ECHELON®

ISI Programmer's Guide Version 3

078-0299-01F

Echelon Corporation
www.echelon.com

Introduction

This guide describes how you can use Interoperable Self-Installation (ISI) to create networks of control devices that interoperate, without requiring the use of an installation tool. This guide also describes how to use Echelon's ISI Library to develop devices that can be used in both self-installed as well as managed networks.

This guide refers to version 3.01 of Echelon's ISI Libraries.

Overview

Control networks consist of intelligent devices like switches, thermostats, pumps, motors, valves, controllers, and a variety of other sensors and actuators that communicate with each other to provide distributed monitoring and control. A control network may be a small, simple network consisting of a few devices; may be a large network in a building, factory, or ship consisting of tens of thousands of devices; or may even be a very large regional network consisting of millions of devices. In every case, the devices in the network must be configured to become part of a common network, and to exchange data amongst themselves. The process of configuring devices in a control network to establish communication among the devices is called *network installation*.

Networks can be categorized by the method used to perform network installation. The two categories of networks are *managed networks* and *self-installed* networks. A managed network is a network where a shared *network management server* is used to perform network installation. A user typically uses a tool to interact with the server and define how the devices in the network should be configured and how they should communicate. Such a tool is called a *network management tool*. For example, Echelon's LonMaker® Integration Tool is a network management tool that uses a network management server called the LNS® Server to install devices in a network. Although a network management tool and a server are used to initially establish network communication, they need not be present all the time for the network to function. The network management tool and server are only required whenever changes are made to the network's configuration.

In a managed network, the network management tool and server allocate various network resources, such as device and data point addresses. The network management server is also aware of the network topology, and can configure devices for optimum performance within the constraints of the topology.

The alternative to a managed network is a self-installed network. There is no central tool or server that manages all of the network configuration in a self-installed network. Instead, each device contains code that replaces parts of the network management server's functionality, resulting in a network that no longer requires a special tool or server to establish network communication or to change the configuration of the network.

Devices in a self-installed network cannot rely on a network management server to coordinate their configuration. Since each device is responsible for its own configuration, a common standard is required to ensure that devices configure themselves in a compatible way. The standard protocol for performing self-installation in LONWORKS networks is called the *LONWORKS Interoperable Self-Installation (ISI) Protocol*. The ISI protocol can be used for networks of up to 200 devices that meet topology and connection constraints described in this guide. Larger or more complex networks must either be installed as managed networks, or must be partitioned into multiple smaller subsystems, where each subnetwork has no more than 200 devices and meets the ISI topology and connection constraints. Devices that conform to the LONWORKS ISI protocol are called *ISI devices*.

This guide presents a library for Neuron® C—called the *Neuron C ISI library*—to create interoperable self-installed devices. This library implements the ISI protocol. This guide details the application programming interface (API) that you will use to interact with the ISI library. For a detailed description of the ISI protocol, see the *LONWORKS ISI Protocol Specification*.

Networks can start out as self-installed networks using ISI and, as size or complexity grows beyond the ISI limits, can be upgraded into a managed network. A self-installed network may also be transitioned to a managed network to take advantage of the additional flexibility and capability provided by a network management tool and server. This guide also details recommended procedures when transitioning from a self-installed network to a managed network. By following these guidelines, self-installed networks can be easily transitioned to managed networks while maintaining all of the configuration information from the self-installed network.

The ISI protocol is a licensed protocol. The ISI Developer's Kit and Mini EVK Evaluation Kit both include a license for development use of the ISI library with Echelon's FT 3120®/FT 3150®/PL 3120/PL 3150/PL 3170™ Smart Transceivers or Echelon's FTT-10A/LPT-11/PLT-22 Transceivers used in conjunction with Neuron Chips. By signing a Revision J or newer OEM License Agreement (or an amendment to a prior version that includes rights to the ISI protocol), manufacturers can acquire a royalty-free license to produce devices incorporating the ISI library and using an Echelon FT 3120/FT 3150/PL 3120/PL 3150/PL 3170 Smart Transceiver or an Echelon FTT-10A/LPT-11/PLT-22 Transceiver used in conjunction with a Neuron Chip.

To use the ISI trademark or logo in your products, you must first certify your products to the *LONMARK 3.4 Application-Layer Guidelines* (or newer), and sign the 15 July 2005 or newer version of the LONMARK Certified Products Logo License Agreement.

To use the ISI libraries in products designed for us in a home environment, you must also have a Digital Home Alliance Agreement in effect with Echelon.

Changes since Revision 2

This document describes version 3.01 of the ISI Library. Revision 3 of this document added details on how to use network variables heartbeats, turnaround connections, controlled enrollment, and also describes several new ISI libraries. Revision 3.01 of this document adds details about developing applications for the PL 3170 Smart Transceiver, and also describes the new **IsiPl3170.lib** library.

Table of Contents

Installing ISI Devices

ISI devices may support *plug-and-play* or *plug-touch-and-play* installation. For plug-and-play installation, installation is performed by plugging in the device. No user interaction is required in this case. This is suitable for devices where connections can be determined automatically. For example, all appliances in a home may automatically connect to a home gateway. If power line transceivers are used, then the network connection is created by plugging in the device so no other steps are required to install an appliance in the home network.

ISI devices may support plug-touch-and-play installation, in which case some minimal user interaction is required to either join a network or create a connection. The interaction may be with a user interface device such as a user interface panel. Alternatively, the interaction may be with the devices themselves. For example, the user may push a button on a device to create a connection. This button is called the *Connect button*. Feedback may be provided to the user using an LED, called the *Connect light*. A lighting system may have a Connect button and light on each switch and each lamp actuator. In this example, the user selects switches and lights to be connected by pressing the Connect buttons on the devices to be connected.

On a simple device, the Connect button may be the same as the Service button, and the Connect light may be the same as the Service light. In this case, the same button may be used to join an ISI network, join a managed network, join a connection, and to restore the device's self-installation data to factory defaults. More complex devices may require multiple Connect buttons and lights. For example, a device that supports multiple manual connections to multiple devices, may use multiple Connect buttons and lights that are not shared with the Service button and light.

ISI Types

There are two types of ISI networks—*ISI-S* for simple and standalone ISI networks, and *ISI-DA* for self-installed networks that support more devices than ISI-S, more complex topologies, and unique domain IDs. An ISI-DA network must include one or more *domain address server (DAS)* devices, and all the devices in an ISI-DA network must be ISI-DA compatible. The DAS devices are present to help manage the ISI-DA network. The protocol implemented by the domain address servers is called the *ISI-DAS* protocol. The domain address servers do not take on the full roll of network management servers. Instead, they are only used to coordinate assignment of unique domain IDs and to maintain an estimate of network size to optimize use of available channel bandwidth.

ISI Messages

The ISI protocol defines a standard set of messages that are used to coordinate the installation of devices in an ISI network. The ISI engine that is part of the ISI library automatically generates and processes most ISI messages. It is sometimes useful to view ISI messages when debugging an ISI application. The ISI Developer's Kit includes an ISI Packet Monitor application that you can use during development for capturing and interpreting ISI messages. The ISI Packet Monitor application is described in *Developing and Debugging the ISI Implementation* in Chapter 4. A few of the key ISI messages are introduced in this section. All of the ISI messages are described and documented in the *ISI Protocol Specification*. Following are a few of the most important ISI messages:

- *Device Resource Usage Message (DRUM)*—this message is periodically broadcast by all ISI devices. It includes the physical address (Neuron ID), logical address (domain, subnet, node IDs), non unique ID, and channel type for the device. The extended version of this message adds a device class and usage field for use in device tracking. You can enable the extended version by passing **isiFlagExtended** into **IsiStart*()** (see *Starting and Stopping Self-Installation* in Chapter 2).

- *Connection Status Messages (CSMs)*—this group of messages is used to create, maintain, and delete connections. There are multiple types of connection status messages, including messages to manually create a new connection (CSMO), automatically create a new connection (CSMA and CSMR), and delete a connection (CSMD). The CSMO, CSMA, and CSMR messages include the group ID, primary functional profile, primary network variable type and direction, variant number, and number of network variables for an offered connection. Devices that receive these messages can use the information—plus optional user interaction—to determine whether or not to join the connection. The extended version adds fields to determine if the connection is acknowledged or polled, the scope of the connection and parts of the program id, and the primary network variable member. You can enable the extended version by passing **isiFlagExtended** into **IsiStart*()** (see *Starting and Stopping Self-Installation* in Chapter 2).

- *Timing Guidance Message (TIMG)*—this message is periodically broadcast by all domain address servers. It includes information about network size and latency. It is an optional message, but if available, ISI devices use this information to schedule all periodic message based on network size. This ensures efficient use of the channel bandwidth and minimizes the overhead of the ISI protocol

ISI Limits

This section describes ISI limits. Some of the limits depend on options selected by your device application, and some depend on which ISI library you choose to link with your application. The ISI libraries and features of each are described in *Optimizing the Footprint of ISI Applications*. Those who use your devices will need to know the resulting limits for your devices. Guidelines for documenting these limits will be available at www.echelon.com/isi.

Device Count Limits

ISI networks support up to 32 devices for ISI-S networks and up to 200 devices for ISI-DA networks. ISI networks will not immediately stop functioning if these limits are exceeded. Increasing the number of devices over the supported limits increases the network bandwidth consumed for administrative ISI messages, possibly preventing regular network operation due to an increased collision rate.

Networks should be designed with some headroom. A reasonable limit is 80%. ISI-S networks that reach 26 devices should be considered for an upgrade to ISI-DA (which might be as simple as adding a DAS), and ISI-DA networks exceeding 160 devices are prime candidates for an upgrade to a managed LNS network.

Channel Types and Limits

The supported channel types for the ISI protocol are PL-20 power line and TP/FT-10 free topology twisted pair.

The maximum channel limit for a device using the **IsiCompactManual** or **IsiCompactAuto** library is one channel. For devices using any of the other libraries, the limit is two channels. ISI-DAS devices always support two channels.

The **IsiCompactManual** and **IsiCompactAuto** libraries are provided for feature-limited implementations of ISI devices with minimum memory footprint. The functionality of the ISI libraries is described in *Optimizing the Footprint of ISI Applications*.

Supported Topology and Routers

ISI-S networks are limited to a single channel that is segmented with physical layer repeaters according to the standard channel properties—none for PL-20 channels, or multiple for TP/FT-10 channels provided there is never more than one physical layer repeater between any two points of communication. In other words, you can have one N-way repeater, much in the way of an N-port Ethernet hub. A physical layer repeater is similar to a hub (signal booster without filtering logic).

ISI-DA networks can have one or two channels. ISI-DA networks with two channels must include a router configured as a repeater. Each channel must meet the same requirements as a channel for ISI-S without a DAS described above. The router must be preconfigured to be compatible with ISI networks, or otherwise capable of joining an ISI network.

If a domain address server is used in a two-channel network with a PL-20 and TP/FT-10 channel, it should be located on the PL-20 channel. One of the functions of the domain address server is to determine the slowest channel of the network that it is located on. If a domain address server is located on the PL-20 channel, it will start-up with knowledge of the slowest channel. If it is located on the TP/FT-10 channel, it will have to learn of the existence of the PL-20 channel by discovering one of the PL-20 devices. This may take some time. Conversely, if the domain address server is located on the TP/FT-10 channel and all PL-20 devices are removed from the network, the domain address server should be reset to relearn the network topology.

ISI does not support redundant routers, and the user is responsible for avoiding looping topologies. The network topologies described in this section will not cause looping topologies.

Connection Complexity

A single device can join multiple connections; limited by available address, alias, and connection table space on the device. Devices can map one or more network variables to a single selector, but it is the device's responsibility to ensure that at most one input network variable is mapped to a single selector. The number of network variables in any given connection and on any single device is only limited by device resources (alias, address, and connection table space). Connections can include an unlimited number of devices. Devices supporting aliases (those not built with the **IsiCompactManual** or **IsiCompactAuto** library) can extend connections; that is, a single network variable on a device can join multiple connections at the same time. Devices using the **IsiCompactManual** or **IsiCompactAuto** libraries cannot extend connections (no support for aliases) and cannot replace or remove connections (no support for removal) other than by returning the device to factory settings. A standard mechanism is supported with each ISI device to return to factory defaults.

Identification of Connections

Connections are established using a process called *enrollment*. A device may accept a connection invitation and join a connection on the basis of a single network variable type alone. For example, a device can choose to join a connection that uses a **SNVT_switch** network variable. A device may accept a connection invitation and join a connection on the basis of a single functional block. For example, a device can choose to join a connection that offers data from a **SFTPclosedLoopSensor** functional block with the **SNVT_xxx** output implemented as a **SNVT_amp** network variable. *Standard connections* are those that can be understood (and accepted or refused) solely by knowledge derived from the standard resource file set. Enrollment procedures that require additional knowledge are collectively named *manufacturer-specific connections*, although such a connection may not be limited to a single manufacturer. For example, a group of manufacturers may share knowledge required in the understanding (accepting) of those connections. The complexity of manufacturer-specific connections is unlimited (but cannot exceed 63 selectors, and should not exceed 4 selectors). For example, a single manufacturer-specific open enrollment message can contain a number of different standard and non-standard functional profiles. The simple case of a manufacturer-specific connection allows enrollment of user network variables and profiles.

Address Table Size

Neuron C applications should maximize the address table size using the **#num_addr_table_entries** compiler directive. The maximum size for Neuron C is 15. Even though most ISI devices require fewer address table entries when self-installed, implementing a 15-entry address table if space is available allows for versatile and complex connections when used in a managed network.

When used in a self-installed network, an ISI device will typically only require one address table entry for each group it can join. Since the ISI application has complete control over the groups it might belong to (by means of the **IsiGetAssembly()**, **IsiGetNextAssembly()**, and **IsiCreateCsmo()** overrides), the maximum size address table can be determined by the application developer. This is not the case in a managed network, where a 15-entry address table is normally required for flexible use of a device.

Alias Table Size

When designing an application that is to be used in a managed network, a rule of thumb is to declare a minimum number of alias table entries of at least $A + NVs / Z$, with A and Z being 3 or 5 and NVs equal to the number of NVs declared by the application. You can declare a maximum of 62 aliases. This rule-of-thumb often provides a good size estimate, although you may have a better understanding of the expected alias table requirements based on analysis of typical use-cases and expected connection scenarios for the device.

The **IsiCompactManual** and **IsiCompactAuto** libraries do not support aliases, and therefore do not use the alias table at all. In these cases, declaring an alias table following the above guidelines is recommended to allow for use of the device in a managed network.

Other ISI libraries support aliases, and allocate alias table entries each time an ISI connection is extended (using the **IsiExtendEnrollment()** function call). One alias table entry is typically required for each network variable that is associated with the assembly that is used with the **IsiExtendEnrollment()** function, unless the assembly is not yet bound at that time. An API is provided to determine the connection status of a given assembly (**IsiIsConnected()** function), and a function is provided to determine the number of remaining, unused, alias table entries (**IsiGetFreeAliasCount()**). The number of **IsiExtendEnrollment()** calls made for the same assembly over time cannot be predetermined (unless the application never calls this function, and never accepts automatic enrollment). For those applications, the above rule-of-thumb is recommended for a reasonable minimum alias table size.

Automatic enrollment always calls **IsiExtendEnrollment()**, unless the ISI library does not support connection extensions. In this case, **IsiCreateEnrollment()** is used instead.

Domain Table Size

The domain table holds two entries by default. This is the required minimum size of the domain table for an ISI device. If you use the **#num_domain_entries** compiler directive to construct an ISI application that implements just one domain table entry, the device will not function correctly. The error log contained in the device will report an **invalid_domain (138)** error, resulting from an attempt to start the ISI engine with less than two domain table entries.

Earshot Problems

Open media such as power line may experience occasional communication outages due to interference from other power line devices or neighboring networks. In addition, an open media device may receive packets from devices using the same media in neighboring networks. The ISI protocol handles transient communication outages gracefully: when devices fail to recognize network problems or changes to network address or connection information due to transient outages, devices will recover once the outage has come to an end. Occasional and unexpected receipt of network data from distant sites will cause no harm, but if this possibility exists, a domain address server and ISI-DA can be used to logically isolate the networks and prevent inadvertent connections between devices in neighboring networks. Critical processes such as device and domain ID acquisition are protected with a user-confirmed protocol, preventing devices from being hijacked by other sites in earshot.

The following table shows a summary of key limits:

Limit	Value	Notes
Maximum device count ISI-S	32	See text on previous page.
Recommended maximum device count ISI-S	26	
Maximum device count ISI-DA	200	
Recommended maximum device count ISI-DA	160	
Maximum number of network variables per device	254	Devices may contain larger numbers of NVs or aliases, but those with an index > 254 cannot be used with an ISI network. If an ISI device is moved to a managed network, it can reveal extra functionality with additional NVs and aliases.
Maximum number of aliases per device	254, optional	
Maximum number of connections per device	Limited by device resources, but cannot exceed 254	The ISI libraries provide a default implementation of an ISI connection table with 8 entries, but the application may override this with a larger or smaller table.
Maximum number of connection assemblies per device	254	This is determined by the NV maximum, but cannot exceed 254.

Limit	Value	Notes
Maximum number of selectors per assembly	63	
Recommended maximum number of selectors per assembly	4	

ISI and Energy Storage Devices

In simple devices, such as a light or a switch, a common implementation uses an energy storage power supply, as described in the *PL 3120, PL 3150 and PL 3170 Power Line Smart Transceiver Data Book*. A device using this type of power supply can be referred to as an *energy storage device*. In these devices, under certain worst-case circumstances, the maximum packet size that can be sent is limited. ISI implements two versions of DRUM, CSMO, CSMA, and CSMR messages (See *ISI Messages* in this chapter and the *ISI Protocol Specification*). The normal version is short enough to be sent by an energy storage device, and is restricted to usage with standard network variable types and standard functional profiles only. Since energy storage devices can receive any length message, they can only host connections that use SNVTs and SFPTs, but can join a connection that uses UNVTs and UFPTs. Compound assemblies that are based on a single functional profile and hosted on an energy storage device also need to start with member 1.

The extended versions of these messages contain additional functionality, but result with the message potentially being too long to be transmitted by an energy storage device. The extended message types do not have the limitations summarized above.

By default, the ISI engine recognizes all supported message types, but will only issue the shorter versions of the DRUM, CSMO, CSMA, and CSMR messages. To enable the use of the extended DRUMEX, CSMOEX, CSMAEX, and CSMREX messages, specify *isiFlagExtended* when starting the ISI engine.

All ISI devices may use the standard message formats if the functionality provided by the extended formats is not required, but energy storage devices may not use the extended formats. Energy storage devices may be capable of successful transmission of the extended messages under certain conditions. However, this should not be relied upon, since these conditions include the momentary line condition and part tolerance details that cannot be relied upon for mass production.

ISI domain address servers for power line channels cannot be built with energy-storage power supplies. These devices would fail to transmit DIDRM and DIDCF messages under worst-case conditions (line voltage, line impedance, and part tolerances).

2

Quick Start

This chapter provides a quick start for ISI developers. The source code for a simple ISI application is described.

The code examples used in this document do not describe a complete application and are provided as example code only. Example code is not optimized for code size or performance. Any code used from this document should be tested with your application and you may be able to reduce code size by optimizing the code for your application. See the ISI Web page at www.echelon.com/isi for complete working ISI examples.

Example ISI Application

Most of the ISI protocol is implemented by the *ISI engine* that is part of the ISI library, and much of the related application development is to make calls to the ISI engine's API and override some of the ISI engine's default implementations with application-specific versions.

The ISI engine sends and receives ISI messages and manages the network configuration of your device. You can create a simple ISI application by starting the ISI engine, periodically calling the ISI engine, and processing any messages that arrive. The following program is an example of a simple ISI application that performs these tasks:

```
#pragma num_alias_table_entries 6
#include <isi.h>

when (reset) {
    // Clear all tables and start the ISI engine
    scaled_delay(31745UL); // 800ms delay
    IsiStartS(isiFlagNone);
}

mtimer repeating isiTimer = 1000ul / ISI_TICKS_PER_SECOND;

when (timer_expires(isiTimer)) {
    // Call the ISI engine to perform periodic tasks
    IsiTickS();
}

when (msg_arrives) {
    if (IsiApproveMsg()) {
        // Process an incoming ISI message
        (void) IsiProcessMsgS();
    }
}
```

The first line includes a required compiler directive, followed by the standard **isi.h** header file. This file specifies the available ISI library functions. These functions are described in the rest of this guide.

The first **when** task is the reset task. In this task, a call to the **IsiStartS()** library function initializes and starts the ISI engine. For ongoing maintenance, the second **when** task periodically calls the **IsiTickS()** function 4 times each second. Finally, the last **when** task identifies incoming application messages as ISI messages with the **IsiApproveMsg()** library function, and processes them with the **IsiProcessMsgS()** function.

To build an ISI application, you must link the application with one of the ISI libraries as described in *Optimizing the Footprint of ISI Applications* in Chapter 5.

3

Self-Installation Basic Procedures

This chapter describes the basic ISI procedures that will be implemented by most ISI applications. Chapter 5 describes more advanced procedures. The functions described in this chapter are further described in Appendices B and C and the data structures used by these functions are documented in Appendix A.

Starting and Stopping Self-Installation

void **IsiPreStart**(void);

void **IsiStart**(IsiType *Type*, IsiFlags *Flags*);

void **IsiStartS**(IsiFlags Flags);

void **IsiStartDA**(IsiFlags Flags);

void **IsiStartDAS**(IsiFlags Flags);

void **IsiStop**(void);

void **IsiIsRunning**(void);

You can start and stop the ISI engine. The ISI engine sends and receives ISI messages and manages the network configuration of your device. You will typically start the ISI engine in your reset task when self-installation is enabled, and you will typically stop the ISI engine when self-installation is disabled.

You can use the **IsiStart()** function to start the ISI engine using any ISI type. You can use specialized versions of the **IsiStart()** function to minimize the memory footprint of your application. Devices that only support a single ISI type may use one of the following functions:

- **IsiStartS()**—starts the ISI engine for a device that does not support domain acquisition.
- **IsiStartDA()**—starts the ISI engine for a device that supports domain acquisition, but is not a domain address server.
- **IsiStartDAS()**—starts the ISI engine for an ISI-DAS application that supports domain acquisition and is a domain address server.

For PL 3170 devices, you must call the **IsiPreStart()** function from the **when(reset)** task before calling any other ISI functions. This function establishes the runtime links between the ISI engine in the read-only memory (ROM) of a PL 3170 Smart Transceiver and the callbacks in the application. You must call this function even if you do not plan to start the ISI engine.

The **IsiPreStart()** function is supported only for PL 3170 devices, and is not supported for other device types.

You can stop the ISI engine by calling the **IsiStop()** function. When you stop the ISI engine, callbacks into the application will no longer occur. Most ISI functions that the application might invoke, such as the **IsiTickS()** function introduced earlier, have a benign behavior when the engine is stopped. The application need not track the engine's state therefore, and may make the same set of ISI API calls in any state. The behavior of each ISI function in the idle state is described later in this document. The **IsiIsRunning()** function may be called at any time and returns a non-zero value if the engine is running.

All ISI devices must have a standard way to enable and disable self-installation. This enables self-installed devices to be installed into managed networks as described in Chapter 6. To provide this interface, include a **SCPTnwrkCnfg** configuration property implemented as a configuration network variable that applies to your application's Node Object functional block, if available, or applies to the entire device if there is no Node Object functional block. The configuration property has two values—**CFG_LOCAL** and **CFG_EXTERNAL**. When set to **CFG_LOCAL**, your application can enable self installation. When set to **CFG_EXTERNAL**, your application must disable self installation. Network management tools automatically set this value to **CFG_EXTERNAL** to prevent conflicts between self-installation functions and the network management tool. See *Implementing a **SCPTnwrkCnfg** CP* for more details.

To maximize compatibility with network management tools used for managed networks, insert an 800 millisecond to one-and-a-half second delay before calling any of the **IsiStart()** functions. This delay can be implemented with a call to the **delay()** or **scaled_delay()** function, other application processing, or a combination of application processing plus a call to the **delay()** or **scaled_delay()** function. Without this delay, a network tool may fail to confirm a state change when commissioning the device for the first time, or for the first time after a change to the device's application.

EXAMPLE 1

The following example declares a **SCPTnwrkCnfg** configuration property that applies to the device, tests its value on startup, waits for 800ms, and starts the ISI engine without support for domain acquisition.

```
network input SCPTnwrkCnfg cp cp_info(reset_required,
device_specific) cpNetConfig;

device_properties {
   cpNetConfig = CFG_EXTERNAL
};

when (reset) {
   if (cpNetConfig == CFG_LOCAL) {
      scaled_delay(31745UL); // 800ms delay
      IsiStartS(isiFlagNone);
   }
}
```

See *Implementing a **SCPTnwrkCnfg** CP* for more important considerations.

Implementing Periodic Services

void **IsiTick(**IsiType *Type***)**;

void **IsiTickS(**void**)**;

void **IsiTickDa(**void**)**;

void **IsiTickDas(**void**)**;

You must periodically call the **IsiTick()** function after you have started the ISI engine as described in the previous section. You should call this function approximately every 250ms. You can use a timer task to implement the periodic service.

If your device supports a single ISI type, you can use one of the three specialized versions of the **IsiTick()** function to minimize the memory footprint of your application.

The **IsiTick()** and **IsiTickDas()** functions are not supported for PL 3170 devices.

EXAMPLE

The following example calls the **IsiTickS()** function every 250ms:

```
#include <isi.h>

mtimer repeating isiTimer = 1000ul / ISI_TICKS_PER_SECOND;

when (timer_expires(isiTimer)) {
    IsiTickS();
}
```

Handling an Incoming Message

boolean **IsiApproveMsg**(void);

boolean **IsiProcessMsg**(IsiType *Type*);

boolean **IsiProcessMsgS**(void);

boolean **IsiProcessMsgDa**(void);

boolean **IsiProcessMsgDas**(void);

boolean **IsiApproveMsgDas**(void);

boolean **IsiProcessResponse**(void);

You can determine if an incoming message is an ISI message, and you can pass all ISI messages to the ISI engine for processing. To determine if a message is an ISI message, call the **IsiApproveMsg()** function. This function returns a non-zero value if the incoming message is an ISI message and the ISI engine is running. To process an ISI message, call one of the **IsiProcessMsg()** functions. If the **IsiProcessMsg()** function returns **FALSE**, then the message has been recognized.

You can use the **IsiProcessMsg()** function to process an ISI message on a device that supports multiple ISI types. If your device supports a single ISI type, you can use one of the three specialized versions of the **IsiProcessMsg()** function to minimize the memory footprint of your application.

The **IsiProcessMsg()** functions pass the received message to the ISI engine, which handles all of the processing. You can perform any application-specific processing for the received message before calling the **IsiProcessMsg()** function.

Domain address servers need to use the **IsiApproveMsgDas()** function to approve an incoming message. This is necessary for the device acquisition process watching for service pin messages. Domain address servers also need to implement the **IsiProcessResponse()** function. Not using either of these functions will cause the domain or device acquisition processes to fail. Both **IsiApproveMsgDas()** and **IsiProcessResponse()** share the same return values as **IsiApproveMsg()** and **IsiProcessMsg()**, respectively.

EXAMPLE 1

The following example for a device without domain acquisition tests for incoming ISI messages, and calls **IsiProcessMsgS()** to process them:

```
when (msg_arrives) {
    if (IsiApproveMsg() && IsiProcessMsgS()) {
        // TODO: process unprocessed ISI messages here (if any)
    } else {
        // TODO: process other application messages here (if any)
    }
}
```

EXAMPLE 2

The following partial DAS example tests for incoming ISI messages and responses on a domain address server. For a complete implementation of a DAS device, the **IsiStartDas()** and **IsiTickDas()** functions must also be called at a minimum.

```
when (msg_arrives) {
    if (IsiApproveMsgDas() && IsiProcessMsgDas()) {
        // TODO: process unprocessed ISI messages here (if any)
    } else {
        // TODO: process other application messages here (if any)
    }
}

when (resp_arrives) {
   if (IsiProcessResponse()) {
      // TODO: process unprocessed responses here (if any)
   }
}
```

Acquiring a Domain Address

void **IsiAcquireDomain**(boolean *SharedServicePin*);

void **IsiStartDeviceAcquisition**(void);

void **IsiCancelAcquisition**(void);

void **IsiCancelAcquisitionDas**(void);

void **IsiFetchDomain**(void);

void **IsiFetchDevice**(void);

There are three methods to assign a domain to an ISI device:

1. The domain may be fixed and assigned by the device application. All ISI devices must initially support this method since an initial application domain is assigned prior to acquiring a domain using one of the other methods. This enables all devices to be used in an ISI-S network.

2. A device that supports domain acquisition can acquire a unique domain address from a domain address server. If a domain address server is not available, domain acquisition will fail and the ISI engine will continue to use the most recently assigned domain (initially, the default domain). Devices that support domain acquisition also support multiple, redundant, domain address servers. Domain address acquisition is initiated by the user and controlled by the device acquiring the domain,

not the domain address server. This method allows the device to make intelligent decisions about retries, preventing enrollment during the domain acquisition. It also allows the device to increase automatic enrollment performance following the completion of domain acquisition.

3. A domain address server can assign a domain to a device without a request from the device. This minimizes the code required in the device, and can be used with all devices. This process is called *fetching a device*.

4. A domain address server can fetch the domain from any of the devices in a network and assign it to itself. This keeps multiple domain address servers in a network synchronized with each other, or allows a replacement domain address server to join an existing ISI network. This process is called *fetching a domain*.

A domain address server must support both methods 2, 3 and 4. It can allow a device that supports domain acquisition to acquire a domain, it can fetch any ISI device, and it can fetch a domain from another device.

A domain address server can fetch a domain from any device in the network to be joined. To install a replacement domain address server, or to install redundant domain address servers into an existing, previously configured, network, each domain address server can query the current domain configuration from any previously configured device in the network. This is called *fetching a domain*, and this process is typically initiated by the user via the newly installed domain address server.

The following table summarizes the first three procedures:

	Acquire Domain	Fetch Device	Fetch Domain
Description	Devices that support domain acquisition use this procedure to obtain a unique domain address from a domain address server	A domain address server assigns a unique domain address to an ISI device	A domain address server acquires the domain from another previously installed device in the network
Initiated on	Device after enabling device acquisition on the domain address server	Domain address server	Domain address server
Code required	On device and on domain address server	On domain address server	On domain address server

Key advantage	Active process—device is in control of proceedings and aware of success or failure	Passive process, supporting resource-restricted device implementations	Supports installation of replacement or redundant domain address servers

Acquiring a Domain Address from a Domain Address Server

To acquire a domain address from a domain address server, start the ISI engine using the **IsiStartDA()** function, or using the **IsiStart()** function with the **isiTypeDa** type.

