SELEthernetController

IEC 61131 Library for ACSELERATOR RTAC® Projects

SEL Automation Controllers
Table of Contents

Section 1: SELEthernetController

   Introduction ................................................................. 3
   Supported Firmware Versions ....................................... 4
   Enumerations ............................................................... 4
   Aliases ........................................................................ 4
   Functions ..................................................................... 4
   Classes ....................................................................... 6
   Benchmarks ................................................................... 22
   Examples ...................................................................... 25
   Release Notes ............................................................... 35
**RTAC LIBRARY**

**SELEthernetController**

**Introduction**

The SELEthernetController library provides the ability to stream data through TCP and UDP ports. It provides client and server functionality for both TCP and UDP connections. Various methods are used to initiate new read or write operations.

**Special Considerations**

- Copying classes from this library causes unwanted behavior. This means the following:
  1. The assignment operator “:=” must not be used on any class from this library; consider assigning pointers to the objects instead.

```
// This is bad and in most cases will provide a compiler error such as:
// "C0328: Assignment not allowed for type class_SocketObject"
mySocketObject := otherSocketObject;
```

```
// This is fine
someVariable := mySocketObject.value;
// As is this
pt_mySocketObject := ADR(mySocketObject);
```

- Classes from this library must never be VAR_INPUT or VAR_OUTPUT members in function blocks, functions, or methods. Place them in the VAR_IN_OUT section or use pointers instead.

- Classes in this library have memory allocated inside them. As such, they should only be created in environments of permanent scope (e.g., Programs, Global Variable Lists, or VAR_STAT sections).

- Opening and closing large numbers of Ethernet connections can take significant time. Take great care when using this library on a system with hard real-time requirements.

**NOTE:** See the acSELEATOR RTAC Library Extensions Instruction Manual (LibraryExtensionsIM) for more information about the concepts used by the object-oriented extensions to the IEC 61131-3 standard.
This library does not have the ability to open ports in the RTAC firewall to allow inbound communication. This means that for the library to receive UDP packets or act as a TCP server, those ports must be opened as Ethernet Incoming Access Points in the AcRTAC project.

Supported Firmware Versions

You can use this library on any device configured using acSELERATOR RTAC® SEL-5033 Software with firmware version R143 or higher.

Versions 3.5.0.6 and older can be used on RTAC firmware version R133 and higher.

Enumerations

Enumerations make code more readable by allowing a specific number to have a readable textual equivalent.

enum_SocketState

<table>
<thead>
<tr>
<th>Enumeration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LISTENING</td>
<td>The server socket is ready to receive communication requests.</td>
</tr>
<tr>
<td>OPEN</td>
<td>The socket is ready to send and receive UDP data.</td>
</tr>
<tr>
<td>CONNECTED</td>
<td>The socket has an active client server TCP session.</td>
</tr>
<tr>
<td>CLOSED</td>
<td>The socket will no longer allow communication.</td>
</tr>
<tr>
<td>ERROR</td>
<td>The socket has encountered an error and needs reconfiguration.</td>
</tr>
</tbody>
</table>

Aliases

This section lists aliases that this library defines.

SESSION_HANDLE

<table>
<thead>
<tr>
<th>Alias</th>
<th>IEC 61131 Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SESSION_HANDLE</td>
<td>POINTER TO BYTE</td>
</tr>
</tbody>
</table>

Functions

This library provides the following functions.
fun_StringToInaddr

Convert basic strings to the INADDR type this library uses. Strings provided to this function must be in the format xxx.xxx.xxx.xxx, where xxx is a number between 0 and 255.

**Inputs**

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ipAddrString</td>
<td>STRING(15)</td>
<td>The string to convert to an INADDR.</td>
</tr>
</tbody>
</table>

**Outputs**

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ipAddr</td>
<td>INADDR</td>
<td>The converted INADDR.</td>
</tr>
</tbody>
</table>

**Return Value**

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE if the string could be converted.</td>
</tr>
</tbody>
</table>

**Processing**

The fun_StringToInaddr() function does the following:
- Parses the provided string into four bytes.
- Outputs those bytes ordered in an INADDR.
- Returns false and outputs an IP address of 0.0.0.0 if the string could not be converted.

fun_InaddrToString

Converts an INADDR to a string in the format xxx.xxx.xxx.xxx, where xxx is a number between 0 and 255.

