOLING 1 EVAP_COOLING 2 EVAP_EMERG_COOLING EVAP_NUL	Object not performing cooling (off cycle or disabled) Object currently cooling Object performing emergency cooling 0xFF
SNVT_file_req	(request field)	typedef file: SNVT_FR.H typedef name: file_request_t
	0 FR_OPEN_TO_SEND 1 FR_OPEN_TO_RECEIVE 2 FR_CLOSE_FILE 3 FR_CLOSE_DELETE_FILE 4 FR_DIRECTORY_LOOKUP 5 FR_OPEN_TO_SEND_RA 6 FR_OPEN_TO_RECEIVE_RA FR_NUL	sequential access read sequential access write close and save file close and delete file retrieve directory entry random access read random access write 0xFF
SNVT_file_status	(status field)	typedef file: SNVT_FS.H typedef name: file_status_t
	0 FS_XFER_OK 1 FS_LOOKUP_OK 2 FS_OPEN_FAIL 3 FS_LOOKUP_ERR 4 FS_XFER_UNDERWAY 5 FS_IO_ERR 6 FS_TIMEOUT_ERR 7 FS_WINDOW_ERR 8 FS_AUTH_ERR 9 FS_ACCESS_UNAVAIL 10 FS_SEEK_INVALID 11 FS_SEEK_WAKE FS_NUL	file transfer successful directory lookup successful error on opening file error on directory lookup file transfer in progress error on reading / writing file file transfer timed out window sequence error authentication failure access mode not supported random access beyond EOF 0xFF
SNVT_hvac_emerg		typedef file: SNVT_EM.H typedef name: emerg_t
	0 EMERG_NORMAL 1 EMERG_PRESSURIZE 2 EMERG_DEPRESSURIZE 3 EMERG_PURGE 4 EMERG_SHUTDOWN EMERG_NUL	No emergency mode Emergency pressurize mode Emergency depressurize mode Emergency purge mode Emergency shutdown mode 0xFF

Table 3
Enumeration Lists
continued

Name	Enum Definition	Notes		
SNVT_hvac_mode, SNVT_hvac_status	(mode field)	typedef file: SNVT_HV.H typedef name: hvac_t		
	0 HVAC_AUTO	Controller automatically changes mode		
	1 HVAC_HEAT	Heating only		
	2 HVAC_MRNG_WRMUP	Application-specific morning warmup		
	3 HVAC_COOL	Cooling only		
	4 HVAC_NIGHT_PURGE	Application-specific night purge		
	5 HVAC_PRE_COOL	Application-specific pre-cool		
	6 HVAC_OFF	Controller not controlling outputs		
	7 HVAC_TEST	Equipment being tested		
	8 HVAC_EMERG_HEAT	Emergency heat mode (heat pump)		
	9 HVAC_FAN_ONLY	Air not conditioned, fan turned on		
	HVAC_NUL	0xFF		
SNVT_hvac_overid	(state field)	typedef file: SNVT_HVO.H typedef name: hvac_overid_t		
	0 HVO_OFF	not overridden		
	1 HVO_POSITION	position percentage - use percent field		
	2 HVO_FLOW_VALUE	flow in liters / sec - use flow field		
	3 HVO_FLOW_PERCENT	flow percentage - use percent field		
	4 HVO_OPEN	position = 100%		
	5 HVO_CLOSE	position = 0%		
	6 HVO_MINIMUM	configured minimum		
	7 HVO_MAXIMUM	configured maximum		
	HVO_NUL	0xFF		
SNVT_lev_disc		typedef file: SNVT_LEV.H typedef name: discrete_levels_t		
		2-state device	3-state device	4-state device
	0 ST_OFF	off	off	off
	1 ST_LOW	on	low	low
	2 ST_MED	on	high	med
	3 ST_HIGH	on	high	high
	4 ST_ON	on	high	high
	ST_NUL	0xFF		

Table 3
Enumeration Lists
continued

Name	Enum Definition	Notes
SNVT_obj_request	(object_request field)	typedef file: SNVT_RQ.H typedef name: object_request_t
	0 RQ_NORMAL 1 RQ_DISABLED 2 RQ_UPDATE_STATUS 3 RQ_SELF_TEST 4 RQ_UPDATE_ALARM 5 RQ_REPORT_MASK 6 RQ_OVERRIDE 7 RQ_ENABLE 8 RQ_RMV_OVERRIDE 9 RQ_CLEAR_STATUS 10 RQ_CLEAR_ALARM 11 RQ_ALARM_ NOTIFY_ ENABLED 12 RQ_ALARM_ NOTIFY_ DISABLED 13 RQ_MANUAL_CTRL 14 RQ_REMOTE_CTRL 15 RQ_PROGRAM RQ_NUL	enable object and remove override disable object just report object status perform object self-test update alarm status report status bit mask override object enable object remove object override clear object status clear object alarm enable alarm notification disable alarm notification enable object for manual control enable object for remote control enable programming of special configuration properties 0xFF
SNVT_occupancy		typedef file: SNVT_OC.H typedef name: occup_t
	0 OC_OCCUPIED 1 OC_UNOCCUPIED 2 OC_BYPASS 3 OC_STANDBY OC_NUL	Area is occupied Area is unoccupied Area is temporarily occupied for the bypass period Area is temporarily unoccupied 0xFF
SNVT_override		typedef file: SNVT_OV.H typedef name: override_t
	0 OV_RETAIN 1 OV_SPECIFIED 2 OV_DEFAULT OV_NUL	retain current level go to specified level go to default level 0xFF
SNVT_scene	(function field)	typedef file: SNVT_SC.H typedef name: scene_t
	0 SC_RECALL 1 SC_LEARN SC_NUL	recall value for specified scene learn current value for specified scene 0xFF

Table 3
Enumeration Lists
continued

Name	Enum Definition	Notes
SNVT_scene_cfg	(function field)	typedef file: SNVT_SCF.H typedef name: scene_config_t
	0 SCF_SAVE 1 SCF_CLEAR 2 SCF_REPORT 3 SCF_SIZE 4 SCF_FREE SCF_NUL	overwrite this scene with new data delete this scene from the list display this scene's data report the number of programmed scenes report the number of free scene storage spaces 0xFF
SNVT_setting	(function field)	typedef file: SNVT_SET.H typedef name: setting_t
	0 SET_OFF 1 SET_ON 2 SET_DOWN 3 SET_UP 4 SET_STOP 5 SET_STATE SET_NUL	setting off setting on decrease setting by specified value increase setting by specified value stop action setting on at specified value 0xFF
SNVT_telcom		typedef file: SNVT_TEL.H typedef name: telcom_states
	0 TEL_NOTINUSE 1 TEL_OFFHOOK 2 TEL_DIALING 3 TEL_DIALCOMP 4 TEL_RINGBACK 5 TEL_INCOMING 6 TEL_RINGING 7 TEL_ANSWERED 8 TEL_CONNECTED 9 TEL_TALKING 10 TEL_HANGINGUP 11 TEL_HUNGUPX 12 TEL_HOLD 13 TEL_UNHOLD 14 TEL_RELEASE 15 TEL_FULLDUP 16 TEL_BLOCKED 17 TEL_CWAIT 18 TEL_DESTBUSY 19 TEL_NETBUSY 20 TEL_ERROR TEL_NUL	"Null State (U0)" not in use "Call Initiated (U1)" "Overlap Sending (U2)" "Outgoing Call Proceeding (U3)" "Call Delivered (U4)" hearing ringback "Call Present (U6)" incoming call has not yet started ringing "Call Received (U7)" incoming call when the user has not yet answered "Connect Request (U8)" user has answered the call and is waiting to be awarded the call "Call Connected (U9)" "Active (U10)" parties are exchanging data "Disconnect Request (U11)" user has hung up "Disconnect Indication (U12)" the other side hung up "Suspend Request (U15)" user has requested the network suspend the call "Resume Request (U17)" resume a held call "Release Request (U19)" user has requested the network to release "Overlap Receiving (U25)" user has acknowledged the call and is prepared to receive additional information connection with blocking, (call waiting disabled) call waiting coming in destination busy problem, network problem, non-network 0xFF

Table 3
Enumeration Lists continued

Name	Enum Definition	Notes
SNVT_therm_mode		typedef file: SNVT_THM.H typedef name: therm_mode_t
	0 THERM_NO_CONTROL 1 THERM_IN_OUT 2 THERM_MODULATING THERM_NUL	Thermostat disabled Cut in / out control Modulating control 0xFF
SNVT_trans_table	(interp_pts_*m*_to_*n* fields)	typedef file: SNVT_IP.H typedef name: interp_t
	0 IP_LINEAR 1 IP_CUBIC_SPLINE IP_NUL	linear interpolation cubic spline interpolation 0xFF

Junction Box and Wiring Guidelines For Twisted Pair LONWORKS® Networks

August 1994 **LONWORKS Engineering Bulletin**

Introduction

This paper identifies the different types of junction boxes and interconnections that may be used in twisted pair LONWORKS networks in building and industrial control applications. The recommendations presented herein are intended to assist electrical system designers, interconnection device manufacturers, and cable manufacturers, and are provided for informational purposes only. For interoperability issues, refer to the *LONMARK™ Layers 1-6 Interoperability Guidelines*, order number 078-0014-01.

Three types of junction box topologies are discussed below:

- Pass-Thru
- Stub
- Local Loop

The junction box provides an interface between the twisted pair cable and the LONWORKS application node. The twisted pair cabling used between junction boxes will depend on the type of transceiver being installed. For systems using TP/XF-78, TP/XF-1250, TP-RS485, TPT/XF-78, or TPT/XF-1250 modules the bus wiring is 22AWG (0.65mm), while either 22 or 24AWG (0.5mm) cabling may be used on the stub between the junction box and the LONWORKS application node (figure 1). For free topology twisted pair systems, including the FTT-10 Free Topology Transceiver, LPT-10 Link Power Transceiver (figure 2) and the PLT-10/20/30 Power Line Transceivers (figure 3), the wiring is typically specified separately for each transceiver.

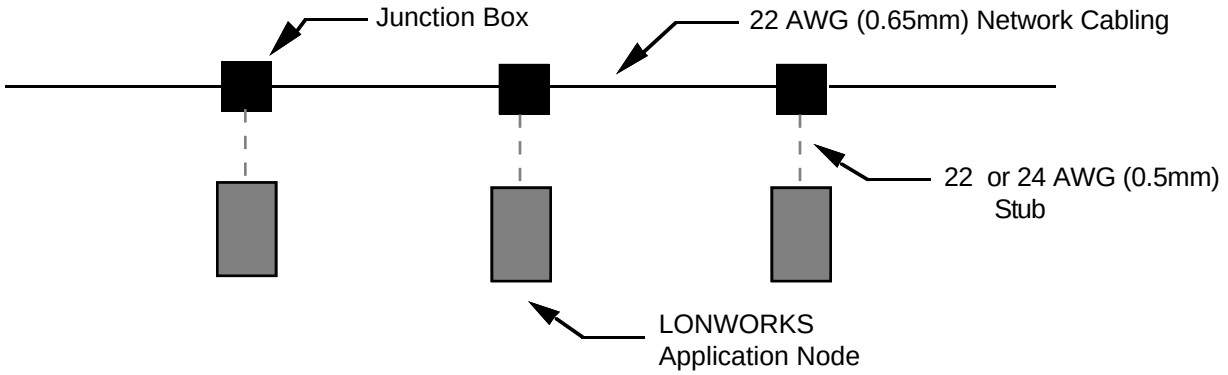


Figure 1 Typical Network Topology for 78kbps, 1.25Mbps, and RS-485 Networks

Echelon Part Number 005-0023-01 REV D

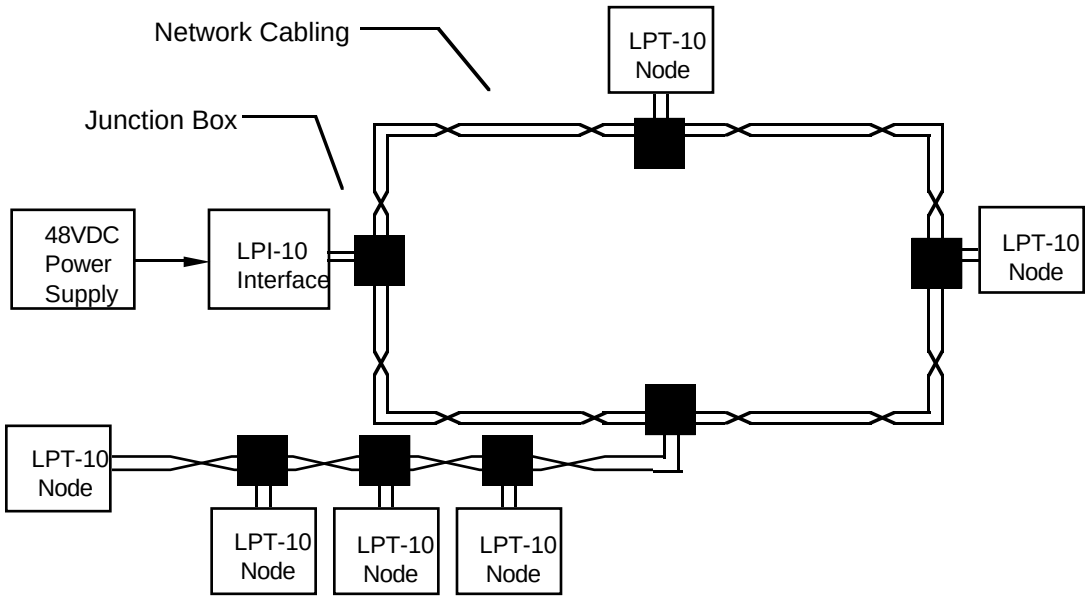


Figure 2 Typical Network Topology for Free Topology Networks
(LPT-10 Link Power Transceiver shown)

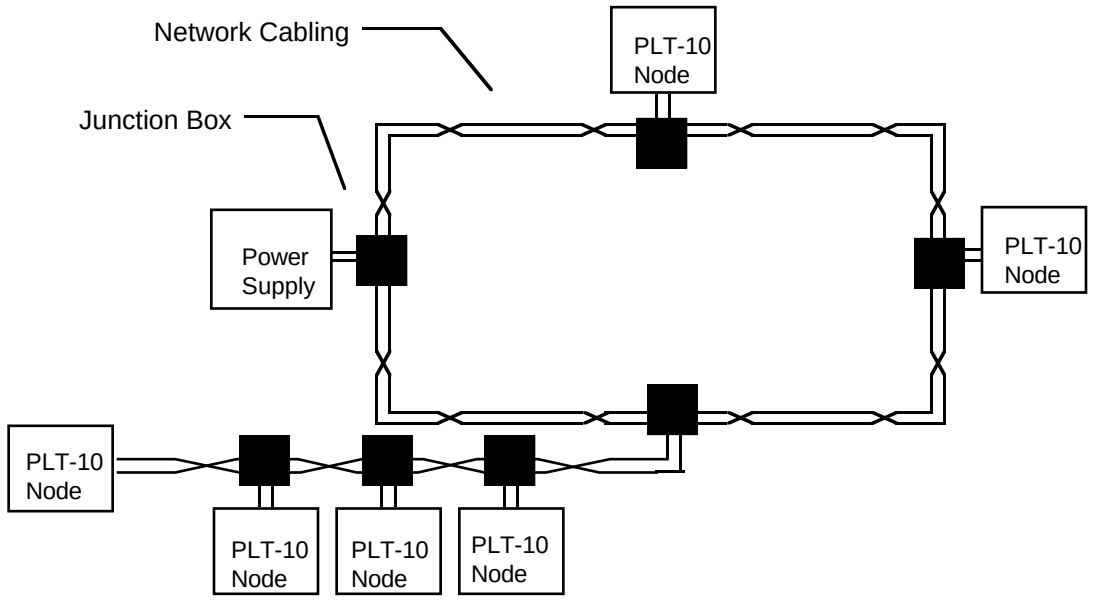


Figure 3 Typical Network Topology for PLT-based Twisted Pair Networks
(PLT-10 Power Line Transceiver shown)

The Pass-Thru Junction Box is intended to be a splice point only and does not include interconnections for a local node. The Stub Junction Box provides both a stub for a local node and a convenient splice point for cables passing to the previous and next junction boxes. The Local Loop Terminal Junction Box provides terminals for a local loop and a splice point for cables passing to the previous and next junction boxes. These two junction boxes are intended to be used when the distance between the junction box and the local node exceeds the allowed stub length, or when a loop-style architecture is desirable.

The connector that will be used to connect the twisted wire pair to the application node is typically defined by the industry application. That said, Echelon recommends the use of Weidmüller BLZ (or equal) connectors and receptacles because of their ease of use, ability to support solid and stranded wires from 12 to 22AWG, and high current capacity for link power applications. BLZ (or equal) terminals are also ideal for use as junction box connectors.

Definition of Terms

Color Code	The color of the LONWORKS network cable conductors. In systems using a single twisted wire pair (two wire), the colors will be white/blue (W/B) and blue (B). In systems using a dual twisted wire pair (four wire), the colors will be white/blue (W/B), blue (B), white/orange (W/O)), and orange (O). A shield drain wire (S) may be provided in either single or dual twisted wire pair networks, and will be clearly designated by color coding or other markings as being a shield or drain wire.
IN	Wiring that originates at the previous node.
Insulation Displacement Terminal	A connector arrangement in which wiring is affixed to insulation displacement terminals. Multiple wires can be attached to a single connector bus either by permitting two conductors to be inserted in the same displacement terminal or by providing two or more displacement terminals on a common bus.
Local Loop	A loop of bus cable that exits from a junction box, interconnects with a node, and returns to the junction box.

Local Power	A power source that provides operating power to a locally powered node.
OUT	Wiring that originates at the current node.
Pass-Thru	A junction box at which wires are spliced, and to which no nodes are connected.
Screw Terminal	A connector arrangement in which wiring is affixed to screw terminals. Washers are provided to separate multiple conductors attached to a single screw.
Stub	A T-tap from a bus cable that originates at a junction box and terminates at a node.
2-Wire Link Power	2-wire link power permits the transmission of both node power and LonTalk® network data on a common pair of wires. Link power nodes may or may not require local power depending on the system topology and local power consumption levels.

Pass-Thru Junction Box

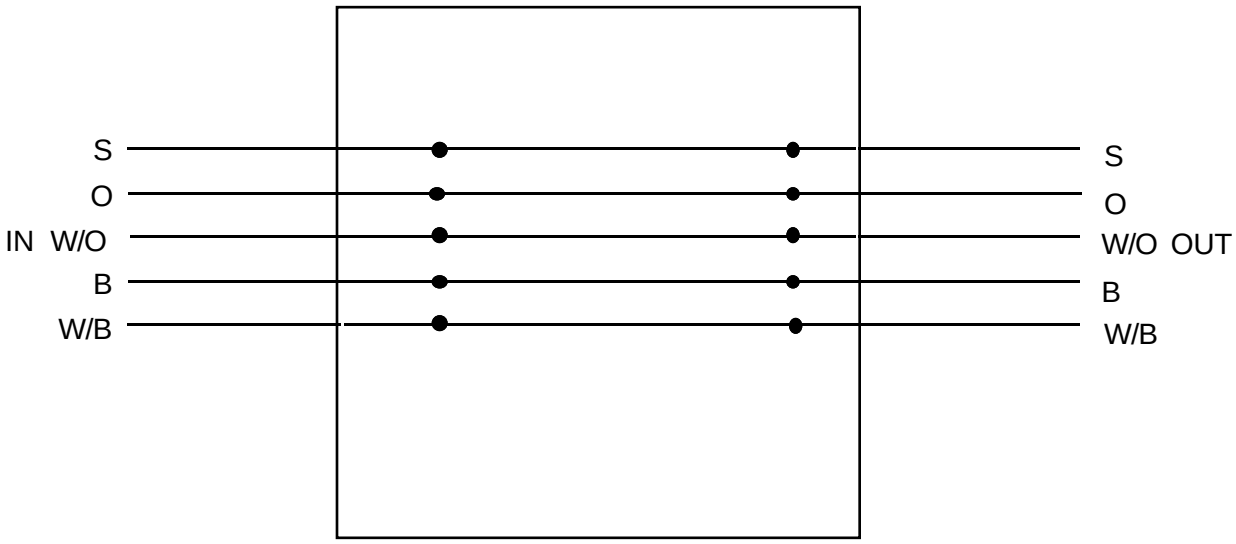
A pass-thru junction box provides a convenient point at which to splice two cables. No nodes or connectors are provided at a pass-thru junction box. There are three primary methods of implementing a splice at a pass-thru junction box:

1. Screw Terminals: IN and OUT wires are stripped and wrapped around screw terminals, which are tightened to retain the wires and make a secure electrical contact. Each screw is supplied with two or more washers. The washers separate conductors where multiple wires are landed at one screw, thereby preventing wires from being ejected as the screw is tightened. IN terminals 1-5 are connected directly by buses, circuit card traces, or wire jumpers, to OUT terminals 1-5, respectively. These connections provide the "pass-thru" function by routing the incoming signals to the outgoing terminals.

2. Insulation Displacement Terminals: IN and OUT wires are punched down on barrel or telco insulation displacement terminals. IN terminals 1-5 are connected directly by buses, circuit card traces, or wire jumpers, to OUT terminals 1-5, respectively. These connections provide the "pass-thru" function by routing the incoming signals to the outgoing terminals.

3. Crimp Connectors: IN and OUT wires are spliced together using crimp connectors. The connectors are then fitted inside the junction box, which itself contains no terminals.