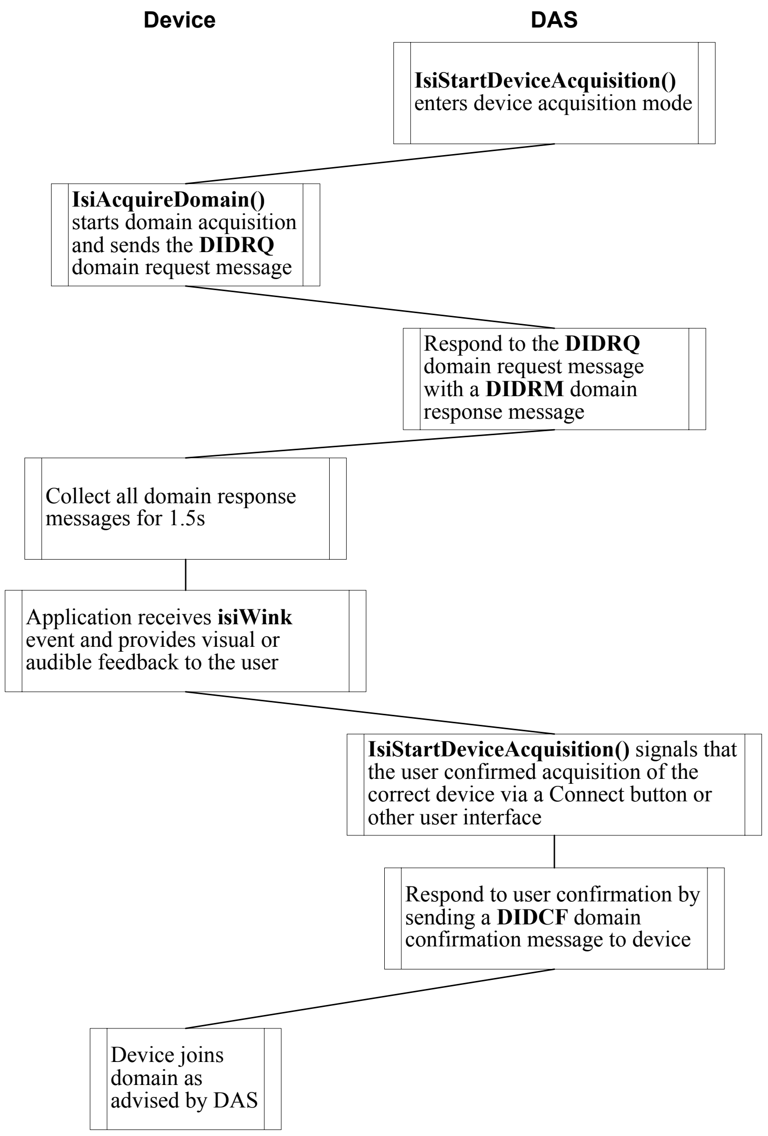
A domain address server must be in device acquisition mode to respond to domain ID requests. To start device acquisition mode on a domain address server, call the **IsiStartDeviceAcquisition()** function.

To start domain acquisition on a device that supports domain acquisition, call the **IsiAcquireDomain()** function.

A typical implementation starts the domain acquisition process when the Connect button is activated and a domain is not already assigned. If **SharedServicePin** is set to **FALSE**, the **IsiAcquireDomain()** function also issues a standard service pin message—this allows using the same installation paradigm in both a managed and an unmanaged environment. If the application uses the physical service pin to trigger calls to the **IsiAcquireDomain()** function, the system image will have issued a service pin message automatically, and the **SharedServicePin** flag should be set to **TRUE** in this case.

When calling **IsiAcquireDomain()** with **SharedServicePin** set to **FALSE** while the ISI engine is not running, a standard service pin message is issued nevertheless, allowing the same installation paradigm and same application code to be used in both self-installed and the managed networks.

After domain acquisition has been enabled by calling **IsiStartDeviceAcquisition()** on the domain address server and it has been started on the device by calling **IsiAcquireDomain()**, the device responds to the **isiWink** ISI event with a visible or audible response. For example, a device may flash its LEDs. The user confirms that the correct device executed its wink routine and the DAS application then confirms the device by calling **IsiStartDeviceAcquisition()** once again. Once confirmed, the domain address server grants the unique domain ID to the device. The device notifies its application with ISI events accordingly.


Device
DAS
IsiStartDeviceAcquisition() enters device acquisition mode
IsiAcquireDomain() starts domain acquisition and sends the DIDRQ domain request message
Respond to the DIDRQ domain request message with a DIDRM domain response message
Collect all domain response messages for 1.5s
Application receives isiWink event and provides visual or audible feedback to the user
IsiStartDeviceAcquisition() signals that the user confirmed acquisition of the correct device via a Connect button or other user interface
Respond to user confirmation by sending a DIDCF domain confirmation message to device
Device joins domain as advised by DAS

The device automatically cancels domain acquisition if it receives multiple, but mismatching, domain response messages in step 4. This may happen if multiple domain address servers with different domain addresses are in device acquisition mode, and all respond to the device's query.

Devices should support the domain acquisition process whenever possible (device resources permitting) over the Fetch Device process described below—the domain acquisition process provides a more robust process with features such as automatic retries and other desirable side effects, like automatic connection reminders.

The IsiCancelAcquisition() function causes a device to leave domain acquisition mode. The cancellation applies to both device and domain acquisition. After this function call is completed, the ISI engine calls IsiUpdateUserInterface()with the IsiNormal event. On a domain address server, use the IsiCancelAcquisitionDas() function instead.

Example 1

The following example starts domain acquisition mode on a domain address server when the user presses a Connect button on the server:

```
when (connect_button_pressed) {
   IsiStartDeviceAcquisition();
}
```

Once started, the domain address server remains in this state for 5 minutes unless cancelled with an **IsiCancelAcquisitionDas()** call. Each successful device acquisition retriggers this timeout.

Example 2

The following example starts domain acquisition on a device when the user pushes a Connect button on the device:

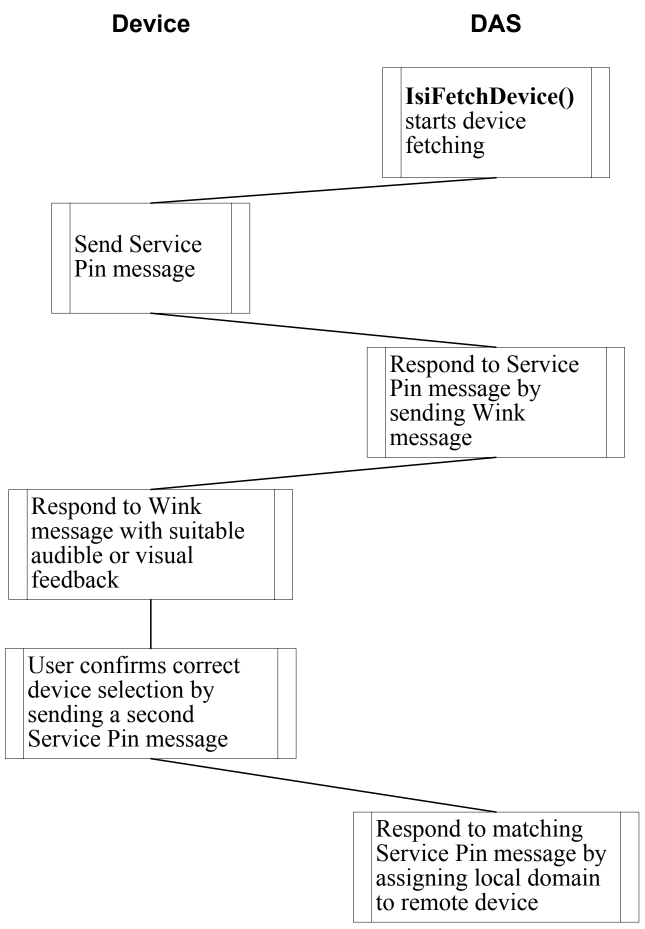
```
when (connect_button_pressed) {
   IsiAcquireDomain(FALSE);
}
```

Fetching a Device from a Domain Address Server

A domain address server can use the **IsiFetchDevice()** function to assign the DAS' unique domain ID to any device. Unlike the **IsiAcquireDomain()** function, the **IsiFetchDevice()** function does not require any action, or special library code, on the device. To fetch a device, call the **IsiFetchDevice()** function on the domain address server.

DAS devices must make this feature available to the user. With this feature, it is not required that devices support domain acquisition in order to participate in an ISI network that uses unique domain IDs.

Similar to the domain acquisition process detailed above, fetching a device also requires a manual confirmation step to ensure that the correct device is paired with the correct domain address server:


Device
DAS
IsiFetchDevice() starts device fetching
Send Service Pin message
Respond to Service Pin message by sending Wink message
Respond to Wink message with suitable audible or visual feedback
User confirms correct device selection by sending a second Service Pin message
Respond to matching Service Pin message by assigning local domain to remote device

EXAMPLE 3

The following example fetches a device on a domain address server when the user presses the Connect button on the server:

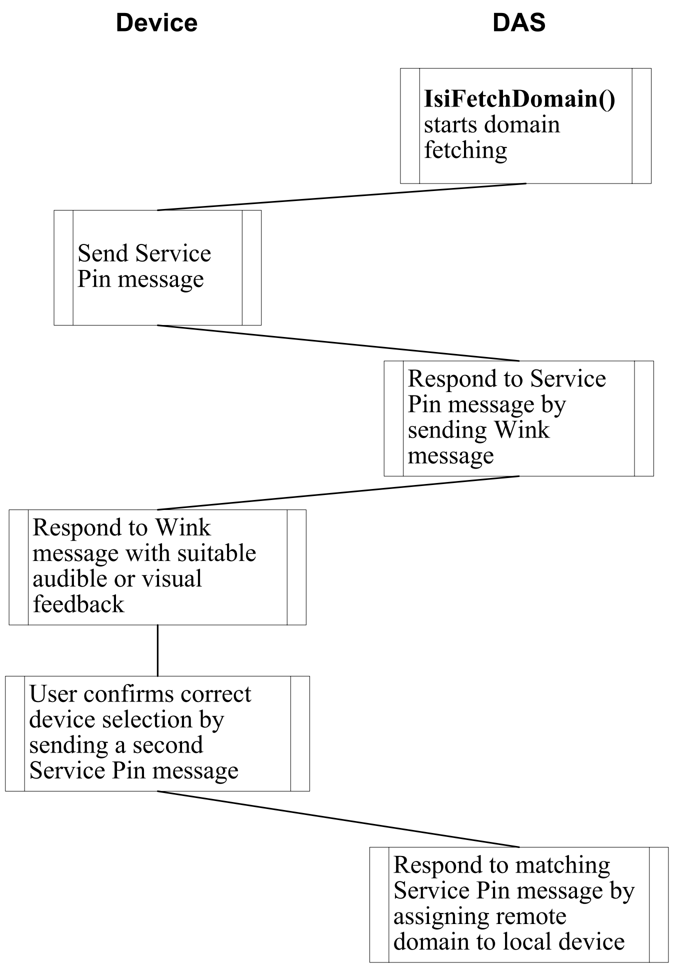
```
when (connect_button_pressed) {
   IsiFetchDevice();
}

//Handle responses to requests in IsiFetchDevice()
when (resp_arrives) {
   if (IsiProcessResponse()) {
      // TODO: process unprocessed responses here (if any)
   }
}
```

Fetching a Domain for a Domain Address Server

A domain address server can use the **IsiFetchDomain()** function to obtain a domain ID. Unlike the **IsiAcquireDomain()** function, the **IsiFetchDomain()** process does not require a domain address server to provide the domain ID information, and does not use the DIDRM, DIDRQ, and DIDCF standard ISI messages. Instead, the domain address server uses the **IsiFetchDomain()**

function to obtain the current domain ID from any device in the network, even from those that only implement ISI-S, or that do not implement or execute ISI at all. This is typically used when installing replacement or redundant domain address servers in a network: a domain address server will normally use the **IsiGetPrimaryDid()** override to specify a unique, non-standard, primary domain ID. A replacement domain address server, or a redundant domain address server, needs to override this preference by using the domain ID that is actually used in the network. This is provided with the **IsiFetchDomain()** function.


Device
DAS
IsiFetchDomain() starts domain fetching
Send Service Pin message
Respond to Service Pin message by sending Wink message
Respond to Wink message with suitable audible or visual feedback
User confirms correct device selection by sending a second Service Pin message
Respond to matching Service Pin message by assigning remote domain to local device

EXAMPLE 3

The following example fetches a domain on a domain address server when the user presses the Connect button on the server:

```
when (connect_button_pressed) {
   IsiFetchDomain();
}

//Handle responses to requests in IsiFetchDomain()
when (resp_arrives) {
   if (IsiProcessResponse()) {
      // TODO: process unprocessed responses here (if any)
   }
}
```

If no unambiguous domain ID is already present on the network, the domain address server will use its default domain ID, as advised with the **IsiGetPrimaryDid()** callback, as a unique domain ID.

Enrolling in a Connection

You can exchange data between devices by creating *connections* between network variables on the devices. Connections are like virtual wires, replacing the physical wires of traditional hard-wired systems. A connection defines the data flow between one or more output network variables to one or more input network variables. The process of creating a self-installed connection is called *enrollment*. Inputs and outputs join a connection during open enrollment, much like students join a class during open enrollment. This section describes the ISI connection model and describes the procedures required to create a connection.

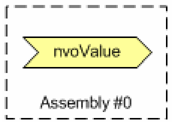
ISI Connection Model

Connections are created during an *open enrollment* period that is initiated by a user, a connection controller, or a device application. Once initiated, a device is selected to open enrollment—this device is called the *connection host*. Any device in a connection may be the connection host—the connection host is responsible for defining the open enrollment period and for selecting the connection address to be used by all network variables within the connection. Connection address assignment and maintenance is handled by the ISI engine, and is transparent to your application.

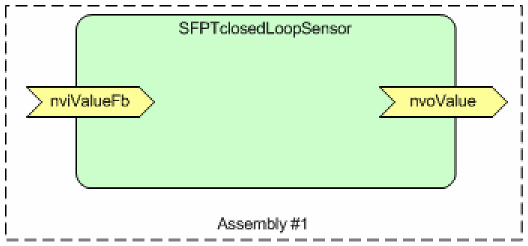
Even though any device in a connection may be the connection host, if you have a choice of connection hosts, network resource utilization will be optimized if you pick the natural hub as the connection host. For example, in a connection with one switch and multiple lights, the switch is the natural hub. In a connection with one light and multiple switches, the light is the natural hub. If there is no natural hub—multiple switches connected to multiple lights for example—using one of the devices with an output network variable will optimize network resource utilization.

A connection host opens enrollment by sending a *connection invitation*. Once a connection host opens enrollment then any number of devices may join the connection.

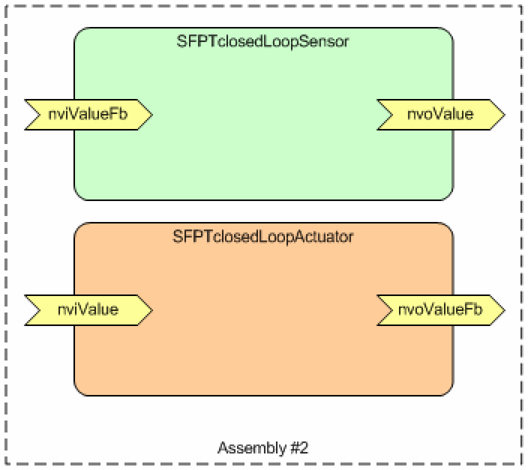
Connections are created among *connection assemblies*. A connection assembly is a block of functionality, much like a functional block. A simple assembly refers to a single network variable:



A connection assembly that consists of a single network variable is called a *simple assembly*. A single assembly can include multiple network variables in a functional block, can include multiple network variables that span multiple functional blocks, or can exist on a device that does not have any functional blocks; an assembly is simply a collection of one or more network variables that can be connected as a unit for some common purpose. A connection assembly consisting of more than one network variable is called a *compound assembly*:



For example, a combination light-switch and lamp ballast controller may have both a switch and a lamp functional block, which are paired to act as a single assembly in an ISI network, but may be handled as independent functional blocks in a managed network:



To communicate and identify an assembly to the ISI engine, the application assigns a unique number to each assembly. This assembly number must be in the 0 – 254 range sequentially assigned starting at 0. Required assemblies for standard profiles must be first, assigned in the order the profiles are declared in the application. Standard ISI profiles that define multiple assemblies must specify the order the assemblies are to be assigned.

Each assembly has a width which is typically equal to the number of network variables in the assembly. In the previous figures, for example, assembly 0 has a width of 1, assembly 1 typically has a width of 2, and assembly 2 typically has a width of 4. All assemblies must have a width of at least 1. Simple assemblies have a width of 1; compound assemblies typically have a width of greater than 1. One of the network variables in a compound assembly is designated as the *primary network variable*. If the primary network variable is part of a functional block, that functional block is designated as the *primary functional block*. Information about the primary network variable may be included in the connection invitation.

To open enrollment, the connection host broadcasts a connection invitation that may include the following information about the assembly on offer: the network variable type of the primary network variable in the assembly, the functional profile number of the primary functional profile in the assembly, and the connection width. Other devices on the network receive the invitation and interpret the offered assembly to decide whether they could join the new connection.

In the case of assembly 0, the connection invitation may just specify a width of one and the network variable type. This is a case similar to the one employed by a generic switch device where the switch offers a **SNVT_switch** network variable that is not tied to a specific functional profile.

Assembly 1 demonstrates a more specialized example. A switch may offer this assembly and describe it as an implementation of the **SFPTclosedLoopSensor** profile, with a width of two, and a **SNVT_switch** input and output. The ISI protocol defines how multiple network variable selectors are mapped to the individual network variables offered.

Since the invitation includes no more than one functional profile number, a compound assembly is typically limited to a single functional block on each device. To include multiple functional blocks in an assembly, a *variant* may be specified. A variant is an identifier that customizes the information specified in the connection invitation. Variants may be defined for any device category and/or any functional profile/member number pair. For example, a variant can be specified with the **SFPTclosedLoopSensor** functional block offered in assembly 2 above to specify that the **SFPTclosedLoopActuator** functional block is included in the assembly. Standard variant values are defined in standard functional profiles that are published by LONMARK International, and manufacturers may specify manufacturer-specific variant values for manufacturer-specific assemblies.

Each assembly on a device has a unique number that is assigned by the application. Each network variable on a device may be assigned to an assembly. The ISI engine calls the **IsiGetNvIndex()** and **IsiGetNextNvIndex()** callback functions to map a member of an assembly to a network variable on the device.

Opening Enrollment

void **IsiOpenEnrollment(**unsigned *Assembly***)**;

void **IsiCreateCsmo(**unsigned *Assembly*, IsiCsmoData* *pCsmoData***)**;

unsigned **IsiGetPrimaryGroup(**unsigned *Assembly***)**;

unsigned **IsiGetWidth(**unsigned *Assembly***)**;

void **IsiInitiateAutoEnrollment(**const IsiCsmoData* *pCsma*, unsigned *Assembly***)**;

void **IsiUpdateUserInterface(**IsiEvent *Event*, unsigned *Parameter***)**;

You can create a connection using *automatic*, *controlled*, or *manual* enrollment. When you use controlled or manual enrollment, user intervention is required to identify devices or assemblies to be connected. Controlled enrollment is initiated by a centralized tool such as a controller or user interface panel. This centralized tool is called the *connection controller*. Manual enrollment is initiated from the devices to be connected, typically with a push button called the *Connect button*. When you use automatic enrollment, connections are automatically created and no user intervention is required.

To join a connection, a device must support at least one type of enrollment. A device may support multiple types of enrollment—a device may even support all three types of enrollment. For example, a lamp actuator may support automatic enrollment to a gateway, controlled enrollment configured by a user interface panel, and manual enrollment with switch devices. Devices that support controlled enrollment must also support connection recovery as described in *Recovering Connections* in Chapter 5. Standard functional profiles may require support for specific types of enrollment.

An event triggers your application to open enrollment. The type of event depends on the type of enrollment:

- *Manual enrollment*—a user input on the device itself typically triggers manual enrollment. The input may be a simple button push, or a device may have a more complex user interface that allows the user to request a connection.
- *Controlled enrollment*—a request from a connection controller typically triggers controlled enrollment. This request is typically initiated by some user input to the connection controller and arrives in a control request (*CTRQ*) message. The CTRQ message identifies an ISI function and an optional parameter.
- *Automatic enrollment*—the **isiWarm** event in the **IsiUpdateUserInterface()** callback function typically triggers automatic enrollment.

To open manual enrollment, call the **IsiOpenEnrollment()** function on the connection host, passing in the assembly number to be offered for this connection. The ISI engine then sends a connection invitation by broadcasting an *open enrollment message* (*CSMO)*. The CSMO message is the invitation for other devices to join this connection. The ISI engine creates the CSMO message by calling the **IsiCreateCsmo()** function, which fills the relevant fields of an **IsiCsmoData** data structure with the values needed to describe the connection type and data that is offered to the network. The default implementation of this function uses the **IsiGetPrimaryGroup()** function to obtain the associated group ID, and sets all fields to zero except the **Application** field (which is filled with data from the device's program ID), the **Width** field (which is set by the value returned by **IsiGetWidth()**), the **Direction** field (which is set to **isiDirectionAny**), and the **NvType** field (which is set to the assembly's primary network variable's SNVT ID). The default implementation of **IsiCreateCsmo()** is sufficient for simple devices, but you will typically override it with an application-specific, implementation. After calling **IsiCreateCsmo()**, the ISI engine constructs the remainder of the CSMO message and broadcasts the connection invitation to the network. In order to create a compound connection (one with an assembly width larger then 1), you must override the **IsiGetWidth()** callback.

Controlled enrollment is initiated and controlled by the connection controller as described in *Creating a Connection with Controlled Enrollment* in Chapter 5. In summary, the connection controller opens the controlled enrollment by sending a CTRQ message specifying the **IsiOpenEnrollment()** function, and also specifying the assembly number to be offered. The application must respond to the CTRQ message with a control response (*CTRP*) message indicating that it implements the requested operation.

To open automatic enrollment, wait for the **isiWarm** event from the **IsiUpdateUserInterface()** callback function, and then call the **IsiInitiateAutoEnrollment()** function, passing a pointer to an **IsiCsmoData** structure containing the invitation, and an the assembly number to be offered for this connection. The ISI engine then sends a connection invitation by broadcasting an automatic enrollment (*CSMA)* message. The ISI engine will also send periodic reminders about the automatic connection by sending CSMR messages. The reminder ensures that new devices have an opportunity to join the automatic connections. Whenever a CSMR is due, the ISI engine calls **IsiCreateCsmo()** to create the message. The CSMA and CSMR messages are the invitations for other devices to enroll in this connection. Opening automatic enrollment through **IsiInitiateAutoEnrollment()** is an immediate action, and once the call is made the connection is implemented for the assembly that the call was made with, regardless of whether there are any members for the connection or not.

The ISI engine automatically transmits the extended CSMOEX, CSMAEX, or CSMREX message (as appropriate) if **isiFlagExtended** was specified during the start of the engine. Otherwise, the ISI engine automatically clips the **Extended** sub-structure of the **IsiCsmoData** structure and issues the regular CSMO, CSMA, or CSMR message.

You can provide feedback to the user while enrollment is open, for example by starting a Connect light to flash. This is typically only done with manual enrollment. The ISI engine informs your application of significant ISI events by calling an **IsiUpdateUserInterface()** callback function.

EXAMPLE 1

The following example opens automatic enrollment:

```
void IsiUpdateUserInterface(IsiEvent Event, unsigned Parameter) {
    if (Event == isiWarm && !IsiIsConnected(MyAssembly)) {
        IsiInitiateAutoEnrollment(&MyCsmoData, MyAssembly);
    }
}
```

EXAMPLE 2

The following example opens manual enrollment for a simple assembly with one network variable, using the network variable's global index as the application-specific assembly number:

```
void startEnrollment(void) {
   unsigned myAssembly;
   myAssembly = nvoValue::global_index;
   IsiOpenEnrollment(myAssembly);
}
```

EXAMPLE 3

The following example opens controlled enrollment for a simple assembly with one network variable, using the network variable's global index as the application-specific assembly number:

```
void sendControlResponse(boolean success) {
    IsiMessage ctrlResp;

    ctrlResp.Header.Code = isiCtrp;
    ctrlResp.Ctrp.Success = success;
    memcpy(ctrlResp.Ctrp.NeuronID,
           read_only_data.neuron_id, NEURON_ID_LEN);

    resp_out.code = isiApplicationMessageCode;
    memcpy(resp_out.data, &ctrlResp,
           sizeof(IsiMessageHeader)+sizeof(IsiCtrp));
    resp_send();
}

when (msg_arrives) {
   IsiMessage inMsg;
   unsigned myAssembly;

   if (IsiApproveMsg()) {

      memcpy(&inMsg, msg_in.data, sizeof(IsiMessage));
      myAssembly = nvoValue::global_index;
      if (inMsg.Header.Code == isiCtrq) {
         if (inMsg.Ctrq.Control == isiOpen &&
              inMsg.Ctrq.Parameter == myAssembly) {
             sendControlResponse(TRUE);
             IsiOpenEnrollment(myAssembly);
             // Other requests deleted for this example
             …
         } else {
                sendControlResponse(FALSE);
         }
       } else {
            (void) IsiProcessMsgS();
      }
```

```
        }
    }
```

EXAMPLE 4

The following example opens manual enrollment for a compound assembly with four selectors. The **isiGetWidth()** returns the library's default value. In this example, enrollment is being opened in response to the user pressing a Connect button. Enrollment can only be opened when the ISI engine is in the normal state. The **ProcessIsiButton()** function is called in response to the Connect button being pressed:

```
IsiEvent isiState;

void IsiCreateCsmo(....) {
   // set pCsmoData as desired
}

unsigned IsiGetWidth(unsigned Assembly) {
   return Assembly == myAssemblyNumber ?
          4 : isiGetWidth(Assembly);
}

void ProcessIsiButton(unsigned Assembly) {
   switch(isiState) {
       ...
       case isiNormal:
           IsiOpenEnrollment(Assembly);
           break;
           ... //Processing for other states
    }   // end of switch(state)
}
```

EXAMPLE 5

The following example opens controlled enrollment for a compound assembly with four selectors. The **isiGetWidth()** returns the library's default value:

```
IsiEvent isiState;

void IsiCreateCsmo(....) {
   // set pCsmoData as desired
}

unsigned IsiGetWidth(unsigned Assembly) {
   return Assembly == myAssemblyNumber ?
          4 : isiGetWidth(Assembly);
}

void sendControlResponse(boolean success) {
    IsiMessage ctrlResp;

    ctrlResp.Header.Code = isiCtrp;
    ctrlResp.Ctrp.Success = success;
    memcpy(ctrlResp.Ctrp.NeuronID,
           read_only_data.neuron_id, NEURON_ID_LEN);

    resp_out.code = isiApplicationMessageCode;
    memcpy(resp_out.data, &ctrlResp,
           sizeof(IsiMessageHeader)+sizeof(IsiCtrp));
    resp_send();
}
```

```
when (msg_arrives) {
   IsiMessage inMsg;
   unsigned myAssembly;

   if (IsiApproveMsg()) {
      memcpy(&inMsg, msg_in.data, sizeof(IsiMessage));
      myAssembly = nvoValue::global_index;
      if (inMsg.Header.Code == isiCtrq) {
         if (inMsg.Ctrq.Control == isiOpen &&
               inMsg.Ctrq.Parameter == myAssembly) {
            sendControlResponse(TRUE);
            IsiOpenEnrollment(myAssemblyNumber);
            // Other requests deleted for this example
            …
         } else {
              sendControlResponse(FALSE);
         }
       } else {
           (void) IsiProcessMsgS();
       }
    }
}
```

EXAMPLE 6

The following refines example 1 and provides a more comprehensive example of opening automatic enrollment for a simple assembly with one network variable:

```
// MyCsmoData defines the enrollment details for the automatic ISI
// network variable connection offered by this device.
static const IsiCsmoData MyCsmoData = {
      ISI_DEFAULT_GROUP,      // Group
      isiDirectionOutput,     // NV direction
      1,                      // Width
      2,                      // Profile number (2 =
                              //     SFPTopenLoopSensor)
      76u,                    // Network variable type index
                              //  (76 = SNVT_freq_hz)
      0                       // Variant (0 = standard)
};

// Call InitiateAutoEnrollment in response to isiWarm
void IsiUpdateUserInterface(IsiEvent Event, unsigned Parameter) {
   if (Event == isiWarm && !IsiIsConnected(myAssemblyNumber)) {
      // We waited long enough and we are not connected already,
      // so let's open an automatic connection:
      IsiInitiateAutoEnrollment(&MyCsmoData, myAssemblyNumber);
   }
}
```

EXAMPLE 7

The following example opens automatic enrollment for a compound assembly with four selectors, offering enrollment for member network variables 1 to 4 of an implementation of the **SFPTsceneController** profile (the **nviScene**, **nvoSwitch**, **nviSetting**, and **nviSwitch** members):

```
// MyCsmoData defines the enrollment details for the automatic ISI
// network variable connection offered by this device
static const IsiCsmoData MyCsmoData = {
        ISI_DEFAULT_GROUP,      // Group
        isiDirectionVarious,    // NV direction
        4,                      // Width
        3251,                   // Profile number (3251 =
                                //     SFPTsceneController)
        0,                      // Network variable type index
                                //  (0 = determined by SFPT)
        0                       // Variant (0 = standard)
};

// Call InitiateAutoEnrollment in response to isiWarm
void IsiUpdateUserInterface(IsiEvent Event, unsigned Parameter) {
    if (Event == isiWarm && !IsiIsConnected(myAssemblyNumber)) {
        // We waited long enough and we are not connected already,
        // so let's open an automatic connection:
        IsiInitiateAutoEnrollment(&MyCsmoData, myAssemblyNumber);
    }
}
```

EXAMPLE 8

For a complete example that implements connection management for multiple assemblies, see the MgDemo example application that is available for free download from www.echelon.com/isi.

Receiving an Invitation

unsigned **IsiGetAssembly(**const IsiCsmoData* *pCsmoData*, boolean *Auto***)**;

unsigned **IsiGetNextAssembly(**const IsiCsmoData* *pCsmoData*, boolean *Auto*, unsigned *Assembly***)**;

You can receive a connection invitation and specify which assemblies are eligible to join the ISI connection. When an ISI device receives a CSMO, CSMA, or CSMR connection invitation message, the ISI engine first checks on the availability of device resources that are required in order to implement the connection. If any of these resources is forseeably missing or insufficient, such as address or connection table space, the invitation is dropped. If the ISI engine believes there are sufficient resources, it calls the **IsiGetAssembly()** and **IsiGetNextAssembly()** functions with the received CSMO, CSMA, or CSMR message. These functions return all assembly numbers that are provisionally approved to join the connection. The **Auto** argument of **IsiGetAssembly()** and **IsiGetNextAssembly()** indicates whether the enrollment is manual or controlled (CSMO) or automatically (CSMA or CSMR) initiated, with **FALSE** meaning the enrollment was initiated manually or by a connection controller. On devices that do not support connection removal, the assembly is ignored if it is already engaged in another connection.

When a device receives an extended CSMOEX, CSMAEX, or CSMREX message, all fields of the **IsiCsmoData** structure are passed to the application, and the fields in the **Extended** sub-structure are all valid.

When a device receives a regular CSMO, CSMA, or CSMR message, the extended fields will automatically be set to all zeros with exception of the **Extended.Member** field, which will be set to one.

Applications need not distinguish between regular and extended incoming messages.

You can provide feedback to the user when an invitation is received and provisionally approved, for example by causing a Connect light to flash while enrollment is open. This is typically only done with a manual connection. The ISI engine informs your application that an eligible invitation has been received and provisionally approved by calling the **IsiUpdateUserInterface()** callback function for each assembly that is provisionally approved to join the connection, sending the **isiPending** event code. This allows the application to indicate the provisionally approved, but not yet accepted, connection invitations.