**Inputs**

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ipAddr</td>
<td>INADDR</td>
<td>The INADDR to convert to a string.</td>
</tr>
</tbody>
</table>

**Return Value**

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRING(15)</td>
<td>The resulting IP address as a string.</td>
</tr>
</tbody>
</table>
Processing

The `fun_InaddrToString()` function does the following:

- Parses the provided INADDR by its four bytes.
- Returns a string in the format `xxx.xxx.xxx.xxx`, where `xxx` is a number between 0 and 255.

Classes

Classes are a particular implementation of a Function Block (FB). They provide methods and properties, which a normal FB does not provide.

class_UdpSocket (Class)

This class provides a socket for sending and receiving using through use of the UDP protocol. Once enabled, the class creates, binds, sends, and receives to the configured ports.

Initialization Inputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxPacketSize</td>
<td>DINT</td>
<td>The maximum packet size, in bytes, to obtain through use of this socket. A value of zero or less results in the class imposing no limit on inbound packet size.</td>
</tr>
</tbody>
</table>

Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>enum_SocketState</td>
<td>R</td>
<td>The state of this socket.</td>
</tr>
<tr>
<td>LocalPort</td>
<td>UINT</td>
<td>R</td>
<td>The port number with which this socket interfaces locally.</td>
</tr>
<tr>
<td>LocalIPAddr</td>
<td>INADDR</td>
<td>R</td>
<td>The IP address with which this socket interfaces locally.</td>
</tr>
<tr>
<td>pt_Error</td>
<td>POINTER TO STRING</td>
<td>R</td>
<td>An error message describing any present error condition.</td>
</tr>
</tbody>
</table>

Properties are internal values made visible through Get and Set accessors. Access is defined as R (read), W (write), or R/W (read/write).

bootstrap_SetLocalIP (Method)

Perform a one-time setting of the local IP address and port this socket will use.
Inputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>localPort</td>
<td>UINT</td>
<td>The port number through which this socket receives and sends data through.</td>
</tr>
<tr>
<td>localIPAddr</td>
<td>INADDR</td>
<td>The IP address on the local box this socket uses. Use 0.0.0.0 to allow all local IP addresses.</td>
</tr>
</tbody>
</table>

Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE if the IP address and port number are set.</td>
</tr>
</tbody>
</table>

Processing

The bootstrap_SetLocalIP() method does the following:

- Immediately returns FALSE if the port and IP address are already set.
- Sets the IP address and port on the local machine for use by this socket.

Open (Method)

Configure the class_UdpSocket to allow communication.

Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE if the port is ready to send data.</td>
</tr>
</tbody>
</table>

Processing

The Open() method does the following:

- Opens the port for sending and receiving communication.
- Returns FALSE if the socket was unable to bind.

Close (Method)

Unconfigure the class_UdpSocket to disable communication.

Processing

The Close() method does the following:

- Discards any unread messages.
- Makes the socket unable to send or receive data.
- Forces reopening of the socket before processing additional communication.
**SendData (Method)**

Send a block of data to the socket.

**Inputs**

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pt_data</td>
<td>POINTER TO BYTE</td>
<td>The data to send.</td>
</tr>
<tr>
<td>numBytes</td>
<td>DINT</td>
<td>The quantity of data in bytes.</td>
</tr>
<tr>
<td>destIPAddr</td>
<td>INADDR</td>
<td>The IP address to which data are sent.</td>
</tr>
<tr>
<td>destPort</td>
<td>UINT</td>
<td>The destination port to which data are sent.</td>
</tr>
</tbody>
</table>

**Return Value**

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DINT</td>
<td>The number of bytes sent.</td>
</tr>
</tbody>
</table>

**Processing**

The `SendData()` method does the following:

- Sends data if `Open()` has been successfully called.
- Validates access to the pointer provided.
-Limits `numBytes` to a positive number.
- Sends `numBytes` of data, starting at `pt_data`, to the socket.
- Returns the number of bytes successfully sent.