Pass-Thru Terminal Junction Box



Terminal Legend

Terminal	Wire Color	Function
1	White/Blue	Data comm. or + for 2-wire link power
2	Blue	Data comm. or - for 2-wire link power
3	White/Orang e	Power + if locally powered
4	Orange	Power GND if locally powered
5	S - Shield	Cable shield if used

Stub Junction Box

A stub junction box provides a convenient point at which to splice two cables and provide a stub for servicing a local node. There are two primary methods of implementing a stub connectorized junction box:

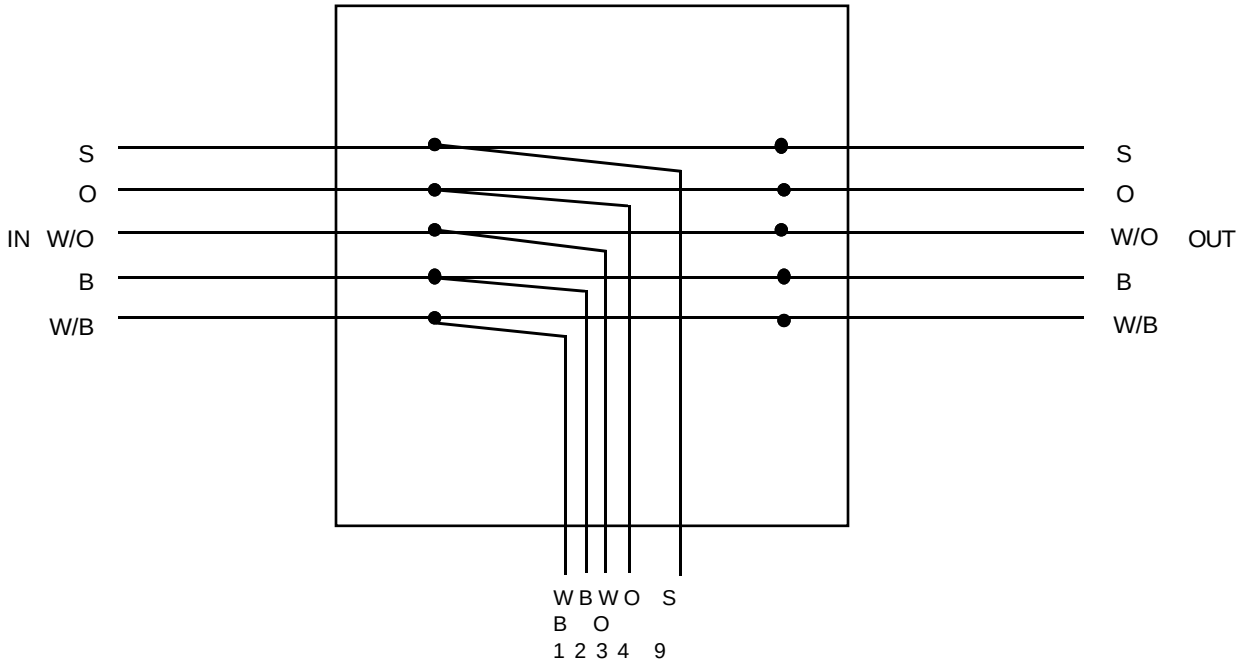
1. Screw Terminal 10 Wire Pin Connector: IN and OUT wires are stripped and wrapped around screw terminals, which are tightened to retain the wires and make a secure electrical contact. Each screw is supplied with two or more washers. The washers separate conductors where multiple wires are landed at one screw, thereby preventing wires from being ejected as the screw is tightened.

2. Insulation Displacement 10 Wire Pin Connector: IN and OUT wires are punched down on barrel or telco insulation displacement terminals. Both stranded and solid conductor wires are supported.

The following section applies to both versions of a stub connectorized junction box.

IN terminals 1-5 are connected directly by buses, circuit card traces, or wire jumpers, to OUT terminals 1-5, respectively. These connections provide the "pass-thru" function by routing the incoming signals to the outgoing terminals.

Stub Junction Box



Terminal Legend

Terminal	Wire Color	Function
1	White/Blue	Data comm. or + for 2-wire link power
2	Blue	Data comm. or - for 2-wire link power
3	White/Orange	Power + if locally powered
4	Orange	Power GND if locally powered
5	S - Shield	Cable shield if used

Tap Legend

Terminal	Wire Color	Function
1	White/Blue	IN Data comm. or + for 2-wire link power
2	Blue	IN Data comm. or - for 2-wire link power
3	White/Orange	IN Power + if locally powered
4	Orange	IN Power GND if locally powered
5 -8	Not used	
9	S - Shield	IN Cable shield
10	Not used	

Local Loop Terminal Junction Box

A local loop terminal junction box provides a convenient point at which to terminate two cables and provide a wiring loop for servicing a local node. There are two primary methods of implementing a local loop terminal junction box:

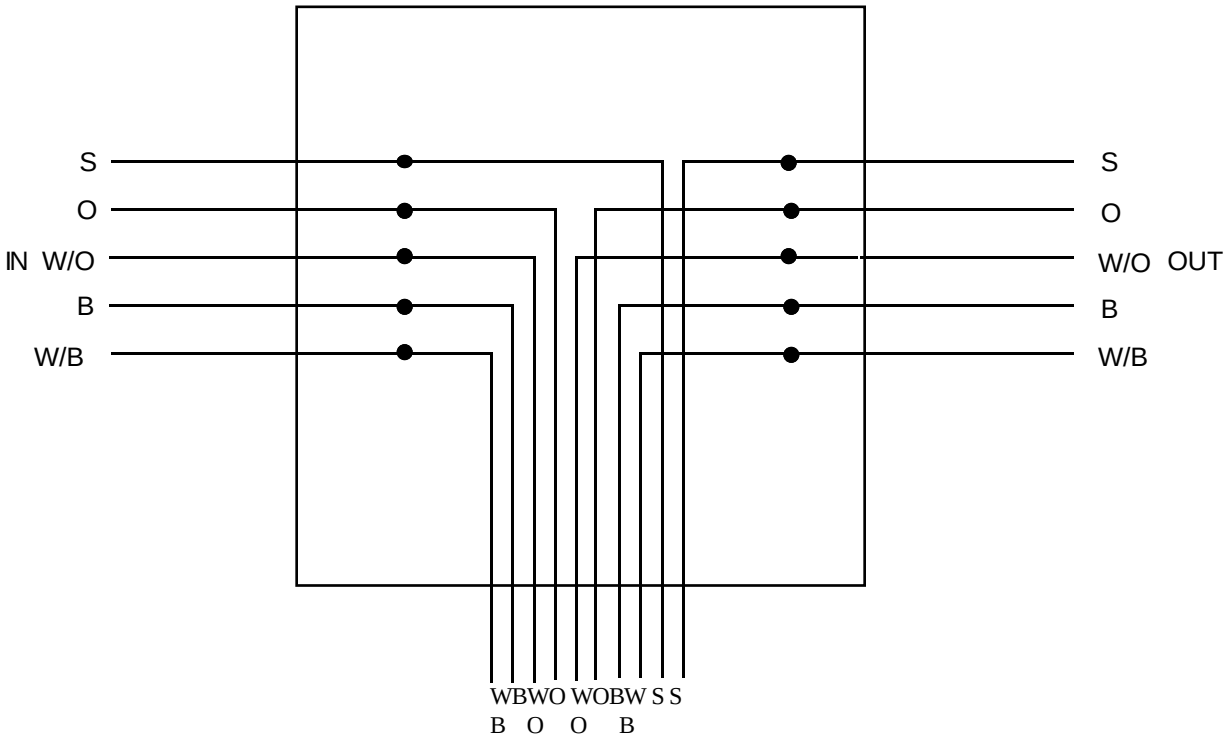
1. Screw Terminal 10 Wire: IN and OUT wires are stripped and wrapped around screw terminals, which are tightened to retain the wires and make a secure electrical contact. Each screw is supplied with two or more washers. The washers separate conductors where multiple wires are landed at one screw, thereby preventing wires from being ejected as the screw is tightened. The local node is wired directly to the In and OUT terminals.

2. Insulation Displacement 10 Wire: IN and OUT wires are punched down on barrel or telco insulation displacement terminals. The local node is wired directly to the In and OUT terminals.

The following section applies to both versions of a local loop connectorized junction box.

IN terminals and OUT terminals are isolated from each other: this junction box does not perform a "pass-thru" function.

Local Loop Terminal Junction Box



Terminal Legend

Terminal	Wire Color	Function
1	White/Blue	Data comm. or + for 2-wire link power
2	Blue	Data comm. or - for 2-wire link power
3	White/Orange	Power + if locally powered
4	Orange	Power GND if locally powered
5	S - Shield	Cable shield if used

Local Loop Legend

Terminal	Wire Color	Function
1	White/Blue	IN Data comm. or + for 2-wire link power
2	Blue	IN Data comm. or - for 2-wire link power
3	White/Orange	IN Power + if locally powered
4	Orange	IN Power GND if locally powered
5	White/Orange	OUT Power + if locally powered
6	Orange	OUT Power GND if locally powered
7	Blue	OUT Data comm. or - for 2-wire link power
8	White/Blue	OUT Data comm. or + for 2-wire link power
9	S - Shield	IN Cable shield
10	S - Shield	OUT Cable shield

Junction Box Suppliers

Hoffman
900 Ehlen Drive
Anoka, Minnesota 55303-7504
Phone: +1-612-421-2240
FAX: +1-612-421-1556

Commercial and industrial junction boxes. The customer must specify a terminal kit assembly (8 to 24 screw terminals per block). Customer must also specify the size of the enclosure and base plate, color, and other options when ordering.

Pass-Thru	Stub	Local Loop	Model No.	Description
√	√	√	D-xxxIS	DesignLine instrumentation enclosure with terminal kit assembly
√	√	√	A-xxxCHQR series	Continuous hinge quick-release CHRQ box with terminal kit assembly
√	√	√	A-xxxCHNF series	Continuous hinge Type 4 CHNF box with terminal kit assembly
√	√	√	A-xxxSC series	Screw cover SC box with terminal kit assembly
√	√	√	A-xxxLP series	Lift-off cover LP box with terminal kit assembly
√	√	√	A-xxxxCHNFSS series	Stainless steel continuous hinge Type 4X CHNFSS junction box with terminal kit assembly
√	√	√	A-xxxCHSCFG series	Fiberglass hinged cover Type 4X enclosure with terminal kit assembly
√	√	√	A-xxxJFG series	Type 4X fiberlass small enclosure with terminal kit assembly
√	√	√	A-xxHxxxxGQR LP series	Type 4X fiberlass medium enclosure with terminal kit assembly
√	√	√	Ultrix™ series	Type 4X fiberlass enclosure with terminal kit assembly
√	√	√	A-xRxxHCLO series	Hinged cover lift-off type 3R enclosure with terminal kit assembly

Leviton Telecom
2222 222nd Street S.E.
Bothell, Washington 98021-4422
Phone: +1-206-486-2222
FAX: +1-206-485-9170

Commercial and residential junction boxes.

Pass-Thru	Stub	Local Loop	Model No.	Description
√	√	√	40236-I	6 screw terminal plastic connecting block with plastic cover
√	√	√	40219-I	4 screw terminal plastic connecting block with plastic cover - not for 4 wire plus shield applications
	√		40278-G	8 screw terminal plastic connecting block with RJ45 connector
	√		41018-GYQ	8 insulation displacement terminal plastic connecting block with RJ45 connector

Suttle Apparatus
P.O. Box 548
Hector, Minnesota 55342
Phone: +1-612-848-6711
FAX: +1-612-848-6218

Commercial and residential junction boxes.

Pass-Thru	Stub	Local Loop	Model No.	Description
√	√	√	SE-42A6	6 screw terminal plastic connecting block with plastic cover
√	√	√	SE-44A and SE-101A cover	10 screw terminal plastic connecting block with plastic cover
√	√	√	SEA-66C1-6	6 pair insulation displacement terminal connecting block
	√		SE-625A28NK	8 screw terminal plastic connecting block with RJ45 connector
	√		SE-625A38NK	8 insulation displacement terminal plastic connecting block with RJ45 connector

Ultrak
2400 Industrial Lane
Suite 2000A
Broomfield, Colorado 80020
Phone: +1-303-466-7333
FAX: +1-303-469-8336

Commercial and residential junction boxes. Some boxes are supplied with tamper switches for security applications.

Pass-Thru	Stub	Local Loop	Model No.	Description
√	√	√	701	7 screw terminal plastic connecting block with plastic cover and tamper switch
√	√	√	711	6 screw terminal round plastic connecting block with plastic cover
√	√	√	720	10 screw terminal plastic connecting block with plastic cover and tamper switch
√	√	√	730	24 screw terminal plastic connecting block with plastic cover and tamper switch
√	√	√	735	10 screw terminal round plastic connecting block with plastic cover and tamper switch

Level 4 Cable Specifications

The Level 4 cable specification used by Echelon and as originally defined by the National Electrical Manufacturers Association (NEMA) differs from the Category IV specification proposed by the Electronic Industries Association / Telecommunication Industry Association (EIA / TIA). The Level 4 cable specifications used by Echelon are presented below, and are followed by a list of Level 4 cable suppliers.

Specifications apply to shielded or unshielded 22AWG (0.65mm) cable, 24AWG (0.5mm) cable shown in brackets [ ] if different	
D-C Resistance (Ohms / 1000 feet at 20°C) maximum for a single copper conductor regardless of whether it is solid or stranded and is or is not metal-coated.	18.0 [28.6]
D-C Resistance Unbalance (percent) maximum	5
Mutual Capacitance of a Pair (pF / foot) maximum	17
Pair-to-Ground Capacitance Unbalance (pF / 1000 feet) maximum	1000
Impedance (Ohms)	
772kHz	102±15% (87-117)
1.0MHz	100±15% (85-115)
4.0MHz	100±15% (85-115)
8.0MHz	100±15% (85-115)
10.0MHz	100±15% (85-115)
16.0MHz	100±15% (85-115)
20.0MHz	100±15% (85-115)
Attenuation (dB/1000 feet at 20°C) maximum	
772kHz	4.5 [5.7]
1.0MHz	5.5 [6.5]
4.0MHz	11.0 [13.0]
8.0MHz	15.0 [19.0]
10.0MHz	17.0 [22.0]
16.0MHz	22.0 [27.0]
20.0MHz	24.0 [31.0]
Worst-Pair Near-End Crosstalk (dB) minimum. Values are shown for information only. The minimum NEXT coupling loss for any pair combination at room temperature is to be greater than the value determined using the formula NEXT (F_{MHz})>NEXT(0.772)-15$\log_{10}(F_{MHz} / 0.772)$ for all frequencies in the range of 0.772MHz-20MHz for a length of 1000 feet.	
772kHz	58
1.0MHz	56
4.0MHz	47
8.0MHz	42
10.0MHz	41
16.0MHz	38
20.0MHz	36

Level 4 Cable Suppliers

Mr. Paul Sullivan
Anixter
4711 Golf Road
Skokie, Illinois 60076

Phone: +1-708-677-2600
FAX: +1-708-677-2668

Mr. Steve Strange
Anixter (UK)
Unit 17, Poyle 14
Newland Drive
Colnbrook, Slough SL3 0DX
Phone: +44 (0)753-686884
FAX: +44 (0)753-817122

Anixter has indicated that they stock the following cables which they will cut to order.

Part No.	Description
9D220150	22 AWG (0.65mm)/1 pair solid, unshielded, PVC
9F220154	22 AWG (0.65mm)/1 pair solid, shielded, PVC
9D220250	22 AWG (0.65mm)/2 pair solid, unshielded, PVC
9F220254	22 AWG (0.65mm)/2 pair solid, shielded, PVC
9H2201504	22 AWG (0.65mm)/1 pair solid, unshielded, plenum
9J2201544	22 AWG (0.65mm)/1 pair solid, shielded, plenum
9H2202504	22 AWG (0.65mm)/2 pair solid, unshielded, penum
9J2202544	22 AWG (0.65mm)/2 pair solid, shielded, plenum

Mr. Eric L. Rand
Connect-Air International, Inc.
50-37th Street N.E.
Auburn, Washington 98002
Phone: +1-206-813-5599
FAX: +1-206-813-5699

The following table lists cables stocked by Connect-Air. For information on direct burial and aerial messenger cables, contact Eric Rand directly.

Part No.	Description
W221P-1002	22AWG (0.65mm)/1 pair stranded, unshielded, PVC
W222P-1004	22AWG (0.65mm)/2 pair stranded, unshielded, PVC
W221P-1003	22AWG (0.65 mm)/1 pair stranded, shielded, PVC
W222P-1005	22AWG (0.65mm)/2 pair stranded, shielded, PVC
W221P-2001	22AWG (0.65mm)/1 pair stranded, unshielded, plenum
W221P-2003	22AWG (0.65mm)/2 pair stranded, unshielded, plenum

W221P-2002	22AWG (0.65 mm)/1 pair stranded, shielded, plenum
W222P-2004	22AWG (0.65mm)/2 pair stranded, shielded, plenum

Link Power/Free Topology Cable Suppliers (Level 4 22AWG (0.65mm) cables may also be used)

Belden
P.O. Box 1980
Richmond, Indiana 47375
Phone: +1-317-983-5200

Part No.	Description
8471	16AWG (1.3mm)/1 pair stranded, unshielded, PVC
85102	16AWG (1.3mm)/1 pair stranded, unshielded, plenum

Anixter Deutschland GmbH
Gottlieb-Daimler-Strasse 55
D-7141 Murrbei Stuttgart
Germany
Phone: +(07144) 2694-0
FAX: +(07144) 23609

Part No.	Description
4QJB2	J-Y(st)Y 20.4AWG (0.8mm)/2 pair solid, shielded, PVC

File Transfer

November 1996 **LONWORKS Engineering Bulletin**

Introduction

The purpose of this bulletin is to establish a standard method of exchanging large amounts of data between nodes in a LONWORKS network. The largest practical amount of data that can be transferred in a single LonTalk® packet is 228 bytes, but this file transfer protocol breaks up data files into packets containing 32 bytes of data and transfers the packets sequentially. The size of the packet is fixed at 32 bytes for interoperability and low node cost, but may be increased if there is no interoperability requirement. Larger packets may require off-chip RAM to store application and network buffers on a Neuron® 3150® Chip and thus increase the cost of the node. This file transfer method can be used with nodes based on stand-alone Neuron Chips, or nodes based on host processors attached to LONWORKS network interfaces, such as the Serial LonTalk Adapter (SLTA), the PC LonTalk Adapter (PCLTA), or interfaces based on the Microprocessor Interface Program (MIP).

Data exchange between nodes in the network is modeled as a file transfer, where a file is a stream of bytes, accessed sequentially either for read or for write. The implementation of a file system is not part of this specification. The file system is dependent on the environment of each node, for example whether it is a stand-alone Neuron Chip, or a host-based node. If it is a host-based node, no assumptions are made about the host file system, for example the format of a file name, the existence of any particular directory structure, or file typing. In this way, any node may participate in a file transfer.

Three logical nodes participate in a file transfer operation: the Initiator, the Sender, and one or more Receivers. The Initiator node may also be the Sender or one of the Receivers. In this way, a node may initiate the transfer of a file from another node to itself, or a node may initiate the transfer of a file from itself to one or more other nodes.

The set-up of a file transfer is implemented with network variables of standard types (SNVTs) that allow the Initiator to communicate with the Sender and the Receivers. The actual transfer itself is implemented with explicit messages, using a windowed protocol.

Echelon Part Number 005-0025-01 REV D

Windowed Transfer Protocol

Most data packets are sent with unacknowledged service, with a request/response packet sent every six packets. These five unacknowledged packets and the sixth request/response packet constitute a data window. This windowing protocol avoids the overhead of acknowledging every packet, but allows recovery from a lost packet no more than six packets later. Each packet contains 32 bytes of data, so that standard buffer sizes may be used on each node, without the need for additional RAM space. The response from the Receivers to the Sender includes an indication of the last packet in the window that was successfully received, incremented by one. The Sender needs to buffer the last data window, in case it receives a request from the Receivers to retransmit one or more packets from that window. For example, if no errors occur, figure 1 shows two successive data windows. Packets marked U are unacknowledged. RQ is a request packet from the Sender, and RSP is a response from the Receiver containing the number of the last packet successfully received incremented by one, which is 5 + 1 = 6.

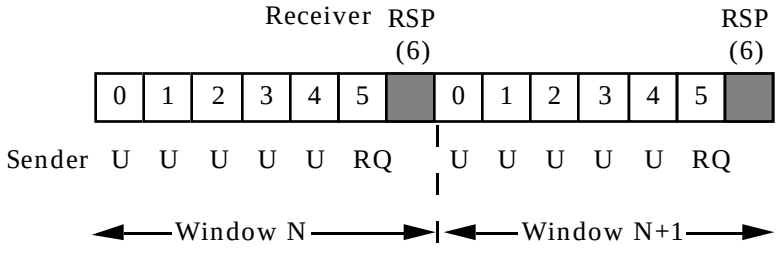


Figure 1 Window successfully received

Figure 2 shows an example of error recovery. If packet 3 in the window is not received correctly, the response packet RSP contains a 3, and the Sender retransmits the window starting with packet 3. Note that packets 3, 4, and 5 are retransmitted to avoid the need for the receiver to buffer packets 4 and 5. In the case of a multicast transfer, the error recovery begins with the lowest numbered packet not received.