EXAMPLE 1

The following example receives and provisionally approves a connection invitation, and blinks a Connect light until the invitation is accepted, or the connection is confirmed or canceled:

```
//IsiUpdateUserInterface is called with isiPending as the IsiEvent
//parameter in response to receiving a CSMO
void IsiUpdateUserInterface(IsiEvent Event, unsigned Parameter) {
   ...  //Optional event processing
   isiState = (Event == isiPending || Event == isiApproved
              || Event > isiWarm) ? Event : isiNormal;
}

unsigned IsiGetAssembly(const IsiCsmoData* pCsmoData, boolean
Auto) {
   if (pCsmoData->Group == ISI_LIGHTING_CATEGORY
       && pCsmoData->Extended.Scope == isiScopeStandard
       && pCsmoData->NvType == SNVT_SWITCH_2_INDEX
       && !(pCsmoData->Variant & 0x60)
       && !pCsmoData->Extended.Acknowledged
       && !pCsmoData->Extended.Poll) {
      // Recognized CSMO, return appropriate assembly number
      return myAssemblyNumber;
   }
   ...
}

mtimer repeating IsiTimer = 1000ul / ISI_TICKS_PER_SECOND;

when (timer_expires(IsiTimer)) {
    unsigned Assembly;

    IsiTickS();
    // drive the ISI-related LED:
    switch(isiState) {
        ...
        case isiPending:
            SetConnectLed(LED_BLINKING);
            break;
        ...
    }
}
```

Accepting a Connection Invitation

void **IsiCreateEnrollment**(unsigned *Assembly*);

void **IsiExtendEnrollment**(unsigned *Assembly*);

For manual and controlled enrollment, you can accept a connection invitation to join the offered connection. When you accept a connection invitation, the ISI engine sends an enrollment acceptance message (CSME) to the connection host. Accepting an invitation only sends an acceptance to the connection host—the connection is not implemented until the connection host confirms the new connection.

You can only accept enrollment for an assembly that has been provisionally approved. To provisionally approve an assembly, the **IsiGetAssembly()** or **IsiGetNextAssembly()** function must have returned the assembly number for the **CsmoData** structure currently under inspection, and the **IsiUpdateUserInterface()** callback function must have identified the assembly in question to be in the **isiPending** state.

For manual enrollment, a connection invitation will typically be accepted based on user input. For example, LEDs may blink on a device when invitations are received and provisionally approved as described in the previous section, and the user may then push the related Connect button to accept a specific invitation.

For a controlled enrollment, a connection invitation will typically be accepted based on a request from a connection controller. This request is typically initiated by some user input to the connection controller.

For automatic enrollment, a connection invitation will typically be accepted based on some application-specific criteria. For example, a home gateway may open automatic enrollment for its inputs and outputs, and newly installed home devices may automatically accept all eligible connection invitations from the home gateway. The actual establishment of an automatic connection is handled by the ISI engine, and requires no call to **IsiCreateEnrollment()** or **IsiExtendEnrollment()**. The ISI engine extends the connection if the library supports connection extension, or creates the extension if the library does not support connection extension and the assembly is not already connected, or if the library supports connection removal.

For devices that support connection removal, you can create a connection that replaces all existing connections for an assembly. For devices that support connections extension, you can add a new connection to an assembly that may already be enrolled in other connections. To create a connection that replaces all existing connections for an assembly, call **IsiCreateEnrollment()**. To add a connection to an assembly without overriding any existing connections associated with the same assembly, call **IsiExtendEnrollment()**. You can extend a non-existent connection; **IsiExtendEnrollment()** has the same functionality as **IsiCreateEnrollment()** if no connection exists for the assembly in question.

Extending a connection consumes network resources. Each extension to a connection requires one or more new aliases and connection table entries, and results in additional network transactions for every update to the connection.

You can eliminate this additional resource usage by deleting and re-creating a connection instead of extending it.

You can provide feedback to the user when an invitation is accepted, for example by changing the state of the Connect light from flashing to solid on when the connection invitation is accepted. This is typically only done with manual enrollment. The ISI engine informs your application that a connection invitation has been accepted by calling the **IsiUpdateUserInterface()** callback function, assigning the **isiApproved** or **isiApprovedHost** state to the respective assembly. This allows the application to indicate the accepted connection invitation.

EXAMPLE 1

The following manual enrollment example accepts a connection invitation when the user presses a Connect button. The user can press the Connect button for a long or short period, causing a destructive or constructive operation. In the context of accepting a connection invitation, the constructive operation is to accept. No destructive operation exists at this stage, but once the connection invitation has been accepted but not implemented yet, the destructive operation would be to cancel out from the previous acceptance. Using a second button or some other suitable means, the Connect button's meaning also can be altered much in the way a Control or Alt key on a computer keyboard does. In the context of accepting a connection invitation, the regular operation could be to create a connection by replacing any previous connection related with the same assembly, if any. The alternative behavior could be to always extend the connection, thus preserving any previous connections, if any.

```
IsiEvent isiState;

void ProcessIsiButton(unsigned Assembly, boolean Constructive,
  boolean Alternative) {
   switch(isiState) {
       ...
       case isiPending:
           if (Constructive) {
               if (Alternative) {
                   IsiExtendEnrollment(Assembly);
               } else {
                   IsiCreateEnrollment(Assembly);
               }
           }
           break;
           ... //Processing for other states
    }   // end of switch(state)
}

// IsiUpdateUserInterface is called with isiApproved as the
// IsiEvent parameter in response to accepting the enrollment
void IsiUpdateUserInterface(IsiEvent Event, unsigned Parameter) {
   ...  //Optional event processing
   isiState = (Event == isiPending || Event == isiApproved
              || Event > isiWarm) ? Event : isiNormal;
}

mtimer repeating IsiTimer = 1000ul / ISI_TICKS_PER_SECOND;

when (timer_expires(IsiTimer)) {
   unsigned Assembly;
```

```
    // drive the ISI-related LED:
    switch(isiState) {
        ...
        case isiApproved:
        case isiApprovedHost:
            SetConnectLed(LED_ON);
            break;
        ...
    }
}
```

EXAMPLE 2

The following example opens controlled enrollment for a simple assembly with one network variable and accepts the invitation when requested by the connection controller:

```
void sendControlResponse(boolean success) {
     IsiMessage ctrlResp;

     ctrlResp.Header.Code = isiCtrp;
     ctrlResp.Ctrp.Success = success;
     memcpy(ctrlResp.Ctrp.NeuronID,
             read_only_data.neuron_id, NEURON_ID_LEN);

     resp_out.code = isiApplicationMessageCode;
     memcpy(resp_out.data, &ctrlResp,
             sizeof(IsiMessageHeader)+sizeof(IsiCtrp));
     resp_send();
}

when (msg_arrives) {
    IsiMessage inMsg;
    unsigned myAssembly;

    if (IsiApproveMsg()) {
        memcpy(&inMsg, msg_in.data, sizeof(IsiMessage));
        myAssembly = nvoValue::global_index;
        if (inMsg.Header.Code == isiCtrq) {
            if (inMsg.Ctrq.Control == isiOpen &&
                    inMsg.Ctrq.Parameter == myAssembly) {
                sendControlResponse(TRUE);
                IsiOpenEnrollment(myAssembly);
            } else if (inMsg.Ctrq.Control == isiCreate &&
                    inMsg.Ctrq.Parameter == myAssembly) {
                sendControlResponse(TRUE);
                IsiCreateEnrollment(myAssembly);
            } else if (inMsg.Ctrq.Control == isiFactory) {
                sendControlResponse(TRUE);
                IsiReturnToFactoryDefaults();
            } else {
                   sendControlResponse(FALSE);
            }
          } else {
            (void) IsiProcessMsgS();
          }
     }
}
```

Implementing a Connection

In a manual or controlled enrollment, when a connection host sends a connection invitation by broadcasting an open enrollment message, one or more devices may accept the connection invitation and respond with an enrollment acceptance message (CSME). When the host receives at least one CSME message, the host indicates this to the host's application by calling the **IsiUpdateUserInterface()** callback function. Typically, the host's application will change the state of the related Connect light from flashing to solid on.

Once the host assembly is in that state (**isiApprovedHost**), the connection can be cancelled or implemented. See *Canceling a Connection* for details about cancellation.

To implement a connection on a connection host, call either **IsiCreateEnrollment()** or **IsiExtendEnrollment()**. The connection host joins the connection and issues a connection enrollment confirmation message (CSMC). When calling **IsiCreateEnrollment()**, any connection that exists for the same assembly will be removed (See *Deleting a Connection*, later in this chapter). When calling **IsiExtendEnrollment()**, the new connection is added to any existing connections for the same assembly, consuming an alias table entry for each NV in the assembly.

Once the connection host confirms the connection, devices that have previously accepted the connection invitation join the connection by replacing or extending an existing connection, depending on the function that was used to accept the invitation.

When a device joins a connection, the ISI engine on that device updates the network configuration for the device, and the accepted connection becomes active.

The ISI engine automatically implements the connections for the accepted assembly. To determine the NVs to be connected, the ISI engine calls the **IsiGetNvIndex()** and **IsiGetNextNvIndex()** functions for each selector used with the connection.

You can provide feedback to the user when a connection has been joined, for example by turning off the Connect light. This is typically only done with manual connections. The ISI engine informs your application that a connection has been implemented by calling the **IsiUpdateUserInterface()** callback function. This allows the application to indicate the new connection.

EXAMPLE 1

The following manual enrollment example implements a connection on a connection host when the user presses the Connect button a second time. It turns off the Connect light to indicate the acceptance on the host.

```
void ProcessIsiButton(unsigned Assembly, boolean Constructive,
  boolean Alternative) {
    switch(isiState) {
        ...
        case isiApprovedHost:
            // An approved host is a connection host that has
            // received at least one enrollment acceptance
            // (CSME) message.
            IsiCreateEnrollment(Assembly);
            break;
            ... // Processing for other states
    }   // End of switch(state)
}

// IsiUpdateUserInterface is called with isiApprovedHost as the
// IsiEvent parameter in response to receiving a CSME;
// IsiUpdateUserInterface is called with isiImplemented as the
// IsiEvent parameter in response to invoking
// IsiExtendEnrollement() or IsiCreateEnrollment()
void IsiUpdateUserInterface(IsiEvent Event, unsigned Parameter) {
   if (Event == isiApprovedHost){
      SetConnectLed(LED_ON);
   }
   else if (Event == isiImplemented || Event == isiCancelled) {
      isiState = isiNormal;
      SetConnectLED(LED_OFF);
   }
   ...  //Processing for other events
}
```

Canceling a Connection

You can cancel a pending enrollment on the connection host at any stage, and on any device that has accepted the connection invitation. Cancellation is no longer possible once the connection is implemented; see *Deleting a Connection* for these cases.

Pending enrollment sessions are automatically cancelled if:

- On the connection host, if no connection enrollment acceptance message (CSME) is received within T_{enroll} after the **IsiOpenEnrollment()** function call.
- On the connection host, if the connection is not implemented by a **IsiCreateEnrollment()** or **IsiExtendEnrollment()** function call within T_{enroll} after the receipt of a connection enrollment confirmation message (CMSE).
- On an accepting device, if the connection has been accepted and no connection enrollment confirmation message (CMSC) has been received within T_{enroll} after the acceptance.

To explicitly cancel a pending enrollment, call the **IsiCancelEnrollment()** function.

When a connection host cancels a pending enrollment session, it issues a connection enrollment cancellation message (CSMX). Devices that have accepted the related connection invitation automatically cancel in result of receiving a related CSMX message.

When a connection member cancels a pending enrollment session, the cancellation only has local effect—the approved assembly changes to the **isiCancelled** state. Since the connection host may continue re-sending invitation messages (CSMOs), the same device may, once again, conditionally approve the assembly and move it to the **isiPending** state. The user may now accept the connection invitation once again (by causing the application to call

IsiCreateEnrollment() or **IsiExtendEnrollment()**), or simply do nothing. The pending assembly remains pending until the enrollment is closed, and automatically returns to the **isiNormal** state.

Deleting a Connection

You can delete an implemented connection using one of three methods:

- The device can restore factory defaults. To do so, call the **IsiReturnToFactoryDefaults()** function. This function clears all system tables, stops the ISI engine, and resets the device. See *Deinstalling a Device* and *Appendix B* for more details about this function.
- The device can delete a connection. To do so, call the **IsiDeleteEnrollment()** function. This function causes the connection information to be removed from the local device as well as on all other devices that are members of the same connection. The **IsiDeleteEnrollment()** function may be called on the connection host, and on any other device that has joined the connection.
- The device can opt out of an existing connection, leaving other devices that have joined the same connection unchanged. To leave a connection locally, call the **IsiLeaveEnrollment()** function. Calling this function on the connection host has the effect of **IsiDeleteEnrollment()**—a connection host cannot leave a connection, but must always delete the connection.

The ISI engine calls the **IsiUpdateUserInterface()** function with the **isiDeleted** event to notify the application on the completion of a deleting operation.

Handling ISI Events

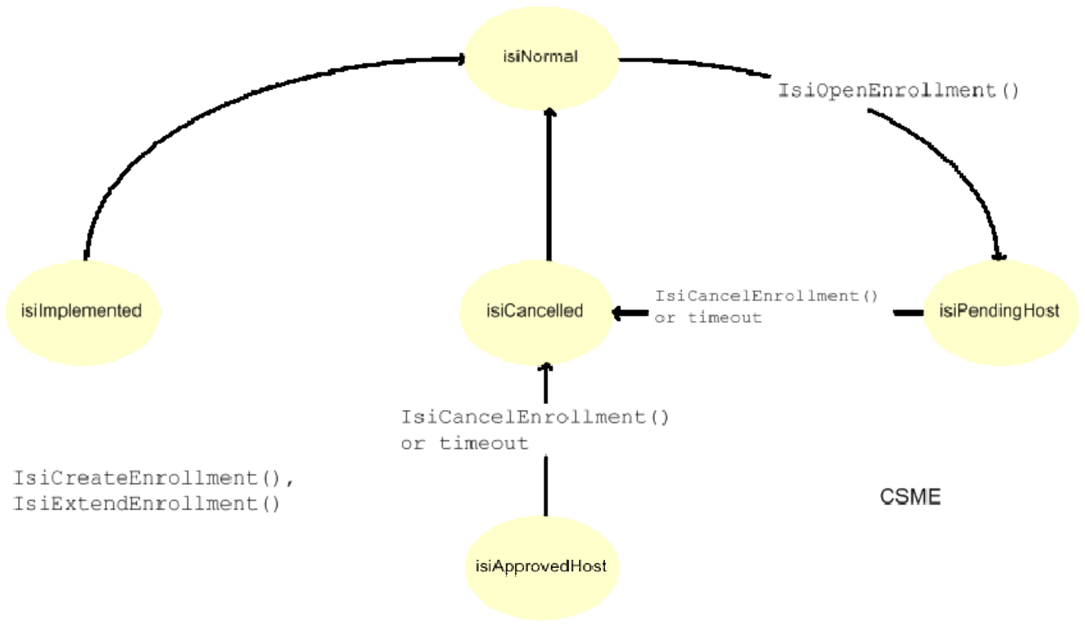
You can signal the progress of the enrollment process to the device user. This will typically be done for devices that use manual connections, since automatic connections do not require user interaction. User feedback may be as simple as a single Connect light and button, possibly shared with the Service light and button. A more complex gateway or controller may have a richer user interface. To receive status feedback from the ISI engine, override the **IsiUpdateUserInterface()** callback function. The ISI engine calls this function with the **IsiEvent** parameter set to one of the values in the following table when the associated event occurs. Some of these events carry a meaningful value in the numeric parameter, as detailed in the table.

IsiEvent	Value	Note
isiNormal	0	The ISI engine has returned to the normal, or idle, state for an assembly. The related assembly is encoded in the parameter; a parameter value of **ISI_NO_ASSEMBLY** indicates that the event applies to all assemblies.
isiRun	1	The ISI engine has been successfully started (**parameter** = 1) or stopped (**parameter** = 0).
isiPending	2	The connection related to the assembly given with the numerical parameter has entered the pending state. The event means the device has received and provisionally approved a connection invitation, but has not yet accepted the connection invitation. This event only applies to a connection member. For a connection host see **isiPendingHost**.
isiApproved	3	The connection related to the assembly given with the numerical parameter changed from the pending state to the approved state. This event occurs when a connection invitation has been provisionally approved and accepted. This event only applies to a connection member. For a connection host see **isiApprovedHost**.
isiImplemented	4	The connection related to the assembly given with the numerical parameter has been implemented. This event occurs on a connection host after calling **IsiCreateEnrollment()** or **IsiExtendEnrollment()** to implement a connection and close enrollment, and on a connection member after receiving an enrollment confirmation message (CSMC).
isiCancelled	5	The connection related to the assembly given with the numerical parameter has been cancelled by a timeout, user intervention, or network action. An assembly number of **ISI_NO_ASSEMBLY** indicates that all pending enrollments are cancelled.
isiDeleted	6	The connection related to the assembly given with the numerical parameter has been deleted.

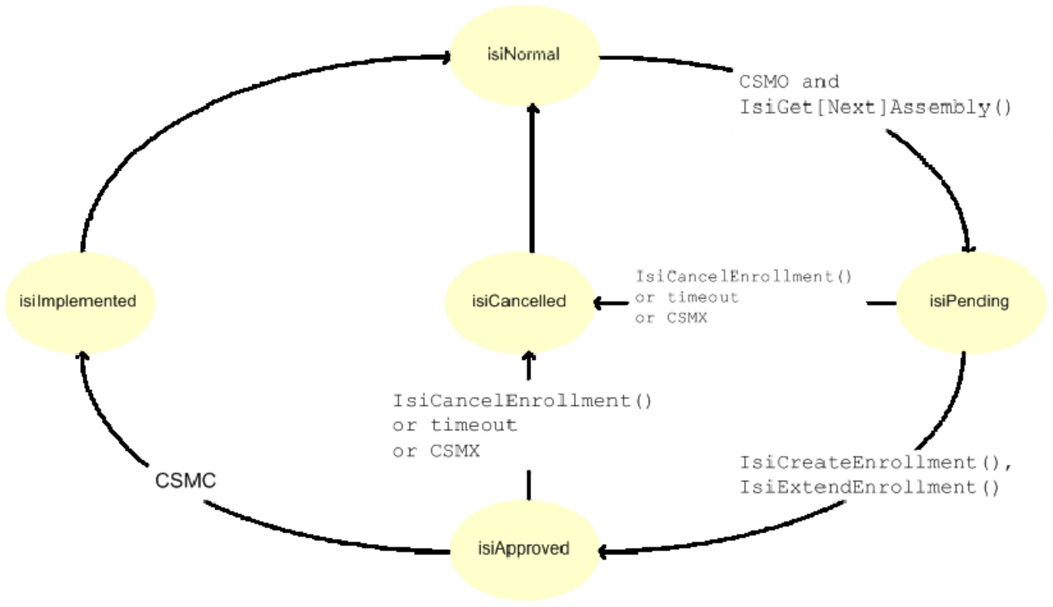
IsiEvent	Value	Note
isiWarm	7	The ISI engine has warmed up (i.e. a predetermined time with a random component has passed since the last reset). From this moment on, the application may call the **IsiInitiateAutoEnrollment()** function. This event occurs no sooner than the expiry of the $\mathbf{T_{auto}}$ ISI protocol timer, but may occur later.
isiPendingHost	8	The connection related to the assembly given with the numerical parameter has entered the pending state. This event occurs on a connection host after it has issued a connection invitation (CSMO) but not yet received any enrollment acceptance messages (CSMEs). This event only applies to a connection host. For a connection member see **isiPending**.
isiApprovedHost	9	The connection indicated with the numerical parameter changed from the pending state to the approved state. This event occurs on a connection host at the receipt of the first connection enrollment acceptance message (CSME). This event only applies to a connection host. For a connection member see **isiApproved**.
isiAborted	10	The device stopped domain or device acquisition. The parameter is a member of the **IsiAbortReason** enumeration and indicates the reason for the abort.
isiRetry	11	The device is retrying the device acquisition procedure. The parameter is the remaining number of retries.
isiWink	12	The device should perform its wink function. The specific function is application-dependent, but should provide some visible feedback to the user. For example, the application may blink an LED on the device.
isiRegistered	13	This event indicates either acquisition start or successful acquisition completion on either an ISI-DA or ISI-DAS device. The parameter indicates either a successful start (**parameter = 0**) or completion (**parameter = 0xFF**).

You can override the **IsiUpdateUserInterface()** callback function with an application-specific function to provide application-specific user feedback. The default implementation of this function does nothing, and is only useful for devices that exclusively use automatic enrollment.

The following figure summarizes the typical sequence of events for a connection host using manual or controlled enrollment. The sequence of events is similar for a connection host using automatic enrollment, except that the connection host will skip the **isiApprovedHost** event and go straight to the **isiImplemented** event. The sequence of events shown in this figure is typical—the actual sequence of events passed to the **IsiUpdateUserInterface()** callback may vary from this diagram.



The following figure summarizes the typical sequence of events for a connection member. As with the previous diagram, the sequence of events shown in this figure is typical—the actual sequence of events passed to the **IsiUpdateUserInterface()** callback may vary from this diagram.


isiNormal
CSMO and IsiGet[Next]Assembly()
isiImplemented
isiCancelled
IsiCancelEnrollment() or timeout or CSMX
isiPending
IsiCancelEnrollment() or timeout or CSMX
CSMC
IsiCreateEnrollment(), IsiExtendEnrollment()
isiApproved

You can get more detailed ISI diagnostic events. These events are useful for debugging ISI applications and are not typically used for production products. To receive notification of diagnostic events, enable diagnostics in the **IsiStart()** function by raising the **isiFlagSupplyDiagnostics** flag, and override the **IsiUpdateDiagnostics()** callback function. This callback is normally disabled and the default implementation of **IsiUpdateDiagnostics()** does nothing. The ISI engine calls this function with the **IsiDiagnostic** parameter set to one of the values in the following table when the associated event occurs—some of these events carry a meaningful value in the numeric parameter, as detailed in the following table:

IsiDiagnostic	Value	Note
isiSubnetNodeAllocation	1	A local subnet/node ID has been allocated.
isiSubnetNodeDuplicate	2	A duplicate subnet/node ID has been detected.
isiReceiveDrum	4	DRUM message received.
isiReceiveTimg	5	TIMG message received.
isiSendPeriodic	6	Periodic message other than an NV heartbeat message (see the **isiSendHeartbeat** event for these) sent. The parameter contains the ISI message code for the message sent.
isiSelectorDuplicate	7	NV selector duplicate has been detected. The parameter indicates the associated assembly.
isiSelectorUpdate	8	NV selector update has been detected. The parameter indicates the associated assembly.
isiReallocateSlot	9	Period broadcasting slot has been reallocated as a result of message spreading.

Deinstalling a Device

You can deinstall a device to remove all network configuration data, including the domain addresses, network addresses, and connection configurations. For devices that do not provide direct connection removal, this is the only way to remove a device from a connection. You can use this procedure to re-enable self-installation for an ISI device that was installed in a managed network. You can also use this procedure to return a device to a known state. You can deinstall a device to move it from a managed network to a self-installed network, or to move a self-installed device to a new self-installed network. All ISI devices must support deinstallation.

To deinstall a device, set the **SCPTnwrkCnfg** configuration property to **CFG_LOCAL** to enable self-installation and then call the **IsiReturnToFactoryDefaults()** function. You will typically deinstall a device in response to an explicit user action. For example, the user might be required to press and hold the service pin for five seconds to trigger deinstallation.

The **IsiReturnToFactoryDefaults()** function clears and reinitializes all system tables, stops the engine, and resets the device. Due to the device reset, the call to the **IsiReturnToFactoryDefaults()** function never returns.

EXAMPLE

The following example deinstalls a device after the service pin is held for a long period:

```
when (timer_expires(ServicePinHoldTimer)) {
   nciNetConfig = CFG_LOCAL;
   IsiReturnToFactoryDefaults();
}
```

Recovering from a Programming Error

The **IsiReturnToFactoryDefaults()** function described in the previous section assists with recovering from some programming errors. For example, you can safely remove incorrectly established connections with this tool. However, if the application enters a state that causes it to malfunction on an algorithm level, more application-specific code is needed to assist recovery from such a condition. A typical implementation is to monitor the status of the Service button (see the **service_pin_state()** function in the *Neuron C Reference Guide*) to determine if the Service button is activated continuously for a prolonged period, e.g., five seconds, and then call the **IsiReturnToFactoryDefaults()** function. This function call never returns, as it resets the device. Control first returns to the application within the **when (reset)** task. In this task, the application can also check the state of the Service button using the same **service_pin_state()** function. If the Service button is pressed while the device resets, the application can re-initialize the application's state and variables to return to orderly behaviour, and might use the Neuron C **active_service_led** built-in variable to signal completion of recovery operations without using any other application I/O.

Declaring Network Variable Arrays

Network variable arrays must be declared with the **bind_info(expand_array_info)** modifier if you use the default implementation of **IsiCreateCsmo()**, **IsiGetAssembly()**, or **IsiGetNextAssembly()**. This includes the forwardees **isiCreateCsmo()**, **isiGetAssembly()**, and **isiGetNextAssembly()** functions (*See Forwarders in Chapter 5*). Applications that override all of these forwarders with application-specific implementations that do not use the related forwardees do not need to specify the **expand_array_info** attribute. This attribute allows the ISI library to determine the SNVT type ID for every network variable on the device, at the expense of a larger amount of memory required for self-identification data.

Using the run_unconfigured Compiler Directive

ISI makes use of the **run_unconfigured** compiler directive to enable the device application to run without network configuration. This is declared in the **isi.h** standard header file and no extra steps are needed. Side-effects of this directive are described in the *Neuron C Reference Guide*.

Implementing a SCPTnwrkCnfg CP

ISI applications must implement a **SCPTnwrkCnfg** configuration property that is implemented as a configuration network variable. This configuration property must apply to your application's Node Object, if available, or apply to the entire device if there is no Node Object. This configuration property provides an interface for network tools to disable self-installation on an ISI device. This allows for the same device to be used in both self-installed and managed networks. The **cp_info(reset_required)** attribute is typically used with the declaration of the **SCPTnwrkCnfg** CP. This allows you to simply check the current ISI state in the device's **when (reset)** task. See the ISI example applications for examples.

The default value of the **SCPTnwrkCnfg** configuration property must be **CFG_EXTERNAL**. This allows for the device to be transitioned to a managed network without error. When setting this as the default for a device that will use self-installation, detect the first start with a new application image and, in this case only, change the value of the CP to **CFG_LOCAL** so that the ISI engine can come up running with the first power-up.

EXAMPLE

```
network input SCPTnwrkCnfg cp cp_info(reset_required) cpNetConfig
= CFG_EXTERNAL;

device_properties {
   cpNetConfig
};

eeprom SCPTnwrkCnfg oldNetConfig = CFG_NUL;

when (reset) {
   SCPTnwrkCnfg netConfig;
   netConfig = oldNetConfig;

   if (netConfig == CFG_NUL) {
      // First application start, enable self-installation
      cpNetConfig = CFG_LOCAL;
   }
   oldNetConfig = cpNetConfig;

   if (cpNetConfig == CFG_LOCAL) {
      if (netConfig == CFG_EXTERNAL) {
         // Managed application has returned to self-installation
         IsiReturnToFactoryDefaults(); // Call NEVER returns!
       }
      // Self-installed network--start the ISI engine
      scaled_delay(31745UL); // 800ms delay
      IsiStartS(isiFlagExtended);
   }
}
```

4

Quick Start, Revisited

This chapter revisits the quick start example in Chapter 1 to add connection enrollment support and user feedback.

Here is the quick start example from Chapter 1:

```
#include <isi.h>

when (reset) {
    // Clear all tables and start the ISI engine
    scaled_delay(31745UL); // 800ms delay
    IsiStartS(isiFlagNone);
}

mtimer repeating isiTimer = 1000ul / ISI_TICKS_PER_SECOND;

when (timer_expires(isiTimer)) {
    // Call the ISI engine to perform periodic tasks
    IsiTickS();
}

when (msg_arrives) {
    if (IsiApproveMsg()) {
        // Process an incoming ISI message
        (void) IsiProcessMsgS();
    }
}
```

While this application can attach to a network, send status messages, and receive the status messages from other devices, it has no way of establishing a connection to any other device in the network. It also has no way of interacting with the user, which means it cannot relay information to the user, nor can it receive input from the user. This chapter expands the example to implement support for one connection, with simple user interaction of the connection status.

To support connections, you must define at least one network variable, call **IsiOpenEnrollment()** to open manual or controlled enrollment, call **IsiInitiateAutoEnrollment()** to start automatic enrollment, and call **IsiCreateEnrollment()** or **IsiExtendEnrollment()** to accept a connection invitation for manual or controlled enrollment. For the example, one network variable is defined that controls the state of a light. This network variable is created with the following code:

```
network input SNVT_switch nviLight;
```

The **IsiCreateEnrollment()** or **IsiExtendEnrollment()** call is typically made when the user activates the Connect button for an assembly that is in the pending state. To track the state for each assembly, you can override the **IsiUpdateUserInterface()** callback function. Because the example application supports only one assembly, state tracking is simple and can be implemented with a single variable.

The following code overrides the **IsiUpdateUserInterface()** callback function to get updates of the state of the ISI engine:

```
// Last known state variable.  This is used when providing
// user input back to the ISI engine (below)
IsiEvent deviceState = isiNormal;

// IsiUpdateUserInterface() override
```

```
void IsiUpdateUserInterface(IsiEvent event,
                            unsigned parameter) {
   if (event == isiPending || event == isiPendingHost) {
      SetConnectLed(LED_BLINKING);
   } else if (event == isiApproved ||
              event == isiApprovedHost) {
      SetConnectLed(LED_ON);
   } else {
      SetConnectLed(LED_OFF);
   }
   deviceState = event;
}
```

SetConnectLed() is a device-specific function that is defined elsewhere. The function sets the device's LED into the three states mentioned above. While the default implementation of the **IsiUpdateUserInterface()** function contained in the ISI implementation library does nothing, the overridden implementation above connects the ISI engine with the user interface—a single LED in this case.

The second parameter of **IsiUpdateUserInterface()** generally indicates the assembly number to which the event applies. Since the sample device only has one assembly, the parameter is disregarded here.

ISI requires no specific user-interface, but a simple user interface is typically required for devices that implement manual connections. A minimal user interface can be implemented by monitoring state changes to **isiPendingHost**, **isiPending**, **isiApprovedHost**, **isiApproved**, and **isiNormal** for each assembly. To relay a user command to the ISI engine, you can detect the related user interface operation and call the related ISI command function. For a small device this user input device can be as simple as a single Connect button, which may be in one of three states: 1) not pressed, 2) pressed for a short while, or 3) pressed for a prolonged period of time. This single-button input is often combined with the current state of the related assembly, as indicated with the **DeviceState** tracking variable introduced above.

The following table summarizes the response to a button press, based on the **DeviceState** value.

DeviceState	Connection Host	Response to Connect Button
isiPendingHost	Yes	A pending connection host is one that has issued a connection invitation (CSMO message) but has not yet received a single CSME. In this state, only cancellation is a valid operation. Call **IsiCancelEnrollment()** to cancel the open enrollment.
isiPending	No	A pending connection member is one that has received and provisionally approved a connection invitation. Call **IsiCreateEnrollment()** or **IsiExtendEnrollment()** to accept the connection.

DeviceState	Connection Host	Response to Connect Button
isiApprovedHost	Yes	An approved connection host is one that has received at least one connection enrollment acceptance message (CSME). Call **IsiCreateEnrollment()** or **IsiExtendEnrollment()** to implement the connection. Call **IsiCancelEnrollment()** to cancel the enrollment.
isiApproved	No	An approved connection member is one that awaits a connection enrollment confirmation (CSMC) or cancellation (CSMX) message from the host. Call **IsiCancelEnrollment()** to cancel the acceptance (opt out of the open enrollment).
isiNormal	N/A	In the normal state, devices can become hosts (**IsiOpenEnrollment()**) or leave or delete existing connections through **IsiLeaveEnrollment()** or **IsiDeleteEnrollment()**.