**SendQueuedData (Method)**

Send a block of data to the socket from the front of a queue.

**Inputs**

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>destIPAddr</td>
<td>INADDR</td>
<td>The IP address to which data are sent.</td>
</tr>
<tr>
<td>destPort</td>
<td>UINT</td>
<td>The destination port to which data are sent.</td>
</tr>
</tbody>
</table>

**Inputs/Outputs**

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue</td>
<td>class_ByteDeque</td>
<td>The queued data to send.</td>
</tr>
</tbody>
</table>
### Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DINT</td>
<td>The number of bytes sent.</td>
</tr>
</tbody>
</table>

### Processing

The `SendQueuedData()` method does the following:

- Sends data if `Open()` has been successfully called.
- Sends all data, starting at the front of `queue`, to the socket.
- Returns the number of bytes successfully sent.
- Removes bytes sent from the front of `queue`.

### ReceiveFrom (Method)

Overwrites `data` with the first packet available to the socket.

### Inputs/Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>class_ByteVector</td>
<td>The container to which the data are written.</td>
</tr>
</tbody>
</table>

### Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fromIpAddr</td>
<td>INADDR</td>
<td>The IP address from which the data came.</td>
</tr>
<tr>
<td>fromPort</td>
<td>UINT</td>
<td>The port from which the data came.</td>
</tr>
</tbody>
</table>

### Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DINT</td>
<td>The number of bytes loaded.</td>
</tr>
</tbody>
</table>

### Processing

The `ReceiveFrom()` method does the following:

- Does nothing if the socket is not open or a new packet is not available.
- Deletes any data found in `data`.
- Places the first packet—as many as `maxPacketSize` bytes—from this socket in `data`.
- Returns the number of bytes loaded into `data`.
ReceiveToQueueFrom (Method)

Overwrites \textit{data} with the first packet available to the socket.

\textbf{Inputs/Outputs}

\begin{center}
\begin{tabular}{|l|l|l|}
\hline
\textbf{Name} & \textbf{IEC 61131 Type} & \textbf{Description} \\
\hline
\textit{data} & \text{class.ByteDeque} & The container to which the data are written. \hline
\end{tabular}
\end{center}

\textbf{Outputs}

\begin{center}
\begin{tabular}{|l|l|l|}
\hline
\textbf{Name} & \textbf{IEC 61131 Type} & \textbf{Description} \\
\hline
\textit{fromIpAddr} & \text{INADDR} & The IP address from which the data came. \hline
\textit{fromPort} & \text{UINT} & The port from which the data came. \hline
\end{tabular}
\end{center}

\textbf{Return Value}

\begin{center}
\begin{tabular}{|l|l|}
\hline
\textbf{IEC 61131 Type} & \textbf{Description} \\
\hline
\text{DINT} & The number of bytes loaded. \hline
\end{tabular}
\end{center}

\textbf{Processing}

The \texttt{ReceiveToQueueFrom()} method does the following:
\begin{itemize}
  \item Does nothing if the socket is not open or a new packet is not available.
  \item Deletes any data found in \textit{data}.
  \item Places the first packet—as many as maxPacketSize bytes—from this socket in \textit{data}.
  \item Returns the number of bytes loaded into \textit{data}.
\end{itemize}

\textbf{class\_TcpClient (Class)}

This class provides a client socket for the TCP protocol. Once enabled, the class creates, binds, and sends to the configured port. Because TCP is session-based communication, the client should close the session upon completion of the communication to conserve server resources.

\textbf{Properties}

\begin{center}
\begin{tabular}{|l|l|l|l|}
\hline
\textbf{Name} & \textbf{IEC 61131 Type} & \textbf{Access} & \textbf{Description} \\
\hline
\textit{State} & \text{enum\_SocketState} & \text{R} & The state of this socket. \hline
\textit{LocalPort} & \text{UINT} & \text{R} & The port number provided by the user for use locally. A value of zero means an OS-chosen port value will be used. \hline
\textit{LocalIPAddr} & \text{INADDR} & \text{R} & The IP address with which this socket interfaces locally. \hline
\end{tabular}
\end{center}
Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DestIPAddr</td>
<td>INADDR</td>
<td>R</td>
<td>The IP address to which any triggered message is sent.</td>
</tr>
<tr>
<td>DestPort</td>
<td>UINT</td>
<td>R</td>
<td>The port number to which any triggered message is sent.</td>
</tr>
<tr>
<td>pt_Error</td>
<td>POINTER TO STRING</td>
<td>R</td>
<td>An error message describing any present error condition.</td>
</tr>
</tbody>
</table>

Properties are internal values made visible through Get and Set accessors. Access is defined as R (read), W (write), or R/W (read/write).