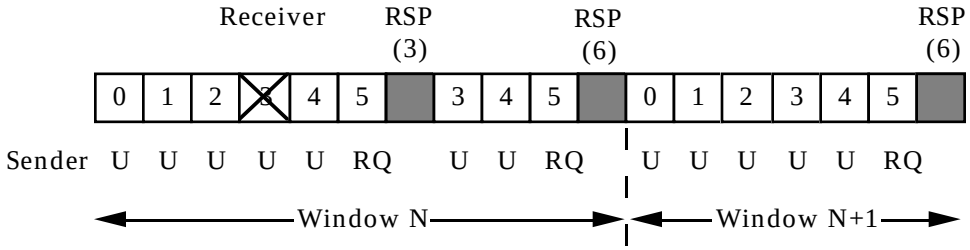


Figure 2 Window with retransmission

Setting Up a File Transfer

There are two Standard Network Variable Types used to set up a file transfer. `SNVT_file_req` is used for communication from the Initiator to the Sender and Receivers. `SNVT_file_status` is used for communication from the Sender and Receivers to the Initiator. These data structures are pre-defined in the Neuron C Compiler. See Appendix A, and also The *SNVT Master List and Programmer's Guide,* part of the *LONWORKS Interoperability Packet.* In the case where the Initiator is the same as one of the other nodes, the network variables are turn-around, meaning that outputs are connected to inputs on the same node. Note that the Initiator can choose to monitor and control the file transfer NVs either by being bound to them or by using explicit updates and polling.

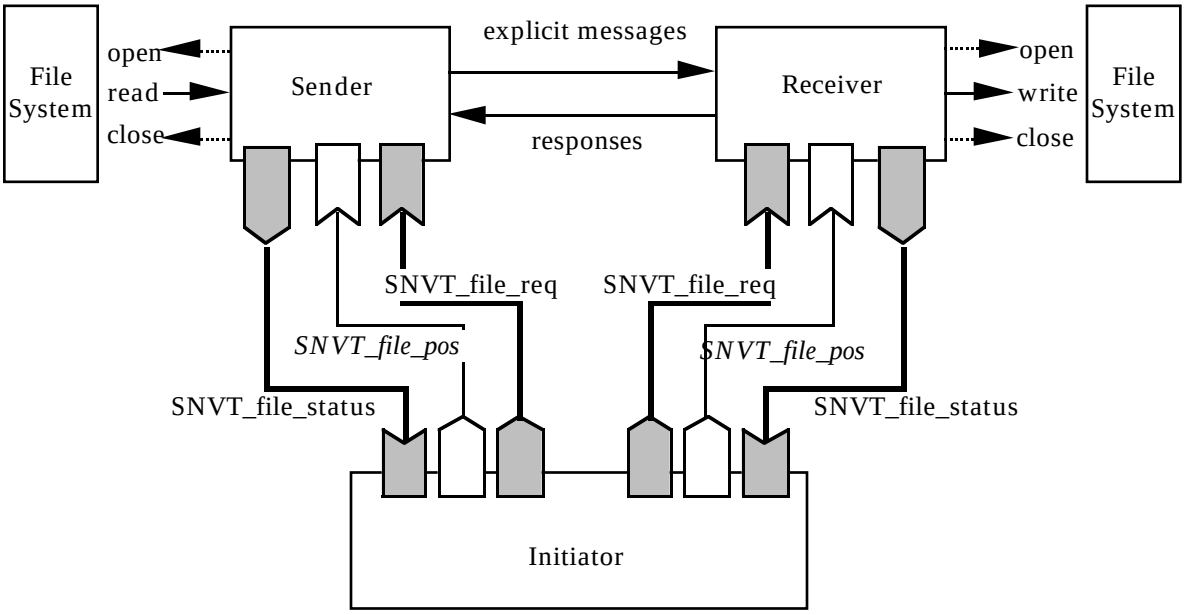


Figure 3 File Transfer Architecture

To start the transfer, the Initiator updates the network variable of type SNVT_file_req on the Receivers in order to open the destination files for writing. The result of the file open operation is returned in an update to the network variable of type SNVT_file_status on the Receivers. If all the Receiver files were successfully opened, the Initiator then uses the same procedure to open the source file on the Sender for reading. At the same time, the Initiator passes the network address of the Receivers to the Sender. The Sender uses this address to pass the file data in explicit messages to the Receivers. At the conclusion of the transfer, the Sender informs the Initiator that it has transmitted the whole file, and the Initiator then informs the Sender to close the file and Receivers to close and save the files.

During data transfer and setup, several potential errors can occur. The Sender or Receivers can fail to open the specified files. In this case, the initiator backs the other nodes out of the operation appropriately. The Sender can detect a read error (other than end-of-file), and the Receivers can detect a write error. If a Receiver fails to receive a data packet within a configurable time-out, it closes and restores its output file, aborts the transfer and informs the Initiator. The Initiator passes this timeout value to the Receivers as part of the setup request. Similarly, if the Sender fails to receive a response to one of its requests to the Receivers, it closes its input file, aborts the transfer and informs the Initiator. Any error prior to the complete file transfer causes any partially received files to be restored to their original state.

Random Access

The random access protocol allows a file to be opened and then to be transferred in a piecemeal fashion. The Initiator initiates transfers of a subset of the file by performing a seek operation first on the Receiver and then on the Sender. The seek includes an offset and a length. Multiple seeks may be executed within the scope of a single open/close by the initiator. The length must be the same in both the Sender and Receivers while the offsets may differ.

Each seek causes an exchange which is virtually identical to that of a full file transfer. The term data exchange is used to refer to the process which occurs following each seek as well as that during a full file transfer. A full file data exchange differs from that of a seek in that a full file data exchange always starts with window 0 whereas each seek data exchange starts with a window number one greater than the previous (starting with 0).

Delayed Responses

An Initiator normally expects that following the update of the file transfer request NV, the status NV will immediately be updated to indicate a success or failure condition. However, it is possible that a node accessing files on a disk or network file system may require some extra time to prepare the status NV response. To this end, the Initiator must tolerate a "working on it" response to either an open, close, or look-up operation. Read, write and seek operations must be responded to immediately. This may require buffering of data in the node to satisfy this requirement.

The "working on it" response to an open or look-up operation is `FS_XFER_OK`. Note that for a look-up operation, the status NV union must contain the requester address rather than directory information data. The "working on it" response to a close request is either `FS_XFER_UNDERWAY` or `FS_SEEK_WAIT` depending on whether a random access transfer is under way.

In a polling situation, upon getting a "working on it" response, the Initiator should wait for a period of time, such as one second, before trying again to get the status.

Completing a Data Exchange

Senders define normal completion of a data exchange as the transmittal of all data within the file or, for random access, transmittal of the number of bytes specified in the seek operation. Receivers define normal completion of a data exchange as the receipt of a file transfer packet with less than the maximum amount of data. It is implementation dependent whether a Receiver treats receipt of more or less data than the file length (or seek length for random access) as an error. However, it is recommended that an `FS_IO_ERR` be returned if the data exchange is too long. Note that it is incumbent upon the Sender to always transmit the last packet for a data exchange with a length less than the maximum, even if it means transmitting a packet with zero data.

Completing a File Transfer

Whether there are errors or not, it is always the Initiator that closes the files on the Sender and the Receivers. This guards against race conditions which could otherwise occur in multiple initiator scenarios. To guard against the Initiator never closing the file, the Sender and the Receivers must have the ability to do a local close in the event that the Initiator does not close the file in a timely manner, for example, within one minute after an error condition or normal completion. Furthermore, the Initiator must close the file on the Sender first then the

Receivers. This protects against race conditions where the Sender is still sending while the Receiver is closed and reopened by another Initiator.

Multicast File Transfers

A multicast file transfer requires that the Sender be given a group number as the destination. The group size specified by the Initiator should include the Sender even if the sender is not a member of the group. The group may also include nodes which are neither Senders or Receivers as long as there is no chance of ambiguity when they receive the file transfer messages. That is, these nodes can not also be potential Receivers. Such nodes are not included in the group size and thus do not respond. The typical case of this is where the Initiator is bound to the Receivers via a group and then tells the Sender to use that same group for the transfer.

Concurrency

It is possible for a single Initiator or multiple Initiators to concurrently conduct file transfers involving the same Sender and/or Receivers. This requires that the Sender and/or Receivers have multiple sets of file transfer NVs. It is the Initiators job to find an available set of file transfer NVs. If multiple file transfer NVs are defined, they must be defined in NV arrays. A set of file transfer NVs is thus grouped based on their common array index. Note that an exception to this is that LONMARK® nodes may delineate a set of NVs by virtue of their belonging to a common LONMARK object.

Note that regardless of the number of file transfer NVs, a Receiver can not have multiple incoming files in progress because it has no way to differentiate incoming file transfer messages. An exception to this rule is if the Receiver/Initiator are the same node then there may be multiple simultaneous incoming transfers.

Multiple concurrent file transfers using a single set of NVs is possible but problematic. First, it does not work for random access transfer because SNVT_file_pos does not include a file index. Second, it requires that the Initiators be bound to the Sender/Receivers but there is no means for an Initiator to determine that such bindings are required. Third, it complicates multicast transfers in term of determining which group to use as the Sender's destination. For these reasons, this form of operation is not considered to be interoperable.

`SNVT_file_req` Data Structure

In order to avoid operating system dependencies, a file on a Sender or Receiver is identified with a unique 16-bit number called the file index. This allows up to

65,535 files to be identified on any node. The file index is used as an argument to file open and directory lookup operations.

The network variables of type SNVT_file_req contain an operation code and a file index. When a node receives a network variable update of this type, it performs the indicated operation, and returns the status of that operation in a network variable of type SNVT_file_status. The request operation codes are defined in the Neuron C include file SNVT_FR.H as follows:

```
0 FR_OPEN_TO_SEND
1 FR_OPEN_TO_RECEIVE
2 FR_CLOSE_FILE
3 FR_CLOSE_DELETE_FILE
4 FR_DIRECTORY_LOOKUP
5 FR_OPEN_TO_SEND_RA
6 FR_OPEN_TO_RECEIVE_RA
```

The request functions are:

FR_OPEN_TO_RECEIVE

Opens the indicated file for writing. The Receiver node executes a file open operation. The request also contains a timeout value in milliseconds to be used to recover from Sender node failures. Status returned may be FS_OPEN_FAIL or FS_XFER_UNDERWAY. FS_XFER_OK can also be returned to indicate a delayed response (see "Delayed Responses" above). Once a file is open, the node will reject all further attempts to open a file until the file is closed.

FR_OPEN_TO_SEND

Opens the indicated file for sequential reading. Additional parameters to this request are a destination explicit address, and two booleans to indicate whether authenticated and/or priority messaging should be used. If there is only one Receiver, the destination explicit address is a subnet/node address. If there is more than one Receiver, it is a group address. The explicit address also contains a retry count and a transaction timer to be used for the request/response message at the end of every window. The Sender node executes a file open operation, and begins a file transfer by sending packets to the indicated nodes on the domain in which the FR_OPEN_TO_SEND was received. Status returned may be FS_OPEN_FAIL or FS_XFER_UNDERWAY. FS_XFER_OK can also be returned to indicate a delayed response (see "Delayed Responses" above). Once a file is open, the node will reject all further attempts to open a file until the file is closed.

FR_CLOSE_FILE

Closes and saves the specified file. Status returned is FS_XFER_OK. The status can also be left at its current value to indicate a delayed response (see Delayed Responses above).

FR_CLOSE_DELETE_FILE

Closes and backs out any changes to the specified file. Status returned is FS_XFER_OK. This is used for backing out of an aborted transfer. The file is restored to the state it was in prior to the start of transfer. Note that the "DELETE_FILE" name is somewhat of a misnomer in that the file remains in the directory.

FR_DIRECTORY_LOOKUP

Retrieves directory information for the specified file. Status returned is FS_LOOKUP_OK or FS_LOOKUP_ERR. FS_XFER_OK can be returned to indicate a delayed response (see "Delayed Responses" above).

FR_OPEN_TO_SEND_RA

Same as FR_OPEN_TO_SEND except it opens the indicated file for reading using random access. The normal status is FS_SEEK_WAIT rather than FS_XFER_UNDERWAY.

FR_OPEN_TO_RECEIVE_RA

Same as FR_OPEN_TO_RECEIVE except it opens the indicated file for writing using random access. The normal status is FS_SEEK_WAIT rather than FS_XFER_UNDERWAY.

SNVT_file_status Data Structure

The status field in the file status structure contains the status of the last honored request to that node. As long as the node is the process of a data exchange, the status is FS_XFER_UNDERWAY. If the node is awaiting a seek operation, the status is FS_SEEK_WAIT. At the end of the transfer, the status becomes FS_XFER_OK. If a file read or write operation fails, the status is FS_IO_ERR. If the transfer was aborted due to a time-out or transaction failure, the status is FS_TIMEOUT_ERR. If a window is received out of sequence, the Receiver node's status is FS_WINDOW_ERR. The returned status codes are defined in the Neuron C include file SNVT_FS.H as follows:

```
0 FS_XFER_OK
1 FS_LOOKUP_OK
2 FS_OPEN_FAIL
3 FS_LOOKUP_ERR
```

4 FS_XFER_UNDERWAY
5 FS_IO_ERR
6 FS_TIMEOUT_ERR
7 FS_WINDOW_ERR
8 FS_AUTH_ERR
9 FS_ACCESS_UNAVAIL
10 FS_SEEK_INVALID
11 FS_SEEK_WAIT

The status structure always contains the number of files on the node, and the index of the file that was the subject of the last operation.

If the last operation was an FR_OPEN_TO_SEND, FR_OPEN_TO_RECEIVE, FR_OPEN_TO_SEND_RA or FR_OPEN_TO_RECEIVE_RA operation, the data structure returned from a Sender or Receiver to the Initiator always contains the full (domain, subnet, node) address of the Initiator. This is for the case of multiple Initiators when there may be several operations attempted concurrently on the same set of file transfer NVs. Each Initiator is responsible for checking its own address against the value returned in the file status structure to ensure that it was granted the requested access. An Initiator must not close a file (i.e., as part of its error handling) unless it was granted access.

If the last operation was a successful FR_DIRECTORY_LOOKUP operation, the status structure contains the directory entry for the specified file. The directory entry is composed of a 16-bit file type, a 32-bit file size, and a 16-character file information array. The latter can be used for any purpose though a NULL terminated ASCII string is recommended. Neither the type nor information string fields have any significance to the file transfer software, they are provided as a convenience to the application. For example, the information string may be used as a file name for a host operating system.

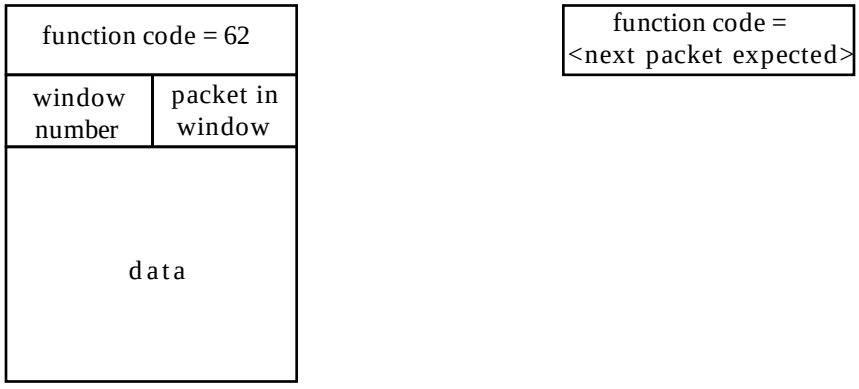
It is recommended that file types be at least 256 to avoid conflicting with file types that may be assigned by the LONMARK Association.

SNVT_file_pos Data Structure

This structure is used when the file is opened for random access. It contains a 32-bit file position value, representing a byte offset from the beginning of the file. If the specified offset is beyond the end of file, a status of FS_SEEK_INVALID is returned. This structure also contains a 16-bit byte count which specifies the length of the next file transfer. The file must be opened using one of the random access modes (FR_OPEN_TO_SEND_RA or FR_OPEN_TO_RECEIVE_RA). If the node does not support random access then it will not have a SNVT_file_pos NV and it returns a status of FS_ACCESS_UNAVAIL to a random access open request.

Application Protocol Data Unit Structure

Each packet of data is transmitted in an explicit message with a message code of 62, a one-byte header, and 32 bytes of data. The message code of 62 is reserved by the *LONMARK™ Application Layer Interoperability Guidelines* for interoperable file transfer. The most significant nibble of the header contains the window number (modulo 16), and the least significant nibble contains the packet number within the window (0-5). The Receivers check that packets are received in sequential order, and respond with a request for retransmission starting from the first packet not received correctly within the current window. Receivers also check that windows are received in sequential order. This is to handle the case where one Receiver has received all of a window successfully, and another Receiver has not received the first packet of that window. The window number allows the Receivers to distinguish a retransmission of the current window from the start of a new window.



Application Protocol Data Unit for packets from Sender to Receiver (Unacknowledged or Request)

Application Protocol Data Unit for packets from Receiver to Sender (Response)

Figure 4 Application Protocol Data Unit structures

Operating System Dependencies

The file transfer software does not implement a file system. This must be provided by the implementor. For example, on a Neuron Chip-hosted node, a file on a Sender node may be a RAM array, or a data stream acquired from some sensor in real time as the transfer is proceeding. On a node implemented with a host computer, a file may be a real disk file, or some other sequential device. The timeouts specified in the file open requests should take into consideration the time needed to obtain the data with a read operation on the Sender node, and to dispose of the data with a write operation on the Receiver node.

The file system must provide the following operations:

```
file_handle open(file_index, operation);
```

This opens the specified file for reading or writing, and returns a file descriptor to be used for later read or write operations.

```
int read(file_handle, void *, int);
```

This reads the specified number of bytes from the file into the specified buffer, and returns the number of bytes read. If this is less than the number of bytes specified, an end-of-file has been reached. If an error has occurred, -1 is returned.

```
int write(file_handle, void *, int);
```

This writes the specified number of bytes from the specified buffer into the file, and returns the number of bytes written. If this is less than the number specified, an error has occurred.

```
void close(file_handle, int backout);
```

This closes the file. If `backout` is `TRUE`, the changes are backed out.

If random access is supported, it must also provide the following:

```
int seek(file_handle, offsetType, lengthType);
```

Implementation

Two model implementations of the file transfer protocol are available. One is in Neuron C for Neuron Chip-hosted nodes, and the second is in C for PC-hosted nodes. These examples may be downloaded from Echelon's website at http://www.lonworks.echelon.com.

The Neuron C model implementation contains five files:

The file `FILE.H` contains the definitions of the types `file_index`, `file_descriptor` and `file_packet`, as well as the defined symbols `FILE_PACKET_SIZE` (32) and `FILE_WINDOW_SIZE` (6).

The file INITIATOR.NC contains the code necessary to initiate a transfer between a Sender and one or more Receivers. Random access file transfer support is not included. The user defines the parameters of the transfer, which are:

Parameter	*Type*	*Meaning*
sender_index	file_index	The index of the source file on the Sender node
receiver_index	file_index	The index of the destination file on the Receiver nodes
auth_on	boolean	Whether to use authentication for the request messages
prio_on	boolean	Whether to use priority for all messaging
receiver_timeout	unsigned long	Timeout for the Receiver nodes in msec
sender_timeout	unsigned long	Timeout for the Sender node in msec
retry_count	unsigned int	Number of retries for the request messages

This implementation makes the following assumptions. These are for simplification of the implementation and are not requirements for interoperability.

- The parameters for all Receivers (in the multicast case) must be the same, specifically, the file index and the receiver timeout.
- The Initiator is bound to the Sender and Receivers. In the multicast case, the group connecting the Initiator to the Receivers contains only the Initiator and Receivers (i.e., no other members, no group overloading).

The file FILEXFER.NC contains the code for the Sender and Receiver nodes. It implements random access file transfer as well. Any of Sender, Receiver or Random Access capability can be removed via conditional compilation. An additional conditional compilation option to simulate network errors is provided and is off by default. FSYS.H contains a skeleton for a file system on the Neuron Chip. The code is marked to show the places in which the real file system operations should be implemented.

On a 1.25 Mbps twisted pair channel, the maximum file transfer throughput for this interoperable model implementation is 2.0 kbytes/second. If the packet size is increased to 225 bytes, and the window size to 15 packets, then the throughput becomes 5.0 kbytes/sec, but this implementation is not interoperable.