In the example code below, the application Connect button **when** task calls enrollment functions based on the **DeviceState**, which was set in the overridden **IsiUpdateUserInterface()** function above. The **if (longPress)** test represents an action that would happen if the button was being held down for an extended period of time, with the implementation details not shown here.

```
when (io_changes(...)) {
   switch(deviceState) {
      case isiPendingHost:
          if (longPress) {
              IsiCancelEnrollment();
          }
          break;
      case isiPending:
          IsiCreateEnrollment(assembly);
          break;
      case isiApprovedHost:
          if (longPress) {
              IsiCancelEnrollment();
          } else {
              IsiCreateEnrollment(assembly);
          }
          break;
      case isiApproved:
          if (longPress) {
              IsiCancelEnrollment();
          }
          break;
```

```
            case isiNormal:
               if (longPress) {
                  IsiDeleteEnrollment(assembly);
               } else {
                  IsiOpenEnrollment(assembly);
               }
               break;
            }
         }
      }
```

With the refinements to the quick start example introduced above the resulting device will do the following:

- Participate in and honor all ISI messages relating to the device's subnet/node address.
- Use timing guidance received from a domain address server, if available, and the application is linked with the **IsiCompactS**, **IsiCompactSHb**, **IsiCompactDa**, **IsiCompactDaHb**, or **IsiFull** library; see *Optimizing the Footprint of ISI Applications* for more details about the different ISI libraries.
- Open enrollment for connections with a width of 1, offering a connection with a **SNVT_switch** network variable.
- Provisionally approve open enrollment messages for manual connections that relate to a single standard network variable with a matching **SNVT_switch** type.

A typical ISI application will proceed and provide further customization. Such customization serves two purposes:

- A customized application can support more complex connection types, such as complex compound assemblies or manufacturer-specific connections.
- Customized applications can be tailored to specific application needs. Customizing ISI can lead to smaller memory footprint, compared to the generic default implementations.

Further customization is discussed in *Self-Installation Advanced Procedures*.

5

Developing and Debugging an ISI Application

This chapter describes how to develop and debug applications using the Neuron C ISI Library.

General Considerations

You can develop applications for the Neuron C ISI library using the NodeBuilder® 3.1 Development Tool or the Mini EVK Evaluation Kit. The Neuron C ISI Library is included with Mini EVK, and is available as a free download for the NodeBuilder 3.1 Development Tool. New updates for the Neuron C ISI Library are posted periodically, so check www.echelon.com/isi for the latest version before starting a new development.

The NodeBuilder tool includes the LonMaker Integration Tool, and the LonMaker tool is used to create a managed network when using the NodeBuilder tool. As a result, without special debugging considerations, self-installation will be disabled for your devices when you try to debug them with the NodeBuilder tool. You can also encounter the same problem if you use the LonMaker tool with the Mini EVK to take advantage of the network debugging capabilities of the LonMaker tool. This chapter describes the special considerations required when debugging an ISI application in a managed network.

As described in Chapter 2, ISI applications use a **SCPTnwrkCnfg** configuration property to enable and disable self-installation. With this configuration property, self-installation functions may be enabled or disabled at any time. When testing an ISI application, test the application with and without self installation enabled (i.e. both settings for the **SCPTnwrkCnfg** configuration property), as the ISI engine impacts the network behavior as well as the timing of the application algorithm.

When developing and debugging an ISI device, there are two aspects of the device that must be developed and debugged—the device application and the ISI implementation.

Developing and Debugging the Application

When developing and diagnosing the application algorithm, such as a washing machine's or compressor's control algorithm, the application may enable or disable the ISI engine. If the ISI engine remains enabled during this work (the **SCPTnwrkCnfg** configuration property remains set to **CFG_LOCAL**), see the next section for considerations. Typically, the operation of the ISI engine will be irrelevant to the correct operation of the application algorithm, and the **SCTPnwrkCnfg** configuration property may be set to **CFG_EXTERNAL** to disable the ISI engine. In a managed environment, the network management tool such as the LonMaker tool will automatically set the **SCPTnwrkCnfg** configuration property to **CFG_EXTERNAL** when commissioning your device. This will disable the ISI engine and prevent any interference with the NodeBuilder tool.

Developing and Debugging the ISI Implementation

You can develop and debug ISI applications with the NodeBuilder Development Tool or the Mini EVK Evaluation Kit. The Mini EVK is fully compatible with ISI applications and does not require any special considerations for development.

To enable debugging with the managed environment provided by the NodeBuilder and LonMaker tools, you must ensure that an ISI application in development cannot modify the primary domain. The primary domain and the device's network address are managed by the NodeBuilder tool; modifying this data as a result of self-installation will prevent testing and diagnosing with the NodeBuilder or LonMaker tools.

To prevent the primary domain from being overridden when using the NodeBuilder tool, override the **IsiSetDomain()** function with one that does nothing, for a debug target as shown in the following example:

```
#ifndef _MINIKIT
#ifdef _DEBUG
void IsiSetDomain(domain_struct* pDomain, unsigned Index) {
   ;
}
#endif
#endif
```

The ISI engine will call this function whenever the primary domain must be updated. The default implementation routes this call to the **update_domain()** standard Neuron C library function. With an override as shown here, an attempt to update the primary domain has no effect.

WARNING: The **IsiSetDomain()** override shown here will disable important aspects of the ISI implementation. This override cannot be used with production-level devices, or devices that are to be used outside the managed NodeBuilder environment.

When using the **IsiSetDomain()** override in the above fashion to allow for development, testing, and debugging of ISI-related code of a self-installed device within a managed NodeBuilder environment, a few other restrictions exist:

The LonMaker Browser will not work with self-installed devices when the ISI engine is running. While monitoring network variable values may work correctly, updating network variable values may not work.

In a typical ISI application, the **SCPTnwrkCnfg** configuration property must be set to **CFG_LOCAL** to enable the ISI code. If it is not, then the ISI engine will be disabled. In the debug environment, you can force the ISI engine to always be running, bypassing the **SCPTnwrkCnfg** CP setting. The following example shows how to do this:

```
network input cp SCPTnwrkCnfg nciNetConfig = CFG_LOCAL;

when (reset) {
#ifndef _DEBUG
   if (cpNetConfig == CFG_LOCAL)
#endif
   {
       scaled_delay(31745UL); // 800ms delay
       IsiStartS();
   }
}   // when (reset)
```

The development network may contain other managed devices. However, managed and self-installed connections cannot coexist within the same domain since the different allocation algorithms used for managed and self-installed connections could result in duplicate resources being assigned. Managed devices and self-installed devices must always be installed in different domains.

The NodeBuilder tool is the best tool for developing and debugging Neuron C applications, but there are three other tools that you can use to aid debugging:

- The ISI Packet Monitor Application is a free utility included with the ISI Developer's Kit. You can use it to monitor and decode most ISI messages. To start the ISI Packet Monitor Application, open the Windows **Start** menu, point to **Programs** > **Echelon Interoperable Self-Installation**, and then click **ISI Monitor Packet Application**. After the Network Interface windows opens, select your channel type and network interface, and then click **Connect**. The ISI Packet Monitor window appears. ISI packets received by the selected network interface are interpreted and listed in the lower pane. You can highlight text in the lower pane, right-click the highlighted text, and then click **Copy** on the shortcut menu to copy the log to the Windows clipboard. You can select one of three tabs in the upper pane to show a summary of devices reporting DRUMs, a summary of connections reporting CSMx's, or a summary of ISI message statistics.

- The NodeUtil Device Utility is a free utility that is available for download from www.echelon.com/downloads. This application allows you to see various aspects of a device, including memory and network variable tables, and also allows you to view and update network variables.

- The LonScanner Protocol Analyzer is an application for monitoring and analyzing low-level network traffic received by a network interface. A free trial edition of the LonScanner tool is included with the Mini EVK Evaluation Kit. You can also download a free trial edition from www.echelon.com/lonscanner. The trial edition runs for a limited time and throws away some of the received packets. You can order a LonScanner activation key to remove the time limit and unlock the full functionality.

6

Self-Installation Advanced Procedures

This chapter describes advanced ISI procedures that are not typically used by all ISI applications, but support additional features not supported by the basic ISI procedures described in Chapter 2. The functions described in this chapter are further described in Appendixes B and C, and the data structures used by these functions are documented in Appendix A.

Overriding a Callback Function

You can have the ISI engine call functions in your application in response to key ISI events. These functions are called *callback functions*, because the ISI engine calls back to your application. To simplify the use of callback functions, the ISI library includes implementations of all ISI API callback functions. As a result, you do not have to provide any callback functions. For example, the ISI library contains a default implementation of the **IsiUpdateUserInterface()** callback function. In this case, the default implementation does nothing. To create your own **IsiUpdateUserInterface()** function, override the function with your own application-specific implementation.

Other common callback functions are the **IsiGetAssembly()** and **IsiGetNextAssembly()** functions. You can override these functions to create application-specific connections.

To override a default implementation, redefine the function with a matching name and prototype in your application. When the linker recognizes a function within the application space, it will no longer link the implementation contained in the library.

When overriding a callback function, avoid calling ISI functions or initiating time-consuming operations from the override function. Doing so may cause the ISI engine to function incorrectly. The exceptions to this are when the function simply returns a flag, such as **IsiIsRunning()** and **IsiIsBecomingHost()**, or when the function is intended to be called from a callback function, like **IsiInitiateAutoEnrollment()**.

The following table lists the callback functions implemented in the ISI library and the reasons to override each one.

Function Name	**Reason to Override**
IsiCreateCsmo()	Override this function to host connections that offer more than the default information, which is the following: the group ID returned by the **IsiGetPrimaryGroup()** function, the **Application** field determined by the device's program ID, the **Width** field set by the value returned by **IsiGetWidth()**, the **Direction** field set to **isiDirectionAny**, the **NvType** field set to the primary network variable's SNVT ID, and all other fields set to zero.yyy
IsiCreatePeriodicMsg()	This function is used to determine when a device should send a periodic message. It should only be overridden if a device needs a hook into the periodic message scheduler.

Function Name	Reason to Override
IsiGetAssembly()	This function should be overridden for applications with compound assemblies involving multiple network variables, connections involving functional blocks, or automatic enrollment. The default implementation returns the assembly number as the network variable's global index, if a compatible network variable exists for a simple connection using standard network variable types. The default implementation always returns **ISI_NO_ASSEMBLY** for an automatic enrollment request. To support joining a connection with automatic enrollment, this callback function must be overridden. When overriding this function, also consider overriding the **IsiGetNextAssembly()** function.
IsiGetConnection()	This function must be overridden when implementing a custom connection table. When this function is overridden, **IsiSetConnection()** and **IsiGetConnectionTableSize()** must also be overridden. The default implementation of the connection table has eight entries stored in EEPROM. The connection table functions are typically overridden to create a connection table with a size fitting the application, or with a persistent data storage model that fits the application's needs. See *Customizing the ISI Connection Table* for more information.
IsiGetConnectionTableSize()	This function must be overridden when implementing a custom connection table. When this function is overridden, **IsiSetConnection()** and **IsiGetConnection()** must also be overridden. The default implementation of the connection table has eight entries stored in EEPROM. The connection table functions are typically overridden to create a connection table with a size fitting the application, or with a persistent data storage model that fits the application's needs. *See Customizing the ISI Connection Table* for more information.

Function Name	Reason to Override
IsiGetNextAssembly()	This function should be overridden when providing for compound assemblies involving multiple network variables, connections involving functional blocks, or automatic enrollment. The default implementation returns the next assembly number if a complementary network variable exists for a simple connection, using standard network variable types. The default implementation of this function always returns **ISI_NO_ASSEMBLY** for an automatic enrollment request. To support joining a connection with automatic enrollment, this callback function must be overridden. When overriding this function, also consider overriding the **IsiGetAssembly()** function.
IsiGetNextNvIndex()	This function must be overridden in order to share a single selector among multiple network variables. The default implementation always returns **ISI_NO_INDEX**. This function is rarely overridden, but if you do, consider also overriding the **IsiGetNvIndex()** function.
IsiGetNvIndex()	This function must be overridden in order to provide for compound assemblies involving multiple network variables. The default implementation returns *Assembly* + *Offset*. This function is commonly overridden. Also consider overriding the **IsiGetNextNvIndex()** function.
IsiGetPrimaryDid()	This function can be overridden when using a non-standard domain ID as the default domain ID. The default implementation returns a 3 byte domain, the ASCII values of the characters “ISI”. Devices using this override may not be ISI compatible.
IsiGetWidth()	This function must be overridden when an assembly has a width greater then 1. The default implementation always returns 1.

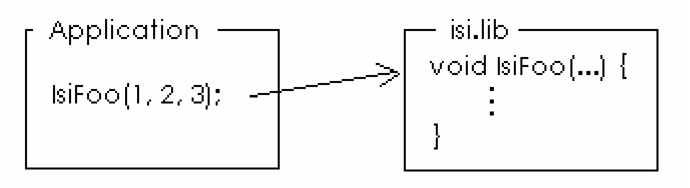
Function Name	Reason to Override
IsiSetConnection()	This function must be overridden when implementing a custom connection table. When this function is overridden, **IsiGetConnectionTableSize()** and **IsiGetConnection()** must also be overridden. The default implementation of the connection table has eight entries stored in EEPROM. The connection table functions are typically overridden to create a connection table with a size fitting the application, or with a persistent data storage model that fits the application's needs. See *Customizing the ISI Connection Table* for more information.
IsiSetDomain()	This function should only be overridden to allow ISI to run in a managed development and debugging environment by disabling the ability to change the domain. See *Developing and Debugging an ISI Application* for more information and important considerations.
IsiUpdateDiagnostics()	This function is used to provide feedback about the internal state of the ISI engine. The application may override this function for enhanced diagnostics during development and testing. This function is rarely overridden and should only be used in a debugging or testing environment. This function may not be supported by all ISI libraries.
IsiUpdateUserInterface()	This function is used to provide feedback about the state of the ISI engine to the user. It should be overridden in any application that needs to provide feedback to a user.

Forwarders

You can create your own compatible implementations of most of the ISI library functions, which will then be used instead of the library function. You may frequently override some of the functions. For example, the **IsiGetAssembly()** function determines if an incoming open enrollment message describes an acceptable connection for this device, and to which local assembly the enrollment could apply. The ISI library contains default implementations of these functions in order to simplify development; however, most developers will tailor the ISI implementation by overriding the default implementations. You cannot replace the functions that indicate they cannot be overridden in Appendix B.

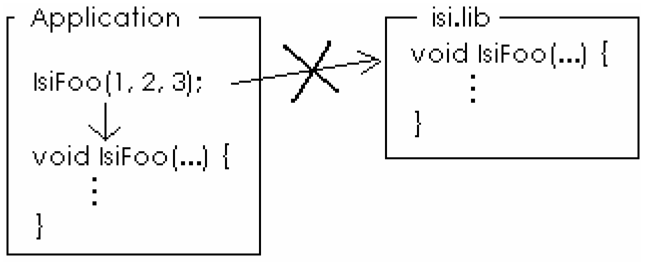
The C language has no concept of overloaded functions, so an overridden function can no longer call the default implementation: as both have the same name, calling the function with the same name will result in a recursive function call.

The following illustrates the standard C library case: a library provides a utility "**IsiFoo()**," which an application may call:



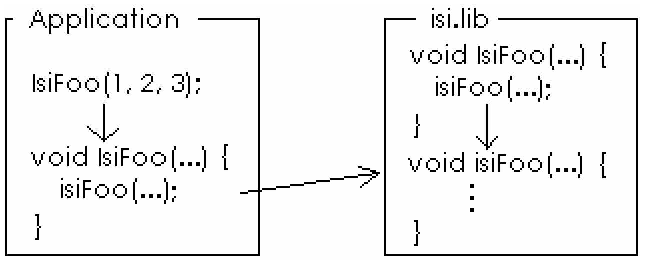
Standard Library Case

Overloading the utility function with an application-specific implementation prevents accessing the implementation provided with the library:



Overridden Library Function

You can use a *forwarder* to override an ISI function and call its default implementation at the same time. You can choose not to use the default implementation, or you can provide additional functionality and continue to call the default implementation. Every ISI function that supports forwarding has a sister function with the same definition that starts with a lower-case "i". For example, **IsiFoo()** has a sister function called **isiFoo()**. The API is defined as the plain function (i.e. **Isi *()**). If you choose to override this function, your override code may still call the **isiFoo()** function to benefit from the default implementation:



Forwarders are implemented in a way that eliminates any overhead for the indirection; there is no benefit in calling the sister function directly other than from within an overridden function.

Forwarders are often used to provide an application-specific implementation in a certain aspect, and to fall back to standard behavior in all other cases. For example, consider this override of the **IsiGetWidth()** function:

```
unsigned IsiGetWidth(unsigned Assembly) {
   return Assembly == SPECIAL ? 3 : isiGetWidth(Assembly);
}
```

The overridden function provides a width of 3 for a single, special, assembly, and returns the standard width for all other assemblies.

Assembly Number Allocation

Assembly numbers must be in the 0 – 254 range and sequentially assigned by the device application starting at 0. Required assemblies for standard profiles must be first, assigned in the order the profiles are declared in the application. Standard ISI profiles that define multiple assemblies must specify the order the assemblies are to be assigned.

Many of the default ISI functions use a default assembly numbering scheme where the assembly number is equal to the associated network variable's index. For assemblies containing multiple network variables, the lowest index of all associated network variables is used.

The following ISI functions use the default assembly numbering scheme. These functions are forwardees. If you create a custom assembly number scheme, you must override all of the related forwarders to use your scheme, and may no longer use these forwardees:

- isiGetNvIndex()
- isiGetNextNvIndex()
- isiCreateCsmo()
- isiGetAssembly()
- isiGetNextAssembly()

Supporting Compound Assembly Connections

You can support connections of compound assemblies with multiple network variables, either in a single functional block or multiple functional blocks, or without any functional blocks at all. The default implementation of ISI functions such as **IsiGetAssembly()**, **IsiGetNextAssembly()**, **IsiCreateCsmo()**,

IsiGetNvIndex(), and **IsiGetWidth()** handle connections of simple assemblies referring to a single network variable of a standard network variable type.

For example, a device supporting a compound assembly may implement a closed-loop actuator with an input and output network variable—the actuator is an implementation of the **SFPTclosedLoopActuator** profile called **myClosedLoopActuator** with a **SNVT_amp** input and output called **nviValue** and **nvoValue**. The following code opens enrollment for connections to the **myClosedLoopActuator** functional block:

```
IsiOpenEnrollment(myAssemblyNumber);
```

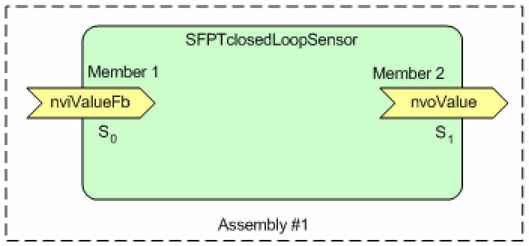
In turn, the ISI engine calls the **IsiGetWidth()** and **IsiCreateCsmo()** callbacks, which you may override to honor this particular assembly:

```
unsigned IsiGetWidth(unsigned Assembly) {
   // Return 2 for the myClosedLoopActuator assembly
   return Assembly == myAssemblyNumber ?
      2 : isiGetWidth(Assembly);
}
```

When a device becomes a connection host for a compound assembly, the device issues an open enrollment message (CSMO), which includes the first selector used in this enrollment **S_0**, and the number of network variable selectors used with this enrollment **Width**. For compound assemblies, i.e. assemblies with **Width** > 1, devices that accept the connections must understand how to apply the **Width** different network variable selector values to the local network variables.

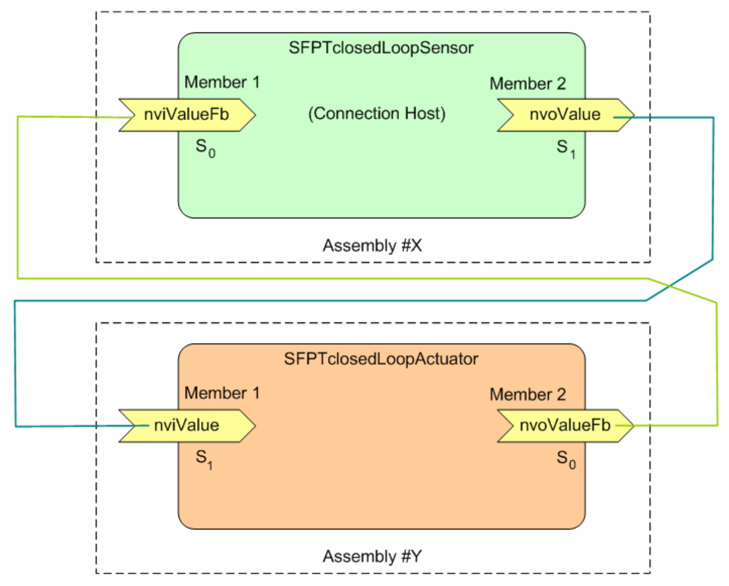
An accepting device derives that knowledge from the CSMO. For example, if the CSMO refers to a standard functional profile that is recognized by the receiving device, and if the CSMO's **Variant** field is zero, the receiving device knows that the enrollment contains **Width** selector values starting with S_0, where subsequent selector values S_1, S_2 … $S_{Width-1}$ follow sequentially to a maximum of 0x2FFF, with any following selector values continuing at 0 (unless a different mapping is specified in the profile). If not otherwise specified in the profile, the selector values are applied to the host's network variables in rising order of the network variable member number within the functional profile, starting with the member number contained in the CSMOEX's **Member** field.

In case the host offers a **SFPTclosedLoopSensor** standard functional profile with **Width** 2, the host will therefore apply the selector values to the network variables as shown here:



Assembly Member Allocation

With that knowledge, an accepting device that implements a **SFPTclosedLoopActuator** functional block must associate S_1 with its local input network variable, and S_0 with its output network variable. The completed connections are shown next:



Compound Assembly Connections

When the accepting **SFPTclosedLoopActuator** functional block implements the connection, the ISI implementation will query the mapping from selector values to network variables using the **IsiGetNvIndex()** callback.

The **IsiGetNvIndex()** function provides an assembly number and offset. The application must return the index of the matching network variable, or **ISI_NO_INDEX** if no matching network variable exists.

In this example, the **IsiGetNvIndex()** function will be called twice on the connection host because the CSMO indicated a width of 2. The **Assembly** parameter will be set to *Y* (which was provided through **IsiGetAssembly()** callback earlier). When the callback occurs with offset 0, the application returns the global index of the input network variable. The index of the output is provided for offset 1. The code regarding **AmBecomingHost** is included to avoid having an input network variable on the first device having the same selector value as the input network variable on the second device, and an output network variable on the first device having the same selector value as the output network variable on the second device:

```
unsigned IsiGetAssembly(...) {
   if ( ... ) {
      // Recognized SFPTclosedLoopSensor with width 2, variant 0,
      // and a compatible network variable type
      return Y;
   }
   ...
}

unsigned IsiGetNvIndex(unsigned Assembly, unsigned Offset) {
   unsigned Result;
   if (Assembly == Y) {
      if (Offset) {
         Result = AmBecomingHost ? nviValue::global_index
                                  : nvoValue::global_index;
      } else {
         Result = AmBecomingHost ? nvoValue::global_index
                               : nviValue::global_index;
      }
   } else {
      Result = isiGetNvIndex(Assembly, Offset);
   }
   return Result;
}
```

The application may also map multiple network variables to the same selector with the **IsiGetNextNvIndex()** callback. The default implementation of **IsiGetNextNvIndex()** returns **ISI_NO_INDEX**, indicating that only one network variable, the one indicated with **IsiGetNvIndex()**, applies to this assembly and offset value pair.

Mapping multiple network variables to a single selector value is an advanced feature that is only used with standard functional profiles that specify its use, or with manufacturer-specific connections. When using this feature, you must be able to ensure that selector values remain unique within the set of input network variables.

When using multiple functional blocks, the majority of the code is the same as the code for a single functional block. There is no difference between referencing a network variable as a global variable and as a functional block member variable, i.e. **nvoOne::global_index** is the same as **fbOne::nvoValue::global_index**, as long as **nvoOne** implements **nvoValue** in **fbOne**.

Creating a Polled Connection

You can create a polled connection where devices with an input network variable in the connection poll one or more output network variables in the connection. Two fields in the CSMOEX message are used to support polled output NVs. One field indicates the direction of the network variable that may be subscribed to the connection on offer. The direction field may have any of the following values: **isiDirectionInput**, **isiDirectionOutput**, **isiDirectionAny**, and **isiDirectionVarious**. The second field indicates a poll Boolean attribute, which is normally cleared. This allows for the following combinations:

1. Connection host enrolls local, normal, input network variable: direction(CSMOEX) = **isiDirectionOutput** or **isiDirectionAny**, poll(CSMOEX) = clear;

2. Connection host enrolls local, normal, output network variable: direction(CSMOEX) = **isiDirectionInput** or **isiDirectionAny**, poll(CSMOEX) = clear;

3. Connection host enrolls local, polling, input network variable: direction(CSMOEX) = **isiDirectionOutput**, poll(CSMOEX) = set;

4. Connection host enrolls local, polled, output network variable: direction(CSMOEX) = **isiDirectionInput**, poll(CSMOEX) = set.

The **isiDirectionAny** value is used in unspecific connections; for example, between multiple switches and multiple lamps. Most connections will be direction-specific, whereas many general-purpose I/O devices will support unspecific-direction connections to simplify the manual-connection scenario. Devices receiving a CSMO include the direction and poll attributes in the consideration of acceptance.

You can use the **isiDirectionVarious** value when the compound assembly contains multiple network variables of different directions, where the network variable direction is determined by the functional profile.

EXAMPLE

The following example creates a CSMOEX with the direction field set to output and the poll field set to on. The **isiFlagExtended** flag is required for transmitting this CSMOEX (but not for receiving).

```
static const IsiCsmoData MyCsmo = {
       ISI_DEFAULT_GROUP,      // Group
       isiDirectionOutput,     // NV direction
       1,                      // Width
       2,                      // Profile number (2 =
                               //     SFPTopenLoopSensor)
       76u,                    // Network variable type index
                               //     (76 = SNVT_freq_hz)
       0,                      // Variant (0 = standard)
       0,                      // Acknowledged
       1,                      // Poll
       isiScopeStandard,       // Scope that defines FPT or NVT
       {0x9F, 0xFF, 0xFF, 0x05, 0x00, 0x05},  // application
       1                       //  network variable member
                               //     number (1 = nvoValue)
};
```

Creating an Acknowledged Connection

You can create a unicast acknowledged connection. The ISI protocol does not support multicast acknowledged connections. An acknowledged connection requires subnet/node ID addressing instead of the group ID addressing typically used by ISI for network variable connections. Devices initiating acknowledged network variable updates must track subnet/node ID allocation using some suitable application-specific algorithm. The ISI library does not provide an implementation of these algorithms, but applications are free to supply support for acknowledged service. Acknowledged service is typically not used because of the additional complexity of tracking subnet/node ID changes. To send an NV update using the acknowledged service, create an explicit message that is formatted as a network variable update. Include an explicit address for the message with addressing details from an application-level address table and identify the message as using the acknowledged service, and then explicitly send the message.

EXAMPLE

The following example shows the construction of a CSMOEX for an acknowledged connection, and monitors DRUMs to update subnet and node IDs in a device that allows connections to one other device. The **isiFlagExtended** flag is required for transmitting this CSMOEX (but not for receiving).

```
static const IsiCsmoData myCsmoData = {
       ISI_DEFAULT_GROUP,      // Group
       isiDirectionOutput,     // NV direction
       1,                      // Width
       2,                      // Profile number (2 =
                               //     SFPTopenLoopSensor)
       76u,                    // Network variable type index
                               //     (76 = SNVT_freq_hz)
       0,                      // Variant (0 = standard)
       1,                      // Acknowledged
       0,                      // Poll
       isiScopeStandard,       // Scope that defines FPT or NVT
       {0x9F, 0xFF, 0xFF, 0x05, 0x00, 0x05},  // application
       1                       // Network variable member
                               //     number (1 = nvoValue)
};

typedef struct {
   unsigned   NeuronId[NEURON_ID_LEN];
   unsigned   subnet;
   unsigned   node;
} AddressTracker;

AddressTracker eeprom addressTracking = {
   {0,0,0,0,0,0}, 0, 0
};

when (msg_arrives) {
   IsiMessage msg;
   if (IsiApproveMsg()) {
       // Prior to calling IsiProcessMsg(), pre-process DRUM
       // messages by updating the AddressTracker
       memcpy(&msg, msg_in.data, sizeof(IsiMessage));
       if (msg.Header.Code == isiDrum) {
           if (memcmp(addressTracking.NeuronId,
                msg.Msg.Drum.NeuronId, NEURON_ID_LEN) == 0) {
              addressTracking.subnet = msg.Msg.Drum.SubnetId;
              addressTracking.node = msg.Msg.Drum.NodeId;
           }
       }
       // Call the ISI message handler
       (void) IsiProcessMsgS();
   }
}
```

The **addressTracking** variable maintains the current subnet/node ID pair for the device with the Neuron ID that is also kept in the **addressTracking** variable. The

Neuron ID to be tracked must be discovered and written to the **addressTracking** variable when a connection invitation is accepted. When sending an (acknowledged) network variable message, this message is explicitly constructed like an application message, with explicit addressing provided using address details from the **addressTracking** variable.

Turnaround Connections

You can create a turnaround connection on an ISI device. A *turnaround connection* is a connection between two network variables on the same device. The following limitations apply for ISI turnaround connections:

- The connection host for the turnaround connection must be the device containing the connected network variables.
- Only one local assembly can enroll with the host assembly, but assemblies on other devices may also be enrolled.
- The host assembly must contain at least one output network variable.
- Applications overriding the **IsiGetNextNvIndex()** function to apply multiple network variables to a shared selector may only support ISI turnaround connections if these multiple network variables are all output network variables.
- ISI turnaround connections can only be created with manual or controlled enrollment. You cannot create turnaround connections with automatic enrollment.

An alternative to creating a turnaround connection is to create an application-level connection. This is a connection where your application passes the data written to an output network variable to the function or when task that reads an input network variable. An application-level connection is easy to implement and requires fewer network resources such as connection table space or NV selectors. The primary disadvantage of an application-level connection is that it is invisible to any network management tools that may later be used to transition an ISI network to a managed network.

To create a manual turnaround connection, select one host assembly on a device, and then add at most one member assembly per device and enrollment. If this member assembly is on the hosting device, a turnaround connection will be created.

Customizing the ISI Connection Table

You can customize the *ISI connection table*. The connection table is a table containing the state and details of all connections that the device has joined. The ISI engine maintains this table and uses the information in this table to host or join connections. By default, the ISI implementation provides a small connection table with eight entries suitable for devices with limited resources.

The connection table contains one entry for each simple connection the device has joined, and for each simple extension to a connection. Connections and connection extensions from compound assemblies require *N* connection table entries, with *N* being **1 + (***Width* **/** 4).

You can replace the connection table to support more simultaneous connections or to support more complex connections, requiring a larger connection table. You can also replace the default connection table to implement one with less than eight entries to suit resource-limited devices. When you replace the connection table, you can have 0 – 254 entries in the table. To replace the connection table, override the **IsiGetConnectionTableSize()***,* **IsiGetConnection()**, and **IsiSetConnection()** functions. If any of these functions are overridden, then all three must be overridden. If some but not all are overridden, the ISI engine will not function properly.

The connection table requires persistent storage; the content of the connection table must not change when the device is reset or looses power. The size of the connection table must be constant between device resets.

EXAMPLE 1

The following example creates a connection table with 16 entries stored in on-chip EEPROM:

```
#define CTABSIZE 16u
static eeprom fastaccess IsiConnection
myConnectionTable[CTABSIZE];
unsigned IsiGetConnectionTableSize(void) {
   return CTABSIZE;
}
const IsiConnection* IsiGetConnection(unsigned index) {
   return myConnectionTable + index;
}
void IsiSetConnection(IsiConnection* pConnection, unsigned index)
{
   myConnectionTable[index] = *pConnection;
}
```

You can also move the connection table to off-chip storage, including an off-chip serial EEPROM for low-cost non-volatile memory. The ISI engine accesses one connection table entry at a time so that you only need to buffer one connection table entry at a time if you move it out of the Neuron memory space. The ISI engine makes no assumptions about the pointer that your application returns to the **IsiGetConnection()** call, except that it points to a single valid connection table entry*.* You can therefore buffer a single connection table entry within the Neuron address space, and always return the same address.