**bootstrap_SetLocalIP (Method)**

Perform a one-time setting of the local IP address and port this socket will use.

**Inputs**

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>localPort</td>
<td>UINT</td>
<td>The port number through which this socket receives and sends data. If this value is zero then the OS will choose a random outgoing port.</td>
</tr>
<tr>
<td>localIPAddr</td>
<td>INADDR</td>
<td>The IP address on the local box this socket uses. Use 0.0.0.0 to allow all local IP addresses.</td>
</tr>
</tbody>
</table>

**Return Value**

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE if the IP address and port number are set.</td>
</tr>
</tbody>
</table>

**Processing**

The `bootstrap_SetLocalIP()` method does the following:

- Immediately returns FALSE if the port and IP address are already set.
- Sets the IP address and port on the local machine for use by this socket.

**SetIP (Method)**

Set the IP address and port to be used on the next `Open()` request.
Inputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>destIPAddr</td>
<td>INADDR</td>
<td>The IP address to which data is sent.</td>
</tr>
<tr>
<td>destPort</td>
<td>UINT</td>
<td>The destination port to which data is sent.</td>
</tr>
</tbody>
</table>

Processing

The `SetIP()` method does the following:

- Sets the IP address and port that will become the new destination the next time `Open()` is called.

Open (Method)

Configure the class TcpClient to allow communication.

Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE if the port is ready to send data.</td>
</tr>
</tbody>
</table>

Processing

The `Open()` method does the following:

- Opens the port for sending and receiving communication, if `SetIP()` has been successfully run.
- Returns false if the socket was unable to connect.

Close (Method)

Unconfigure the class TcpClient, to disable communication.

Inputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>forceClose</td>
<td>BOOL</td>
<td>Close the session without waiting for confirmation.</td>
</tr>
</tbody>
</table>

Processing

The `Close()` method does the following:

- Discards any unread data.
- Allows server to retain session until all data are read, if `forceClose` is FALSE.
- Makes the socket unable to send data or receive replies.
- Forces reopening of the socket before processing additional communication.
SendData (Method)

Send a block of data to the socket.

Inputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pt_data</td>
<td>POINTER TO BYTE</td>
<td>The data to send.</td>
</tr>
<tr>
<td>numBytes</td>
<td>DINT</td>
<td>The quantity of data in bytes.</td>
</tr>
</tbody>
</table>

Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DINT</td>
<td>The number of bytes sent.</td>
</tr>
</tbody>
</table>

Processing

The SendData() method does the following:

- Does nothing if the socket is not open.
- Validates access to the pointer provided.
- Limits numBytes to a positive number.
- Sends all provided data to the socket.
- Returns the number of bytes successfully sent.

SendQueuedData (Method)

Send a block of data to the socket from the front of a queue.

Inputs/Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue</td>
<td>class_BodyDeque</td>
<td>The data to send.</td>
</tr>
</tbody>
</table>

Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DINT</td>
<td>The number of bytes sent.</td>
</tr>
</tbody>
</table>

Processing

The SendQueuedData() method does the following:

- Does nothing if the socket is not open.
- Sends all provided data, starting at the front of queue, to the socket.
Returns the number of bytes successfully sent.

- Removes bytes sent from the front of queue.

**ReceiveData (Method)**

Appends a block of data from the socket to *data*.

**Inputs**

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numBytes</td>
<td>DINT</td>
<td>The number of bytes requested.</td>
</tr>
</tbody>
</table>

**Inputs/Outputs**

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 6131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>class_ByteVector</td>
<td>The container to which the data are written.</td>
</tr>
</tbody>
</table>

**Return Value**

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DINT</td>
<td>The number of bytes loaded.</td>
</tr>
</tbody>
</table>

**Processing**

The `ReceiveData()` method does the following:

- Does nothing if the socket is not open.
- Requests data from the socket until there are either no more or it reaches `numBytes`, whichever happens first.
- Appends all data retrieved to *data*.
- Returns the number of bytes appended.