Appendix A. Definitions of File Transfer SNVTs

These structures are implicitly defined by the Neuron C compiler.

```
typedef struct {
    file_request    request;
    unsigned long   index;
    unsigned long   receive_timeout;
    union {
        struct {
        unsigned    type;    // 1 for subnet/node
        unsigned    domain  : 1;
        unsigned    node    : 7;
        unsigned            : 4;
        unsigned    retry   : 4;
        unsigned            : 4;
        unsigned    tx_timer: 4;
        unsigned    subnet;
        } sn;
        struct {
        unsigned    type    : 1;    // 1 for group
        unsigned    size    : 7;
        unsigned    domain  : 1;
        unsigned            : 7;
        unsigned            : 4;
        unsigned    retry   : 4;
        unsigned            : 4;
        unsigned    tx_timer: 4;
        unsigned    group;
        } gp;
    } dest_address;
    int    auth_on;
    int    prio_on;
} SNVT_file_req;

typedef struct {
    file_status   status;
    unsigned long number_of_files;
    unsigned long selected_file;
    union {
        struct {
            char     file_info[ 16 ];
            unsigned size[ 4 ];
            unsigned long type;
        } descriptor;

        struct {
            unsigned domain_id[ 6 ];
            unsigned domain_length;
            unsigned subnet;
            unsigned node;
        } address;
    } adr;
} SNVT_file_status;

typedef struct {
    unsigned      rw_ptr[ 4 ];    // 32-bit value compatible with s32_type
    unsigned long rw_length;
} SNVT_file_pos;
```

LONWORKS™ Power Line SCADA Systems

September 1992 LONWORKS Marketing Bulletin

Abstract

LONWORKS™ is a general purpose control system technology that can be used to monitor sensors, control actuators, and display system status in a wide variety of applications. One such application is the creation of a LONWORKS supervisory control and data acquisition (SCADA) system for manufacturing processes, machine diagnostics, and fault and safety alarms. With the advent of LONWORKS power line transceivers, SCADA signalling can now be effected through the same AC or DC mains wiring that powers the equipment under supervision. Power line signalling eliminates the need for additional communications cabling, and both reduces the expense and simplifies the task of retrofitting existing installations. LONWORKS power line transceivers are based on a powerful new Echelon custom integrated circuit, and incorporate many technological innovations that enable them to signal reliably on an otherwise hostile and noisy medium. The transceivers meet FCC regulations and are available in several configurations for OEM applications.

Introduction

LONWORKS technology enables the creation of intelligent SCADA systems that can automatically supervise sensors, trigger control output devices, operate displays and control panels, and intercommunicate with or without a central computer. The heart of a LONWORKS SCADA system is the NEURON® CHIP, an integrated circuit that combines a sophisticated communication protocol, three microprocessors, a multitasking operating system, a programmable event driven logic program, and a flexible input/output scheme. Before the advent of the NEURON CHIP, anyone

Echelon Part Number 005-0035-01 REV A

building a SCADA system also had to develop a protocol and microelectronics for transmitting data reliably between the various system devices. The NEURON CHIP's highly refined and very reliable communication scheme - one perfected over the past four years - eliminates the need to develop a proprietary communication system, saving both time and resources.

A typical LONWORKS SCADA system consists of a network of sensors and actuators that are supervised by local controllers, each containing a NEURON CHIP. The local controllers monitor the sensors and trigger the actuators based on operating programs that are executed by the NEURON CHIPS. The controllers are usually interconnected using copper wiring (figure 1). Each sensor and actuator is wired to a local controller, which in turn may be wired to a central auditing and display system. Changes in the status of the devices connected to the system are broadcast over the wires, as are programming commands used to reconfigure the operation of the network devices.

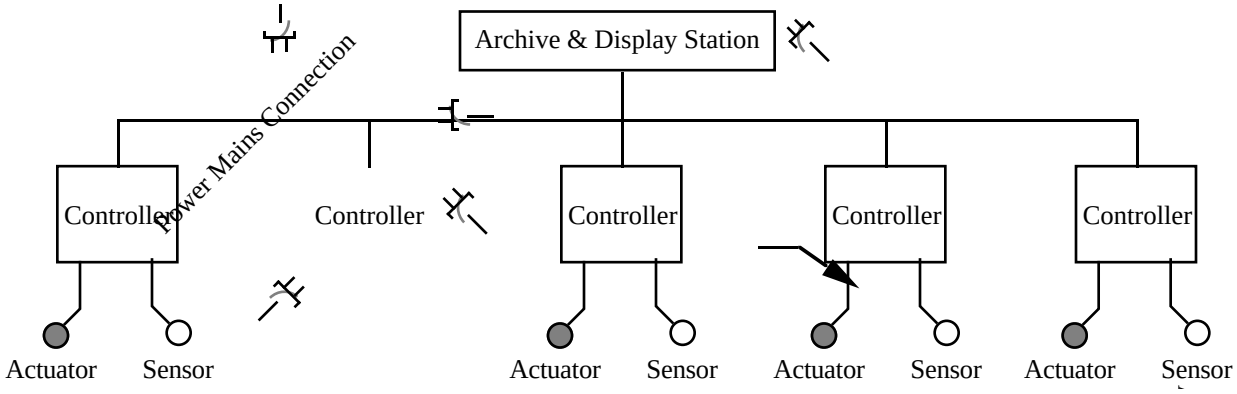


Figure 1
LONWORKS SCADA System Using Copper Wiring

There is a high cost associated with installing and maintaining the cable plant that links together the many elements of the SCADA system. The installation of dedicated cabling often involves the use of expensive conduits to carry the wires and junction boxes for maintenance access. If a cable is severed or its conductors inadvertently shorted or grounded, time must be spent troubleshooting the wiring harness to isolate and repair the fault. Each time a sensor is added or an actuator is moved, the system wiring must be changed accordingly, often resulting in network down time until the new connections can be established (figure 2).

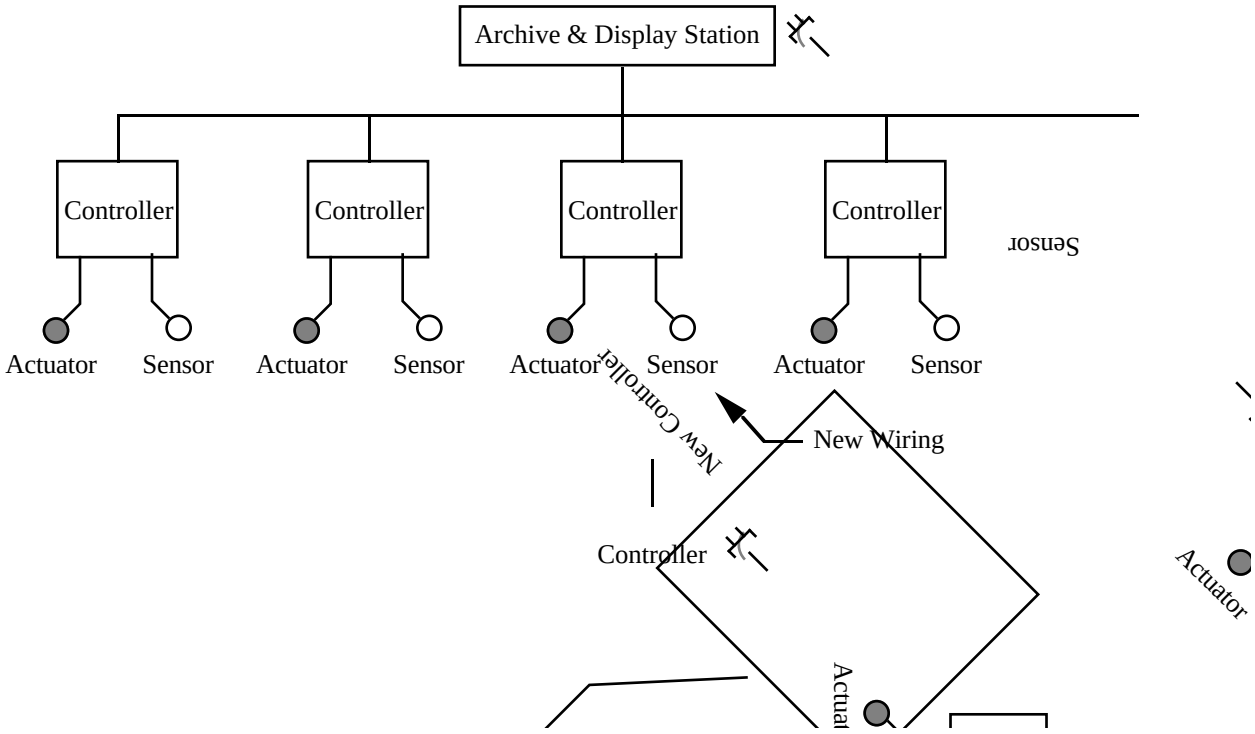


Figure 2
Expanding A LONWORKS SCADA System Using Copper Wiring

One method of simplifying the installation of a SCADA system is to eliminate the use of dedicated wiring by using an existing signal path. Radio and infrared signalling offer two possible solutions. Radio signalling systems employ radio transceivers at each controller, and communicate by broadcasting radio signals at specific frequencies. Infrared transceivers perform the same function but use bursts of infrared light in lieu of radio waves. Both types of transceivers provide reliable signalling provided that a clear line-of-sight path exists between the transceivers. For example, Echelon's RF-10 Transceiver is ideal for short distance radio communications where the path is free of obstacles that might block the radio signals.

Unfortunately, radio signalling is often impractical because of licensing restrictions, the presence of radio frequency interference, and the inability to reliably penetrate many building materials present in large installations (figure 3). Infrared signalling is complicated by packaging problems associated with making the transceiver visible outside of the equipment enclosure, and also by the need for an extensive network of repeaters to redirect signals around equipment and between rooms (figure 4).

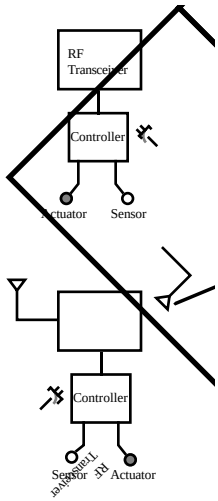


Figure 3
Problematic Application of a Radio Frequency Communication System

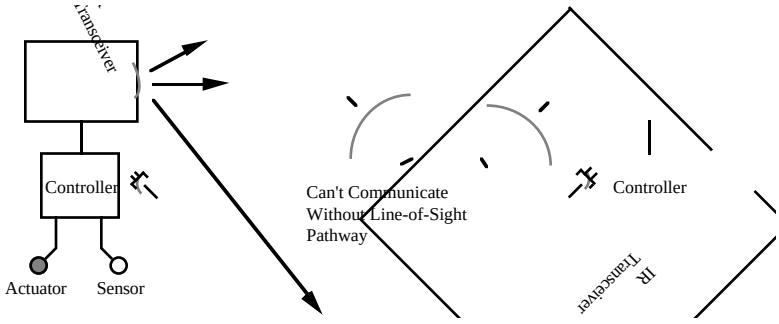


Figure 4
Problematic Application of an Infrared Communication System

Power Line Signalling

One medium that is common to all SCADA systems is the power mains. Power wiring pervades most sensors, actuators, displays, and related equipment since these devices generally require electrical power for normal operation. For this reason, LONWORKS power line technology provides an elegant and inexpensive method of linking together the different elements of a SCADA system.

When a sensor or actuator with a LONWORKS power line transceiver is installed, its connection to the power mains also provides access to the communication medium and, in turn, to the SCADA system itself (figure 5). If a new motor controller, test device, or fault alarm is added to a facility, its power connection becomes the pathway by which it communicates with the rest of the SCADA system. This eliminates the need for dedicated monitoring system wiring, and greatly reduces the cost and complexity of the system installation.

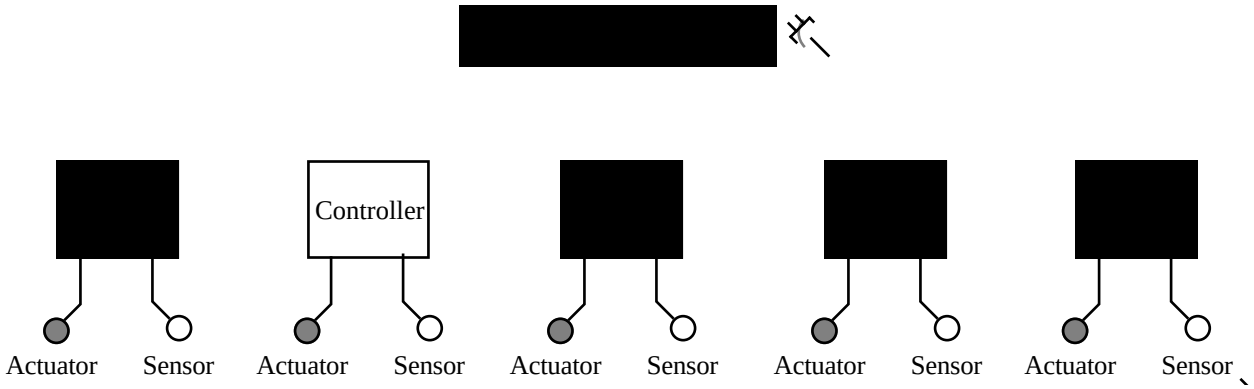


Figure 5
LONWORKS Power Line SCADA System Wiring

Since power wiring is usually present throughout an entire facility, power line signalling also makes it simple to move or add remote displays as they are needed. By simply plugging them into an appropriate power outlet, the displays will receive status information throughout the facility. This permits portable displays to be moved and shared between different applications as required without the need for new cabling.

For example, it might be desirable to interconnect NEURON CHIP-based pump controllers so that a fault in one pump can trigger an orderly shutdown by others pumps on the SCADA system (figure 6). In addition, the SCADA system can independently audit the performance of the entire pump network, and detect if a controller malfunctions, the power supply drops below recommended levels, or any other potentially hazardous condition exists. When a fault is detected, any other machines that rely on the correct operation of the faulty machine will be quickly notified through the power line that is shared by all of the pumps.

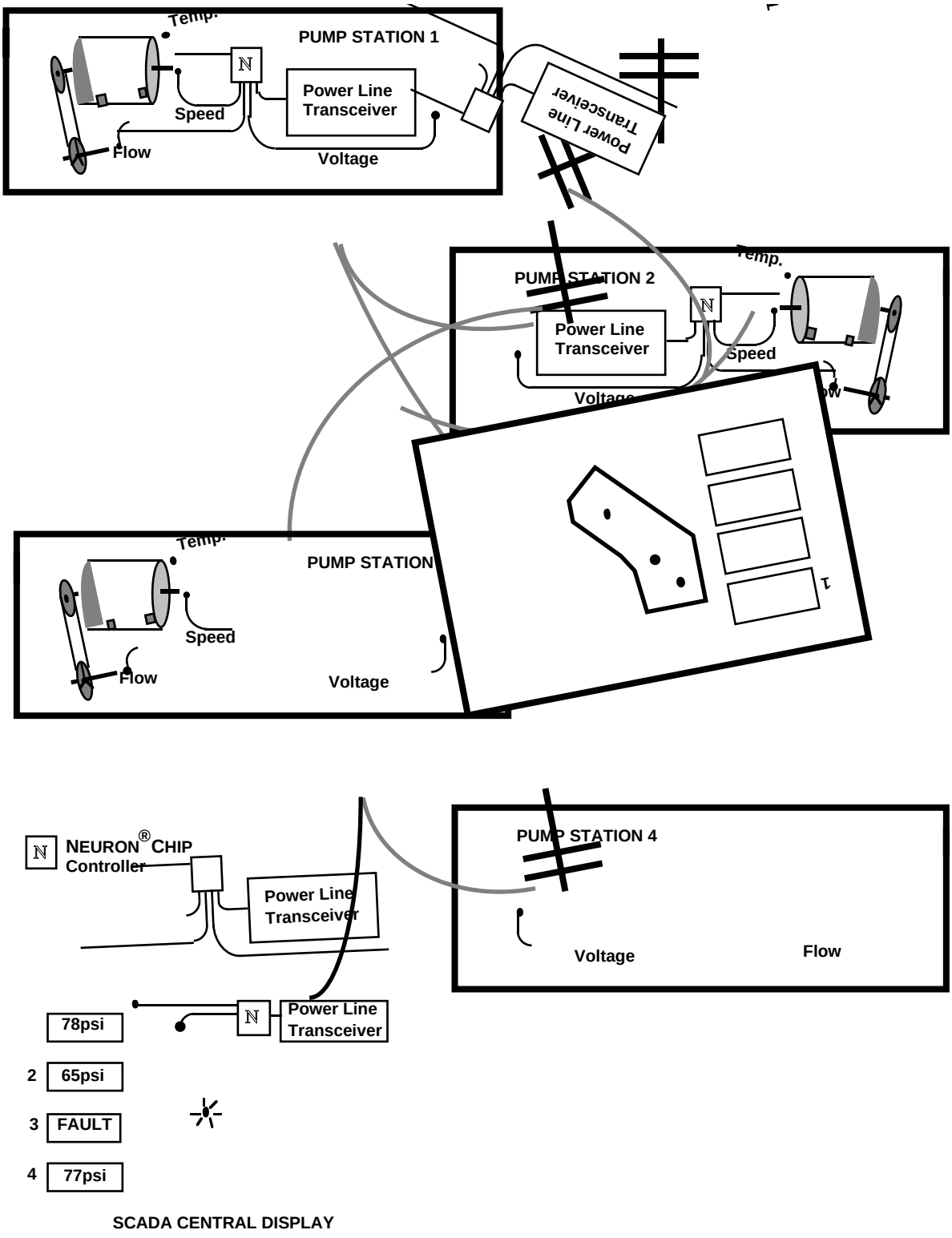


Figure 6
LONWORKS Pump SCADA System

Another typical example would be a NEURON CHIP-based machine in which different configurations of the assembly are sold to different customers, or are added as production needs grow (figure 7). In the example shown below, each modular assembly of the plasma treatment system is equipped with a power line transceiver and SCADA system. Modules 1 to 3 might be installed initially and use the power line for machine control signalling. At a later time, a display station might be required in a supervisor's office. Module 4, a remote display panel, can be added easily by plugging it into the power mains. This design eliminates the need to change the wiring plant by installing a conduit to the supervisor's office: the net result is a simpler, less expensive expansion of the system.

Power Line Technology

The power line is optimized for supplying power at 50/60Hz, but is an extremely hostile communication medium whose electrical characteristics do not readily transmit frequencies needed for useful data rates. The wire inductance and loading effects of electrical devices plugged into the power line can easily result in the attenuation of a transmitted signal by 40dB (a factor of 100) to 60dB (a factor of 1,000). In addition, electrical equipment such as motors, light dimmers, switching power supplies, and fluorescent light ballasts inject noise that can mask low-level power line control signals. Worse yet, these electrical characteristics vary from instant to instant presenting a continuously changing communication channel. Echelon's engineers had to develop unique methods of signal processing and error correction to tame the power line and turn it into a robust communication circuit.

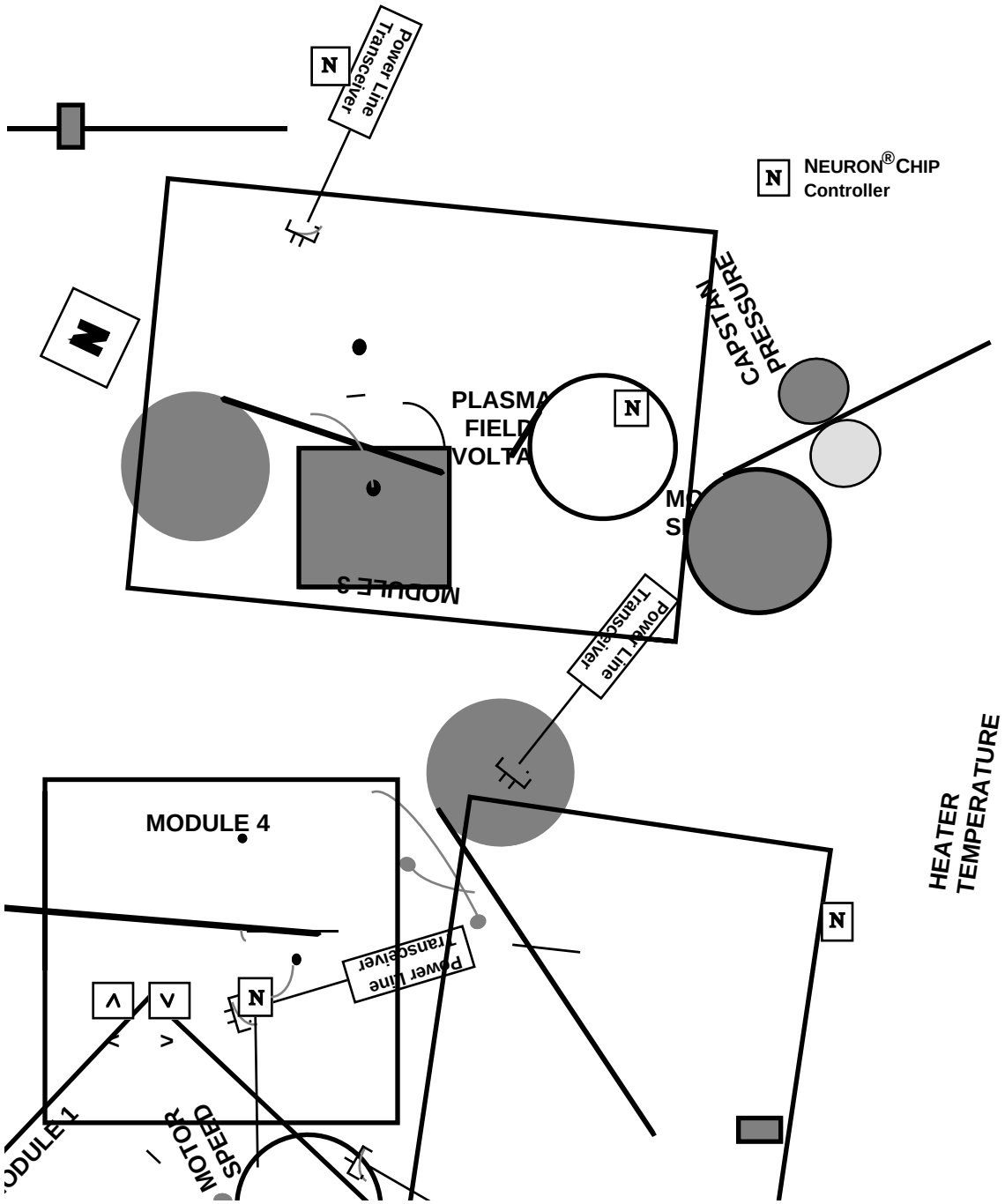


Figure 7
LONWORKS Modular Machine SCADA System

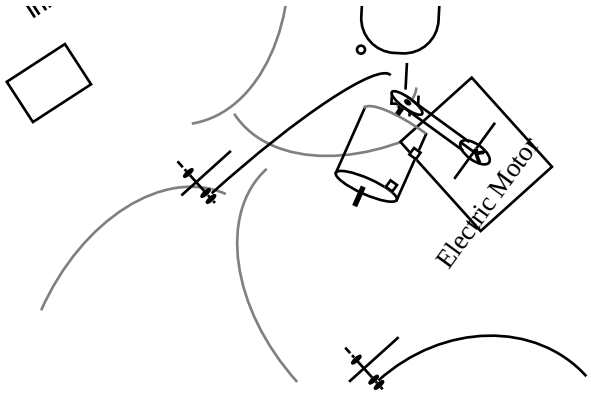


Figure 8
Typical Sources of Power Line Noise and Attenuation

Technical Features

Spread spectrum signalling is the best method of communicating via the power line because of its ability to burst through power line noise. Ideally, spread spectrum transmission should distribute a signal over the widest possible bandwidth for optimum performance against electrical noise. Bandwidth limits for power line signalling in the U.S. are set by the Federal Communications Commissions (FCC). The lowest practical frequency is around 100KHz, since power line noise increases dramatically below this frequency. The practical upper limit is 450KHz since the FCC has mandated that power line signals not interfere with AM radios operating down to 535KHz. To use the greatest possible portion of this 350KHz-wide band, Echelon's engineers developed a modified direct sequence spread spectrum encoding technique with a bit rate of 10 kilobits per second.