The connection table is accessed frequently, so your **IsiGetConnection()** implementation should minimize processing time.

EXAMPLE 2

The following example creates a connection table with 16 entries stored in off-chip storage:

```
#define CONTABSIZE 16u

static IsiConnection connectionBuffer;

unsigned IsiGetConnectionTableSize(void) {
   return CONTABSIZE;
}

const IsiConnection* IsiGetConnection(unsigned index) {
    // TODO: fetch data for index from external storage,
   // for example by using a I2C serial I/O model; fill
   // connectionBuffer with that data.
   …
   return &connectionBuffer;
}

void IsiSetConnection(IsiConnection* pConnection,
                      unsigned index) {
    // TODO: transfer data referenced by pConnection to
    // external storage, saving it for the given Index
    …
}
```

Getting ISI Version Information

unsigned **IsiImplementationVersion**(void);
unsigned **IsiProtocolVersion**(void);

You can get the version number of the ISI engine, and of the ISI protocol supported by the ISI engine. Two functions are included to provide information about the current implementation of ISI.

To get the version number of the ISI engine, call the **IsiImplementationVersion()** function.

To get the maximum version of the ISI protocol supported by the ISI engine, call the **IsiProtocolVersion()** function. The ISI engine implicitly supports protocol versions less then the version returned by the function unless otherwise noted.

Discovering Devices

You can discover all the devices in an ISI network. All devices in an ISI network periodically broadcast their status by sending out DRUM messages. To discover devices, you can monitor these status messages. This is useful for gateways and controllers that need to maintain a table of all devices in a network, or provide unique capabilities for specific types of devices in a network.

To discover devices, monitor the DRUM messages being sent on the network by other devices and store the relevant information in a *device table*. A device table is a table containing a list of devices and their attributes including their network addresses. The DRUM messages contain all of the relevant information to do explicit messaging. To create a device table, store the relevant DRUM fields, such as subnet ID, node ID, and Neuron ID in a table that you can use to directly communicate with other devices as described in the next section. To detect deleted devices, monitor the time of the last update for each entry in the table and detect devices that have not recently sent a DRUM.

EXAMPLE 1

The following example maintains a device table, adding new devices when discovered, updating subnet and node IDs when they change, and deleting stale devices.

```
#include <isi.h>
#include <mem.h>

#define MAX_DEVICES 16
#define MAX_CREDITS 5
#pragma num_alias_table_entries 6

unsigned deviceCount;

// Struct to hold device information
typedef struct {
   unsigned credits;
   unsigned subnetId;
   unsigned nodeId;
   unsigned neuronId[NEURON_ID_LEN];
} Device;

Device devices[MAX_DEVICES];

when (reset) {
   deviceCount = 0;    // Not required since global RAM is zeroed
   scaled_delay(31745UL); // 800ms delay
   IsiStartS(isiFlagNone);
}

void MaintainDevices(const IsiDrum* pDrum) {
   unsigned i;

   // Iterate through the device list and see if the Neuron ID
   // of the stored device matches that of the new device; if it
   // does, then update the related details
   for (i = 0; i < deviceCount; i++) {
      if (memcmp(Devices[i].neuronId, pDrum->NeuronId,
            NEURON_ID_LEN) == 0) {
         devices[i].credits = MAX_CREDITS;
         devices[i].subnetId = pDrum->SubnetId;
         devices[i].nodeId = pDrum->NodeId;
         break;
      }
   }
   // If i is equal to the device count, then the device was not
   // found, so add it to the device table if possible
   if (i == deviceCount && deviceCount < MAX_DEVICES) {
      memcpy(devices[i].neuronId, pDrum->NeuronId,
            NEURON_ID_LEN);
      deviceCount++;
      devices[i].credits = MAX_CREDITS;
      devices[i].subnetId = pDrum->SubnetId;
      devices[i].nodeId = pDrum->NodeId;
   }
}
```

```
// Iterate through the device table again and decrement the
// credits at each call of IsiCreatePeriodicMsg(), removing any
// devices that have not sent a DRUM in the max possible amount
// of credits.  IsiCreatePeriodicMsg() calls occur at a rate
// roughly equal to the DRUM rate expected from each device;
// this IsiCreatePeriodicMsg() override does not actually cause
// the generation of a periodic message (it always returns
// FALSE), and is used here to decrement each device’s
// credit at the best interval possible
boolean IsiCreatePerodicMsg(void) {
    unsigned i;
    for (i = 0; i < devicecount; i++) {
        devices[i].credits--;
        if (devices[i].credits == 0) {
            devicecount--;
            if (devicecount != i) {
                // Move device from end to this spot’s location
                memcpy(devices+i, devices+devicecount,
                    sizeof(Device));
            }
        }
    }
    return FALSE;
}

when (msg_arrives) {
    IsiMessage message;
    if (IsiApproveMsg()) {
        // Received packets are kept in msg_in;
        // compare the message code of the received msg
        // to that of a DRUM
        memcpy(&message, msg_in.data, sizeof(IsiMessage));
        if (message.Header.Code == isiDrum ||
             message.Header.Code == isiDrumEx) {
            MaintainDevices(&message.Msg.Drum);
        }
        IsiProcessMsgS();
    }
}
```

Accelerating Device Discovery

You can accelerate device discovery to reduce the time to discovery of all devices in an ISI network. Discovering devices using DRUMs as described in the previous section can take a long time due to the automatic bandwidth utilization control implemented by the ISI protocol. For example, discovering five devices in an ISI-S power line network without a domain address server can take up to 5 minutes 30 seconds. Discovering 200 devices in an ISI-DA power line network can take up to 34 minutes.

You can accelerate device discovery by using the 709.1 Query ID message. You can broadcast this message to all devices in the primary domain. All devices will respond with their Neuron ID and program ID. If there are many devices in the network, it is likely that some of the responses will be lost due to the requesting device’s buffers becoming full, or due to network collisions. To accommodate this, the requesting device disables responses from any devices discovered by the first request, and then repeats the process on the remaining devices. To accelerate device discovery, follow these steps:

1. Broadcast a Respond to Query message with a parameter of MODE_ON to the primary domain. This enables all devices to respond to the Query ID message.

2. Broadcast a Query ID message to the primary domain.

3. If there are no responses, exit the procedure.

4. For each response, save the Neuron ID in the device table and then send a Respond to Query message with a parameter of 0 addressed to the Neuron ID. This disables the device from responding to subsequent Query ID messages.

5. Repeat steps 2 – 4.

EXAMPLE

The following example uses 709.1 Query ID messages to discover all devices in the primary domain.

```
#include <msg_addr.h>
#include <netmgmt.h>

msg_tag bind_info(nonbind) discoveryTag;

#define RETRY_COUNT 3
#define ENCODED_RPT_TIMER   8   // 256ms
#define ENCODED_TX_TIMER    8   // 256ms
#define PRIMARY_DOMAIN      0

// The function broadcases a respond_to_query(MODE_ON) command,
// requesting that every receiving device shall respond to query
// ID requests.
void RespondOn(void) {
    msg_out.code = NM_opcode_base | NM_respond_to_query;
    msg_out.service = UNACKD_RPT;
    msg_out.dest_addr.bcast.type = BROADCAST;
    msg_out.dest_addr.bcast.domain = PRIMARY_DOMAIN;
    msg_out.dest_addr.bcast.rpt_timer = ENCODED_RPT_TIMER;
    msg_out.dest_addr.bcast.retry  = RETRY_COUNT;
    msg_out.dest_addr.bcast.tx_timer = ENCODED_TX_TIMER;
    msg_out.dest_addr.bcast.subnet = 0;
    msg_out.tag = discoveryTag;
    msg_out.data[0] = MODE_ON;
    msg_send();
}

//  Issue a single Query ID request.  This function
//  gets called repeatedly as long as positive
//  responses to this request are received.
void QueryNetwork(void) {
    NM_query_id_request queryIdRequest;

    msg_out.code = NM_opcode_base | NM_query_id;
    msg_out.service = REQUEST;
    msg_out.dest_addr.bcast.type = BROADCAST;
    msg_out.dest_addr.bcast.domain = PRIMARY_DOMAIN;
    msg_out.dest_addr.bcast.rpt_timer = ENCODED_RPT_TIMER;
    msg_out.dest_addr.bcast.retry  = RETRY_COUNT;
    msg_out.dest_addr.bcast.tx_timer = ENCODED_TX_TIMER;
    msg_out.dest_addr.bcast.subnet = 0;
    msg_out.tag = discoveryTag;
    queryIdRequest.selector = SELECTED;
    memcpy(msg_out.data, &queryIdRequest,
        sizeof(NM_query_id_request));
    msg_send();
}

// Handle incoming responses.  For successful responses to
// the query ID request, store the response data and process according
// to the application's needs.
// Issue a respond_to_query(MODE_OFF) command to the device that responded
// to the query, requesting that the same device should refrain from responding
// to repeated queries, and repeat the Query ID request to the remaining
// devices in the domain.
when (resp_arrives(discoveryTag)) {
    NM_query_id_response queryIdResponse;

    if (resp_in.code == (NM_resp_success|NM_query_id)) {
        // Save response data:
        memcpy(&queryIdResponse, resp_in.data,
sizeof(NM_query_id_response));

        // To do: Process new device here...
        ...

        // Disable this device's query ID response
        msg_out.dest_addr.nrnid.type = NEURON_ID;
        msg_out.dest_addr.nrnid.domain = PRIMARY_DOMAIN;
        msg_out.dest_addr.nrnid.rpt_timer = ENCODED_RPT_TIMER;
        msg_out.dest_addr.nrnid.retry = RETRY_COUNT;
        msg_out.dest_addr.nrnid.tx_timer = ENCODED_TX_TIMER;
        memcpy(msg_out.dest_addr.nrnid.nid, queryIdResponse.neuron_id,
NEURON_ID_LEN);
        msg_out.code = NM_opcode_base | NM_respond_to_query;
        msg_out.tag = discoveryTag;
        msg_out.service = REQUEST;
        msg_out.data[0] = MODE_OFF;
        msg_send();

        // Try again to query the ID message from other devices
        QueryNetwork();
    }
}

//  Start the device discovery process by enabling each device's
//  response to Query ID request, and issueing such a request.
//  Repeat the query for every
void DiscoverDevices(void)
{
    RespondOn();
    QueryNetwork();
}
```

Creating a Connection with Controlled Enrollment

You can create a connection from a connection controller using controlled enrollment. The connection controller is not a network management tool—it simply takes the place of the push buttons on devices used to create connections using manual enrollment. Based on user input to the connection controller, the controller sends requests to the devices to be connected that are translated to the same ISI API calls used to create manual connections. A connection controller will typically be some sort of user interface device such as a user interface touch panel.

To send a request for controlled enrollment, the connection controller sends a control request (CTRQ) message to the devices to be connected. If the devices support controlled enrollment, they respond with control response (CTRP) messages. The CTRQ and CTRP message are described in Appendix A.

The CTRQ message causes the device to execute the ISI API function indicated with the control value, using the parameter if needed. For example, a CTRQ message specifying an **isiOpen** function with a parameter of 2 may be sent to cause a device to become a manual enrollment host for assembly 2. The device receives this message, sends a positive response, and then calls **IsiOpenEnrollment(2)**. The connection controller must send the CTRQ message using the request/response service. If the device doesn't implement controlled enrollment at all, no response will be received. If the device supports controlled enrollment, but not the requested operation, a negative response will be received. If the device supports the requested operation, a positive response will be received. The request is always sent on the primary domain, typically using Neuron ID addressing. Using the primary domain is required so that the response can reach the connection controller. Using Neuron ID addressing relieves the connection controller from keeping track of Neuron ID versus subnet/node ID tracking, but the connection controller is free to use subnet/node ID addressing,

The connection controller must send requests for the ISI API functions described in *Enrolling in a Connection* in Chapter 2 to create a connection.

When a CTRP response is received from a device, the connection controller can use the Neuron ID contained in the response to correlate the response to its device table.

EXAMPLE

The following connection controller example implements a simple ISI connection from a switch to a light.

```
#include <msg_addr.h>
#include <isi.h>
#define RETRY_COUNT 3
#define ENCODED_TX_TIMER 11 // 768ms
#define ENCODED_RPT_TIMER 2
#define PRIMARY_DOMAIN 0

//  Issue the ISI controlled enrollment command (with parameter
// assembly) to the destination device indicated by neuronId.
void ControlCommand(const unsigned* neuronId, IsiControl command,
unsigned assembly) {
    IsiMessage request;
    msg_out_addr destination;

    request.Header.Code = isiCtrq;
    request.Msg.Ctrq.Control = command;
    request.Msg.Ctrq.Parameter = assembly;

    destination.nrnid.type = NEURON_ID;
    destination.nrnid.domain = PRIMARY_DOMAIN;
    destination.nrnid.rpt_timer = ENCODED_RPT_TIMER;
    destination.nrnid.subnet = 0;
    destination.nrnid.retry  = RETRY_COUNT;
    destination.nrnid.tx_timer = ENCODED_TX_TIMER;
    memcpy(destination.nrnid.nid, neuronId, NEURON_ID_LEN);

    IsiMsgSend(&request, sizeof(IsiMessageHeader)+sizeof(IsiCtrq),
REQUEST, &destination);
}

// State information for this enrollment controller.
// Using this state information allows the application to track
// completion of the entire process, and possibly to cancel out
// upon failure.

enum {
     idle,       // do nothing
     opening,    // open enrollment with host, awaiting response from host
     enrolling,  // enrolling members, awaiting responses from each member
     creating    // close enrollment and create connection
} controlState;

// Hold data to open enrollment on a host, enroll one member, and close
// enrollment with the host.  The structure is filled in by the
// ConnectTwoDevices function and used with the when (resp_arrives) task.

typedef struct {
   unsigned neuronId[NEURON_ID_LEN];
   unsigned assembly;
} ControlledAssembly;

struct {
   ControlledAssembly host;
   ControlledAssembly member;

} controlJob;

when (resp_arrives) {
    IsiMessage response;

    if (resp_in.code == isiApplicationMessageCode) {
     memcpy(&response, resp_in.data, resp_in.len);
     if (response.Header.Code == isiCtrp) {
          // This is a controlled enrollment response
     if (response.Msg.Ctrp.Success) {
          // This is a successful response.  See where we are
          // and proceed to next step, if any:
          switch(controlState) {
             case opening:
                  // Host has opened; enroll member
                  ControlCommand(controlJob.member.neuronId, isiCreate,
controlJob.member.assembly);
                  controlState = enrolling;
```

```
                     break;
                case enrolling:
                     // Just one member for this example; close enrollment
                     ControlCommand(controlJob.host.neuronId, isiCreate,
controlJob.host.assembly);
                     controlState = creating;
                     break;
                case creating:
                     // We're done; rport successful completion
                     // to the user
                     ...
                     controlState = idle;
                     break;
                }
           } else {
                // This is a failure response. For this example, simply
                // cancel the enrollment.
                ControlCommand(controlJob.host.neuronId, isiCancel,
controlJob.host.assembly);
                // Indicate failure via user interface
                ...
                controlState = idle;
           }
      }
   }
}

// Open enrollment with a host device and host assembly, enroll one
// member device and assembly, and closes enrollment.  The function
// initializes the description of the control job and kick-starts
// the process; the when (resp_arrives) task handles the
// completion, or cancellation, of the remaining steps.

void ConnectTwoDevices(unsigned* hostId, unsigned hostAssembly,
    unsigned* memberId, unsigned memberAssembly) {
    // Copy the job description
    memcpy(controlJob.host.neuronId, hostId, NEURON_ID_LEN);
    controlJob.host.assembly = hostAssembly;
    memcpy(controlJob.member.neuronId, memberId, NEURON_ID_LEN);
    controlJob.member.assembly = memberAssembly;
    // Start the process by opening enrollment on the host
    controlState = opening;
    ControlCommand(hostId, isiOpen, hostAssembly);
}
```

Recovering Connections

You can recover existing connections so that a connection controller can display connections not created by the connection controller, and display connections created by the connection controller that are no longer in the connection controller's database. To recover connections, a connection controller must first discover all the devices in the network as described in *Discovering Devices*. To recover the connections, the controller uses the read connection table request (RDCT) message, which allows reading a device's connection table via the network. Support for this message is optional (except for devices that support controlled enrollment), and therefore this message can only be used with devices that support it.

The RDCT message includes optional host and member assembly fields that specify which connection table entries are requested. If they are not supported by the device, or are both set to 0xFF, the connection table entry indicated by the index is requested. If they are supported and the host or member field is not 0xFF, the index provided is the starting index. The first matching connection table entry is returned, if any. If both host and member fields are set to a value different from 0xFF, the connection table entries are returned that match either the host or the member fields, if any.

This allows a connection controller to read the entire connection table, or to read the table selectively in order to provide quick answers to questions like "is assembly Z on device X connected, and is it the host of the connection?"

If the requested data is available, the response to a RDCT message is a read connection table success (RDCS) message. This message contains the requested connection table index and data. If the connection table index does not exist, or if the requested assemblies do not exist, the response is a read connection table failure (RDCF) message.

A connection controller can determine if a device does not support the optional host and member assembly fields by comparing the assembly numbers in the read response to the requested assembly number, or by receiving an RDCF message indicating a failed read. If a device does not support the host and member assembly fields, the connection controller must read every entry in the connection table individually. This has minimal impact for devices with one or two connection table entries, but increases network traffic for devices with many connection table entries.

EXAMPLE 1

The following connection controller example recovers all the connections from a device.

```
#include <msg_addr.h>
#include <isi.h>

#define RETRY_COUNT 3
#define ENCODED_TX_TIMER 11 // 768ms
#define ENCODED_RPT_TIMER 2
#define PRIMARY_DOMAIN 0

// This structure holds information required while reading a remote
// device's connection table
struct {
    unsigned neuronId[NEURON_ID_LEN];
    unsigned index;
} recoveryJob;

// Issue one read connection table request using the global
// recoveryJob variable for destination address and current
// connection table index information.  Increment the index
// kept in that global variable.
void RequestConnectionTable(void) {
    IsiMessage request;
    msg_out_addr destination;

    request.Header.Code = isiRdct;
    request.Msg.Rdct.Index = recoveryJob.index++;
    request.Msg.Rdct.Host = request.Msg.Rdct.Member = ISI_NO_ASSEMBLY;

    destination.nrnid.type = NEURON_ID;
    destination.nrnid.domain = PRIMARY_DOMAIN;
    destination.nrnid.rpt_timer = ENCODED_RPT_TIMER;
    destination.nrnid.subnet = 0;
    destination.nrnid.retry  = RETRY_COUNT;
    destination.nrnid.tx_timer = ENCODED_TX_TIMER;
    memcpy(destination.nrnid.nid, recoveryJob.neuronId, NEURON_ID_LEN);

    IsiMsgSend(&request, sizeof(IsiMessageHeader)+sizeof(IsiRdct), REQUEST,
&destination);
}

// Handle receipt of incoming responses.  This example focusses on
// isiRdcs and isiRdcf responses.
when (resp_arrives) {
    IsiMessage response;
    if (resp_in.code == isiApplicationMessageCode) {
        // This is an ISI response
        memcpy(&response, resp_in.data, resp_in.len);
        if (response.Header.Code == isiRdcf) {
            // The remote device rejected our request, probably because we
            // have queried all available connection table entries already
            // (bad index).  Notify the user interface, if needed.
            ...

        } else if (response.Header.Code == isiRdcs) {
            // The remote device replied to our request with the connection
            // table entry requested, in response.Msg.Rdcs.  Notify the UI
            // and/or process this data further, as needed by the application:
            ...

            // Because we received a positive response, let's try for the
            // next index
            RequestConnectionTable();
        }
    }
}

// Initiate the process of reading a remote device's connection table.
// The function kick-starts the process, where the majority of the work
// is done in the when (resp_arrives) task.  Calling this function before
// the previous connection table read job completes causes the previous
// job to abort, and the new one to start
void ReadRemoteConnectionTable(const unsigned* remoteNeuronId) {
    memcpy(recoveryJob.neuronId, remoteNeuronId, NEURON_ID_LEN);
    recoveryJob.index = 0;
    RequestConnectionTable();
}
```

Example 2

The following ISI device example receives a read connection table request (RDCT) message and then sends the appropriate response.

```
#pragma num_alias_table_entries 6

#include <isi.h>

// Reply with isiRdcf (connection table read failure), or with
// isiRdcs (connection table read success) message for the
//  connection table index provided.
void SendConnectionTableResponse(unsigned index, boolean success) {
    IsiMessage response;
    unsigned   length;

    if (success) {
        response.Header.Code = isiRdcs;
        response.Msg.Rdcs.Index = index;
        memcpy(&response.Msg.Rdcs.Data, IsiGetConnection(index),
sizeof(response.Msg.Rdcs.Data));
        length = sizeof(IsiMessageHeader)+sizeof(IsiRdcs);
    } else {
        response.Header.Code = isiRdcf;
        length = sizeof(IsiMessageHeader);
    }
    memcpy(resp_out.data, &response, length);
    resp_send();
}

when (msg_arrives) {
    IsiMessage request;
    unsigned index, connectionTableSize;
    const IsiConnection* pConnection;

    if (IsiApproveMsg() && IsiProcessMsgS()) {
        // The message is approved (it is a genuine ISI message), but
        // was not processed by IsiProcessMsgS().  This might be a
        // connection table request.

        memcpy(&request, msg_in.data, msg_in.len);
        connectionTableSize = IsiGetConnectionTableSize();

        if (request.Header.Code == isiRdct) {
            // This is a connection table read request

            index = request.Msg.Rdct.Index;

            if (request.Msg.Rdct.Host != ISI_NO_ASSEMBLY) {
                // Try locating a connection table entry that is active and
                // report the requested assembly as the host assembly
                while (index < connectionTableSize) {
                    pConnection = IsiGetConnection(index);
                    if (pConnection->State >= isiConnectionStateInUse
                    && pConnection->Host == request.Msg.Rdct.Host) {
                        break;
                    }
                    ++index;
                }
            } else if (request.Msg.Rdct.Member != ISI_NO_ASSEMBLY) {
                // Try locating a connection table entry that is active and
                // report the requested assembly as the member assembly
                while (index < connectionTableSize) {
                    pConnection = IsiGetConnection(index);
                    if (pConnection->State >= isiConnectionStateInUse
                    && pConnection->Member == request.Msg.Rdct.Member) {
                        break;
                    }
                    ++index;
                }
            }
            SendConnectionTableResponse(index, index < connectionTableSize);
        }
    }
}
```

Sending an NV Update or Polling an NV from a Controller

You can send an individual network variable update to, or poll an individual network variable from, an individual device. This is typically done on a gateway or controller that provides a user interface for individual devices on an ISI network. The process is based on building and maintaining a device table as discussed in the previous section, identifying the individual device from this table, and explicitly constructing explicitly addressed network variable messages to communicate with the remote device. The explicit network variable updates and polls require more code than the equivalent implicit update or poll that occurs when you assign to or poll a network variable.

To send an individual update or poll, first build and maintain a device table as described in the previous section. Based on the criterion needed to identify an individual device, you may need to add more fields to the table, filled with data from the DRUM message, or queried from the device.

When inspecting all live devices held in the table, you can identify types of devices based on the device program IDs—for instance, you can identify all appliances to give the user the capability to control individual appliances.

To select an individual device among multiple candidate devices, additional logic is required. For instance, the gateway device might list three refrigerators. For service and diagnosis, however, your application may want to communicate with only one specific refrigerator at a time.

The gateway application may use addressing details from the table to directly communicate with each instance, and to obtain further details that could assist with identification. For instance, the refrigerators could implement a **SCPTname1** and **SCPTname2** configuration property that lets the gateway determine the name of each unit, or a **SCPTlocation** configuration property that lets the gateway determine the physical location of each unit. Alternatively, the gateway could issue Wink commands and obtain input from the user, guided by the refrigerators' Wink responses, to choose an individual instance. Once the instance has been identified, the gateway application can communicate directly with the remote device.

To send a network variable update to the remote device, create an explicit message that is formatted as a network variable update. Include an explicit address for the message with addressing details that you fetch from the device table.

To poll a network variable output on a device in the device table, create an explicit message that is formatted as a network variable fetch. Include an explicit address for the message with addressing details that you fetch from the device table.

EXAMPLE 1

The following example sends a network variable fetch message to a device with a specified NV index, subnet ID, and node ID. These may be fetched from the device table created in the previous section.

```
#include <msg_addr.h>
#include <netmgmt.h>

#define RETRY_COUNT 3
#define ENCODED_TX_TIMER 11 // 768ms
#define ENCODED_RPT_TIMER 2
#define PRIMARY_DOMAIN 0

msg_tag bind_info(nonbind) fetchNvTag;

// Issue a fetch NV request using the NV index on the remote
// device, and the remote device's subnet/node ID pair for the
// destination address.  Return when the request has been sent;
// see when (resp_arrives) task for processing of the response.
void FetchNv(unsigned nvIndex, unsigned subnetId, unsigned nodeId)
{
    NM_nv_fetch_request fetchRequest;

    memset(&fetchRequest, 0, sizeof(NM_nv_fetch_request));

    fetchRequest.nv_index = nvIndex;
    memcpy(msg_out.data, &fetchRequest,
sizeof(NM_nv_fetch_request));

    // Use Subnet/Node ID addressing
    msg_out.dest_addr.snode.type   = SUBNET_NODE;
    msg_out.dest_addr.snode.domain = PRIMARY_DOMAIN;
    msg_out.dest_addr.snode.subnet = subnetId;
    msg_out.dest_addr.snode.node   = nodeId;
    msg_out.dest_addr.snode.retry  = RETRY_COUNT;
    msg_out.dest_addr.snode.tx_timer = ENCODED_TX_TIMER;
    // Copy the relevant data to msg_out
    msg_out.code = NM_opcode_base | (NM_nv_fetch &
NM_opcode_mask);
    msg_out.service = REQUEST;
    msg_out.tag = fetchNvTag;   // Destination address
    msg_send();
}

//  Handle receipt of responses to the fetch request issued
//  by the FetchNv() function.
when (resp_arrives(fetchNvTag)) {
    NM_nv_fetch_response fetchResponse;

    if (resp_in.code == (NM_resp_success | NM_nv_fetch)) {
        memcpy(&fetchResponse, resp_in.data,
sizeof(NM_nv_fetch_response));
        // Process returned NV Value
        ...
    }
}
```

Example 2

The following example sends an explicit network variable update to a device in the device table created in the previous section after sending a query NV request to determine the selector of the target network variable.

```
#include <msg_addr.h>
#include <netmgmt.h>
#include <isi.h>

#define RETRY_COUNT 3
#define ENCODED_TX_TIMER  8 // 256ms
#define ENCODED_RPT_TIMER 2
#define PRIMARY_DOMAIN 0

#define NV_UPDATE_MESSAGE   0x80u

//  This explicit message tag is used when querying the remote
//  network variable's configuration, and when issuing the
//  acknowledged network variable update.
msg_tag bind_info(nonbind) nvTag;

//  This variable holds the remote device's subnet/node ID pair,
//  used when sending the acknowledged NV update which copies
//  the value from a local network variable, indicated by the
//  localNvIndex detail, to a remote input network variable.
struct {
    unsigned subnetId;
    unsigned nodeId;
    unsigned localNvIndex;
} nvUpdateJob;

// Initiate an acknowledged network variable update by copying
// relevant data into the nvUpdateJob variable.  The function
// then issues a network variable configuration request, which is
// required to obtain the remote network variable's selector value.
// The when (resp_arrives) task, below, handles the response to that
// request and issues the acknowledged network variable update.
void InitiateNvUpdate(unsigned remoteNvIndex, unsigned
localNvIndex,
                      unsigned remoteSubnetId, unsigned
remoteNodeId) {

    NM_query_nv_cnfg_request queryNvRequest;

    // Preserve data needed when the NV configuration response arrives
    nvUpdateJob.localNvIndex = localNvIndex;
    nvUpdateJob.subnetId = remoteSubnetId;
    nvUpdateJob.nodeId = remoteNodeId;

    // Prepare the network variable configuration request
    memset(&queryNvRequest, 0, sizeof(NM_query_nv_cnfg_request));
    queryNvRequest.nv_index = remoteNvIndex;
    memcpy(msg_out.data, &queryNvRequest,
sizeof(NM_query_nv_cnfg_request));

    // Use Subnet/Node ID addressing
    msg_out.dest_addr.snode.type   = SUBNET_NODE;
    msg_out.dest_addr.snode.domain = PRIMARY_DOMAIN;
    msg_out.dest_addr.snode.subnet = remoteSubnetId;
    msg_out.dest_addr.snode.node   = remoteNodeId;
    msg_out.dest_addr.snode.retry  = RETRY_COUNT;
    msg_out.dest_addr.snode.tx_timer = ENCODED_TX_TIMER;
    msg_out.code = NM_opcode_base | (NM_query_nv_cnfg &
NM_opcode_mask);
    msg_out.service = REQUEST;
    msg_out.tag = nvTag;
    msg_send();
}

// Handle receipt of the network variable configuration response, and
// build a network variable update message that copies the value of the
// local network variable to the remote network variable.  This example
// does not ensure that the remote network variable is an input, or
// that the both network variables share the same type.
when (resp_arrives(nvTag)) {
    NM_query_nv_cnfg_response queryNvResponse;
    unsigned localNvLength;
    const unsigned* pLocalNvValue;

    if (resp_in.code == (NM_resp_success | NM_query_nv_cnfg)) {
        memcpy(&queryNvResponse, resp_in.data,
sizeof(NM_query_nv_cnfg_response));
        // queryNvResponse now holds the remote network variable's selector,
        // which will be used to build the acknowledged network variable update

        memset(&(msg_out.dest_addr.snode), 0, sizeof(struct
snode_struct));

        // Use Subnet/Node ID addressing
        msg_out.dest_addr.snode.type = SUBNET_NODE;
        msg_out.dest_addr.snode.subnet = nvUpdateJob.subnetId;
        msg_out.dest_addr.snode.node = nvUpdateJob.nodeId;

        // Construct the network variable update message
        msg_out.code = NV_UPDATE_MESSAGE |
queryNvResponse.nv_selector_hi;
        msg_out.data[0] = queryNvResponse.nv_selector_lo;

        // Copy the value of the local variable
        pLocalNvValue = IsiGetNvValue(nvUpdateJob.localNvIndex,
&localNvLength);
        memcpy(&msg_out.data[1], pLocalNvValue, localNvLength);

        msg_out.dest_addr.snode.retry = RETRY_COUNT;
        msg_out.dest_addr.snode.tx_timer = ENCODED_TX_TIMER;
        msg_out.service = ACKD;
        msg_send();
    }
}
```

Monitoring Data from a Controller and Designing Devices for Monitoring

There are three common methods for a controller to monitor network variables in an ISI network: *polling*, *controlled enrollment*, and *automatic enrollment*. The advantage of polling is that it does not require any additional code in the monitored devices, but the disadvantage is that the controller may require a long time to poll all network variables in a network—this may cause the controller to appear to have a poor response time after a network variable value on a monitored device changes. Controlled and automatic enrollment both address responsiveness by supporting event-driven updates—the controller is notified whenever a monitored network variable value changes. The disadvantage of both controlled and automatic enrollment is that additional code space is required on the monitored device, with automatic enrollment requiring the most code space. This section describes the three methods for both controller and device developers. If you are developing a general-purpose controller, you should support all three methods to be able to monitor any type of device. If you are developing an ISI device, you should consider supporting either controlled or automatic enrollment if polling will not provide adequate monitoring performance. Some standard ISI profiles require controlled enrollment, so using controlled enrollment for monitoring with these profiles does not require additional code space.

