```
   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/.