Echelon's spread spectrum technology is also well suited for certain European SCADA applications. Because of the proliferation of long-wave radio stations operating above 150KHz, European restrictions on power line signalling systems are more severe than in the U.S. The widest bandwidth available is from 9 to 95KHz, which is reserved for utility-related applications, such as remote meter reading and load shedding. Although only 86KHz wide, this bandwidth is large enough to support the direct sequence spread spectrum technology developed for the U.S.

Echelon's 31 chips/bit encoding scheme yields 2 kbits/s for the 9 to 95KHz band in Europe, in contrast with a bit rate of 10 kbits/s for the 100 to 450KHz band in the U.S.

Europe has also allocated the band from 125 to 140KHz for general-purpose applications that are not associated with utilities. The strict limits imposed on the shape of the signal waveform eliminates the possibility of using spread spectrum transmissions in this band. For these applications, Echelon is developing a second custom power line IC that will use narrow band signalling to comply with European regulations for bandwidth and media access protocol. This chip will be available early next year.

SCADA System Response Time

The response time of a SCADA system is the speed at which it can react to a change in the status of a sensor, output device, or operator directive. Changes in the status of sensors, actuators, displays, or the controllers are broadcast as packets of data bits. A data packet generally includes the address of the sending and receiving controllers, command signals, and error correction information. In a LONWORKS control system, the process of formatting and decoding packets is handled by the NEURON CHIP, which is optimized for this purpose. The NEURON CHIP works in concert with Echelon's custom PLIC to deliver between 55 and 60 packets per second for typical applications. System response time depends on how fast the packets can be formatted by the transmitter, broadcast over the communication medium, and then decoded by the receiver. Echelon has devised several techniques to minimize system response time.

One factor contributing to response time of a SCADA system is the overhead needed to prepare a control command for transmission (handled by the NEURON CHIP), and ensuring that the packet is received without error (supported by the PLIC). Echelon included in the PLIC a high performance oversampling correlation filter and adaptive bit timing recovery algorithm that synchronizes instantaneously with the incoming command and adapts to dynamically varying power line impairments. The net effect is faster system response time and less traffic congestion on the

communication lines.

If a command is corrupted in transit, such as might happen if noise interferes with the signal transmission, it must either be reconstructed at the receiving end or retransmitted in its entirety. Packet reconstruction is faster than packet retransmission. Echelon developed a unique error correction system that adds a premium of only 6% to the length of the command. Classical methods of error correction add up to 50% to the length of a packet, and since these additional error correction bytes take time to transmit and decode, they decrease system response time.

A special purpose mode of the NEURON CHIP tightly couples it with the PLIC. This feature provides a frame based serial protocol that allows the PLIC to pass data asynchronously with the NEURON CHIP. The net effect of closely coupling the PLIC and the NEURON CHIP is to increase by 1.5 times the effective data rate compared with other transceivers that operate using symbol encoding.

A combination of these and other technical advancements allows a LONWORKS power line SCADA system to react quickly to status changes, even in the face of impediments on the power line medium. The fast response time of the LONWORKS system makes it suitable for a wide range of industrial and machine control applications, as well as for building and home automation projects.

Product Range

Echelon's new power line solutions are offered in two different packages, both based on the firm's custom spread spectrum integrated circuit (PLIC). These include the PLT-10 Transceiver Module and the PLC-10 Control Module. The PLT-10 Transceiver Module consists of a 1.5 x 1.7 x 0.5 inch encapsulated module containing a direct sequence spread spectrum PLIC integrated circuit, crystal oscillator, receive front end, and transmitter amplifier and filter. The PLT-10 Transceiver Module can be mounted on or inside of an OEM product, and requires the addition of an Echelon NEURON® CHIP, power line coupling network, and power supply. The PLT-10 Transceiver Module can be used to couple LONWORKS control signals to virtually

any AC or DC power line, regardless of voltage, through the selection of an appropriate coupling circuit.

The PLC-10 Control Module consists of a 2.5 x 3.5 x 1 inch circuit card containing a NEURON 3150 CHIP, PROM socket, direct sequence spread spectrum PLIC integrated circuit, transformer isolated power supply, 120VAC power line coupling circuit, and connectors for power and I/O. The PLC-10 Control Module is a complete LONWORKS power line controller, and can be connected directly to the sensors, outputs, displays, or application electronics that the module will control. Both differential (2-wire) and common mode (3 wire) coupling networks are available. Differential models are required for use in older homes where no ground exists, or where sensitive ground fault interrupters are attached to the power line.

Separate versions of these products will also be offered for Europe in both a transceiver module configuration and a control module. Both versions will employ a second Echelon-developed IC using narrow-band signalling and will address the growing European home and building automation markets. These products will be introduced in early 1993, and will meet the latest European power line signalling regulations.

Summary

With the advent of LONWORKS power line transceivers, SCADA systems can now be effected through the same mains wiring that powers the equipment under supervision. Power line signalling eliminates the need for additional communications cabling, and both reduces the expense and simplifies the task of retrofitting existing installations. LONWORKS power line transceivers are based on a powerful new Echelon custom integrated circuit, and incorporate many technological innovations that enable them to signal reliably on an otherwise hostile and noisy medium. The transceivers meet FCC carrier current regulations and are available in several configurations for OEM applications.

Developing a Network Driver for the PC LonTalk® Adapter

June 1994

LONWORKS® Engineering Bulletin

Introduction

This engineering bulletin is intended for those developers needing to port the PCLTA network driver for DOS to PC operating systems not based on DOS. The registers and protocol used to communicate with the PCLTA are described.

Overview

The network driver provides a hardware-independent interface between the host application and the PCLTA. The driver communicates with the PCLTA by reading and writing to a set of 8-bit registers, with optional hardware interrupts. A LONWORKS standard network driver must supply four basic functions: `ldv_open()`, `ldv_read()`, `ldv_write()` and `ldv_close()`. These functions provide a suitable operating-system independent definition for the network driver.

The network driver carries out these tasks:

- Transfers data to and from the PCLTA over the parallel interface.
- Buffers data moving in both directions over the interface.
- Responds to interrupts from the PCLTA (optional).

This diagram shows how the network driver fits into the host application architecture.

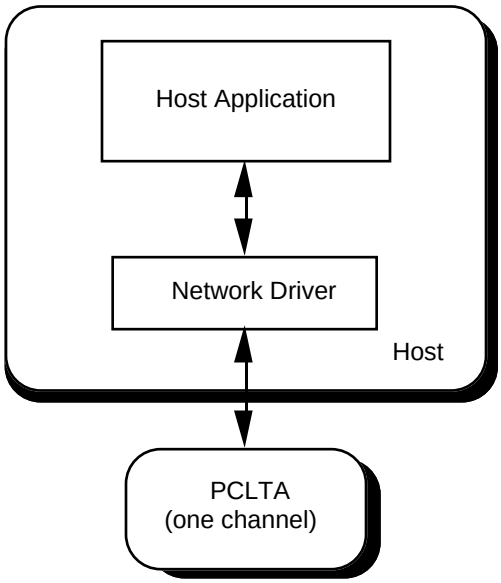


Figure 1 PCLTA Host Software Architecture

To avoid ambiguity when describing the driver interface, the terms *uplink* and *downlink* are used to describe the direction of data travel. These terms use a "host-centric" perspective. The downlink direction is from the host and uplink is toward the host. The host writes data downlink to the PCLTA. The PCLTA moves data downlink to the network. Uplink data moves from the transceiver into buffers in the PCLTA. The PCLTA moves the data uplink to the host.

Implementing a Network Driver

The LonTalk protocol defines a standard calling convention for network driver functions. A network driver must provide these four functions, which are called by the host application. The syntax for these functions may be dependent on the operating system. The PCLTA Starter Kit includes a DOS implementation of a PCLTA network driver. The source code for this network driver is available on the LonLink™ bulletin board, and provides a convenient starting point for creating a network driver for other operating systems.

Functions Implemented by the Network Driver

A LONWORKS standard network driver implements the following four functions. Detailed specifications follow.

`ldv_open()`	Opens the network driver, allocates necessary resources, and initializes the PCLTA
`ldv_close()`	Closes the network driver and de-allocates resources associated with the open device.
`ldv_read()`	Retrieves an available message from the network driver. The function returns immediately with an error indication if no messages are available.
`ldv_write()`	Delivers a message to the network driver. The function returns immediately with an error indication if no downlink buffers are available.

In the following functional specifications, the data type `LDVCode` is an enumeration defining all the status codes returned from the network driver. The data type `LNI` is a `typedef` for the device handle returned from `ldv_open()`.

ldv_open()

Purpose

This function opens the network driver, establishes communications with the PCLTA, and returns a network interface handle for use with other network driver functions. The specified PCLTA and associated data structures are initialized.

Syntax

```
LDVCode ldv_open( const char * device_name, LNI * pHandle );
```

Input Parameter

`device_name`	Null terminated string containing the name of the device associated with the network driver. This name is operating-system dependent. For the PCLTA DOS network driver, this name is of the form LONx, where x is the device number specified when the network driver is loaded in `CONFIG.SYS`.

Output Parameter

`pHandle`	A pointer to an operating-system dependent handle for use with the other network driver functions. The returned value is valid only if the result is `LDV_OK`.

Result Code

`LDV_OK`	0	The network driver was successfully opened
`LDV_NOT_FOUND`	1	The device name supplied could not be found
`LDV_ALREADY_OPEN`	2	An attempt to open a device that was already open
`LDV_DEVICE_ERR`	4	Hardware or operating system error
`LDV_NO_RESOURCES`	8	Unable to allocate resources necessary to open device

ldv_close()

Purpose

This function closes the network driver, and recovers all resources associated with the open device.

Syntax

```
LDVCode ldv_close( LNI Handle );
```

Input Parameter

`Handle`	An operating-system dependent handle returned from `ldv_open`

Result Code

`LDV_OK`	0	The network driver was successfully closed
`LDV_NOT_OPEN`	3	The specified network driver was not open
`LDV_DEVICE_ERR`	4	Hardware or operating system error
`LDV_INVALID_DEVICE_ID`	5	The specified handle was not associated with a LONWORKS network interface

ldv_read()

Purpose

This function reads a network interface command or application buffer from the specified network driver.

Syntax

```
LDVCode ldv_read( LNI handle, void * pMsg, unsigned length );
```

Input Parameters

Handle	An operating-system dependent handle returned from ldv_open
length	The number of bytes available in the supplied buffer

Output Parameter

pMsg	A pointer to a buffer which will contain either an immediate network interface command, or an application buffer. See the *Host Application Programmer's Guide* for more details. The returned message is only valid if the function result is LDV_OK.

Result Codes

LDV_OK	0	The function returned an uplink message
LDV_NOT_OPEN	3	The specified network driver was not open
LDV_DEVICE_ERR	4	Hardware or operating system error
LDV_INVALID_DEVICE_ID	5	The specified handle was not associated with a LONWORKS network interface
LDV_NO_MSG_AVAIL	6	No uplink message was pending in the network driver or the network interface
LDV_NO_RESOURCES	8	Unable to allocate resources necessary to read a message
LDV_INVALID_BUF_LEN	9	The specified buffer length was insufficient for the next uplink message

ldv_write()

Purpose

This function writes a network interface command or application buffer to the specified network driver.

Syntax

```
LDVCode ldv_write( LNI handle, void * pMsg, unsigned length );
```

Input Parameters

`Handle`	An operating-system dependent handle returned from `ldv_open`
`pMsg`	A pointer to a buffer which contains either an immediate network interface command, or an application buffer. See the *Host Application Programmer's Guide* for more details.
`length`	The total number of bytes in the supplied `buffer`

Result Codes

`LDV_OK`	0	The buffer was successfully passed to the network driver. This does not mean that the message was successfully transmitted to the network interface or the network. The uplink completion event can be checked to determine this. See Chapter 6 for more details.
`LDV_NOT_OPEN`	3	The specified network driver was not open
`LDV_DEVICE_ERR`	4	Hardware or operating system error
`LDV_INVALID_ DEVICE_ID`	5	The specified handle was not associated with a LONWORKS network driver
`LDV_NO_BUF_AVAIL`	7	No downlink application buffer was available in the network interface or the network driver.
`LDV_NO_RESOURCES`	8	Unable to allocate resources necessary to write the message

Implementing a PCLTA Network Driver

This section suggests one possible procedure for implementing a PCLTA network driver for operating systems other than DOS or Windows.

1. Implement and test low-level I/O to the PCLTA. Make sure that your driver can read the Status register successfully, and that the value read makes sense.
2. Implement and test a function to write an immediate downlink network interface command to the data register. To test this, write an `niRESET` byte, and observe whether the PCLTA is resetting properly. When the PCLTA is reset, the service LED blinks momentarily.
3. Implement and test a function to read an immediate uplink network interface command. To test this, read the `niRESET` byte that the PCLTA passes uplink after it is reset.
4. Implement and test a function to write an network interface buffer into the PCLTA using polled I/O. To test this, write a local network management Set Node Mode command, specifying the Reset mode. Observe the service LED to see whether the PCLTA resets properly.
5. Implement and test a function to read an application buffer from the PCLTA using polled I/O. To test this, write a local network management Request message, and then read the resulting Response message. A suitable message is the Query Status network management message.
6. Implement the open, close, read and write functions for your particular operating environment.
7. Start porting the example host application to your target. The code for this example is on the LonLink bulletin board as `HA.ZIP`. If the example host application can run successfully, the driver is basically working.
8. Implement and test interrupt support in the network driver to improve real-time response.
9. Implement and test input and output buffering in the network driver to reduce congestion problems.

Network Driver Architecture

The network driver manages the physical interface with the PCLTA, implements the network interface protocol, performs flow control, manages input and output buffers, and provides a read/write interface to the host application. The following figure illustrates how the network driver fits into the host application architecture.

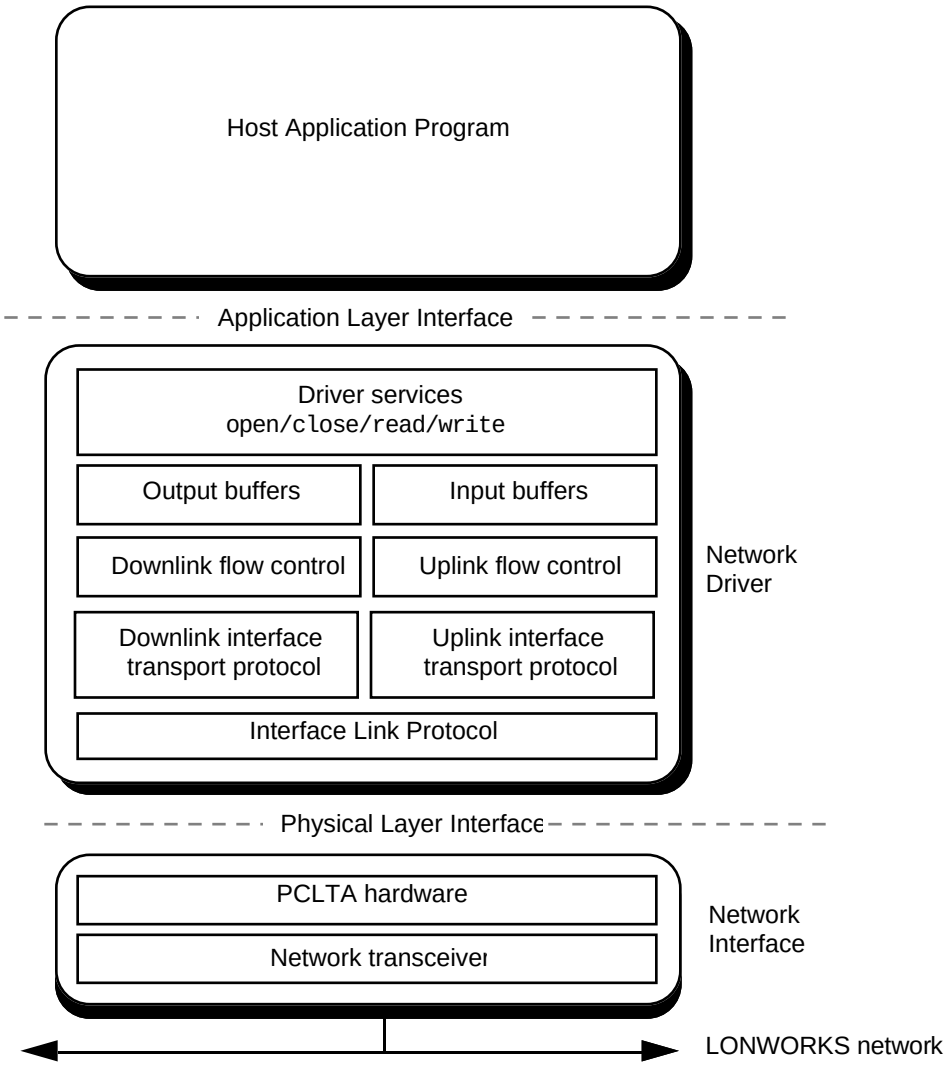


Figure 2 Detailed Architecture of Network Driver

PCLTA Physical Layer Interface

The PCLTA uses a simple and efficient register-based interface to exchange messages with the host processor. For a description of these registers, see Appendix A of the *PCLTA User's Guide.*

Setting the Transceiver Type

When the network driver first accesses the PCLTA hardware, it should read the transceiver ID register to determine how to set the clock rate Neuron Chip on the PCLTA. The clock rate is set using the Clock and Reset Control register described in Appendix A of the *PCLTA User's Guide.* The default input clock rate is 10MHz. Table 2.3 of the *PCLTA User's Guide* defines the clock rate for the standard transceiver IDs. For custom transceivers, the network driver must have some way of ascertaining the correct input clock rate if it is not 10MHz. For example, the DOS driver uses the /C switch for this purpose. For standard transceivers, the PCLTA firmware automatically loads the correct communications parameters into the configuration data structure of the Neuron Chip on the PCLTA. When the transceiver ID is set to type 30 (custom), the PCLTA firmware sets the communications port pins to high impedance to avoid damage. In this case, either the host application or the network driver must load the communications parameters into the Neuron Chip using local network management (`niNETMGMT`) write memory commands. For more details on the communications parameters, see the *Neuron Chip Data Book.*

Interface Link Layer Protocol

Before data can be moved across the interface, the PCLTA must be in the `READY` state and also have come out of `RESET`. The host can test these flags in the Status register, and optionally set the corresponding bits in the Interrupt Request Control register to cause the desired uplink interrupts. The host controls the direction of the interface. To transfer data downlink, the host writes data to the Data register. This causes `READY` to be de-asserted. The host should then wait for `READY` to be asserted again before writing the next byte to the interface. The host can poll the Status register by reading the register, or alternatively, it can avoid polling by enabling an uplink interrupt for the `DLREADY` condition.

To transfer data uplink, the host reads data from the Data register. This causes `READY` to be de-asserted. The host should then wait for `READY` to be asserted again before reading the next byte from the interface. Once an uplink transfer begins, it completes at the Neuron Chip parallel I/O speed, and so is normally implemented with programmed I/O rather than interrupts.

Once a link-layer transfer is underway, the Neuron Chip application processor is blocked executing a fast I/O instruction. The transfer must complete within approximately 840msec (for an input clock of 10MHz) to avoid a watchdog time-out on the Neuron Chip. This can occur, for example, if the network driver is being debugged, and there is a breakpoint in the transfer loop.

Interface Transport Protocol

The interface transport protocol provides data framing, and an orderly turnaround of the interface between uplink and downlink transfers.