Polling

Polling is the simplest of the three methods. The controller uses a device table as described in *Discovering Devices* in this chapter to create network variable fetch requests that it sends to the monitored devices. The 709.1 protocol engine in each of the monitored devices automatically responds to the fetch requests with the requested network variable value, so no special code is required in the monitored device. This works whether the network variable is in a connection or not since the fetch request uses a network variable index that does not change when the network variable enrolls in a connection. The problem with polling is performance. On a power line channel, a controller should not do more than one poll per second on a network with a power line channel or 12 polls per second on an exclusive TP/FT-10 network. Controllers supporting multiple outgoing transactions should have no more than one outstanding polling transaction at a time. In a network with 100 devices and one monitored network variable per device, the controller will require 10 minutes to poll every device at 10 polls per minute. If the monitored devices are lamps, the controller will take up to 10 minutes to indicate a change in the state of the lamp. This is probably not satisfactory for a lamp, but may be satisfactory for a device with slowly changing data like an air temperature sensor.

Controlled Enrollment

Controlled enrollment as described in *Creating a Connection with Controlled Enrollment* in this chapter provides an easy way for a controller to orchestrate event-driven updates from monitored devices, at a cost of at least 100 bytes on the monitored devices. Some standard ISI profiles require controlled enrollment, in that case there is no additional code or code space required on the device to support controlled enrollment for monitoring. When a controller uses controlled enrollment for monitoring, it must also provide a user interface for creating peer-to-peer connections with controlled enrollment. The reason is that a connection created by a controller for monitoring will make the assembly used for monitoring no longer available for a peer-to-peer connection if the device does not support connection replacement or connection extension—this will be the case for many simple devices. By supporting peer-to-peer controlled enrollment, the controller can re-build the required connections when the user adds a monitored device to a peer-to-peer connection. To support controlled enrollment, the controller must maintain a device description table that lists:

- the program IDs that support controlled enrollment,
- which network variables on each of those devices are to be monitored by the controller,
- the assembly containing each monitored network variable, and
- which of the devices support connection replacement and which support connection extension.

For devices with program IDs that are not in the device description table, the controller may query the device for the required information. The controller can query and interpret the device's node self-documentation string to get a list of the functional blocks implemented by the device, then use the standard functional profiles to determine the required network variables, and then use the standard assembly number to functional block number mapping described in *ISI Connection Model* in Chapter 2 to determine the assemblies.

To support controlled enrollment for monitoring, a controller follows these steps upon discovering a device as described in *Discovering Devices* in this chapter:

1. The controller looks up the discovered device in the device description table and determines whether it supports controlled enrollment, connection replacement, and/or connection extension.

2. If the device supports controlled enrollment and connection extension, the controller extends any existing connection for the monitored assembly by sending **isiOpen** and **isiExtend** requests to the device, and then recovers connections to confirm that the connection was successfully extended. If the connection was not successfully created, the controller proceeds with the remaining steps.

3. If the device supports controlled enrollment and connection replacement, but does not support connection extension or could not extend the existing connection, the controller determines if the monitored NV is already in a connection. It does this by using connection recovery as described in *Recovering Connections* in this chapter. If the monitored NV is not already in a connection, the controller creates a connection from the monitored NV to the controller using the **isiOpen** and **isiCreate** requests. If the monitored NV is already in a connection, the controller rebuilds the connection using the **isiOpen** and **isiCreate** requests on all assemblies in the connection, adding the controller to the connection.

4. If the device supports controlled enrollment but does not support connection replacement or connection extension, the controller determines if the monitored NV is already in a connection. It does this by using connection recovery as described in *Recovering Connections* in this chapter. If the monitored NV is not already in a connection, the controller creates a connection from the monitored NV to the controller using the **isiOpen** and **isiCreate** requests. If the monitored NV is already in a connection, the controller sends the **isiFactory** request to the device to clear all of its existing connections, and then rebuilds all connections

for the device and all devices that it is connected to using the **isiOpen** and **isiCreate** requests.

To support controlled enrollment for monitoring, a controller must also recreate monitored connections to a device when the device overwrites on clears any connections created by the controller. There are two cases where this may happen. A device that supports connection replacement may overwrite a controller connection by creating a new connection that replaces the controller connection. Any device may overwrite a controller connection by resetting the device to factory defaults, clearing all connections. Either case can occur if the device supports manual enrollment and the user manually adds the device to a connection, or if the device supports automatic enrollment and the device automatically joins a connection. In the first case, the controller can monitor all CSMO and CSMA messages, and silently add itself to the connection if the connection will overwrite a monitored connection. A controller silently adds itself by not issuing a CSMA message, but otherwise doing all the steps described in *Enrolling in a Connection* in Chapter 2 to enroll in a connection. In the second case, the controller repeats the procedure described above to create a monitored connection using controlled enrollment. The controller can also periodically recover the connections from all devices in the network to ensure any monitoring connections are still intact. If the controller discovers any missing monitoring connections, it can recreate them. This periodic traffic should be minimized to reduce network overhead. For example, sending one request every 10 seconds to recover a connection should be sufficient.

Automatic Enrollment

Automatic enrollment, as described in *Enrolling in a Connection* in Chapter 2, provides an easy way for a controller and monitored device to work together to support event-driven updates. This provides a method that requires fewer network transactions than controlled enrollment, and no wait states, at a cost of at least a few hundred bytes on the monitored devices. This method requires the monitored device to support automatic connections and connection replacement, and also requires the device to be more involved in creating the monitored connection, since the monitored device will determine which network variables on the device will be connected to the controller.

For a controller to use automatic enrollment for monitoring, the controller becomes the automatic connection host for data used by the controller. Standard ISI profiles define standard functional profiles called *monitor points* that controllers can use to request data to be monitored from monitored devices, and that devices can identify as connections offered for monitoring use.

For a device to support automatic enrollment for monitoring, when a monitored device receives a monitor point automatic invitation, the device application follows these steps:

1. If the device supports automatic enrollment for monitoring and also supports connection extension, the device extends the existing connection by accepting the connection invitation. If the extension fails, the device proceeds with the next step.

2. If the device supports automatic enrollment for monitoring and but does not support connection extension, the device calls the the **IsiIsConnected()** function to determine if the monitored NV is already in a connection. If the monitored NV is not already in a connection, the device accepts the connection invitation from the monitor point. If the monitored NV is already in a connection, the device does not accept the invitation.

If a device has accepted an automatic monitoring connection, and later needs to create a connection due to a manual request or other automatic request, it either extends the connection if it supports connection extension, or it replaces the monitoring connection if it does not or if it cannot extend the connection due to lack of an available alias and connection table entries. In the latter case, the controller will lose its monitoring connection. The controller can detect this by monitoring CSMO and CSMA messages. When either is received from a monitored device, the controller must map the assembly number to monitored network variables as described in *ISI Connection Model* in Chapter 2 and silently join the connection. If the device supports heartbeats, the controller can also detect lost monitoring connections by detecting lost heartbeat messages.

Selecting a Monitoring Method

A controller should implement all three monitoring methods described in this section to support all device types with the best response time supported by each. Device manufacturers can choose from the monitoring methods described in this section depending on how time-critical monitored data is for each device, and depending on available code space. The following are the recommended monitoring methods for devices, in decreasing order code size and functionality:

1. Support automatic enrollment to monitor points with connection extension and heartbeats.
2. Support automatic enrollment to monitor points with connection extension.
3. Support automatic enrollment to monitor points with connection replacement.
4. Support controlled enrollment with connection extension.
5. Support controlled enrollment with connection replacement.
6. Support controlled enrollment without connection extension or connection replacement.
7. Support polling.

Sending Periodic Heartbeats

boolean **IsiQueryHeartbeat(**unsigned *NvIndex***)**;

boolean **IsiIssueHeartbeat(**unsigned *NvIndex***)**;

You can use periodic processing in the ISI engine to schedule and send periodic heartbeat updates for any of the output network variables in your application. To send periodic heartbeats, follow these steps:

1. Start the ISI engine with the **isiFlagHeartbeat** flag. This causes the ISI engine to periodically call the **IsiQueryHeartbeat()** callback. This function returns **TRUE** if a heartbeat has been propagated, and **FALSE** otherwise.

2. Implement an **IsiQueryHeartbeat()** callback function and call the **IsiIssueHeartbeat()** function from within the function to send a heartbeat update. The **IsiIssueHearbeat()** function sends a network variable heartbeat for the indicated network variable index and all associated aliases. These heartbeats are sent with unacknowledged service with one repeat.

The **IsiIssueHeartbeat()** function only propagates network variables using group addressing. For example, if you use subnet/node ID addressing to implement an acknowledged connection as described in *Creating an Acknowledged Connection*, the output will not be propagated by the **IsiIssueHeartbeat()** function. To propagate outputs that do not use group addressing; use the **propagate()** Neuron C library function in combination with an **IsiCreatePeriodicMessage()** callback described in the next section for customized but scheduled network variable heartbeats.

EXAMPLE

The following starts the ISI engine with heartbeats enabled and issues a heartbeat in response to the **IsiQueryHeartbeat()** callback.

```
when (reset) {
   IsiStartDa(isiFlagHeartbeat);
}

boolean IsiQueryHeartbeat(unsigned nv) {
   // Agree to all heartbeats suggested by the ISI engine
   return IsiIssueHeartbeat(nv);
}
```

Sending Application-Specific Periodic Messages

You can send periodic messages, taking into account network size so that you optimize use of available channel bandwidth. The ISI engine automatically sends periodic messages in a controlled and scheduled manner to limit channel utilization for both ISI and application heartbeat messages. When each cycle of periodic messages is completed, the ISI engine calls the **IsiCreatePeriodicMsg()** callback function. The default implementation of this function always returns **FALSE**. To send application-specific periodic messages, follow these steps:

1. Start the ISI engine with the **isiFlagApplicationPeriodic** flag.

2. Override the **IsiCreatePeriodicMsg()** callback function to set an application-specific flag and return **TRUE** to signal that the application has a periodic message to send. Do not send the message from within

this function; just set the application-specific flag if there is a message to send and return control to the ISI engine immediately.

3. Use the application-specific flag in a separate **when** task to send the periodic message soon after the **IsiCreatePeriodicMsg()** function is completed. If you are sending an ISI message, you can use the **IsiMsgSend()** function. This function sets header fields for an ISI message, and then passes the message on to **IsiMsgDeliver()**, which propagates the message. If you are sending a network variable update, use the **propagate()** function to send the update.

A typical use-case is an application that does not participate in the default implementation of the network variable heartbeat scheme provided with ISI, but chooses to insert more specialized network variable heartbeat messages into the ISI cycle of periodic messages.

EXAMPLE

```
boolean sendApplicationPeriodic = FALSE;

when (reset) {
    …
    IsiStartS(isiFlagApplicationPeriodic);
    …
}

boolean IsiCreatePeriodicMsg(...) {
    if (have something to do) {
        // have something to do may always be TRUE
        sendApplicationPeriodic = TRUE;
    }
    return sendApplicationPeriodic;
}

when (sendApplicationPeriodic) {
    sendApplicationPeriodic = FALSE;
    // Send periodic message, for example, with IsiMsgSend()
    // For network variable heartbeats, use propagate()
}
```

Optimizing the Footprint of ISI Applications

The ISI implementation is packaged in several different libraries named **IsiFull.lib**, **IsiCompactAuto.lib**, **IsiCompactManual.lib, IsiCompactS.lib, IsiCompactSHb.lib, IsiCompactDa.lib, IsiCompactDaHb.lib**, and **IsiPl3170.lib**. When creating an application that uses the ISI implementation, you must choose one of these libraries to link to the application. If you are developing an ISI application for a PL 3170 device, you must use the **IsiPl3170.lib** library.

The libraries vary in the supported features and required resources. This section discusses the differences, and helps you choose the best library for your application.

IsiFull

This library contains the complete feature set as defined in this document. The **IsiFull** library may require device resources for library code and constant data that are unavailable for a specific target platform, especially if the platform is based on a 3120 chip. If there are insufficient resources on a device for the **IsiFull** library, then choose one of the other libraries.

IsiCompactAuto

This library supports features contained under *All Compact Libraries*, below, with the added limitation that this implementation only supports automatic connections. The **IsiOpenEnrollment()**, **IsiCreateEnrollment()**, and **IsiExtendEnrollment()** functions are not present in this library, and ISI network messages related to manual enrollment are ignored.

Network variable heartbeats are not supported by this version of the ISI engine, and devices created with this library are not ISI-DA compatible and do not support domain acquisition.

IsiCompactManual

This library supports features contained under *All Compact Libraries*, below, with the added limitation that this implementation only supports manual, but no automatic, connections. The **IsiInitiateAutoEnrollment()** function is not present in this library, and ISI network messages related to automatic enrollment are ignored.

Network variable heartbeats are not supported by this version of the ISI engine, and devices created with this library are not ISI-DA compatible and do not support domain acquisition.

IsiCompactS

This library supports features contained under *All Compact Libraries*, below, with the addition of timing guidance support, allowing devices created with this library to be installed in ISI-DA networks with up to 200 devices, and aliases, which allows for one network variable to participate in multiple connections and usage of the **IsiExtendEnrollment()** call. In addition, it also supports both manual and automatic enrollment.

Network variable heartbeats are not supported by this version of the ISI engine, and devices created with this library do not support domain acquisition.

IsiCompactSHb

This library supports the same features as the IsiCompactS library, with the addition of support for network variable heartbeats, as described in *Sending Periodic Heartbeats*.

Devices created with this library do not support domain acquisition.

IsiCompactDa

This library supports the same features as IsiCompactS, with the addition of support for domain acquisition.

Network variable heartbeats are not supported by this version of the ISI engine.

IsiCompactDaHb

This library supports the same features as IsiCompactDa, with the addition of support for network variable heartbeats, as described in *Sending Periodic Heartbeats.*

All Compact Libraries

All compact ISI libraries support all features detailed under *IsiFull*, except those specifically excluded in the description of the particular version of compact ISI library. The following features are also not supported by compact ISI libraries:

- Direct removal or replacement of existing network variable connections is not supported. Connection invitations will be ignored for assemblies that are already in a connection. The **IsiLeaveEnrollment()** and **IsiDeleteEnrollment()** functions are not present. To remove an existing connection from a device that uses a compact ISI library, call the **IsiReturnToFactoryDefaults()** function to return the device to the factory defaults. To replace an existing connection on such a device, return the device to the factory defaults, and then create the new connection.

- Network variable turnaround connections are not supported with the compact libraries.

- The compact libraries require support by the application to pre-validate resources prior to enrolling with a new connection, whereas the IsiFull library contains more comprehensive, built-in, support for this validation. See the **IsiGetFreeAliasCount()** function for more information.

- Diagnostics are not supported; the **isiFlagSupplyDiagnostics** flag is ignored.

The following table compares the features of each library.

To identify the smallest possible ISI implementation that meets your application needs, start with the leftmost column (**IsiCompactManual.lib**) and proceed towards the right until you have found a library that supports all required features. When in doubt and device resources allowing, prefer using **IsiFull.lib**.

	IsiCompactManual.lib	IsiCompactAuto.lib	IsiCompactS.lib	IsiCompactSHb.lib	IsiCompactDa.lib	IsiCompactDaHb.lib	IsiFull.lib
Core Functions	X	X	X	X	X	X	X
Manual Enrollment	X		X	X	X	X	X
Automatic Enrollment		X	X	X	X	X	X
Aliases (extending enrollment)			X	X	X	X	X
Heartbeats				X		X	X
Connection Removal							X
Turnaround Connections							X
Domain Acquisition					X	X	X
Timing Guidance			X	X	X	X	X
Diagnostics							X

IsiPl3170

The core ISI functions for an ISI-S or ISI-DA device are embedded in the read-only memory (ROM) of a PL 3170 Smart Transceiver. The **IsiPl3170.lib** library supports all of the core features of ISI, except the **IsiUpdateDiagnostics()** callback function.

For a PL 3170 device, you must use the **IsiPl3170.lib** library; if you select a different library, the ISI engine will not start. This library is not supported for other device types.

You must use the Neuron Linker Version 4.04.08, or later, to link your application with the **IsiPl3170.lib** library. Previous versions of the linker do not support this library. Neuron Linker Version 4.04.08 is available with Service Pack 4 or later for the NodeBuilder 3.1 Development Tool.

The ISI engine for a PL 3170 device is delivered in two parts (one part in device ROM and the other in the **IsiPl3170.lib** library). Because the links between the ISI engine in the PL 3170 ROM and the callbacks in the application are defined when the device resets, you must call the **IsiPreStart()** function from the **when(reset)** task before calling any other ISI functions. You must call this function even if you do not plan to start the ISI engine.

The PL 3170 Smart Transceiver does not support the ISI-DAS functions. Therefore, your PL 3170 ISI application should not use DAS-specific functions, such as **IsiProcessMsgDas()** or **IsiApproveMsgDas()**. In addition, your PL 3170 ISI application should not use generic ISI functions, such as **IsiStart()** or

IsiTick(), because using these functions causes the linker to include the more specific functions (including the DAS functions) with the application.

If your application uses the CENELEC Configuration Library functions to enable or disable the CENELEC Media Access protocol at runtime, your ISI application does not need to link with the **cenelec.lib** library because the PL 3170 system image includes these functions.

7

Transitioning between Managed and Self-Installed Networks

This chapter describes how you can transition a self-installed network to a managed network. This is useful when you want to introduce the benefits of a network management tool into an operating self-installed network.

You can upgrade a network from a self-installed network to a managed network. You may do this to add non-ISI devices to an ISI network, to extend the network beyond the ISI limits, or to introduce managed monitoring and control tools to the network.

You can use the LonMaker Integration Tool or another application that implements LNS recovery to automatically analyze an ISI network and create a LNS database reflecting the present network configuration. The LonMaker tool will also automatically create a LonMaker drawing that shows all the channels, devices, functional blocks, and connections in the ISI network. You can then use the LonMaker tool to further enhance and extend the network.

To transition an ISI network to a managed network using the LonMaker tool, follow these steps:

1. Ensure all devices in the network are powered on.

2. Recover the network as described in *Recovering a Network* in the *LonMaker User's Guide*. One of the steps of the recovery procedure is to optionally create device templates based on device interface (XIF) files for your devices. This step is typically optional, but is required for any devices created with the **disable_snvt_si** compiler directive. These devices do not provide a network accessible definition of their interface, and can only be recovered if an XIF file is available. For all devices, an XIF file may have additional information that is not reported by the device itself, and may therefore improve recovery.

3. Open the recovered LonMaker network and verify that all devices were correctly discovered.

4. Change the network to the OnNet management mode.

5. Resynchronize the devices in the network to the LNS database as described in *Resynchronizing the Drawing, Database, and Network* in the *LonMaker User's Guide*. This will update the **SCPTnwrkCnfg** configuration property that is implemented in all ISI devices to disable self-installation and enable configuration by the LonMaker tool and other LNS applications. This will also correct network configuration errors identified during recovery.

You can read the **SCPTnwrkCnfg** configuration property in an ISI device to determine if the device is currently in the self-installed or managed state. If the **SCPTnwrkCnfg** CP is set to **CFG_LOCAL**, then the device is currently self-installed. If it is set to **CFG_EXTERNAL**, then the device is in the managed state.

For more considerations, and example code, see *Implementing a **SCPTnwrkCnfg** CP*.

You can also transition a device from a managed network back to a self-installed network by deinstalling it as described in *Deinstalling a Device* in Chapter 2.

Appendix A

API Data Types

This appendix describes the data types used by the ISI library. Byte offset values shown refer to the **IsiMessage** structure, starting with IsiMessageHeader at index 0.

This ISI API reference describes the ISI API for Neuron C, and uses the Neuron C programming model. The Neuron C language employs a 1-byte-centric, big-endian, programming model. The **char**, **int**, **short**, and **enum** types all use a single byte. The **long** type uses 2 bytes in big-endian ordering. Bitfields are arranged in big-endian order within a single byte boundary. Aggregate packing is 1, aggregate padding is 0.

The Neuron C language boolean constant **FALSE** has a value of 0 (zero) and the boolean constant **TRUE** has a value of 1, Any non-zero value is evaluated as true in a boolean expression, and any function returning a boolean value may return any non-zero value.

ConnDesc Structure

Contains details describing a connection. Used in the **IsiConnection** structure.

Data Type	Field Name	Note
unsigned : 6	Offset	Offset into the larger selector set for connections from compound assemblies with more than four selectors, 0 for connections from simple assemblies, connections from compound assemblies covering four or less than selectors, or the first connection table entry related to a connection from a compound assembly with a width > 4. Also see CSMI.
unsigned : 1	Auto	1 for connections created with automatic enrollment; 0 for connections created with manual or controlled enrollment.
unsigned : 1		Reserved. Set to zero when writing; ignore when reading.

IsiAbortReason Enumeration

Identifies the cause of domain or device acquisition termination. Used as the parameter for the **isiAbort** event for the**IsiUpdateUserInterface()** function.

IsiAbortReason	Value	Note
isiAbortUnsuccessful	1	Domain acquisition stopped after 20 retries, or any other failure to obtain an unambiguous domain ID.
isiAbortMismatchingDidrm	2	Domain acquisition stopped due to arrival of mismatching DIDRM.
isiAbortMismatchingDidcf	3	Domain acquisition stopped due to arrival of mismatching DIDCF.
isiAbortMismatchService	4	Domain acquisition stopped due to mismatching confirmation service message.

IsiCid Structure

Contains the unique connection ID for a connection. The data held by this structure uniquely identifies a connection in an ISI network.

Byte offset	Data Type	Field Name	Note
N	unsigned [5]	UniqueId	A unique identifier for the connection host, based on the host's Neuron ID.
N+5	unsigned long	SerialNumber	Connection host-allocated serial number.

The byte offset N is 1 for the **IsiCid** or **IsiConnectionHeader** structure used with ISI message types, and is 0 (zero) when used with the **IsiConnection** structure.

IsiConnection Structure

Represents a row in the connection table that is read with **IsiGetConnection()** and written using **IsiSetConnection()**.

Data Type	Field Name	Note
IsiConnectionHeader	Header	See **IsiConnectionHeader**
unsigned	Host	Local assembly number that acts as host for this connection, or **ISI_NO_ASSEMBLY** if none.
unsigned	Member	Local assembly number that is member to this connection, or **ISI_NO_ASSEMBLY** if none.
unsigned : 2	State	Use **IsiConnectionState** enumerated values
unsigned : 1	Extend	For pending enrollment: this is a pending connection extension.
unsigned : 1	HaveCsme	For pending enrollment hosts: at least one member has enrolled.
unsigned : 4	Width	Width, as governed by this connection table entry. A single connection table entry can govern up to 4 selectors; connections from compound assemblies with a larger width are split into multiple connection table entries.
ConnDesc structure	Desc.Bf	For—sometimes more efficient—access via 8 bit cardinals, use the **Desc.OffsetAuto** construct; see **isi.h** for definition details.

IsiConnectionHeader Structure

Contains the connection identifier (CID) and selector value for a connection-related ISI message.

Byte offset	Data Type	Field Name	Note
N	IsiCid	Cid	See **IsiCid.**
N+7	unsigned long	Selector	Selector value 0 – 0x2FFF. The most significant 2 bits must be cleared.

The byte offset N is 1 for the **IsiConnectionHeader** structure used with ISI message types, and is 0 (zero) when used with the **IsiConnection** structure.

IsiConnectionState Enumeration

Specifies the state of a connection table entry.

IsiConnectionState	Value	Note
isiConnectionStateUnused	0	The connection table entry is unused.
isiConnectionStatePending	1	This connection table entry refers to a currently or previously open enrollment. A new open enrollment message may override this record.
isiConnectionStateInUse	2	This connection table entry is in use.
isiConnectionStateTcsmr	3	This device will broadcast a CSMR for this connection table entry following the completion of domain acquisition. After sending the CSMR, the connection state returns to **isiConnectionStateInUse**. This state does not apply to devices that do not support domain acquisition.

IsiControl Enumeration

Specifies the requested operation for a controlled enrollment request contained in a control request (CTRQ) message.

IsiControl	Value	Note
isiNoop	0	No action.
isiOpen	1	Call **IsiOpenEnrollment()** using the assembly number passed in as the parameter.
isiCreate	2	Call **IsiCreateEnrollment()** using the assembly number passed in as the parameter.

IsiControl	Value	Note
isiExtend	3	Extend a connection by calling **IsiExtendEnrollment()** using the assembly number passed in as the parameter.
isiCancel	4	Cancel an open (pending or approved) enrollment by calling **IsiCancelEnrollment()**.
isiLeave	5	Remove the specified assembly from all enrolled connections on the local device by calling **IsiLeaveEnrollment()** using the assembly number passed in as the parameter.
isiDelete	6	Remove the specified assembly from all enrolled connections on all devices by calling **IsiDeleteEnrollment()** using the assembly number passed in as the parameter.
isiFactory	7	Restore the device's self-installation data to factory defaults by calling **IsiReturnToFactoryDefaults()**.

IsiCsma Structure

An alias for the **IsiCsmo** data type, see there for details. The **IsiCsma** data type is exchanged with automatic enrollment messages (CSMAs).

IsiCsmc Structure

An alias for the **IsiConnectionHeader** data type, see there for details. The **IsiCsmc** data type is exchanged with enrollment confirmation messages (CSMCs).

IsiCsmd Structure

An alias for the **IsiConnectionHeader** data type, see there for details. The **IsiCsmd** data type is exchanged with enrollment deletion messages (CSMDs).

IsiCsme Structure

An alias for the **IsiConnectionHeader** data type, see there for details. The **IsiCsme** data type is exchanged with enrollment acceptance messages (CSMEs).

IsiCsmi Structure

Defines an enrollment information message (CSMI) sent by the ISI engine.

Byte offset	Data Type	Field Name	Note
1	IsiConnection Header	Header	See **IsiConnectionHeader**
10	unsigned : 6	Offset	Offset into the selector set of a connection from a compound assembly with five or more selectors; zero for a connection from a simple assembly or a connection from a compound assembly using less than five selectors.
10	unsigned : 2	Count	Number of selectors governed by this CSMI message minus one, which is a value between zero and three. The value in this field is one less than the width controlled by this CSMI message.

IsiCsmo Structure

Defines a manual open enrollment message (CSMO).

Byte offset	Data Type	Field Name	Note
1	IsiConnection Header	Header	See **IsiConnectionHeader**.
10	IsiCsmoData	Data	See **IsiCsmoData**.

IsiCsmoData Structure

Contains the fields of a CSMO message to be sent by the ISI engine, and is constructed in **IsiCreateCsmo()**.

Byte offset	Data Type	Field Name	Note

Byte offset	Data Type	Field Name	Note
N	unsigned	Group	The group (or device category) this connection applies to. See text for valid group ID values, and allocation of group ID values.
N+1	unsigned : 2	Direction	Direction of the primary network variable on offer. See the **IsiDirection** enumeration.
N+1	unsigned : 6	Width	Number of selector values used with this connection, starting with the value of the **Selector** field. Value 0 is reserved.
N+2	unsigned long	Profile	Functional profile number of the functional profile that defines the functional block containing the primary network variable, or zero if none.
N+4	unsigned	NvType	NV type index of the NV type for the primary network variable, or zero if none specified. The NV type index is an index into resource file that defines the network variable type for the primary network variable on offer.

Byte offset	Data Type	Field Name	Note
N+5	unsigned	Variant	Variant number for the offered connection. Variants may be defined for any device category and/or functional profile/member number pair. Variants allow different interpretations of the offered data, based on a variant value. Set to zero unless otherwise known. Values 1 – 127 are standard variant values specified by the *ISI Protocol Specification* and by ISI profiles published by LONMARK International. Values 128 – 254 are available for use by manufacturer-specific connections. Value 255 is reserved.
N+6	unsigned : 1	Extended. Acknowledged	Normally cleared. For regular CSMO, CSMA, and CSMR messages, this field is always zero. You must specify **isiFlagExtended** when starting the ISI engine to use this field. See text for special considerations.
N+6	unsigned : 1	Extended.Poll	Normally cleared. For regular CSMO, CSMA, and CSMR messages, this field is always zero. You must specify the **isiFlagExtended** flag when starting the ISI engine to use this field. See text for special considerations.

Byte offset	Data Type	Field Name	Note
N+6	unsigned : 2	Extended. Scope	Scope of the resource file containing the functional profile and network variable type definitions specified by the **Profile** and **NvType** fields. See the **IsiScope** enumeration. Values 1 and 2 are reserved. For regular CSMO, CSMA, and CSMR messages, this field is always zero. You must specify the **isiFlagExtended** flag when starting the ISI engine to use this field.
N+6	unsigned : 4		Reserved. Clear when sending, ignore when receiving. For regular CSMO, CSMA, and CSMR messages, this field is always zero.
N+7	unsigned [6]	Extended. Application	The first 6 bytes of the connection host's standard program ID. The last two standard program ID bytes (channel type and model number) are not included. For regular CSMO, CSMA, and CSMR messages, this field is always zero. You must specify the **isiFlagExtended** flag when starting the ISI engine to use this field.
N+13	unsigned	Extended. Member	NV member number within the functional profile for the primary network variable, or zero if none. For regular CSMO, CSMA, and CSMR messages, this field is always one. You must specify the **isiFlagExtended** flag when starting the ISI engine to use this field.

The byte offset N is 10 for the **IsiCsmoData** structure used with ISI message types (**IsiCsmo** structure), and is zero when used with the **IsiGetAssembly()** and **IsiGetNextAssembly()** functions.

IsiCsmr Structure

An alias for the **IsiCsmo** data type, see there for details. The **IsiCsmr** data type is used with automatic enrollment reminder messages (CSMRs).

IsiCsmx Structure

An alias for the **IsiConnectionHeader** data type, see there for details. The **IsiCsmx** data type is used with enrollment cancellation messages (CSMXs).

IsiCtrp Structure

Defines a control response (CTRP) message used for controlled enrollment.

Byte Offset	Data Type	Field Name	Note
1	boolean	Success	Non-zero if the requested operation is supported, **FALSE** otherwise.
2	unsigned[6]	NeuronID	Neuron ID of the device sending the control response.

IsiCtrq Structure

Defines a control request (CTRQ) message used for controlled enrollment.

Byte Offset	Data Type	Field Name	Note
1	IsiControl	Control	Requested operation.
2	unsigned	Parameter	Parameter to be used with the requested operation passed in through the **Control** field. See the **IsiControl** enumeration for more details.

IsiDiagnostic Enumeration

Represents the possible diagnostic values that are passed into **IsiUpdateDiagnostics()**.

IsiDiagnostic	Value	Note

IsiDiagnostic	Value	Note
isiSubnetNodeAllocation	1	A local subnet/node ID has been allocated.
isiSubnetNodeDuplicate	2	A subnet/node ID duplicate has been detected.
isiReceiveDrum	4	DRUM or DRUMEX message received.
isiReceiveTimg	5	TIMG message received.
isiSendPeriodic	6	Periodic message other than an NV heartbeat message sent (see **IsiQueryHeartbeat()** for these). The ISI message code for the message sent is in the numeric parameter. DRUM and DRUMEX messages are reported with the same message code (0), and CSMR and CSMREX are reported with the same message code (6).
isiSelectorDuplicate	7	Selector duplicate detected. The parameter indicates the associated assembly.
isiSelectorUpdate	8	Selector update detected. The parameter indicates the associated assembly.
isiReallocateSlot	9	Broadcasting slot has been reallocated as a result of message spreading.

IsiDidcf Structure

An alias of the **IsiDidrm** data type, see there for details. The **IsiDidcf** data type is used with domain confirmation messages (DIDCFs).

IsiDidrm Structure

Defines a domain response message.

Byte Offset	Data Type	Field Name	Note
1	unsigned : 3	DidLength	Number of significant bytes in the DID array, may be 1, 3, or 6.
1	unsigned : 5		Reserved; set to zero when sending, ignore when receiving.

Byte Offset	Data Type	Field Name	Note
2	unsigned [6]	Did	Domain ID. This field is always 6 bytes long; the number of significant bytes is contained in the **DIDlength** field
8	unsigned [6]	NeuronId	Neuron ID of the DAS, used for matching DIDRM messages with DIDCF messages.
14	unsigned	DeviceCountEst	Estimated device count used for throttling periodic messages. The device count is an approximation and will typically be larger than the actual network size.
15	unsigned	ChannelType	The least performing channel ID observed.

IsiDidrq Structure

Defines a domain request message.

Byte Offset	Data Type	Field Name	Note
1	unsigned [6]	NeuronId	Neuron ID of the requesting device.
7	unsigned	Nuid	Non-unique device ID. This ID is a random number between 0 and 255 that may be used by a DAS to approximate the number of devices within earshot. This number does not need to be unique between devices, but a statistic distribuition of **Nuid** values is required.

IsiDirection Enumeration

Specifies the direction of the primary network variable on offer in a CSMO.

IsiEvent	Value	Note
isiDirectionOutput	0	Indicates the network variable is an output NV.
isiDirectionInput	1	Indicates the network variable is an input NV.

IsiEvent	Value	Note
isiDirectionAny	2	Indicates that both input and output network variables may be applicable to this connection. Used to support N:M connections.
isiDirectionVarious	3	Indicates the network variable direction is distinct and a combination of inputs and outputs. Used in compound assemblies, where the network variable directions are defined by the functional profile.

IsiDrum Structure

Defines a domain resource usage message (DRUM/DRUMEX).

Byte Offset	Data Type	Field Name	Note
1	unsigned : 3	DidLength	States the number of significant bytes in the DID array, may be 1, 3, or 6.
1	unsigned : 3		Reserved. Set to zero when sending, ignore field when receiving DRUM message unless otherwise known.
1	unsigned : 2	UserDefined	Reserved. Set to zero when sending, ignore when receiving.
2	unsigned [6]	DomainId	Primary domain ID (see **DidLength** for number of significant bytes).
8	unsigned [6]	NeuronId	Sender's Neuron ID.
14	unsigned	SubnetId	Sender's current subnet ID.
15	unsigned	NodeId	Sender's current node ID.
16	unsigned	Nuid	Sender's non-unique device ID. See the **IsiDidrq** structure for a description.
17	unsigned	ChannelType	Value from the *Channel Type* standard program ID field.

Byte Offset	Data Type	Field Name	Note
18	unsigned long	Extended.Device Class	Value from the *Device Class* standard program ID field. For regular DRUM messages, this field is always zero. You must specify the **isiFlagExtended** flag when starting the ISI engine to use this field.
20	unsigned	Extended.Usage	Value from the *Usage* standard program ID field. For regular DRUM messages, this field is always zero. You must specify the **isiFlagExtended** flag when starting the ISI engine to use this field.

IsiEvent Enumeration

Specifies an event that is passed into **IsiUpdateUserInterface()** when the ISI engine does a callback.

IsiEvent	Value	Note
isiNormal	0	User interface needs to be reset to the normal, or idle, state. The parameter value is always set to **ISI_NO_ASSEMBLY**.
isiRun	1	ISI engine successfully started (**parameter = 1**) or stopped (**parameter = 0**).
isiPending	2	Connection member related to the assembly given with the numerical parameter entered the pending state. The device has received an acceptable CSMO, but has not accepted the enrollment yet.
isiApproved	3	Connection member related to the assembly given with the numerical parameter changed from pending to approved. Occurs when a pending enrollment has been accepted.
isiImplemented	4	Approved connection has been implemented. The parameter identifies the assembly. For a connection host this occurs when the device invokes **IsiCreateEnrollement()** or **IsiExtendEnrollment()**. For a connection member this event occurs at the receipt of the first CSMC.

IsiEvent	Value	Note
isiCancelled	5	Connection has been cancelled by a timeout, user, or network action. The parameter is always set to **ISI_NO_ASSEMBLY**.
isiDeleted	6	Connection has been deleted. The parameter identifies the assembly.
isiWarm	7	ISI engine has warmed up—i.e., some reasonable, but random, time has passed since the last reset. From this moment on, the application may call the **IsiInitiateAutoEnrollment()** function.
isiPendingHost	8	Connection host related to the assembly given with the numerical parameter entered the pending state. The device has issued the CSMO but not yet received a CSME.
isiApprovedHost	9	Connection host related to the assembly given with the numerical parameter changed from the pending state to the approved state. Occurs at the receipt of the first CSME.
isiAborted	10	Device stopped the domain or device acquisition. The parameter is a member of the **IsiAbortReason** enumeration and indicates the reason for the abort.
isiRetry	11	The device is retrying device acquisition. The parameter is set to the remaining number of retries.
isiWink	12	Device should perform its wink function. The specific function is application-dependent, but should provide some visible feedback to the user. For example, the application may blink an LED on the device.
isiRegistered	13	Indicates either acquisition start or successful acquisition completion process on either an ISI-DA or ISI-DAS device. The parameter indicates either a successful start (**parameter = 0**) or completion (**parameter = 0xFF**).

IsiFlags Enumeration

Specifies option flags for the **IsiStart()** functions used to start the ISI engine. These flags may be combined.

IsiFlags	Value	Note
isiFlagNone	0x00	Typical default value with no flags specified.

IsiFlags	**Value**	**Note**
isiFlagExtended	0x01	Enables the use of the extended DRUMEX, CSMOEX, CSMAEX, and CSMREX messages.
isiFlagHeartbeart	0x02	Enables automatic network variable heartbeats. Heartbeats are disabled by default. Automatic heartbeats apply to all bound output network variables and related aliases. See *Optimizing Footprint of ISI-enabled Applications* for which libraries this feature can be used in.
isiFlagApplicationPeriodic	0x04	Enables the **IsiCreatePeriodicMsg()** callback, which allows an application to participate in the periodic broadcast schedule. May not be supported by all versions of the ISI library; implementations that do not support this feature ignore this flag, if set. The feature is disabled by default.
isiFlagSupplyDiagnostics	0x08	Enables additional diagnostics data to be provided to the application via the **IsiUpdateDiagnostics()** callback. This callback must be explicitly enabled by raising the **isiFlagSupplyDiagnostics** flag. See *Optimizing Footprint of ISI-enabled Applications* for which libraries this feature can be used in.

IsiMessage Structure

Contains the header and body of an ISI protocol message. These messages are typically sent using **IsiSendMsg()**.

Byte Offset	IsiMessage	Data Type	Note
0	Header	IsiMessageHeader	Message Header
1	Msg	union	Contents of the message. Union of the following ISI messages: IsiCsma Csma, IsiCsmc Csmc, IsiCsmd Csmd, IsiCsme Csme, IsiCsmi Csmi, IsiCsmo Csmo, IsiCsmr Csmr, IsiCsmx Csmx, IsiCtrp Ctrp, IsiDidcf Didcf, IsiDidrm Didrm, IsiDidrq Didrq, IsiDrum Drum, IsiRdct Rdct, IsiRdcs Rdcs, IsiTimg Timg

IsiMessageCode Enumeration

Specifies the ISI message type in an **IsiMessageHeader** structure.

IsiMessageCode	Value	Note
isiDrum	0x00	Domain resource usage information
isiDrumEx	0x01	Extended domain resource usage information
isiCsmo	0x02	Connections: open enrollment
isiCsmoEx	0x03	Connections: extended open enrollment
isiCsma	0x04	Connections: automatic open enrollment
isiCsmaEx	0x05	Connections: extended automatic open enrollment
isiCsmr	0x06	Connections: automatic enrollment reminder
isiCsmrEx	0x07	Connections: extended automatic enrollment reminder
isiLastEx	isiCsmrEx	Last extended message code
isiDidrq	0x08	Domain ID request

IsiMessageCode	Value	Note
isiDidrm	0x09	Domain ID response
isiDidcf	0x0A	Domain ID confirmation
isiTimg	0x0B	Timing guidance message
isiCsmx	0x0C	Connections: cancel enrollment
isiCsmc	0x0D	Connections: close and confirm enrollment
isiCsme	0x0E	Connections: accept enrollment
isiCsmd	0x0F	Connections: delete connection
isiCsmi	0x10	Connections: status and resource information
isiCtrq	0x11	Controlled enrollment: control request
isiCtrp	0x12	Controlled enrollment: control response
isiRdct	0x13	Controlled enrollment: read connection table request
isiRdcs	0x14	Controlled enrollment: read connection table success
isiRdcf	0x15	Controlled enrollment: read connection table failure
isiLastCode	isiRdcf	Last message code
isiCodeMask	0x1F	Mask for ISI message codes

IsiMessageHeader Structure

Contains the header that is sent with all ISI messages.

Byte offset	Data Type	Field Name	Note
0	IsiMessageCode	Code	ISI message code. See **IsiMessageCode** enumeration.

IsiRdcs Structure

Defines a read connection table success (RDCS) response message used for controlled enrollment.

Byte offset	Data Type	Field Name	Note
1	unsigned	Index	Index of the connection table that is being sent.
2	IsiConnection	Data	Connection table entry.

IsiRdct Structure

Defines a read connection table (RDCT) request message used for controlled enrollment.

Byte offset	Data Type	Field Name	Note
1	unsigned	Index	Connection table index to return, or the starting index to start searching from (if **Host** or **Member** are not **ISI_NO_ASSEMBLY**).
2	unsigned	Host	Host assembly to search for in the connection table, or **ISI_NO_ASSEMBLY**. Support for this field is optional. Devices supporting the RDCT request but not the **Host** field may always return the entry specified by the **Index** field, or **isiRdcf** if an index is not specified.
3	unsigned	Member	Member assembly to search for in the connection table, or **ISI_NO_ASSEMBLY**. Support for this field is optional. Devices supporting the RDCT request but not the **Member** field may always return the entry specified by the **Index** field, or **isiRdcf** if an index is not specified.

IsiScope Enumeration

Specifies the scope of the data types referenced in a CSMO message. This value represents the scope of the resource file containing the functional profile and network variable type definitions specified by the **Profile** and **NvType** fields.

IsiScope	Value	Note
isiScopeStandard	0	Indicates that the **Profile** and **NvType** values are defined in the standard resource file

IsiScope	Value	Note
isiScopeManufacturer	3	Indicates that **Profile** and **NvType** values refer to user-defined scope 3 – 5 resource files that match the applicable fields in the **ProgramIDShort** field

All other values are reserved.

IsiTimg Structure

Defines a timing guidance message (TIMG).

Byte Offset	Data Type	Field Name	Note
1	unsigned : 4	Originator	Set to 8 for a regular DAS. All other values are reserved.
1	unsigned : 4		Reserved.
2	unsigned	DeviceCountEst	Estimated device count used for throttling periodic messages. The device count is an approximation and will typically be larger than the actual network size.
3	unsigned	ChannelType	The least performing channel type ID observed.

IsiType Enumeration

Specifies the ISI protocol to be implemented by the ISI engine.

IsiType	Value	Note
isiTypeS	0	ISI without domain acquisition.
isiTypeDa	1	ISI with domain acquisition.
isiTypeDas	2	ISI-DAS.

Appendix B

API Functions

This appendix describes the functions provided by the ISI library.

Important: To minimize its memory requirements, the ISI engine performs only limited function parameter verification. You must ensure that your ISI application passes correct parameters to the ISI functions.

IsiAcquireDomain()

void **IsiAcquireDomain**(boolean *SharedServicePin*);

Starts or re-starts the domain ID acquisition process in a device that supports domain acquisition. The **IsiAcquireDomain()** function must not be called from a DAS or device that does not support domain acquistion.

The **IsiFetchDevice()** function provides an alternate method to assign a domain from a DAS. A DAS must support both methods.

When *SharedServicePin* is set to **FALSE**, calling this function will always result in a service pin message, regardless of the state of the ISI engine. When it is set to **TRUE**, a service pin message will not be sent. If your application senses the service pin state and uses that to call the **IsiAcquireDomain()** function, call it with *SharedServicePin* set to **TRUE**. This indicates that the device has already sent the service pin message, and will not send its own. If your application uses some other UI to start the domain acquisition procedure, then call **IsiAcquireDomain()** with *SharedServicePin* set to **FALSE**. With the ISI engine running, this begins the domain acquisition procedure, and with the engine stopped it simply issues a service pin message.

EXAMPLE

The following example starts domain acquisition on an ISI-DA device when the user pushes a Connect button on the device:

```
when (connect_button_pressed) {
    IsiAcquireDomain(FALSE);
}
```

IsiApproveMsg()

boolean **IsiApproveMsg**(void);

Determines if an incoming message is an ISI message by verifying the message code and length. If approved, the received message may be processed by one of the **IsiProcessMsg()** functions.

The **IsiApproveMsg()** function always returns **FALSE** if the ISI engine is stopped.

When implementing a domain address server, use **IsiApproveMsgDas()** instead.

EXAMPLE

The following example tests for incoming ISI messages, and calls **IsiProcessMsgS()** to process them:

```
when (msg_arrives) {
    if (IsiApproveMsg() && IsiProcessMsgS()) {
        // TODO: process unprocessed ISI messages here (if any)
    } else {
        // TODO: process other application messages here (if any)
    }
}
```

IsiApproveMsgDas()

boolean **IsiApproveMsgDas**(void);

Determines if an incoming message is an ISI or service pin message by verifying the message code and length. If approved, the received message may be processed by one of the **IsiProcessMsg()** functions. This function must be used on ISI-DAS devices instead of **IsiApproveMsg()**. The function has the same functionality as **IsiApproveMsg()**, but also approves additional messages required by a domain address server. This function does not approve messages belonging to the response queue; use **IsiProcessResponse()** to process those.

The **IsiApproveMsgDas()** function always returns **FALSE** if the ISI engine is stopped.

EXAMPLE

The following partial example tests for incoming ISI messages and responses on a domain address server. For a complete implementation of a DAS device, the **IsiStartDas()** and **IsiTickDas()** functions must also be called at a minimum:

```
when (msg_arrives) {
    if (IsiApproveMsgDas() && IsiProcessMsgDas()) {
        // TODO: process unprocessed ISI messages here (if any)
    } else {
        // TODO: process other application messages here (if any)
    }
}

when (resp_arrives) {
    if (IsiProcessResponse()) {
        // TODO: process unprocessed responses here (if any)
    }
}
```

IsiCancelAcquisition()

void **IsiCancelAcquisition**(void);

Cancels both device and domain acquisition. Following the completion of this function call, **IsiUpdateUserInterface()** is called with the **isiNormal** event.

You must use **IsiCancelAcquisitionDas()** on a domain address server instead.

This function has no effect unless the ISI engine is running and the device is either in device or domain acquisition mode.

IsiCancelAcquisitionDas()

void **IsiCancelAcquisitionDas**(void);

The **IsiCancelAcquisitionDas()** function is only available to domain address servers. The function provides a superset to the regular **IsiCancelAcquisition()** function, additionally resetting the extended state information unique to a domain address server.

This function has no effect unless the ISI engine is running and the device is either in device or domain acquisition mode.

IsiCancelEnrollment()

void **IsiCancelEnrollment**(void);

Cancels an open (pending or approved) enrollment. When used on a connection host, a CSMX connection cancellation message is issued to cancel enrollment on the connection members. When used on a device that has accepted, but not yet implemented, an open enrollment, this causes the device to opt out of the enrollment locally.

The function has no effect unless the ISI engine is running and in the pending or approved state.

EXAMPLE

The following example cancels enrollment on a connection host if the Connect button is pushed prior to any devices accepting the invitation, or if the Connect button is held down for an extended period:

```
when (io_changes(...)) {
   switch(DeviceState) {
      …
      case isiApprovedHost:
          if (Long) {
              IsiCancelEnrollment();
          } else {
              IsiCreateEnrollment(Assembly);
          }
          break;
      case isiApproved:
          if (Long) {
              IsiCancelEnrollment();
          }
          break;
      …
    }
}
```

IsiCreateEnrollment()

void **IsiCreateEnrollment**(unsigned *Assembly*);

Accepts a connection invitation. This function may be called after the application has received and approved a CSMO open enrollment message. If the assembly is not already in a connection, or if the assembly is in a connection and the device supports direct connection removal, the connection will be created anew. If the assembly is already in a connection, any previous connection information will be replaced. This function must not be called with an assembly that is already in a connection on a device that does not support direct connection removal.

On a connection host that has received at least one CSME enrollment acceptance message, this command completes the enrollment and implements the connection as new, replacing any previously existing enrollment information associated with this assembly.

Calling this function on a device that does not support connection removal while indicating an assembly number that is already engaged in another connection, will not implement the new connection. The **isiImplemented** event will not be fired in this case. The application may use the **IsiIsConnected()** function to determine if a given assembly is currently engaged in a connection.

Where supported and unless application requirements dictate otherwise, the **IsiExtendEnrollment()** function should be used instead.

The ISI engine must be running and in the correct state for this function to have any effect. For a connection host, the ISI engine must be in the approved state. Other devices must be in the pending state.

EXAMPLE

The following example accepts a connection invitation on a connection member if the Connect button is pushed when a connection is pending:

```
when (io_changes(...)) {
   switch(DeviceState) {
         …
      case isiPending:
          IsiCreateEnrollment(Assembly);
          break;
      …
      }
   }
}
```

IsiDeleteEnrollment()

void **IsiDeleteEnrollment(**unsigned *Assembly***)**;

Removes the specified assembly from all connections, and sends a CSMD connection deletion message to all other devices in each connection to remove them from the connection as well. This function has no effect if the ISI engine is stopped.

EXAMPLE

The following removes an assembly from all connections when the Connect button is held for a long period during normal operation.

```
when (io_changes(...)) {
   switch(DeviceState) {
      …
      case isiNormal:
         if (Long) {
            IsiDeleteEnrollment(Assembly);
         } else {
            IsiOpenEnrollment(Assembly);
         }
         break;
      }
   }
}
```

IsiExtendEnrollment()

void **IsiExtendEnrollment**(unsigned *Assembly*);

Accepts a connection invitation on a device that supports connection extension. This function may be called after the application has received and approved a CSMO open enrollment message. The connection will be added to any previously existing connections. If no previous connection exists for *Assembly*, a new connection will be created. This function must not be called on a device that does not support connection extension.

Where supported and unless application requirements dictate otherwise, this function should be used instead of the **IsiCreateEnrollment()** function.

On a connection host that has received at least one CSME enrollment acceptance message, this command completes the enrollment and extends any existing connections. If no previous connection exists for *Assembly*, a new connection will be created.

Devices using the IsiFull library automatically use comprehensive resource validation included with the full version of the ISI library. See **IsiGetFreeAliasCoung()** for discussion and recommendations when using this function with devices built with one of the compact ISI libraries.

The ISI engine must be running and in the correct state for this function to have any effect. For a connection host, the ISI engine must be in the approved state. Other devices must be in the pending state.

EXAMPLE

The following example accepts a connection invitation for a connection member if the Connect button is pushed when a connection is pending, or extends an existing connection if the **Alternative** flag is set:

```
IsiEvent isiState;

void ProcessIsiButton(unsigned Assembly, boolean Alternative) {
   switch(isiState) {
      ...
      case isiPending:
         if (Alternative) {
            IsiExtendEnrollment(Assembly);
         } else {
            IsiCreateEnrollment(Assembly);
         }
         break;
      ...
   }
}
```

IsiFetchDevice()

void **IsiFetchDevice**(void);

Fetches a device by assigning a domain to the device from a domain address server (DAS). The **IsiFetchDevice()** function must not be called from a device that is not a domain address server.

To use the **IsiFetchDevice()** function, you must implement a **when (resp_arrives)** task and call the **IsiProcessResponse()** function from the domain address server's application to complete the processing involved with fetching a remote device.

The **IsiFetchDevice()** operation does not require application code on the remote device. The remote device remains unaware of the change to its primary domain.

An alternate method to assign a domain to a device is for the device to use the **IsiAcquireDomain()** function. This function provides more immediate recovery from network addressing conflicts and more immediate maintenance of automatic connections. The disadvantage of using the **IsiAcquireDomain()** function is that it typically requires more code on the device, and it requires that the device support ISI-DA. The **IsiFetchDevice()** function may be used with any device. A DAS must support both methods.

The ISI engine must be running for this function to have any effect, and this function only operates on a domain address server.

EXAMPLE

The following example is part of the domain address server's application. The example assigns the domain address server's domain to a device when the user presses the Connect button on the server:

```
when (connect_button_pressed) {
   IsiFetchDevice();
}

// Handle responses to requests in IsiFetchDomain()
when (resp_arrives) {
   if (IsiProcessResponse()) {
      // TODO: process unprocessed responses here (if any)
   }
}
```

IsiFetchDomain()

void **IsiFetchDomain**(void);

Starts or re-starts the fetch domain process in a domain address server, described above in *Fetching a Domain for a DAS*. The **IsiFetchDomain()** function must not be called from a device that is not a domain address server.

The ISI engine must be running for this function to have any effect, and this function only operates on a domain address server.

To use the **IsiFetchDomain()** function, you must implement a **when (resp_arrives)** task and call the **IsiProcessResponse()** function function with the domain address server's application to complete the processing involved with fetching a domain ID.

Example

The following example is part of the domain address server's application. The example fetches the domain ID from a remote device and assigns it to a domain address server when the user presses the Connect button on the server:

```
when (connect_button_pressed) {
   IsiFetchDomain();
}

// Handle responses to requests in IsiFetchDomain()
when (resp_arrives) {
   if (IsiProcessResponse()) {
      // TODO: process unprocessed responses here (if any)
   }
}
```

IsiGetFreeAliasCount()

unsigned **IsiGetFreeAliasCount**(void);

Returns the number of unused network variable aliases on the local device. This function may be used with applications using one of the compact ISI libraries that do not supply fully automated resource validation, or with connection hosts. Devices that share a single network variable among multiple network variable selectors within one connection also must validate available resources with this function.

On devices that are built with the IsiFull library, which includes fully automated, built-in, comprehensive resource validation, an attempt to extend an existing connection with an explicit **IsiExtendEnrollment()** call or with the acceptance of an automatic enrollment, leads to this validation step. The step, implemented with the ISI engine, determines exactly how many local aliases are currently unused, and how many are required to implement the pending connection. If those required exceed those available, the connection will not be implemented. On a host, the enrollment will be cancelled.

The compact ISI libraries do not include this validation step, but applications may use the **IsiGetFreeAliasCount()** function to implement a more compact, and possibly overestimating, version of this validation step. This more compact code may take application-specific knowledge into account, and it may ignore previous connection history for simplicity and brevity, thereby possibly leaning towards an over-cautious decision.

Example 1:

This example illustrates how an application built with one of the compact ISI libraries may use the **IsiGetFreeAliasCount()** function to implement an application-specific resource validation step. This simple implementation assumes that every connection extension requires one network variable alias per selector, and rejects the open enrollment message if insufficient network variable aliases are available to meet that assumption:

```
unsigned IsiGetAssembly(const CsmoData* pCsmo, boolean Auto) {
    unsigned candidateAssembly;
    // Determine an eligible assembly based on local knowledge and
    // inspection of data provided with pCsmo and Auto arguments
    candidateAssembly = …
    return IsiGetFreeAliasCount() >= pCsmo->Width ?
        candidateAssembly : ISI_NO_ASSEMBLY;
}
```

Example 2:

This example illustrates how a connection host can verify that sufficient local resources are available using **IsiGetFreeAliasCount ()**, prior to opening enrollment:

```
void myValidatedOpenEnrollment (unsigned assembly) {
    if (IsiGetFreeAliasCount (assembly) >= IsiGetWidth (assembly)) {
        IsiOpenEnrollment (assembly) ;
    }
}
```

This example assumes each network variable that is part of the connection requires a network variable alias. The assumption ignores that some of the network variables may not require an alias because they are not enrolled in a connection, and ignores advanced connection schemes that use network variable selector or network variable sharing. This is a worst-case assumption.

Typical applications may have better knowledge of the resource requirements, and will often be able to provide a more efficient resource validation step. For example,say you have an application that always uses **IsiCreateEnrollment()** and never calls **IsiExtendEnrollment()** for the assembly in question. That application never needs to validate available aliases for this assembly. Or, an application that only implements a single assembly may find more efficient ways to ensure the availability of sufficient network variable aliases.

IsiImplementationVersion()

unsigned **IsiImplementationVersion**(void);

Returns the version number of this ISI implementation, which is 3 for the Release 3 and Release 3.01 ISI libraries. No forwarder is provided for the **IsiImplementationVersion()** function. This function operates in any state of the ISI engine.

IsiInitiateAutoEnrollment()

void **IsiInitiateAutoEnrollment**(const IsiCsma* *pCsma*, unsigned *Assembly*);

Starts automatic enrollment. The local device will become the connection host. Automatic enrollment may replace previous connections, if any. No forwarder exists for this function. Once this call returns, then the ISI connection is implemented for the associated assembly.

This function cannot be called before the **isiWarm** event has been signaled in the **IsiUpdateUserInterface()** callback.

This function does nothing when the ISI engine is stopped.

EXAMPLE

The following example opens automatic enrollment:

```
void IsiUpdateUserInterface(IsiEvent Event, unsigned Parameter) {
    if (Event == isiWarm && !IsiIsConnected(MyAssembly)) {
        IsiInitiateAutoEnrollment(&MyCsmoData, MyAssembly);
    }
}
```

IsiIsBecomingHost()

boolean **IsiIsBecomingHost**(unsigned *Assembly*);

Returns **TRUE** if **IsiOpenEnrollment()** has been called for the specified assembly and the enrollment has not yet timed out, been cancelled, or confirmed. The function returns **FALSE** otherwise.

You can also override the **IsiUpdateUserInterface()** callback function and track the state of the assembly. **IsiIsBecomingHost()** returns **TRUE** if that state is **isiPendingHost** or **isiApprovedHost**.

The **IsiIsBecomingHost()** function does not support a forwarder. The function operates whether the ISI engine is running or not.

IsiIsConnected()

boolean **IsiIsConnected**(unsigned *Assembly*);

Returns **TRUE** if the specified assembly is enrolled in a connection.

No forwarder is provided for **IsiIsConnected()**. The function operates whether the ISI engine is running or not. The function returns **FALSE** if the ISI engine is stopped.

IsiIsRunning()

boolean **IsiIsRunning**(void);

Returns **TRUE** if the ISI engine is running and **FALSE** if the ISI engine is stopped. Non-zero values are considered **TRUE**.

No forwarder is provided for this function. The function operates in all states of the ISI engine.

IsiIssueHeartbeat()

boolean **IsiIssueHeartbeat**(unsigned *NvIndex*);

Sends a network variable update for the specified network variable index and its aliases, as long as the network variable is a bound output network variable using group addressing. The function returns **TRUE** if at least one update has been propagated, and **FALSE** otherwise. **IsiIssueHeartbeat()** uses the address table for addressing information, but always uses unacknowledged service with one repeat when issuing the heartbeat. **IsiIssueHeartbeat()** only sends a network variable if it uses group addressing (the default ISI addressing mode)—it skips updates that use a different addressing mode.

This function is typically called in an **IsiQueryHeartbeat()** callback. If the function is called outside of this callback to create a custom heartbeat scheme, the application must only call this function for bound output network variables.

This function requires that the ISI engine has been started with the **isiFlagHeartbeat** flag passed in.

EXAMPLE

The following starts the ISI engine with heartbeats enabled and issues a heartbeat in response to the **IsiQueryHeartbeat()** callback.

```
when (reset) {
    IsiStartDa(isiFlagHeartbeat);
}

boolean IsiQueryHeartbeat(unsigned nv) {
    // Agree to all heartbeats suggested by the ISI engine
    return IsiIssueHeartbeat(nv);
}
```

IsiLeaveEnrollment()

void **IsiLeaveEnrollment**(unsigned *Assembly*);

Removes the specified assembly from all enrolled connections as a local operation only. When used on the connection host, the function will automatically be interpreted as **IsiDeleteEnrollment()**.

No forwarder is provided for this function. This function has no effect if the ISI engine is stopped.

IsiMsgDeliver()

void **IsiMsgDeliver(**IsiMessage* *pMsg*, unsigned *Length*, msg_out_addr* *pDestination***)**;

Sends an ISI message to the address specified in the *pDestination* parameter.

An ISI application must implement the **IsiMsgDeliver()** function. The function is used by the ISI library for sending messages, and should not be used by the application itself. In most cases, the application code need not explicitly declare the **IsiMsgDeliver()** function; it is automatically implemented by including the **isi.h** header file. The function that should be used for sending messages is **IsiMsgSend()**.

No forwarder is provided for this function. This function operates in any state of the ISI engine.

IsiMsgSend()

void **IsiMsgSend(**IsiMessage* *pMsg*, unsigned *IsiMessageLength*, service_type *Service*, msg_out_addr* *pDestination***)**;

Sends an ISI message of the specified length using the specified service to the specified address, for example from a function invoked via an **IsiCreatePeriodicMsg()** event.

No forwarder is provided for this function. This function operates in any state of the ISI engine.

IsiOpenEnrollment()

void **IsiOpenEnrollment(**unsigned *Assembly***)**;

Opens manual enrollment for the specified assembly. This operation turns the device into a connection host for this connection and sends a CSMO manual connection invitation to all devices in the network.

No forwarder is provided for this function. The ISI engine must be running, and in the idle state.

EXAMPLE

The following example opens manual enrollment for a simple assembly with one network variable, using the network variable's global index as the application-specific assembly number:

```
void startEnrollment(void) {
    unsigned myAssembly;
    myAssembly = nvoValue::global_index;
    IsiOpenEnrollment(myAssembly);
}
```

IsiPreStart()

void **IsiPreStart**(void);

Establishes runtime links between the ISI engine in the read-only memory (ROM) of a PL 3170 Smart Transceiver and the callbacks in the application.

For PL 3170 devices, you must call the **IsiPreStart()** function from the **when(reset)** task before calling any other ISI functions. You must call this function even if you do not plan to start the ISI engine.

Important: The **IsiPreStart()** function is supported only for PL 3170 devices, and is not supported for other device types.

EXAMPLE 1

The following example demonstrates the use of the **IsiPreStart()** function within the **when(reset)** task.

```
network input SCPTnwrkCnfg cp cp_info(reset_required,
device_specific) cpNetConfig;

device_properties {
   cpNetConfig = CFG_EXTERNAL;
};

when (reset) {
   IsiPreStart();
   if (cpNetConfig == CFG_LOCAL) {
      // Self-installed network--start the ISI engine
      scaled_delay(31745UL); // 800ms delay
      IsiStartS(isiFlagNone);
   }
}
```

EXAMPLE 2

The following example demonstrates reset tracking, in addition to the use of the **IsiPreStart()** function within the **when(reset)** task.

```
network input SCPTnwrkCnfg cp cp_info(reset_required) cpNetConfig
= CFG_EXTERNAL;
eeprom SCPTnwrkCnfg oldNetConfig = CFG_NUL;

when (reset) {
   SCPTnwrkCnfg netConfig;
   netConfig = oldNetConfig;

   // For a PL 3170 device only, establish links to callbacks
   IsiPreStart();

   if (netConfig == CFG_NUL) {
      // First application start, enable self-installation
      cpNetConfig = CFG_LOCAL;
   }
   oldNetConfig = cpNetConfig;

   if (cpNetConfig == CFG_LOCAL) {
      if (netConfig == CFG_EXTERNAL) {
         // Managed application has returned to self-installation
```

```
            IsiReturnToFactoryDefaults(); // Call NEVER returns!
         }
        // Self-installed network--start the ISI engine
        scaled_delay(31745UL); // 800ms delay
        IsiStartS(isiFlagExtended+isiFlagHeartbeat);
    }
}
```

IsiProcessMsg()

boolean **IsiProcessMsg(**IsiType *Type***)**;

boolean **IsiProcessMsgDa(**void**)**;

boolean **IsiProcessMsgDas(**void**)**;

boolean **IsiProcessMsgS(**void**)**;

Processes an ISI message that has been verified as an ISI message by the **IsiApproveMsg()** function. The function returns **FALSE** if the message received has been recognized and processed, and returns **TRUE** if the message is unrecognized and not processed.