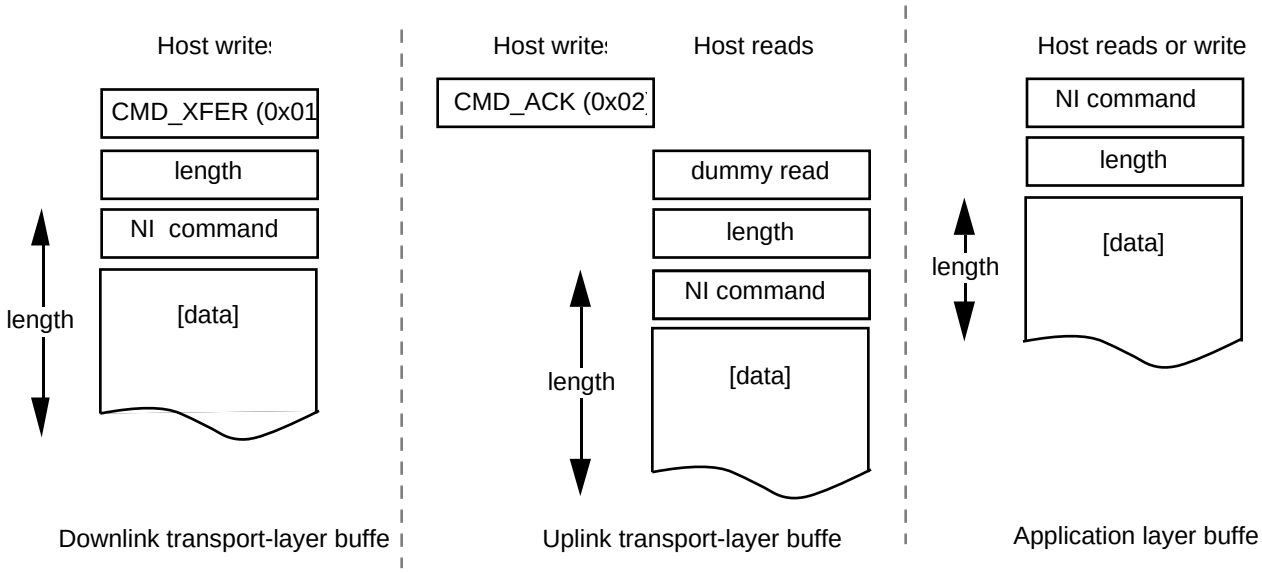


Figure 3 Buffer Formats for PCLTA Network Driver

Downlink Transport Protocol

The downlink transfer sequence may be initiated by the host any time the interface is idle. In addition, if the network interface command is `niCOMM` or `niNETMGMT`, there must be an appropriate downlink buffer available. See the description of the downlink flow control protocol for details. The host first writes a `CMD_XFER` byte (01 hex) to the PCLTA using the link protocol described above. The host then writes a byte containing the number of bytes of data to follow (1 to 255), followed by the data bytes themselves. The first byte of the data is the network interface command. See Appendix B of the *PCLTA User's Guide* for more details on network interface commands. Note that the application layer buffer is slightly different from the downlink transport layer buffer; the length byte and the network interface command byte are swapped, and the value in the length byte differs by one. It is the responsibility of the driver to present the standard application layer buffer to the host application.

The `DLREADY` interrupt request solves the problem of the latency between the host writing the first byte of a downlink transfer and the PCLTA reading it and proceeding with the rest of the transfer. This latency is typically a few tens of microseconds, but could be much longer (up to tens of milliseconds) if a time-consuming network management function is being handled at that time. In this way a host can initiate a downlink transfer by writing the `CMD_XFER` byte and then back off and let the interrupt service handle the next byte of the transfer. Once the PCLTA asserts `READY` for the `CMD_XFER` byte, it transfers the length byte under

software control. This takes several microseconds. Once the PCLTA asserts READY for the length byte, however, the rest of the transfer (starting with the network interface command byte) occurs at parallel I/O speeds (2.4µs per byte for a Neuron Chip with a 10MHz input clock). For most hosts, the time to enter and leave an interrupt service routine exceeds 2.4µs, so that best performance is normally achieved when the rest of the transfer is implemented with programmed I/O rather than interrupts.

Uplink Transport Protocol

The host can initiate an uplink transfer when the PCLTA notifies the host by setting the ULREADY bit in the Status register. The PCLTA can optionally generate an interrupt request if the ULREADY bit is set in the Interrupt Request Control register. Following this, the host acknowledges the uplink transfer by writing a CMD_ACK (02 hex) to the PCLTA, and the host then issues a dummy read to set the READY signal to its proper state for the rest of the transfer. After discarding the dummy byte, the host reads a byte containing the number of bytes of data to follow (1 to 255), followed by the data bytes themselves. The first byte of the data is the network interface command. See Appendix B of the *PCLTA User's Guide* for more details on network interface commands. Note that the application layer message is slightly different from the downlink transport layer message; the length byte and the network interface command byte are swapped, and the value in the length byte differs by one. It is the responsibility of the driver to present the standard application layer buffer to the host application.

Downlink Flow Control

The first byte in a downlink buffer is the network interface command. If the command is niCOMM (a command to the PCLTA to send a message to the network) or niNETMGMT (a command to the NSS-10 module to execute a local network management function), then the PCLTA requires an application output buffer to contain the rest of the message. There are two pools of application output buffers, priority and non-priority. The availability of buffers in these two pools is indicated by the DLBA and DLPBA (respectively) bits in the Status register. The host must not send any niCOMM or niNETMGMT downlink network interface commands unless an appropriate buffer is available. The PCLTA may be configured to generate an interrupt on the host when buffers become available. This is controlled by the DLBA and DLPBA bits in the Interrupt Request Control register. In this way, the host is relieved of the need to poll the PCLTA to determine whether an application output buffer is available. If the network interface command is any of the immediate commands, it may be executed without waiting for an application output buffer. See Appendix B of the *PCLTA User's Guide* for more details on network interface commands.

Uplink Flow Control

The PCLTA indicates that it wishes to communicate an uplink message by asserting ULREADY in the Status register. The PCLTA may be configured to generate an interrupt on the host when an uplink message is pending. This is controlled by the ULREADY bit in the Interrupt Request Control register. In this way, the host is relieved of the need to poll the PCLTA to determine whether an uplink message is pending. Note that ULREADY is asserted for uplink local network interface commands, as well as niCOMM and niNETMGMT commands. If the host is temporarily unable to process uplink traffic, it simply avoids writing the CMD_ACK byte to acknowledge the ULREADY state. In this case, the PCLTA will buffer incoming messages in its application input buffers until they are exhausted. When this occurs, any further incoming messages are discarded until an application input buffer is read by the host and made available again to the PCLTA.

Determinism in Industrial ComputerControl Network Applications

January 1995 Echelon White Paper

Introduction

A control network is a system of sensors, actuators, displays, and logging devices (referred to as "nodes") that are linked together to monitor and control electrical devices. Supervisory functions are typically handled automatically and require no manual intervention except to respond to faults that the system cannot itself correct. In industrial applications, a control network may monitor equipment fault conditions, adjust speed and flow rates, monitor analog inputs, sequentially turn equipment on or off, interface with host processors, and display network status on a computer or custom status display.

The list of criteria that are typically considered essential in an industrial control network include:

- Predictable response times
- Inexpensive node cost
- Compatibility with existing systems
- Control architecture
- Connectionless data transfer
- Presetable environments
- Fault tolerance
- Network-wide synchronization
- Predictable network latency
- Multiple media in the network
- Peer-to-peer architecture
- Inexpensive development cost
- Open architecture
- Application Programming Interface (API)
- Connection-based topology
- Object orientation
- Efficiency
- Multicast data channels
- Consistency with the OSI model
- Open availability and support

The first item on the list, predictable response times, stands out because it is often confused with deterministic behavior. An industrial control network needs predictable response time, especially during periods of network overload. Deterministic behavior actually interferes with the ability of an industrial computer control network to operate with predictable response times, though the reasoning behind this statement may not be obvious.

This paper will deal with the importance of response time, and likewise the unimportance of determinism, as an element in an industrial control network. We will begin with a review of determinism, and then review different protocols to assess their level of determinism. We will then review what factors are critical to the operation of a control network under conditions of overload,

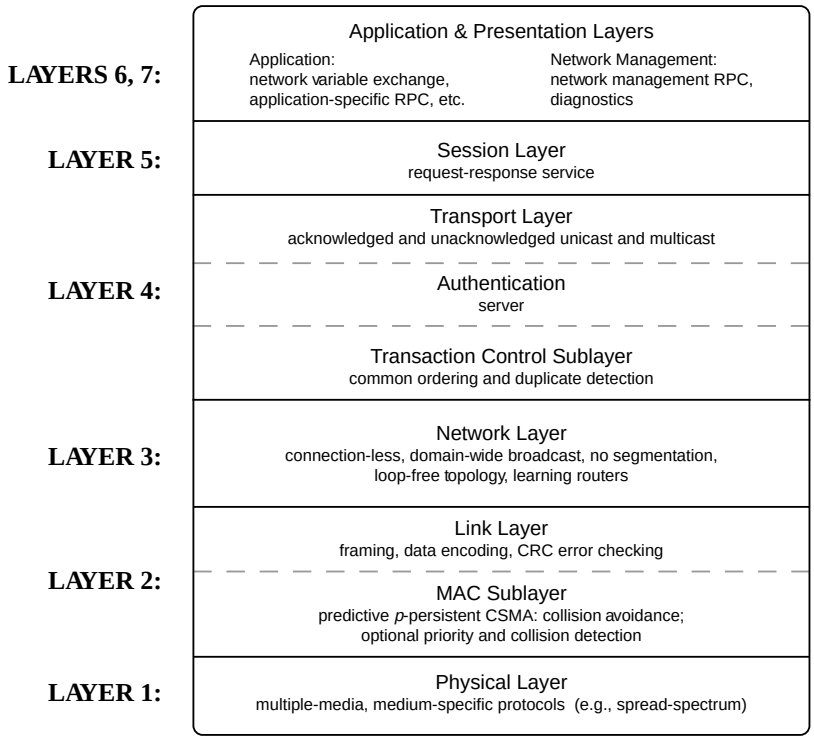
Echelon Part Number 005-0060-01 REV A

and examine why many factors other than determinism are necessary to ensure reliable performance under these conditions.

Determinism

By determinism, it is usually meant that access to the control network by a node may be delayed by at most some time t, where t is known. Every node thus has fair and equal access to the network since no one node will be delayed from gaining access for longer than time t. Using the OSI reference model shown in table 1 as a framework, a node's ability to access the network is a function of the Media Access Protocol (MAC), which is a sublayer of the Link layer known as OSI layer 2.

Table 1 OSI Reference Model



Proponents of determinism claim that knowing the value of t makes it much easier to design control networks, and that networked industrial control systems should only use deterministic protocols. The proponents also claim that the carrier sense multiple access (CSMA) family of protocols is non-deterministic and therefore not suitable for industrial computer control applications. In reality, what happens above the MAC sublayer and at the application determines whether a control network will function or not. As an investigation of protocols will reveal, deterministic protocols actually undermine the performance of an

industrial control network under overload conditions, while properly implemented CSMA protocols are in fact ideal for industrial control applications.

What Is A Deterministic Protocol?

There are four basic families of media access protocols: time division multiplexing (TDM), token bus, token ring, and CSMA. Of these four, all but CSMA are commonly considered deterministic.

TDM Media Access

TDM protocols assign a unique time for each node relative to a clock pulse (commonly called "start of frame") on the medium. Each node counts down to its time and transmits typically a single byte per frame. As long as all nodes have a unique time to transmit, and all nodes receive the same start of frame signal at the same time, the delay to access the network is bounded by the number of time slots assigned. For example, if there are 32 nodes and 32 slots each 1 byte wide, then each node may transmit a single byte for each 32 byte times on the medium, less the overhead for the start of frame signal.

The principle weaknesses of TDM protocols are that they waste bandwidth if all the nodes do not always have something to send, and a loss of synchronization to the start of frame signal results in the loss of all communications - a single point of failure condition. Furthermore, high levels of input/output activity at a node may cause more message traffic to be generated than can be handled by the assigned time slot. Under conditions of high traffic, system responses to input and output can vary considerably in time.

Token Passing Protocols

Token bus and token ring protocols, typified by ARCnet™ and IEEE 802.5, both pass a special message called the "token" which, upon receipt, grants the right to transmit on the medium. Each node may only hold the token for a limited time before passing it on to the next node in the network. In this way, access to the network is bounded by the maximum latency of the token as it is passed to each of the nodes. In a token ring network, only the next station on the wire receives the token. In a token bus network, all stations receive the token and then must decide if it is intended for them.

Relying on topology to route the token message is an important advantage for a token ring because it eliminates the need for addressing information in the token message. Even more important, eliminating the need for addressing information also eliminates the need for each station on the network to know

the address of the next station on the ring. This means that stations may enter and leave the ring without any configuration of the existing stations on the ring. On the other hand, reliance on topology to route the token precludes the use of communications media for which there can be no topological control, e.g., radio frequency and power line.

A weaknesses common to both of these token-based protocols is that the token may be lost due to error and then may be difficult to recover quickly. Most protocols use a random (non-deterministic) method to recover tokens; ARCnet is a typical example.

Errors may also result in the generation of multiple tokens. The presence of multiple tokens results in the loss of all tokens, and cause the network to initiate a token recovery process.

Knowing which node is the "next" node to pass the token is also a problem in token bus systems. If two nodes have the same address - each believes that it is the "next" node - then they will both acknowledge the token and cause a temporary loss of communications. This communication outage will repeat indefinitely and at regular intervals until the error is corrected.

One final problem occurs when nodes enter or leave the token bus. When this occurs the entire bus must automatically reconfigure itself in order to determine the sequence of node addresses on the bus and allow orderly token passing to resume. The time required for reconfiguration is proportional to the number of nodes on the bus, so if the failure mode is one where a node comes and goes due to, say, a watchdog timer reset, then the bus will effectively go down due to continuous reconfigurations.

CSMA-based Protocols

CSMA is a listen-before-transmit scheme in which a node with a message to transmit first listens to the network. If no message traffic is detected, evidenced by the absence of a carrier signal, then the node will transmit. Unlike the other media access protocols, the CSMA protocol family has many variants. The best known form of the CSMA protocol is Ethernet, also known IEEE 802.3. Ethernet was not designed for control systems and exhibits very poor characteristics near network overload; for this reason it is not often used for control systems. The limitations of Ethernet have created the impression that CSMA protocols are not suitable for computer control applications, though at least one variation of CSMA is ideally suited for such an application.

CSMA is fully deterministic when used in master-slave operation, however, it cannot be deterministic in peer-to-peer operation because nodes are not provided

with equal access to the network. Nodes transmit on the basis of their ability to resolve packet collisions, and sometimes on the basis of priority messaging as well, and it is precisely these features which make a properly implemented, non-deterministic CSMA protocol so well suited for control applications.

Since CSMA was invented, there has been a great deal of research focused on modifying the initial protocol to make it perform better near network saturation. These efforts have been successful, as exemplified by non-persistent CSMA protocols (a node waits for a random period of time before checking if a busy channel is free to transmit) and p-persistent CSMA protocols (a node uses probability calculations to determine when and when not to transmit on slotted channels). For example, Echelon's LonTalk® media access protocol uses a predictive p-persistent CSMA protocol - a variant of the p-persistent CSMA protocol - to dynamically adjust the number of packet time slots based on predicted network traffic. By dynamically allocating network bandwidth, the predictive p-persistent CSMA protocol permits the network to continue operating in the presence of very high levels of network traffic without slowing the network during periods of light traffic. The benefits of this technology are its high efficiency, low overhead, low cost hardware, elimination of the need for network wide synchronization, and lack of loss-prone tokens.

CSMA protocols have been criticized because it is theoretically possible that a node could be prohibited from successful media access for an unbounded time due to unresolved collisions. This theoretical result fails the test of determinism because colliding nodes may experience considerable delays accessing the network. If collisions could be resolved with CSMA protocols, so the argument goes, then 100% network utilization could be achieved. Transceivers have in fact been designed for the predictive p-persistent CSMA protocol which resolve collisions; Echelon's power line carrier transceivers are examples of such designs, and since 1992 have been widely used in thousands of industrial control, building management, and home automation applications that require collision resolution. When the predictive p-persistent CSMA protocol is coupled with a transceiver that implements collision resolution, the protocol operates with no packet loses due to collisions. Such a node will achieve successful media access since it will resolve collisions. The predictive p-persistent CSMA protocol has proven its robustness in field applications, and since 1992, Motorola and Toshiba have sold several hundred thousand Neuron® Chips embedded with Echelon's predictive p-persistent CSMA protocol. These firms are currently designing third generations of this family of chips using 0.6 micron geometry.

The predictive p-persistent CSMA protocol also overcomes the issue of unsuccessful media access by employing transceiver designs that limit the number of stations on a single network segment. In addition, each node is limited to a single outgoing transaction at a time; transmitters stop and wait for

an acknowledgment prior to accessing the communications medium again. These two implementation details overcome a key limitation of other CSMA protocols by making it impossible for a working station to be denied access to the communications medium indefinitely.

The predictive p-persistent CSMA protocol is not the only non-deterministic CSMA protocol to offer robust performance. Various automotive protocols, such as Chrysler's Carlink™, use a special encoding scheme to eliminate collisions by resolving them in favor of a single message getting through. As a packet is transmitted, arbitration for access to the media is computed for each bit transmitted. The encoding scheme permits stations to monitor the line during the transmissions of bits corresponding to logic level '1' and not during transmission of logic level '0,' or vise versa. In this way, access to the network is arbitrated a bit at a time with the zero bits dominating the one bits. Thus stations transmitting a 1 bit when another is transmitting a 0 see the other's transmission and stop their own transmission. It should be noted that this protocol does not include fair access. The MAC delay for a packet which has a bit pattern which always loses in the arbitration is unbounded, and this problem must be handled at higher levels in the system.

Network Overload

The proponents of deterministic protocols claim that having determinism makes it much easier to design a networked control system. This assertion requires closer scrutiny.

When a network is lightly loaded (and this should be its condition in normal operation), response times will be good with both deterministic and non-deterministic systems. The reason that light loading should be the condition for normal operation is because the traffic is usually not uniformly distributed over time. Instead it arrives in bursts, and these bursts should not exceed the capacity of a network for very long.

When a network is heavily loaded, as is often the case during periods of emergency or error, response times will be poor with both deterministic and non-deterministic systems. This is because the offered traffic exceeds the bandwidth of the network and messages queue within nodes while awaiting access to the medium. If the overload persists, the nodes may run out of buffer memory to queue additional messages. This will either stop or reset the application in the node, causing further delays. The property of determinism only exacerbates delays during overloads since determinism includes "fair and equal access" to the network as its central feature. Allowing equal access to both critical and non-critical packets is simply not appropriate during emergencies.

When a fluid hose breaks and spills caustic acid, or a motor overheats and starts to smoke, or dynamic equalization must be performed to avoid pressure imbalances, equal access interferes with the real task at hand - namely to send high priority packets to correct the problem ASAP. At these times, offering unbounded delays for non-priority messages actually conserves network bandwidth for critical functions. Thus a network which supports prioritized access and is therefore not deterministic is a better choice for control applications, provided that the non-deterministic protocol also offers high network utilization.

All token passing schemes offer linearly increasing network delays up to a saturation point, but most CSMA protocols do not. The predictive p-persistent CSMA protocol does offer this feature, and represents a distinct improvement over, for example, Ethernet's 1-persistent CSMA approach.

In the case of overload conditions, there are three features which are essential if a protocol is to be used for a control system:

1. Graceful degradation: some number of messages must get through regardless of the offered traffic load. During an overload condition, neither determinism or CSMA will make the system work because the network will not be fast enough to carry all of the messages, however, the system must respond in a controlled manner to such a condition without failing catastrophically;

2. Priority: not all messages will get through in time during overload so it is essential that emergency messages have priority over other messages. Priority messaging works better than token passing in situations of overload because, unlike token passing protocols, high priority messages lock-out low priority messages. This has the effect of dedicating the network bandwidth to emergency traffic and holding off messages that the customer has decided can be deferred. Token passing schemes do not have this important characteristic and allow equal network access to all messages;

3. End-to-end acknowledgment: it is vital for an application either to know that a message got to its destination or that it did not get there within the real time requirements of the system.

Note that determinism is not a necessary requirement. During periods of overload (and graceful degradation), all of the system's real time requirements may not be met. The lack of acknowledgment is a way to detect this failure and to initiate the priority messages to save the process under control.

TDM and token protocols lack these features, even though some of these protocols claim deterministic behavior. The predictive p-persistent CSMA protocol has all three of these features even without a collision resolving communications transceiver. The reason for this situation is simple: the predictive p-persistent CSMA protocol was designed specifically for control network applications, while the other protocols emerged from data transmission network applications. Since control applications must continue operating reliably and/or degrade gracefully during overload conditions, the predictive p-persistent CSMA protocol was tailored to such a mode of operation. Data networks, on the other hand, are optimized for hauling large data files, and do not have the same response time or overload requirements.

One might question whether a predictive p-persistent CSMA protocol can provide well defined network timing given that the number of timing slots vary dynamically. Using the end-to-end acknowledgment service within a predictive p-persistent CSMA protocol (as typified by the LonTalk protocol) allows the application to know whether an operation has succeeded or not within a bounded amount of time. The protocol tracks elapsed time from the point that an application requests that a packet be sent, rather than when the packet is actually sent on the communications medium. For example, suppose that a packet must be sent and acted upon within 50 milliseconds, and if it is not acted upon in that time, the sender of the packet must take immediate action. Such a packet could be sent using acknowledged service with a retry count of 2 and a retry interval of 16 milliseconds. In this way, the application will either know that the transaction completed successfully by receiving the acknowledgment, or the application will know that the transaction failed in 48 milliseconds.