Typically, the **IsiApproveMsg()** and **IsiProcessMsg()** functions are used in a single Boolean expression, as shown:

EXAMPLE

```
when (msg_arrives) {
   if (IsiApproveMsg() && IsiProcessMsgS()) {
      // TODO: Process unrecognized ISI messages here
   }
}
```

You can use the **IsiProcessMsg()** function to process an ISI message on a device that supports multiple ISI types. You can use specialized versions of the **IsiProcessMsg()** function to minimize the memory footprint of your application. Devices that only support a single ISI type may use one of the following functions:

- **IsiProcessMsgS()**—processes an ISI message for a device that does not support domain acquisition.
- **IsiProcessMsgDA()**—processes an ISI message for a device that supports domain acquisition, but is not a domain address server.
- **IsiProcessMsgDAS()**—processes an ISI message for an ISI-DAS application that supports domain acquisition and is a domain address server.

When selecting one of the specialized versions **IsiProcessMsgS()**, **IsiProcessMsgDA()**, or **IsiProcessMsgDAS()**, you must make sure to use the same type of specialized start function (**IsiStartS()**, etc) and tick function (**IsiTickS()**, etc).

No forwarders are provided for these functions. All functions operate in any state of the ISI engine, but must not be called unless **IsiApproveMsg()** returned **TRUE** within the same critical section (**when** task).

IsiProcessResponse()

boolean **IsiProcessResponse**(void);

Processes a response to a request message sent in the domain acquisition process. Only needs to be called on a DAS device.

DAS devices must call this function, or the fetch requests will fail.

EXAMPLE

```
when (resp_arrives) {
    if (IsiProcessResponse()) {
        // TODO: process non-ISI responses
    }
}
```

IsiProtocolVersion()

unsigned **IsiProtocolVersion**(void);

Returns the version of the ISI protocol supported by the ISI engine, which is always one for the Release 3 ISI Library. The number indicates the maximum protocol version supported. The ISI engine also supports protocol versions less than the number returned unless explicitly indicated.

No forwarder is provided for the **IsiProtocolVersion()** function. The function operates in either state of the ISI engine.

IsiReturnToFactoryDefaults()

void **IsiReturnToFactoryDefaults**(void);

Restores the device's self-installation data to factory defaults, causing the immediate and unrecoverable loss of all connection information. This function has the same functionality regardless of whether the ISI engine is running or not. This function calls the **IsiUpdateUserInterface()** callback with the **isiRun** event, stops the ISI engine, and then resets the device to complete the process.

Due to the reset, this function never returns to the caller. Any changes related to returning to factory defaults, such as resetting of device-specific configuration properties to their initial values, must occur prior to calling this function.

No forwarder is provided for this function.

EXAMPLE

The following example returns to factory defaults if the device is returned to self-installation mode after being installed in a managed network:

```
network input SCPTnwrkCnfg cp cp_info(reset_required) nciNetConfig
= CFG_EXTERNAL;
eeprom SCPTnwrkCnfg OldNetConfig = CFG_NUL;

when (reset) {
    SCPTnwrkCnfg cpNwrkConfig;
```

```
        cpNwrkConfig = OldNetConfig;

        if (cpNwrkConfig == CFG_NUL) {
            // For the first application start, set nciNetConfig to
            // CFG_LOCAL, allowing the ISI engine to run by default:
            nciNetConfig = CFG_LOCAL;
        }
        OldNetConfig = nciNetConfig;

        if (nciNetConfig == CFG_LOCAL) {
            if (cpNwrkConfig == CFG_EXTERNAL) {
                // The application has just returned into the self-
                // installed environment.  Make sure to re-initialize
                // the entire ISI engine:
                IsiReturnToFactoryDefaults(); // Call NEVER returns!
             }

    //  We are in a self-installed network:
            //  Start the ISI engine:
            scaled_delay(31745UL); // 800ms delay
            IsiStartS(isiFlagExtended);
        }
    }
```

IsiStart()

void **IsiStart**(IsiType *Type*, IsiFlags *Flags*);

void **IsiStartDAS(**IsiFlags *Flags*);

void **IsiStartS(**IsiFlags *Flags*);

void **IsiStartDA(**IsiFlags *Flags*);

Starts the ISI engine. The ISI engine sends and receives ISI messages, and manages the network configuration of your device. You will typically start the ISI engine in your reset task when self-installation is enabled, and you will typically stop the ISI engine when self-installation is disabled.

The function accepts a combination of flags, defined in the **IsiFlags** enumeration. Certain features of the ISI engine will only become available if the corresponding flag had been raised with the **IsiStart()** function.

You can use the **IsiStart()** function to start the ISI engine using any ISI type. You can use specialized versions of the **IsiStart()** function to minimize the memory footprint of your application. Devices that only support a single ISI type may use one of the following functions:

- **IsiStartS()**—starts the ISI engine for a device that does not support domain acquisition.
- **IsiStartDA()**—starts the ISI engine for a device that supports domain acquisition, but is not a domain address server.
- **IsiStartDAS()**—starts the ISI engine for an ISI-DAS application that supports domain acquisition and is a domain address server.

To maximize compatibility with network management tools used for managed networks, insert an 800 millisecond to one-and-a-half second delay before calling any of the **IsiStart()** functions.

When selecting one of the specialized versions **IsiStartS()**, **IsiStartDA()**, or **IsiStartDAS()**, you must make sure to use the same type of specialized message processor function (**IsiProcessMsgS()**, etc) and tick function (**IsiTickS()**, etc).

No forwarders are provided with the **IsiStart()** functions.

EXAMPLE

The following example starts the ISI engine after confirming the device is in self-installation mode:

```
network input SCPTnwrkCnfg cp cp_info(reset_required) cpNetConfig
= CFG_EXTERNAL;
eeprom SCPTnwrkCnfg oldNetConfig = CFG_NUL;

when (reset) {
    SCPTnwrkCnfg netConfig;
    netConfig = oldNetConfig;

    if (netConfig == CFG_NUL) {
        // First application start, enable self-installation
        cpNetConfig = CFG_LOCAL;
    }
    oldNetConfig = cpNetConfig;

    if (cpNetConfig == CFG_LOCAL) {
        if (netConfig == CFG_EXTERNAL) {
            // Managed application has returned to self-installation
            IsiReturnToFactoryDefaults(); // Call NEVER returns!
         }
        //  Self-installed network--start the ISI engine:
        scaled_delay(31745UL); // 800ms delay
        IsiStartS(isiFlagExtended+isiFlagHeartbeat);
    }
}
```

IsiStartDeviceAcquisition()

void **IsiStartDeviceAcquisition**(void);

Starts or retriggers device acquisition mode on a domain address server. The domain address server will respond to domain ID requests from ISI-DA devices as long as it is in device acquisition mode. When the domain address server is in device acquisition mode and has responded to a domain request (DIDRQ) with a domain response (DIDRM), calling the **IsiStartDeviceAcquisition()** function also confirms that the correct device has been allocated, and triggers the release of a domain confirmation message (DIDCF).

This function has no effect on a device that is not a domain address server, or if the ISI engine is stopped. No forwarder is provided for this function.

EXAMPLE

The following example starts domain acquisition mode on a domain address server when the user presses a Connect button on the server:

```
when (connect_button_pressed) {
    IsiStartDeviceAcquisition();
}
```

IsiStop()

void **IsiStop**(void);

Stops the ISI engine. Use one of the **IsiStart()** functions to re-start the ISI engine.

The **IsiStop()** function has no forwarder. Calling **IsiStop()** when the ISI engine is stopped has no action.

IsiTick()

void **IsiTick**(IsiType *Type*);

void **IsiTickS**(void);

void **IsiTickDa**(void);

void **IsiTickDas**(void);

Performs periodic processing for the ISI engine. You must periodically call one of the **IsiTick()** functions after you have started the ISI engine with one of the **IsiStart()** functions. You should call this function approximately every 250ms. You can use the **IsiTick()** function for an application that supports any ISI type. You can use specialized versions of the **IsiTick()** function to minimize the memory footprint of your application. Devices that only support a single ISI type may use one of the following functions:

- **IsiTickS()**—performs periodic processing for the ISI engine for a device that does not support domain acquisition.
- **IsiTickDA()**—performs periodic processing for the ISI engine for a device that supports domain acquisition, but is not a domain address server.
- **IsiTickDAS()**—performs periodic processing for the ISI engine for an ISI-DAS application that supports domain acquisition and is a domain address server.

When selecting one of the specialized versions (**IsiTickS()**, **IsiTickDa()**, or **IsiTickDas()**) you must make sure to use the same type of specialized message processor function (**IsiProcessMsgS()**, etc) and start function (**IsiStartS()**, etc).

No forwarders are provided with the **IsiTick()** functions. Calling **IsiTick()** when the ISI engine is stopped causes no effect.

EXAMPLE

The following example for a device that does not support domain acquisition declares a timer and calls **IsiTickS()** to do periodic processing:

```
mtimer repeating isiTimer = 1000ul / ISI_TICKS_PER_SECOND;

when (timer_expires(isiTimer)) {
   // Call the ISI engine to perform periodic tasks
   IsiTickS();
}
```

Appendix C

Callback Functions

This appendix describes the callback functions that your application may provide for the ISI library. The ISI library includes default implementations of all ISI API callback functions. As a result, you only have to provide callback functions where you need to customize the default behavior.

IsiCreateCsmo()

void **IsiCreateCsmo**(unsigned *Assembly*, IsiCsmoData* *pCsmoData*);

Constructs the **IsiCsmoData** portion of a CSMO Message. The *pCsmoData* parameter is a pointer to an **IsiCsmoData** structure that is filled by this function call. This function is called by the ISI engine prior to sending a CSMO message. This function has the same effect if the ISI engine is running or not.

IsiCreateCsmo() is a forwarder to **isiCreateCsmo()**.

The **isiCreateCsmo()** forwardee sets the fields of the **IsiCsmoData** structure as follows: it uses the **IsiGetPrimaryGroup()** function to obtain the group ID, and sets all fields to zero except the **Application** field (which is filled with data from the device's program ID), the **Direction** field (which is set to **isiDirectionAny**, which corresponds to the value 2), and the **NvType** field, which is set to the primary network variable's SNVT ID, or zero for a UNVT.

Most applications will override this function to supply the application-specific data for open enrollment messages.

IsiCreatePeriodicMsg()

boolean **IsiCreatePeriodicMsg**(void);

Specifies whether the application has any messages for the ISI engine to send using the periodic broadcast scheduler. Since the ISI engine sends periodic outgoing messages at regular intervals, this function allows an application to send a message at one of the periodic message slots. If the application has no message to send, then this function should return **FALSE**. If it does have a message to send, then this function should return **TRUE**.

To use this function, you must enable application-specific periodic messages using the **isiFlagApplicationPeriodic** flag when you call the **IsiStart()** function.

The default implementation of **IsiCreatePeriodicMsg()** does nothing but return **FALSE**. You can override this function by providing an application-specific implementation of **IsiCreatePeriodicMsg()**.

Do not send any messages, start other network transactions, or call other ISI API functions while the **IsiCreatePeriodicMsg()** callback executes. To call other ISI API functions or start other network transactions, set an application-specific flag in the **IsiCreatePeriodicMsg()** callback function and check the flag in a separate **when** task. This separate **when** task can send the periodic message soon after the **IsiCreatePeriodicMsg()** function is completed.

EXAMPLE

The following example sends a periodic message:

```
boolean SendApplicationPeriodic = FALSE;

boolean IsiCreatePeriodicMsg(...) {
       if (have something to do) {
               SendApplicationPeriodic = TRUE;
       }
       return SendApplicationPeriodic;
}

when (SendApplicationPeriodic) {
       SendApplicationPeriodic = FALSE;
       // Send periodic message, for example, with IsiMsgSend()
       // For network variable heartbeats, use propagate()
}
```

IsiGetAssembly()

unsigned **IsiGetAssembly**(const IsiCsmoData* *pCsmoData*, boolean *Auto*);

Returns the number of the first assembly that may join the enrollment characterized with *pCsmoData*. The function returns **ISI_NO_ASSEMBLY** (0xFF) if no such assembly exists, or an application-defined assembly number 0 – 254. The *Auto* parameter specifies a manually initiated enrollment (*Auto* = **FALSE**) or an automatically initiated enrollment (*Auto* = **TRUE**). The *Auto* flag is true both for initial automatic enrollment messages (CSMA) or reminders that relate to a possibly new connection (CSMR). Automatic enrollment reminder messages that relate to existing connections on the local device are not passed to the application.

The pointer provided with the *pCsmoData* parameter is only valid for the duration of this function call.

IsiGetAssembly() is a forwarder to **isiGetAssembly()**. The **isiGetAssembly()** forwardee returns the assembly number if a compatible network variable exists for a simple connection, using standard network variable types. The default implementation is compatible with the default implementation of **IsiCreateCsmo()**, and is sufficient for simple devices. The **isiGetAssembly()** forwardee always refuses automatic enrollment.

The **isiGetAssembly()** forwardee assumes the default assembly numbering scheme that is described in *Assembly Number Allocation*, earlier.

Applications overriding **IsiGetAssembly()** should also override **IsiGetNextAssembly()**.

The function operates whether the ISI engine is running or not.

IsiGetConnection()

const IsiConnection* **IsiGetConnection**(unsigned *Index*);

Returns a pointer to an entry in the connection table. The default implementation returns a pointer to a built-in connection table with 8 entries, stored in on-chip EEPROM memory. You can override this function to provide an application-specific means of accessing the connection table, or to provide an application table of a different size.

The ISI engine requests only one connection table entry at a time, and makes no assumption on the pointer value. Applications using storage outside the Neuron address space may use a single buffer to transfer all connection table entries, as shown in *Customizing the ISI Connection Table* earlier in this document.

This function is frequently called and should return as soon as possible.

There is no forwarder for this function.

You must override the **IsiSetConnection()** and **IsiGetConnectionTableSize()** functions if you override the **IsiGetConnection()** function.

EXAMPLE

The following example creates a connection table with 16 entries stored in on-chip EEPROM:

```
#define CTABSIZE 16u
static eeprom fastaccess IsiConnection
MyConnectionTable[CTABSIZE];
unsigned IsiGetConnectionTableSize(void) {
   return CTABSIZE;
}
const IsiConnection* IsiGetConnection(unsigned Index) {
   return MyConnectionTable + Index;
}
void IsiSetConnection(IsiConnection* pConnection, unsigned Index){
   MyConnectionTable[Index] = *pConnection;
}
```

IsiGetConnectionTableSize()

unsigned **IsiGetConnectionTableSize**(void);

Returns the number of entries in the connection table. The default implementation returns the number of entries in the built-in connection table (8). You can override this function to support an application-specific implementation of the ISI connection table. You can use this function to provide a larger connection table or to store the connection table outside of the Neuron address space.

The ISI library supports connection tables with 0 to 254 entries. The connection table size is considered constant following a call to **IsiStart()**; you must first stop, then re-start, the ISI engine if the connection table size changes dynamically.

There is no forwarder for this function.

You must override the **IsiSetConnection()** and **IsiGetConnection()** functions if you override the **IsiGetConnectionTableSize()** function.

EXAMPLE

The following example creates a connection table with 16 entries stored in on-chip EEPROM:

```
#define CTABSIZE 16u
static eeprom fastaccess IsiConnection
MyConnectionTable[CTABSIZE];
unsigned IsiGetConnectionTableSize(void) {
   return CTABSIZE;
}
const IsiConnection* IsiGetConnection(unsigned Index) {
   return MyConnectionTable + Index;
}
void IsiSetConnection(IsiConnection* pConnection, unsigned Index){
   MyConnectionTable[Index] = *pConnection;
}
```

IsiGetNextAssembly()

unsigned **IsiGetNextAssembly(**const IsiCsmoData* *pCsmoData*, boolean *Auto*, unsigned *Assembly***)**;

Returns the next applicable assembly following the one indicated with the *Assembly* argument for an incoming CSMO following the specified assembly. The function returns **ISI_NO_ASSEMBLY** (0xFF) if there are no such assemblies, or an application-specific assembly number 1 – 254. This function is called after calling the **IsiGetAssembly()** function, unless **IsiGetAssembly()** returned **ISI_NO_ASSEMBLY**. The *Auto* parameter specifies a manually initiated enrollment (*Auto* = **FALSE**) or an automatically initiated enrollment (*Auto* = **TRUE**). The *Auto* flag is true both for initial automatic enrollment messages (CSMA) or reminders that relate to a possibly new connection (CSMR). Automatic enrollment reminder messages that relate to existing connections on the local device are not passed to the application. The pointer provided with the *pCsmoData* parameter is only valid for the duration of this function call.

IsiGetNextAssembly() is a forwarder to **isiGetNextAssembly()**.

The **isiGetNextAssembly()** forwardee returns the next assembly number if a complementary network variable exists for a simple connection, using standard network variable types. The **isiGetNextAssembly()** forwardee always refuses automatic enrollment. The default implementation is compatible with the default implementation of **IsiCreateCsmo()**, and is sufficient for simple devices.

The **isiGetNextAssembly()** forwardee assumes the default assembly numbering scheme that is described in *Assembly Number Allocation* earlier.

Applications overriding **IsiGetNextAssembly()** should also override **IsiGetAssembly()**.

The function operates whether the ISI engine is running or not.

IsiGetNextNvIndex()

unsigned **IsiGetNextNvIndex(**unsigned *Assembly*, unsigned *Offset*, unsigned *Previous***)**;

Returns the network variable index of the network variable at the specified offset within the specified assembly, following the network variable specified by the *Previous* index. Returns **ISI_NO_INDEX** if there are no more network variables, or a valid network variable index 1– 254 otherwise. The *Offset* parameter is zero-based and relates to the selector number *Offset* within the assembly that is used with the enrollment of this assembly.

This function is a forwarder to **isiGetNextNvIndex()**.

The **isiGetNextNvIndex()** forwardee always returns **ISI_NO_INDEX**.

Applications may override **IsiGetNextNvIndex()** to map multiple local network variables to a single network variable selector. The ISI application is responsible for observing the various binding and protocol limitations when using this feature.

The **isiGetNextNvIndex()** forwardee assumes the default assembly numbering scheme that is described in *Assembly Number Allocation* earlier.

Applications overriding **IsiGetNextNvIndex()** should also override **IsiGetNvIndex()**.

This function operates whether the ISI engine is running or not.

IsiGetNvIndex()

unsigned **IsiGetNvIndex(**unsigned *Assembly*, unsigned *Offset***)**;

Returns the network variable index 0 – 254 of the network variable at the specified offset within the specified assembly, or **ISI_NO_INDEX** if no such network variable exists. This function must return at least one valid network variable index for each assembly number returned by **IsiGetAssembly()** and **IsiGetNextAssembly()**. The *Offset* parameter is zero-based and relates to the selector number *Offset* that is used with the enrollment of this assembly.

This function is a forwarder to **isiGetNvIndex()**.

The **isiGetNvIndex()** forwardee returns *Assembly* + *Offset*.

The **isiGetNvIndex()** forwardee assumes the default assembly numbering scheme that is described in *Assembly Number Allocation* earlier.

Applications overriding **IsiGetNvIndex()** should also override **IsiGetNextNvIndex()**.

This function operates whether the ISI engine is running or not.

IsiGetPrimaryDid()

const unsigned* **IsiGetPrimaryDid**(unsigned* *pLength*);

Returns a pointer to the default primary domain ID for the device. The function also provides the domain ID length in the location provided by the *pLength* parameter. Domain IDs may be 1, 3, or 6 bytes long—the 0-length domain ID cannot be used for the primary domain. Only the number of bytes provided through the *pLength* output parameter must be valid in the returned pointer. You can override this function to override the ISI standard domain ID value. This function is only used to create a non-ISI system.

Both length and value of the domain ID provided are considered constant once the ISI engine is running. To change the primary domain ID at runtime using the **IsiGetPrimaryDid()** callback, stop and re-start the ISI engine.

No forwarder is provided with this function.

Warning: Non ISI devices will not interoperate with ISI devices.

IsiGetPrimaryGroup()

unsigned **IsiGetPrimaryGroup**(unsigned *Assembly*);

Returns the group ID for the specified assembly. The default implementation returns **ISI_DEFAULT_GROUP** (128). This function is only required if the default implementation, or the forwardee, of **IsiCreateCsmo()** is in use.

No forwarder is provided with **IsiGetPrimaryGroup()**.

The function operates whether the ISI engine is running or not.

IsiGetRepeatCount()

unsigned **IsiGetRepeatCount**(void);

Specifies the repeat count used with all network variable connections, where all connections share the same repeat counter. The repeat counter value is considered constant for the lifetime of the application, and will only be queried when the device powers up the first time after a new application image has been loaded and every time **IsiReturnToFactoryDefaults()** runs. Only repeat counts of 1, 2 or 3 are supported. To take full advantage of the secondary frequency on a PL transceiver, only use a repeat count of 1 or 3. This function has no affect on ISI messages.

The default implementation of this function always returns 3.

This function operates whether the ISI engine is running or not.

IsiGetWidth()

unsigned **IsiGetWidth(**unsigned *Assembly***)**;

Returns the width in the specified assembly. The width is equal to the number of network variable selectors associated with the assembly.

This function is a forwarder to **isiGetWidth()**.

The **isiGetWidth()** forwardee always returns one.

Applications must override the **IsiGetWidth()** function to support compound assemblies.

This function operates whether the ISI engine is running or not.

IsiQueryHeartbeat()

boolean **IsiQueryHeartbeat(**unsigned *NvIndex***)**;

Returns **TRUE** if a heartbeat for the network variable with the global index **NvIndex** has been sent, and returns **FALSE** otherwise. When network variable heartbeat processing is enabled and the ISI engine is running, the engine queries bound output network variables using this callback (including any alias connections) whenever the heartbeat is due. This function does not send the heartbeat update—see **IsiIssueHeartbeat()**. For more details on network variable heartbeat scheduling, see the *ISI Protocol Specification*.

The **isiQueryHeartbeat()** forwardee always returns **FALSE**.

IsiSetConnection()

void **IsiSetConnection**(IsiConnection* *pConnection*, unsigned *Index***)**;

Updates an entry in the connection table, which must be kept in persistent, non-volatile, storage.

The default implementation updates an entry in the built-in connection table with 8 entries, stored in on-chip EEPROM memory. You can override this function to provide an application-specific means of accessing the connection table, or to provide an application table of a different size.

The ISI engine requests only one connection table entry at a time, and makes no assumption on the pointer value. Applications using storage outside the Neuron address space may use a single buffer to transfer all connection table entries, as shown by example in *Customizing the ISI Connection Table* earlier in this document. The ISI engine may implement locals of the **IsiConnection** type for use with **IsiSetConnection()** callbacks; the application implementing this override must not maintain a copy of the pointer value.

This function is frequently called and should return as soon as possible.

There is no forwarder for this function.

You must override the **IsiGetConnection()** and **IsiGetConnectionTableSize()** functions if you override the **IsiSetConnection()** function.

IsiSetDomain()

void **IsiSetDomain**(domain_struct* *pDomain*, unsigned *Index*);

Sets the domain, subnet, and node ID in the primary entry of the domain table for a device. You can override this function with an empty function during development in a managed environment to prevent conflicts with the network management tool. For example, if you are using the NodeBuilder Development Tool, the LonMaker tool manages the devices you are developing. To prevent conflicts with the LonMaker tool, the following code disables domain table updates for a NodeBuilder development target:

```
#ifndef _MINIKIT
#   ifdef _DEBUG
    void IsiSetDomain(domain_struct* pDomain,
                         unsigned Index) {
        ;  // do nothing.
    }
#   endif
#endif
```

When you override this function, only the domain, subnet, and node ID management functions of the ISI engine are disabled—all other portions of the ISI protocol will continue to function. This passes control of the subnet and node IDs to an external network management tool, which means they may no longer follow the ISI subnet/node value ranges. When disabled, any detected subnet/node ID conflicts will not be resolved.

Warning: This function can only be overridden during development. Overriding the **IsiSetDomain()** callback in the way shown above allows for debugging of the device application with the ISI engine running, but disables important features of the ISI protocol. Devices using this approach will not function correctly in a self-installed environment.

IsiUpdateDiagnostics()

void **IsiUpdateDiagnostics**(IsiDiagnostic *Event*, unsigned *Parameter*);

Provides optional detailed ISI diagnostic events. These events are useful for debugging ISI applications and are not typically used for production products. To receive notification of diagnostic events, enable diagnostics in the **IsiStart()** function and override the **IsiUpdateDiagnostics()** callback function. This callback is normally disabled and the default implementation of **IsiUpdateDiagnostics()** does nothing. The ISI engine calls this function with the *Event* parameter set to one of the values defined for the **IsiDiagnostic** enumeration in Appendix A. Some of these events carry a meaningful value in the *Parameter* argument, as detailed in the **IsiDiagnostic** definition. Most diagnostics events supplement UI events; use a combination of both events for a complete trace record.

No forwarder is supported for this function.

IsiUpdateUserInterface()

void **IsiUpdateUserInterface**(IsiEvent *Event*, unsigned *Parameter*);

Provides status feedback from the ISI engine. These events are useful for synchronizing the device's user interface with the ISI engine. To receive notification of ISI status events, override the **IsiUpdateUserInterface()** callback function. The default implementation of **IsiUpdateUserInterface()** does nothing. The ISI engine calls this function with the *Event* parameter set to one of the values defined for the **IsiEvent** enumeration in Appendix A. Some of these events carry a meaningful value in the *Parameter* argument, as detailed in the **IsiEvent** definition.

You can use the *Event* parameter passed to the **IsiUpdateUserInterface()** callback function to track the state of the ISI engine. This is a simple way to determine which ISI function calls make sense at any time, and which ones don't.

The following table lists this state information, based on the last *Event* value provided with this callback, over possible ISI actions:

	isiPending	isiPending Host/ isiApproved	isiApproved Host	isiNormal
IsiCancelEnrollment		✓	✓	
IsiCreateEnrollment	✓		✓	
IsiExtendEnrollment	✓		✓	
IsiDeleteEnrollment				✓
IsiLeaveEnrollment				✓
IsiOpenEnrollment				✓
IsiAcquireDomain				✓
IsiStartDeviceAcquisition				✓
IsiReturnToFactoryDefaults	✓	✓	✓	✓

Appendix D

ISI Router Configuration

This appendix provides information for preparing a LONWORKS router for use with an ISI network.

LONWORKS Routers and ISI Networks

To prepare a LONWORKS router for use with an ISI network, configure the device as follows:

	TP/FT-10 Side	PL-20 Side
Domain[0]	0x49-0x53-0x49 (“ISI”)	0x49-0x53-0x49 (“ISI”)
Domain[1]	n/a	n/a
Channel ID	0x04	0x10
Router Mode	Repeater, Online	Repeater, Online

You can find more information about LONWORKS router modules and their configuration from the LONWORKS Router User’s Guide, available for download from http://www.echelon.com/support/documentation/manuals/routers/.

Appendix E

Glossary

This appendix defines the key ISI terms.

Term	Definition
Application message	An ANSI/CEA-709.1 (EN14908-1) message with a message code in the range of 0 to 62. Message codes 48 to 62 are reserved for standard application messages, including messages used for ISI (message code 61) and file transfer (message code 62). Message codes 0 to may be used for proprietary manufacturer-specific messages.
Assembly	See *connection assembly*.
Automatic enrollment	The process of creating an ISI connection without user interaction.
Callback function	A function called by the ISI engine that may be overridden by an application-specific function in order to relay data back and forth between the application and the ISI engine.
Compatible network variable	A network variable of the same type and direction.
Complementary network variable	A network variable of the same type but opposite direction.
Compound assembly	A connection assembly that has more then one network variable.
Connection	A configured data flow or flows between one or more output network variables and one or more input network variables. In ISI, a connection may be a network variable connection or an ISI connection. An ISI connection consists of one or more network variable connections.
Connection assembly	A block of functionality for a device represented by one or more network variables, and one or more functional blocks. Connection assemblies are used to create ISI connections.
Connection controller	A device that initiates enrollment of one or more ISI devices in an ISI connection. Typically a user interface panel or controller.
Connection host	A device that begins the enrollment process by sending a connection invitation specifying a connection assembly.
Connection invitation	A message sent by a connection host to open enrollment for a connection assembly.
Connection member	A device that has joined an ISI connection, but is not the connection host.
Connection message	Messages used to create, delete, and describe ISI connections.
Connection Status Message (CSM)	Periodic messages used to create, delete, and describe ISI connections.
Controlled enrollment	The process of creating an ISI connection requiring a connection

Term	Definition
	controller to specify the devices to be connected.
Domain Address Server (DAS)	A device present in ISI-DA networks that does ISI management tasks such as assigning a domain ID and monitoring and reporting network size. An ISI-DA network may include multiple domain address servers.
Domain Resource Usage Message (DRUM)	Periodic message containing the physical address (Neuron ID), logical address (domain, subnet, and node ID), and part of the program ID for a device.
Enrollment	The process of creating a self-installed ISI connection.
Explicit message	A message that is explicitly constructed by the application. The message may be an application message, network management message, network diagnostic message network variable message, or a foreign-frame message. See also *implicit message*.
Forwarder	A function that allows the default implementation of a function to be called while still allowing for an application-specific implementation.
Implicit message	A message that is constructed by the 709.1 protocol engine within an application. Only used for network variable messages.
ISI connection	See *connection*.
ISI engine	Part of the ISI library that implements most of the ISI protocol and relays information back to the application through use of callback functions.
ISI-S network	A network that does not include a domain address server, or contains any devices that are not ISI-DA compatible. ISI-S networks are limited to 32 devices.
ISI-DA network	A network that includes a domain address server and consists solely of ISI-DA compatible devices. ISI-DA networks are limited to 200 devices.
ISI-DAS protocol	The protocol implemented by a domain address server.
Managed network	A network that is managed by a network management tool, and is therefore not self-installed.
Manual enrollment	The process of creating an ISI connection requiring user interaction with the devices to be connected.
Message	Information sent from a source device to one or more destination devices. In the 709.1 protocol, a message may be an application message, network management message, network diagnostic message, a network variable message, or a foreign-frame message.

Term	Definition
Network management	The act of putting network configuration data into a persistent store (typically a database) with the intent of making the network configuration data in a network of devices consistent with that network configuration data in the persistent store, and maintaining that consistency over time. Just storing the data into a persistent store, for example, is not network management; it is just a backup or snapshot of the data at any one point.
Network management tool	A tool that stores network configuration data into its persistent store and actively maintains consistency between the data in its persistent store and the devices on the network being managed.
Network variable	A data item that a particular device application program expects to get from other devices on a network (an *input network variable*) or expects to make available to other devices on a network (an *output network variable*). Examples are a temperature, switch value, and actuator position setting.
Network variable connection	See *connection*.
Periodic message	A status message that is sent at a specified interval.
Self-installed network	A network that has network addresses and connections created without the use of a network management tool.
Simple assembly	A connection assembly that has one network variable.
Timing Guidance Message (TIMG)	A periodic message that is broadcast by all domain address servers. It includes information about network size and latency. It is an optional message, but if available, ISI devices use this information to schedule all periodic message based on network size. This ensures efficient use of the channel bandwidth and minimizes the overhead of the ISI protocol
Variant	A field in a connection message that allows for different interpretations of the assembly members.

ECHELON®

www.echelon.com