If the sample transaction described above fails, the application might then send an emergency message using the priority feature of the protocol. The priority feature uses separate buffers within each node to allow outgoing priority packets to precede non-priority packets which have already been queued for transmission. Additionally, the priority feature uses dedicated bandwidth (also referred to as "priority slots") at the end of each packet to eliminate contention for the communications medium after the transmission of a packet. Collision resolving transceivers can also be used when the channel bandwidth is limited and/or there is a need to run the network at its maximum capacity for a sustained time.

Conclusion

A control network must monitor and control electrical devices during both normal and emergency conditions. The ability of a control network to function during emergencies, including periods of high network traffic, is dependent on

the network's ability to allocate bandwidth to important messages. Should a network become saturated beyond its capability or experience a fault condition, a graceful failure mechanism should allow for fast recovery without catastrophic failure. The predictive p-persistent CSMA protocol performs well under both normal and overload/fault conditions, and provides the predictable response times needed by an industrial control network. The predictive p-persistent CSMA protocol offers significant performance advantages over TDM, token ring, and token bus protocols under similar circumstances.

While the predictive p-persistent CSMA protocol can be used in a deterministic manner, determinism is neither necessary or sufficient to ensure predictable response times under overload or fault conditions. Yet, these are precisely the circumstances in which an industrial computer control network needs to operate predictably; a broken hose or CPU failure during a batch run - where revenue is at stake - is when a user relies most heavily on the robustness of the control network. The predictive p-persistent CSMA protocol works robustly at the edge of the network's capability, a critical point at which deterministic behavior matters not at all.

Glossary GL

GLOSSARY

A

ABS	average busy stream (a built-in function code)
ALU	arithmetic logic unit
API	Application Programming Interface

B

BIST	built-in self test
BCD	binary coded decimal
BP	Base Page

C

CDet	collision detect
CPU	central processing unit
CRC	cyclic redundancy check
CSMA	carrier sense multiple access

E

EMI	electromagnetic interference
EOM	end of message
ESD	electrostatic discharge

H

HS	handshake

I

I2C	Inter-Integrated Circuit (Philips trademark)
IR	Infrared

L

LRC	Longitudinal Redundancy Check
LSB	least significant bit/byte
LVI	low-voltage inhibit

M

MAC	media access control
MCU	microcontroller unit
MIP	Microprocessor Interface Program
MPU	microprocessor unit
MSB	most significant bit

N

ND	network diagnostic

NEC	National Electric Code
NM	network management
NPO	non-polarized

P

PAL	programmable array logic
PCB	printed circuit board
PCLTA	PC LonTalk Adapter
PTC	positive temperature coefficient

R

RF	radio frequency
RFI	radio frequency interference

S

SCL	serial clock
SCR	silicon-controlled rectifier
SDA	serial data
SIDAC	trademark of Teccor Corp.
SLTA/2	Serial LonTalk Adapter
SMT	surface mount technology
SNVT	Standard Network Variable Type

T

TOS	top of stack

V

VLSI	very large-scale integration

X

XIF	external interface data

Index IND

G

I

L

M

N

O

P

Q

R

S

T

U

W

MOTOROLA AUTHORIZED DISTRIBUTOR & WORLDWIDE SALES OFFICES

NORTH AMERICAN DISTRIBUTORS

UNITED STATES

ALABAMA

Huntsville
- Allied Electronics, Inc. ..... (205)721–3500
- Arrow Electronics ..... (205)837–6955
- FAI ..... (205)837–9209
- Future Electronics ..... (205)830–2322
- Hamilton/Hallmark ..... (205)837–8700
- Newark ..... (205)837–9091
- Wyle Electronics ..... (205)830–1119

Mobile
- Allied Electronics, Inc. ..... (334)476–1875

ARIZONA

Phoenix
- Allied Electronics, Inc. ..... (602)831–2002
- FAI ..... (602)731–4661
- Future Electronics ..... (602)968–7140
- Hamilton/Hallmark ..... (602)736–7000
- Wyle Electronics ..... (602)804–7000

Tempe
- Arrow Electronics ..... (602)966–6600
- Newark ..... (602)966–6340
- PENSTOCK ..... (602)967–1620

ARKANSAS

Little Rock
- Newark ..... (501)225–8130

CALIFORNIA

Agoura Hills
- Future Electronics ..... (818)865–0040

Calabassas
- Arrow Electronics ..... (818)880–9686
- Wyle Electronics ..... (818)880–9000

Culver City
- Hamilton/Hallmark ..... (310)558–2000

Irvine
- Arrow Electronics ..... (714)587–0404
- Arrow Zeus ..... (714)581–4622
- FAI ..... (714)753–4778
- Future Electronics ..... (714)453–1515
- Hamilton/Hallmark ..... (714)789–4100
- Wyle Laboratories Corporate .. (714)753–9953
- Wyle Electronics ..... (714)789–9953

Los Angeles
- FAI ..... (818)879–1234

Manhattan Beach
- PENSTOCK ..... (310)546–8953

Newberry Park
- PENSTOCK ..... (805)375–6680

Orange County
- Allied Electronics, Inc. ..... (714)727–3010

Palo Alto
- Newark ..... (650)812–6300

Rancho Cordova
- Wyle Electronics ..... (916)638–5282

Riverside
- Allied Electronics, Inc. ..... (909)980–6522
- Newark ..... (909)980–2105

Rocklin
- Hamilton/Hallmark ..... (916)632–4500

Roseville
- Wyle Electronics ..... (916)783–9953

Sacramento
- Allied Electronics, Inc. ..... (916)632–3104
- FAI ..... (916)782–7882
- Newark ..... (916)565–1760

San Diego
- Allied Electronics, Inc. ..... (619)279–2550
- Arrow Electronics ..... (619)565–4800
- FAI ..... (619)623–2888
- Future Electronics ..... (619)625–2800
- Hamilton/Hallmark ..... (619)571–7540
- Newark ..... (619)453–8211
- PENSTOCK ..... (619)623–9100
- Wyle Electronics ..... (619)558–6600

San Fernando Valley
- Allied Electronics, Inc. ..... (818)598–0130

CALIFORNIA – continued

San Jose
- Allied Electronics, Inc. ..... (408)383–0366
- Arrow Electronics ..... (408)441–9700
- Arrow Electronics ..... (408)428–6400
- Arrow Zeus ..... (408)629–4789
- FAI ..... (408)434–0369
- Future Electronics ..... (408)434–1122

Santa Clara
- Wyle Electronics ..... (408)727–2500

Santa Fe Springs
- Newark ..... (562)929–9722

Sierra Madre
- PENSTOCK ..... (818)355–6775

Sunnyvale
- Hamilton/Hallmark ..... (408)435–3600
- PENSTOCK ..... (408)730–0300

Thousand Oaks
- Newark ..... (805)449–1480

Woodland Hills
- Hamilton/Hallmark ..... (818)594–0404

COLORADO

Lakewood
- FAI ..... (303)237–1400
- Future Electronics ..... (303)232–2008

Denver
- Allied Electronics, Inc. ..... (303)790–1664
- Newark ..... (303)373–4540

Englewood
- Arrow Electronics ..... (303)799–0258
- Hamilton/Hallmark ..... (303)790–1662
- PENSTOCK ..... (303)799–7845

Thornton
- Wyle Electronics ..... (303)457–9953

CONNECTICUT

Bloomfield
- Newark ..... (860)243–1731

Cheshire
- Allied Electronics, Inc. ..... (203)272–7730
- FAI ..... (203)250–1319
- Future Electronics ..... (203)250–0083
- Hamilton/Hallmark ..... (203)271–5700

Wallingford
- Arrow Electronics ..... (203)265–7741
- Wyle Electronics ..... (203)269–8077

FLORIDA

Altamonte Springs
- Future Electronics ..... (407)865–7900

Clearwater
- FAI ..... (813)530–1665
- Future Electronics ..... (813)530–1222

Deerfield Beach
- Arrow Electronics ..... (305)429–8200
- Wyle Electronics ..... (954)420–0500

Ft. Lauderdale
- FAI ..... (954)428–9494
- Future Electronics ..... (954)426–4043
- Hamilton/Hallmark ..... (954)677–3500
- Newark ..... (954)486–1151

Jacksonville
- Allied Electronics, Inc. ..... (904)739–5920
- Newark ..... (904)399–5041

Lake Mary
- Arrow Electronics ..... (407)333–9300
- Arrow Zeus ..... (407)333–3055

Largo/Tampa/St. Petersburg
- Hamilton/Hallmark ..... (813)507–5000
- Newark ..... (813)287–1578
- Wyle Electronics ..... (813)576–3004

Miami
- Allied Electronics, Inc. ..... (305)558–2511

Maitland
- Wyle Electronics ..... (407)740–7450

Orlando
- Allied Electronics, Inc. ..... (407)539–0055
- FAI ..... (407)865–9555
- Newark ..... (407)896–8350

FLORIDA – continued

Tallahassee
- FAI ..... (904)668–7772

Tampa
- Allied Electronics, Inc. ..... (813)579–4660
- Newark ..... (813)287–1578
- PENSTOCK ..... (813)247–7556

Winter Park
- Hamilton/Hallmark ..... (407)657–3300
- PENSTOCK ..... (407)672–1114

GEORGIA

Atlanta
- Allied Electronics, Inc. ..... (770)497–9544
- FAI ..... (404)447–4767

Duluth
- Arrow Electronics ..... (404)497–1300
- Hamilton/Hallmark ..... (770)623–4400

Norcross
- Future Electronics ..... (770)441–7676
- Newark ..... (770)448–1300
- PENSTOCK ..... (770)734–9990
- Wyle Electronics ..... (770)441–9045

IDAHO

Boise
- Allied Electronics, Inc. ..... (208)331–1414
- FAI ..... (208)376–8080

ILLINOIS

Addison
- Wyle Laboratories ..... (708)620–0969

Arlington Heights
- Hamilton/Hallmark ..... (847)797–7300

Chicago
- Allied Electronics, Inc. (North) .. (847)548–9330
- Allied Electronics, Inc. (South) .. (708)535–0038
- FAI ..... (708)843–0034
- Newark Electronics Corp. ..... (773)784–5100

Hoffman Estates
- Future Electronics ..... (708)882–1255

Itasca
- Arrow Electronics ..... (708)250–0500
- Arrow Zeus ..... (630)595–9730

Lombard
- Newark ..... (630)317–1000

Palatine
- PENSTOCK ..... (708)934–3700

Rockford
- Allied Electronics, Inc. ..... (815)636–1010
- Newark ..... (815)229–0225

Springfield
- Newark ..... (217)787–9972

Wood Dale
- Allied Electronics, Inc. ..... (630)860–0007

INDIANA

Indianapolis
- Allied Electronics, Inc. ..... (317)571–1880
- Arrow Electronics ..... (317)299–2071
- Hamilton/Hallmark ..... (317)575–3500
- FAI ..... (317)469–0441
- Future Electronics ..... (317)469–0447
- Newark ..... (317)844–0047
- Wyle Electronics ..... (317)581–6152

Ft. Wayne
- Newark ..... (219)484–0766
- PENSTOCK ..... (219)432–1277

IOWA

Bettendorf
- Newark ..... (319)359–3711

Cedar Rapids
- Allied Electronics, Inc. ..... (319)390–5730
- Newark ..... (319)393–3800

KANSAS

Kansas City
- Allied Electronics, Inc. ..... (913)338–4372
- FAI ..... (913)381–6800

Lenexa
- Arrow Electronics ..... (913)541–9542

For changes to this information contact Technical Publications at FAX (602) 244-6560

AUTHORIZED DISTRIBUTORS – continued

UNITED STATES – continued

KANSAS – continued

Olathe
- PENSTOCK ..... (913)829–9330

Overland Park
- Future Electronics ..... (913)649–1531
- Hamilton/Hallmark ..... (913)663–7900
- Newark ..... (913)677–0727

KENTUCKY

Louisville
- Allied Electronics, Inc. ..... (502)452–2293
- Newark ..... (502)423–0280

LOUISIANA

New Orleans
- Allied Electronics, Inc. ..... (504)466–7575
- Newark ..... (504)838–9771

MARYLAND

Baltimore
- Allied Electronics, Inc. ..... (410)312–0810
- FAI ..... (410)312–0833

Columbia
- Arrow Electronics ..... (301)596–7800
- Arrow Zeus ..... (410)309–1541
- Future Electronics ..... (410)290–0600
- Hamilton/Hallmark ..... (410)720–3400
- PENSTOCK ..... (410)290–3746
- Wyle Electronics ..... (410)312–4844

Hanover
- Newark ..... (410)712–6922

MASSACHUSETTS

Bedford
- Wyle Electronics ..... (781)271–9953

Boston
- Allied Electronics, Inc. ..... (617)255–0361
- Arrow Electronics ..... (508)658–0900
- FAI ..... (508)779–3111
- Newark 1–800–4NEWARK

Bolton
- Future Corporate ..... (978)779–3000

Burlington
- PENSTOCK ..... (617)229–9100

Lowell
- Newark ..... (978)551–4300

Peabody
- Allied Electronics, Inc. ..... (508)538–2401
- Hamilton/Hallmark ..... (508)532–3701

Wilmington
- Arrow Zeus ..... (978)658–4776

Worchester
- Newark ..... (508)229–2200

MICHIGAN

Detroit
- Allied Electronics, Inc. ..... (313)416–9300
- FAI ..... (313)513–0015
- Future Electronics ..... (616)698–6800

Grand Rapids
- Allied Electronics, Inc. ..... (616)365–9960
- Newark ..... (616)954–6700

Livonia
- Arrow Electronics ..... (810)455–0850
- Future Electronics ..... (313)261–5270
- Hamilton/Hallmark ..... (313)416–5800

Novi
- Wyle Electronics ..... (248)374–9953

Saginaw
- Newark ..... (517)799–0480

Troy
- Newark ..... (248)583–2899

MINNESOTA

Bloomington
- Wyle Electronics ..... (612)853–2280

Burnsville
- PENSTOCK ..... (612)882–7630

Eden Prairie
- Arrow Electronics ..... (612)941–5280
- FAI ..... (612)947–0909
- Future Electronics ..... (612)944–2200
- Hamilton/Hallmark ..... (612)881–2600

MINNESOTA – continued

Minneapolis
- Allied Electronics, Inc. ..... (612)938–5633
- Newark ..... (612)331–6350

MISSISSIPPI

Jackson
- Newark ..... (601)956–3834

MISSOURI

Earth City
- Hamilton/Hallmark ..... (314)770–6300

St. Louis
- Allied Electronics, Inc. ..... (314)240–9405
- Arrow Electronics ..... (314)567–6888
- Future Electronics ..... (314)469–6805
- FAI ..... (314)542–9922
- Newark ..... (314)991–0400

NEBRASKA

Omaha
- Allied Electronics, Inc. ..... (402)697–0038
- Newark ..... (402)592–2423

NEVADA

Las Vegas
- Allied Electronics, Inc. ..... (702)258–1087
- Wyle Electronics ..... (702)765–7117

NEW JERSEY

Bridgewater
- PENSTOCK ..... (908)575–9490

East Brunswick
- Allied Electronics, Inc. ..... (908)613–0828
- Newark ..... (732)937–6600

Fairfield
- FAI ..... (201)331–1133

Marlton
- Arrow Electronics ..... (609)596–8000
- FAI ..... (609)988–1500
- Future Electronics ..... (609)596–4080

Mt. Laurel
- Hamilton/Hallmark ..... (609)222–6400
- Wyle Electronics ..... (609)439–9110

Oradell
- Wyle Electronics ..... (201)261–3200

Pinebrook
- Arrow Electronics ..... (201)227–7880
- Wyle Electronics ..... (973)882–8358

Parsippany
- Future Electronics ..... (201)299–0400
- Hamilton/Hallmark ..... (201)515–1641

NEW MEXICO

Albuquerque
- Allied Electronics, Inc. ..... (505)266–7565
- Hamilton/Hallmark ..... (505)293–5119
- Newark ..... (505)828–1878

NEW YORK

Albany
- Newark ..... (518)489–1963

Buffalo
- Newark ..... (716)631–2311

Great Neck
- Allied Electronics, Inc. ..... (516)487–5211

Hauppauge
- Allied Electronics, Inc. ..... (516)234–0485
- Arrow Electronics ..... (516)231–1000
- FAI ..... (516)348–3700
- Future Electronics ..... (516)234–4000
- Hamilton/Hallmark ..... (516)434–7400
- Newark ..... (516)567–4200
- PENSTOCK ..... (516)724–9580
- Wyle Electronics ..... (516)231–7850

Henrietta
- Wyle Electronics ..... (716)334–5970

Konkoma
- Hamilton/Hallmark ..... (516)737–0600

Pittsford
- Newark ..... (716)381–4244

Poughkeepsie
- Allied Electronics, Inc. ..... (914)452–1470
- Newark ..... (914)298–2810

Purchase
- Arrow Zeus ..... (914)701–7400

NEW YORK – continued

Rochester
- Allied Electronics, Inc. ..... (716)292–1670
- Arrow Electronics ..... (716)427–0300
- Future Electronics ..... (716)387–9550
- FAI ..... (716)387–9600
- Hamilton/Hallmark ..... (716)272–2740

Syracuse
- Allied Electronics, Inc. ..... (315)446–7411
- FAI ..... (315)451–4405
- Future Electronics ..... (315)451–2371
- Newark ..... (315)457–4873

NORTH CAROLINA

Charlotte
- Allied Electronics, Inc. ..... (704)525–0300
- FAI ..... (704)548–9503
- Future Electronics ..... (704)547–1107
- Newark ..... (704)535–5650

Greensboro
- Newark ..... (910)294–2142

Morrisville
- Wyle Electronics ..... (919)469–1502

Raleigh
- Allied Electronics, Inc. ..... (919)876–5845
- Arrow Electronics ..... (919)876–3132
- FAI ..... (919)876–0088
- Future Electronics ..... (919)790–7111
- Hamilton/Hallmark ..... (919)872–0712

OHIO

Centerville
- Arrow Electronics ..... (513)435–5563

Cincinnati
- Allied Electronics, Inc. ..... (513)771–6990
- Newark ..... (513)942–8700

Cleveland
- Allied Electronics, Inc. ..... (216)831–4900
- FAI ..... (216)446–0061
- Newark ..... (216)391–9330

Columbus
- Allied Electronics, Inc. ..... (614)785–1270
- Newark ..... (614)326–0352

Dayton
- FAI ..... (513)427–6090
- Future Electronics ..... (513)426–0090
- Hamilton/Hallmark ..... (513)439–6735
- Newark ..... (937)294–8980

Mayfield Heights
- Future Electronics ..... (216)449–6996

Miamisburg
- Wyle Electronics ..... (937)436–9953

Solon
- Arrow Electronics ..... (216)248–3990
- Hamilton/Hallmark ..... (216)498–1100
- Wyle Electronics ..... (440)248–9996

Toledo
- Newark ..... (419)866–0404

Worthington
- Hamilton/Hallmark ..... (614)888–3313

OKLAHOMA

Oklahoma City
- Newark ..... (405)943–3700

Tulsa
- Allied Electronics, Inc. ..... (918)250–4505
- FAI ..... (918)492–1500
- Hamilton/Hallmark ..... (918)459–6000

OREGON

Beaverton
- Arrow/Almac Electronics Corp. . (503)629–8090
- Future Electronics ..... (503)645–9454
- Hamilton/Hallmark ..... (503)526–6200

Portland
- Allied Electronics, Inc. ..... (503)626–9921
- FAI ..... (503)297–5020
- Newark ..... (503)297–1984
- PENSTOCK ..... (503)646–1670
- Wyle Electronics ..... (503)598–9953

AUTHORIZED DISTRIBUTORS – continued

UNITED STATES – continued

PENNSYLVANIA

Allentown
Newark . . . . . (610)434–7171

Chadds Ford
Allied Electronics, Inc. . . . . (610)388–8455

Coatesville
PENSTOCK . . . . . (610)383–9536

Ft. Washington
Newark . . . . . (215)654–1434

Harrisburg
Allied Electronics, Inc. . . . . (717)540–7101

Philadelphia
Allied Electronics, Inc. . . . . (609)234–7769

Pittsburgh
Allied Electronics, Inc. . . . . (412)931–2774
Arrow Electronics . . . . . (412)963–6807
Newark . . . . . (412)788–4790

SOUTH CAROLINA

Greenville
Allied Electronics, Inc. . . . . (864)288–8835
Newark . . . . . (864)288–9610

TENNESSEE

Knoxville
Newark . . . . . (423)588–6493

Memphis
Newark . . . . . (901)396–7970

TEXAS

Austin
Allied Electronics, Inc. . . . . (512)219–7171
Arrow Electronics . . . . . (512)835–4180
Future Electronics . . . . . (512)502–0991
FAI . . . . . (512)346–6426
Hamilton/Hallmark . . . . . (512)219–3700
Newark . . . . . (512)338–0287
PENSTOCK . . . . . (512)346–9762
Wyle Electronics . . . . . (512)833–9953

Benbrook
PENSTOCK . . . . . (817)249–0442

Brownsville
Allied Electronics, Inc. . . . . (210)548–1129

Carrollton
Arrow Electronics . . . . . (972)380–6464
Arrow Zeus . . . . . (972)380–4330

Dallas
Allied Electronics, Inc. . . . . (214)341–8444
FAI . . . . . (972)231–7195
Future Electronics . . . . . (972)437–2437
Hamilton/Hallmark . . . . . (214)553–4300
Newark . . . . . (972)458–2528

El Paso
Allied Electronics, Inc. . . . . (915)779–6294
FAI . . . . . (915)577–9531
Newark . . . . . (915)772–6367

Ft. Worth
Allied Electronics, Inc. . . . . (817)595–3500

Houston
Allied Electronics, Inc. . . . . (281)446–8005
Arrow Electronics . . . . . (281)647–6868
FAI . . . . . (713)952–7088
Future Electronics . . . . . (713)785–1155
Hamilton/Hallmark . . . . . (713)781–6100
Newark . . . . . (281)894–9334
Wyle Electronics . . . . . (713)784–9953

Richardson
PENSTOCK . . . . . (972)479–9215
Wyle Electronics . . . . . (972)235–9953

San Antonio
FAI . . . . . (210)738–3330

UTAH

Draper
Wyle Electronics . . . . . (801)523–2335

Salt Lake City
Allied Electronics, Inc. . . . . (801)261–5244
Arrow Electronics . . . . . (801)973–6913
FAI . . . . . (801)467–9696
Future Electronics . . . . . (801)467–4448
Hamilton/Hallmark . . . . . (801)266–2022
Newark . . . . . (801)261–5660

West Valley City
Wyle Electronics . . . . . (801)974–9953

VIRGINIA

Herndon
Newark . . . . . (703)707–9010

Richmond
Newark . . . . . (804)282–5671

Springfield
Allied Electronics, Inc. . . . . (703)644–9515

Virginia Beach
Allied Electronics, Inc. . . . . (757)363–8662

WASHINGTON

Bellevue
Almac Electronics Corp. . . . . (206)643–9992
PENSTOCK . . . . . (206)454–2371

Bothell
Future Electronics . . . . . (206)489–3400

Kirkland
Newark . . . . . (425)814–6230

Redmond
Hamilton/Hallmark . . . . . (206)882–7000
Wyle Electronics . . . . . (425)881–1150

Seattle
Allied Electronics, Inc. . . . . (206)251–0240
FAI . . . . . (206)485–6616

Spokane
Newark . . . . . (509)327–1935

WISCONSIN

Brookfield
Arrow Electronics . . . . . (414)792–0150
Future Electronics . . . . . (414)879–0244
Wyle Electronics . . . . . (414)879–0434

Madison
Newark . . . . . (608)278–0177

Milwaukee
Allied Electronics, Inc. . . . . (414)796–1280
FAI . . . . . (414)792–9778

New Berlin
Hamilton/Hallmark . . . . . (414)780–7200

Wauwatosa
Newark . . . . . (414)453–9100

CANADA

ALBERTA

Calgary
FAI . . . . . (403)291–5333
Future Electronics . . . . . (403)250–5550
Hamilton/Hallmark . . . . . (800)663–5500
Newark . . . . . (800)463–9275

Edmonton
FAI . . . . . (403)438–5888
Future Electronics . . . . . (403)438–2858
Hamilton/Hallmark . . . . . (800)663–5500
Newark . . . . . (800)463–9275

Saskatchewan
Hamilton/Hallmark . . . . . (800)663–5500

BRITISH COLUMBIA

Vancouver
Allied Electronics, Inc. . . . . (604)420–9691
Arrow Electronics . . . . . (604)421–2333
FAI . . . . . (604)654–1050
Future Electronics . . . . . (604)294–1166
Hamilton/Hallmark . . . . . (604)420–4101
Newark . . . . . (800)463–9275

MANITOBA

Winnipeg
FAI . . . . . (204)786–3075
Future Electronics . . . . . (204)944–1446
Hamilton/Hallmark . . . . . (800)663–5500
Newark . . . . . (800)463–9275

ONTARIO

Kanata
PENSTOCK . . . . . (613)592–6088

London
Newark . . . . . (519)685–4280

Mississauga
PENSTOCK . . . . . (905)403–0724
Newark . . . . . (905)670–2888

Ottawa
Allied Electronics, Inc. . . . . (613)228–1964
Arrow Electronics . . . . . (613)226–6903
FAI . . . . . (613)820–8244
Future Electronics . . . . . (613)727–1800
Hamilton/Hallmark . . . . . (613)226–1700

Toronto
Arrow Electronics . . . . . (905)670–7769
FAI . . . . . (905)612–9888
Future Electronics . . . . . (905)612–9200
Hamilton/Hallmark . . . . . (905)564–6060
Newark . . . . . (905)670–2888

QUEBEC

Montreal
Arrow Electronics . . . . . (514)421–7411
FAI . . . . . (514)694–8157
Future Electronics . . . . . (514)694–7710
Hamilton/Hallmark . . . . . (514)335–1000

Mt. Royal
Newark . . . . . (514)738–4488

Quebec City
Arrow Electronics . . . . . (418)687–4231
FAI . . . . . (418)682–5775
Future Electronics . . . . . (418)877–6666

INTERNATIONAL DISTRIBUTORS

ARGENTINA
- Electrocomponentes ..... (5–41) 375–3366
- Elko ..... (5–41) 372–1101

AUSTRALIA
- Avnet VSI Electronics (Aust.) ..... (61)2 9878–1299
- Farnell ..... (61)2 9645–8888
- Veltek Australia Pty. Ltd. .... (61)3 9574–9300

AUSTRIA
- EBV Elektronik ..... (43) 189152–0
- Farnell ..... (49) 8961 393939
- SEI/Elbatex GmbH ..... (43) 1 866420
- Spoerle Electronic ..... (43) 1 360460

BELGIUM
- EBV Elektronik ..... (32) 2 716 0010
- Farnell ..... (32) 3 227 3647
- SEI/Belgium ..... (32) 2 460 0747
- Spoerle Electronic ..... (32) 2 725 4660

BRAZIL
- Farnell ..... (5511) 445–7400
- Future ..... (019) 235–1511
- Intertek ..... (011) 266–2922
- Karimex ..... (011) 524–2366
- Masktrade ..... (011) 3361–2766
- Panamericana ..... (011) 223–0222
- Siletek ..... (011) 536–4401
- Tec ..... (011) 5505–2046
- Teleradio ..... (011) 574–0788

BULGARIA
- Macro Group ..... (359) 2708140

CHINA
- Arrow Asia/Pac Ltd ..... (852)2 484–2113
- Avnet WKK Components Ltd. ..... (852)2 357–8888
- China El. App. Corp. Beijing .... (86)10 6828–9951
- Future Advanced Electronics Ltd. .. (852)2 305–3633
- Nanco Electronics Supply Ltd. . (852)2 765–3025
- Qing Cheng Enterprises Ltd. .. (852)2 493–4202

CZECH REPUBLIC
- EBV Elektronik ..... (420) 2 90022101
- Spoerle Electronic ..... (420) 2 71737173
- SEI/Elbatex ..... (420) 2 4763707
- Macro Group ..... (420) 2 3412182

DENMARK
- Arrow Denmark A/S ..... (45) 44 508200
- A/S Avnet EMG ..... (45) 44 880800
- EBV Elektronik – Soeborg ..... (45) 39690511
- EBV Elektronik – Abyhoj ..... (45) 86250466
- Future Electronics ..... (45) 961 00 961

ESTONIA
- Arrow Field Eesti ..... (372) 6503288
- Avnet Baltronic ..... (372) 6397000

FINLAND
- Arrow Finland ..... (358) 9 476660
- Avnet Nortek ..... (358) 9 613181
- EBV Elektronik ..... (358) 9 8557730
- Future Electronics ..... (358) 9 345 5400

FRANCE
- Arrow Electronique ..... (33) 1 49 78 49 78
- Avnet ..... (33) 1 49 65 27 00
- EBV Elektronik ..... (33) 1 40963000
- Farnell ..... (33) 474 659466
- Future Electronics ..... (33) 1 69821111
- Newark ..... (33) 1 30954060
- Sonepar Electronique ..... (33) 1 69 19 89 00

GERMANY
- Avnet EMG ..... (49) 89 4511001
- EBV Elektronik GmbH ..... (49) 89 99114–0
- Farnell ..... (49) 89 61 393939
- Future Electronics GmbH .... (49) 89–957 270
- SEI/Jermyn GmbH ..... (49) 6431–5080
- Newark ..... (49)2154–70011
- Sasco Semiconductor ..... (49) 89–46110
- Spoerle Electronic ..... (49) 6103–304–0

GREECE
- EBV Elektronik ..... (30) 13414300

HONG KONG
- Avnet WKK Components Ltd. ..... (852)2 357–8888
- Farnell ..... (65) 788–0200
- Future Advanced Electronics Ltd. .. (852)2 305–3633
- Nanco Electronics Supply Ltd. .... (852)2 333–5121
- Qing Cheng Enterprises Ltd. . (852)2 493–4202

HUNGARY
- EBV Elektronik KFT ..... (36) 1 4313 495
- Future Electronics ..... (36) 1 2240 510
- Macro Group ..... (36) 1 2030 277
- SEI/Elbatex ..... (36) 1 1409 194
- Spoerle Electronic ..... (36) 1 1294 202

INDIA
- Max India Ltd 0091 11 625–0250

INDONESIA
- P.T. Ometraco ..... (62) 21 619–6166

IRELAND
- Arrow Electronics ..... (353) 14595540
- EBV Elektronik ..... (353) 14564034
- Farnell ..... (353) 18309277
- Future Electronics ..... (353) 6541330
- Macro Group ..... (353) 16766904

ISRAEL
- Future Israel Ltd. ..... (972) 9 9586555

ITALY
- Avnet EMG ..... (39) 02 381901
- EBV Elektronik ..... (39) 02 66096290
- Future Electronics ..... (39) 02 660941
- Silverstar LTD ..... (39) 02 661251

JAPAN
- AMSC Co., Ltd. 81–422–54–6800
- Fuji Electronics Co., Ltd. 81–3–3814–1411
- Marubun Corporation 81–3–3639–8951
- OMRON Corporation 81–3–3779–9053
- Tokyo Electron Device Ltd. .. 81–45–474–7030

KOREA
- Jung Kwang Semiconductors Ltd. .. 82–2–278–5333
- Liteon Korea Ltd 82–2–650–9700
- Nasco Co. Ltd 82–2–3772–6810

LATVIA
- Avnet Baltronic Ltd. ..... (371) 8821118
- Macro Group ..... (371) 7313195

LITHUANIA
- Macro Group ..... (370) 7764937

MALAYSIA
- Farnell ..... (60) 3 773–8000
- Strong Electronics ..... (60) 4 656–3768
- Ultro Technologies Pte. Ltd. .... (65) 545–7811

MEXICO
- Avnet ..... (3) 632–0182
- Dicopel ..... (5) 705–7422
- Future ..... (3) 122–0043
- Semiconductores Profesionales ..... (5) 658–6011
- Steren ..... (5) 325–0925

NETHERLANDS
- HOLLAND
- EBV Elektronik ..... (31) 3465 83010
- Farnell ..... (31) 30 241 2323
- Future Electronics ..... (31) 76 544 4888
- SEI/Benelux B.V. ..... (31) 7657 22500
- Spoerle Electronics – Nieuwegen ..... (31) 3060 91234
- Spoerle Electronics – Veldhoven ..... (31) 4025 45430

NEW ZEALAND
- Arrow Components NZ Ltd ... (64)4 570–2260
- Avnet Pacific Ltd ..... (64)9 636–7801
- Farnell ..... (64)9 357–0646

NORWAY
- Arrow Tahonic A/S ..... (47) 2237 8440
- A/S Avnet EMG ..... (47) 6677 3600
- EBV Elektronik ..... (47) 2267 1780
- Future Electronics ..... (47) 2290 5800

PHILIPPINES
- Alexan Commercial ..... (63) 2241–9493
- Ultro Technologies Pte. Ltd ..... (65) 545–7811

POLAND
- EBV Elektronik ..... (48) 713 422944
- Future Electronics ..... (48) 22 61 89202
- Macro Group ..... (48) 22 224337
- SEI/Elbatex ..... (48) 22 6217122
- Spoerle Electronic ..... (48) 22 6465227

PORTUGAL
- Amitron Arrow ..... (35) 11471 4182
- Farnell ..... (44) 113289 0040
- SEI/Selco ..... (35) 12973 8203

ROMANIA
- Macro Group ..... (401) 6343129

RUSSIA
- EBV Elektronik ..... (7) 095 9761176
- Macro Group – Moscow ..... (7) 095 30600266
- Macro Group – St. Petersburg ..... (7) 81 25311476

SCOTLAND
- EBV Elektronik ..... (44) 141 4202070
- Future ..... (44) 141 9413999

SINGAPORE
- Farnell ..... (65) 788–0200
- Future Electronics ..... (65) 479–1300
- Strong Pte. Ltd ..... (65) 276–3996
- Uraco Technologies Pte Ltd. .... (65) 545–7811

SLOVAKIA
- Macro Group ..... (42) 89634181
- SEI/Elbatex ..... (42) 17295007

SLOVENIA
- EBV Elektronik ..... (386) 611 330216
- SEI/Elbatex ..... (386) 611 597198

S. AFRICA
- Avnet–ASD ..... (27) 11 4442333
- Reutech Components ..... (27) 11 3972992

SPAIN
- Amitron Arrow ..... (34) 91 304 3040
- EBV Elektronik ..... (34) 91 804 3256
- Farnell ..... (44) 113 231 0447
- SEI/Selco S.A. ..... (34) 1 637 10 11

SWEDEN
- Arrow–Th:s AB ..... (46) 8 56265500
- Avnet EMG AB ..... (46) 8 629 14 00
- EBV Elektronik ..... (46) 405 92100
- Farnell ..... (46) 8 730 5000
- Future Electronics ..... (46) 8 441 5470

SWITZERLAND
- EBV Elektronik ..... (41) 1 7456161
- Farnell ..... (41) 1204 6464
- SEI/Elbatex AG ..... (41) 56 4375111
- Spoerle Electronic ..... (41) 1 8746262

TAIWAN
- Avnet–Mercuries Co., Ltd ... (886)2 516–7303
- Solomon Technology Corp. .. (886)2 788–8989
- Strong Electronics Co. Ltd. .. (886)2 917–9917

THAILAND
- Sahapiphat Ltd. ..... (662) 237–9474
- Ultro Technologies Pte. Ltd. .... (65) 540–8328

TURKEY
- EBV Elektronik ..... (90) 216 4631352

UNITED KINGDOM
- Arrow Electronics (UK) Ltd .. (44) 1 234 270027
- Avnet EMG ..... (44) 1 438 788300
- EBV Elektronik ..... (44) 1 628 783688
- Farnell ..... (44) 1 132 636311
- Future Electronics Ltd. ..... (44) 1 753 763000
- Macro Group ..... (44) 1 628 606000
- Newark ..... (44) 1 420 543333

MOTOROLA WORLDWIDE SALES OFFICES

UNITED STATES

ALABAMA
Huntsville ..... (205)464–6800
ALASKA ..... (800)635–8291
ARIZONA
Phoenix ..... (602)302–8056
CALIFORNIA
Calabasas ..... (818)878–6800
Irvine ..... (714)753–7360
Los Angeles ..... (818)878–6800
San Diego ..... (619)541–2163
Sunnyvale ..... (408)749–0510
COLORADO
Denver ..... (303)337–3434
CONNECTICUT
Wallingford ..... (203)949–4100
FLORIDA
Clearwater ..... (813)524–4177
Maitland ..... (407)628–2636
Pompano Beach/Ft. Lauderdale ..... (954)351–6040
GEORGIA
Atlanta ..... (770)729–7100
IDAHO
Boise ..... (208)323–9413
ILLINOIS
Chicago/Schaumburg ..... (847)413–2500
INDIANA
Indianapolis ..... (317)571–0400
Kokomo ..... (765)455–5100
KANSAS
Kansas City/Mission ..... (913)451–8555
MARYLAND
Columbia ..... (410)381–1570
MASSACHUSETTS
Marlborough ..... (508)357–8207
Woburn ..... (781)932–9700
MICHIGAN
Detroit ..... (248)347–6800
MINNESOTA
Minnetonka ..... (612)932–1500
MISSOURI
St. Louis ..... (314)275–7380
NEW JERSEY
Fairfield ..... (973)808–2400
NEW YORK
Fairport ..... (716)425–4000
Fishkill ..... (914)896–0511
Hauppauge ..... (516)361–7000
NORTH CAROLINA
Raleigh ..... (919)870–4355
OHIO
Cleveland ..... (440)349–3100
Columbus/Worthington ..... (614)431–8492
Dayton ..... (937)438–6800
OREGON
Portland ..... (503)641–3681
PENNSYLVANIA
Colmar ..... (215)997–1020
Philadelphia/Horsham ..... (215)957–4100
TENNESSEE
Knoxville ..... (423)584–4841
TEXAS
Austin ..... (512)502–2100
Houston ..... (281)251–0006
Plano ..... (972)516–5100
WASHINGTON
Bellevue ..... (425)454–4160
Seattle (toll free) ..... (206)622–9960
WISCONSIN
Milwaukee/Brookfield ..... (414)792–0122

Field Applications Engineering Available
Through All Sales Offices

CANADA

ALBERTA
Calgary ..... (403)216–2190
BRITISH COLUMBIA
Vancouver ..... (604)606–8502
ONTARIO
Ottawa ..... (613)226–3491
Mississauga ..... (905)501–3500
QUEBEC
Montreal ..... (514)333–3300

INTERNATIONAL

AUSTRALIA
Melbourne ..... (61–3)9887 0711
Sydney ..... (61–2)9437 8944
BRAZIL
Sao Paulo ..... 55(011)3030–5244
CHINA
Beijing 86–10–65642288
Guangzhou 86–20–87537888
Shanghai 86–21–63747668
Tianjin 86–22–25325050
CZECH REPUBLIC
..... (420) 2 21852222
FINLAND
Helsinki ..... (358) 9 6866 880
Direct Sales Lines ..... (358) 9 6866 8844
..... (358) 9 6866 8845
FRANCE
Paris 33134 635900
GERMANY
Langenhagen/Hanover ..... 49(511)786880
Munich 49 89 92103–0
Nuremberg 49 911 96–3190
Sindelfingen 49 7031 79 710
Wiesbaden 49 611 973050
HONG KONG
Kwai Fong 852–2–610–6888
Tai Po 852–2–666–8333
HUNGARY
..... (36) 1 250 83 29
INDIA
Bangalore 91–80–5598615
ISRAEL
Herzlia 972–9–9522333
ITALY
Milan ..... 39(2)82201
JAPAN
Kyusyu 81–92–725–7583
Gotanda 81–3–5487–8311
Nagoya 81–52–232–3500
Osaka 81–6–305–1801
Sendai 81–22–268–4333
Takamatsu 81–878–37–9972
Tokyo 81–3–3440–3311
KOREA
Pusan ..... 82(51)4635–035
Seoul 82–2–3440–7200
MALAYSIA
Penang ..... 60(4)228–2514
MEXICO
Chihuahua ..... 52(14)39–3120
Mexico City ..... 52(5)282–0230
Guadalajara ..... 52(36)78–0750
Zapopan Jalisco ..... 52(36)78–0750
Marketing ..... 52(36)21–2023
Customer Service ..... 52(36)669–9160
NETHERLANDS
Best ..... (31)4993 612 11
PHILIPPINES
Manila ..... (63)2 807–8455
Paranaque ..... (63)2 824–4551
Salcedo Village ..... (63)2 810–0762
POLAND
..... (48) 34 27 55 75
PUERTO RICO
Rio Piedras ..... (787)282–2300
RUSSIA
..... (7) 095 929 90 25
SCOTLAND
East Kilbride ..... (44)1355 565447
SINGAPORE ..... (65)4818188
SPAIN
Madrid ..... 34(1)457–8204
or ..... 34(1)457–8254
SWEDEN
Solna ..... 46(8)734–8800
SWITZERLAND
Geneva ..... 41(22)799 11 11
Zurich ..... 41(1)730–4074
TAIWAN
Taipei ..... 886(2)717–7089
THAILAND
Bangkok ..... 66(2)254–4910
TURKEY
..... (90) 212 274 66 48
UNITED KINGDOM
Aylesbury ..... 44 1 (296)395252

NORTH AMERICA
FULL LINE REPRESENTATIVES

ARIZONA, Tempe
S&S Technologies, Inc. ..... (602)414–1100
CALIFORNIA, Loomis
Galena Technology Group ..... (916)652–0268
INDIANA, Indianapolis
Bailey's Electronics ..... (317)848–9958
NEVADA, Clark County
S&S Technologies, Inc. ..... (602)414–1100
NEVADA, Reno
Galena Tech. Group ..... (702)746–0642
NEW MEXICO, Albuquerque
S&S Technologies, Inc. ..... (602)414–1100
TEXAS, El Paso
S&S Technologies, Inc. ..... (915)833–5461
UTAH, Salt Lake City
Utah Comp. Sales, Inc. ..... (801)572–4010
WASHINGTON, Spokane
Doug Kenley ..... (509)924–2322

NORTH AMERICA
HYBRID/MCM COMPONENT SUPPLIERS

Chip Supply ..... (407)298–7100
Elmo Semiconductor ..... (818)768–7400
Minco Technology Labs Inc. ... (512)834–2022
Semi Dice Inc. ..... (310)594–4631