**ReceiveToQueue (Method)**

Pushes a block of data from the socket to the back of *queue*.

**Inputs**

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numBytes</td>
<td>DINT</td>
<td>The number of bytes requested.</td>
</tr>
</tbody>
</table>
Inputs/Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue</td>
<td>class_ByteDeque</td>
<td>The container to which the data are written.</td>
</tr>
</tbody>
</table>

Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DINT</td>
<td>The number of bytes loaded.</td>
</tr>
</tbody>
</table>

Processing

The `ReceiveToQueue()` method does the following:

- Does nothing if the socket is not open.
- Requests data from the socket until there are either no more or it reaches `numBytes`, whichever happens first.
- Pushes all data retrieved to the back of `queue`.
- Returns the number of bytes pushed.

class_TcpServer (Class)

This class provides a listening socket for the TCP protocol. Once enabled the class creates, binds, and receives on the configured port.

Initialization Inputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxSessions</td>
<td>USINT</td>
<td>The maximum number of concurrent sessions to allow on this socket. This will be forced to at least one.</td>
</tr>
</tbody>
</table>

Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>enum_SocketState</td>
<td>R</td>
<td>The state of this socket.</td>
</tr>
<tr>
<td>LocalPort</td>
<td>UINT</td>
<td>R</td>
<td>The port number with which this socket interfaces locally.</td>
</tr>
<tr>
<td>LocalIPAddr</td>
<td>INADDR</td>
<td>R</td>
<td>The IP address with which this socket interfaces locally.</td>
</tr>
<tr>
<td>NumSessions</td>
<td>USINT</td>
<td>R</td>
<td>The number of client sessions connected to this server block.</td>
</tr>
<tr>
<td>SessionState</td>
<td>enum_SocketState</td>
<td>R</td>
<td>The state of the selected session. CLOSED if nothing is connected or CONNECTED if data can still be read from the socket.</td>
</tr>
<tr>
<td>DestIPAddr</td>
<td>INADDR</td>
<td>R</td>
<td>The IP address to which any triggered message is sent.</td>
</tr>
</tbody>
</table>
Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DestPort</td>
<td>UINT</td>
<td>R</td>
<td>The port to which any triggered message is sent.</td>
</tr>
<tr>
<td>pt_Error</td>
<td>POINTER TO STRING</td>
<td>R</td>
<td>An error message describing any present error condition.</td>
</tr>
</tbody>
</table>

Properties are internal values made visible through Get and Set accessors. Access is defined as R (read), W (write), or R/W (read/write).

**bootstrap_SetLocalIP (Method)**

Perform a one-time setting of the local IP address and port this socket will use.

**Inputs**

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>localPort</td>
<td>UINT</td>
<td>The port number through which this socket receives and sends data.</td>
</tr>
<tr>
<td>localIPAddr</td>
<td>INADDR</td>
<td>The IP address on the local box this socket uses. Use 0.0.0.0 to allow all local IP addresses.</td>
</tr>
</tbody>
</table>

**Return Value**

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE if the IP address and port number are set.</td>
</tr>
</tbody>
</table>

**Processing**

The bootstrap_SetLocalIP() method does the following:

- Immediately returns FALSE if the port and IP address are already set.
- Sets the IP address and port on the local machine used by this socket.

**Open (Method)**

Configure the class_TcpServer to allow for communication.

**Return Value**

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE if the port is ready to receive connections.</td>
</tr>
</tbody>
</table>
Processing

The `Open()` method does the following:

- Configures the server to receive client connections.
- Returns FALSE if the socket was unable to bind.

Close (Method)

Unconfigure the class_TcpServer to disable communication.

Inputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>forceClose</td>
<td>BOOL</td>
<td>Close the session without waiting for confirmation.</td>
</tr>
</tbody>
</table>

Processing

The `Close()` method does the following:

- Discards any unread data.
- Closes all open client sessions: gracefully if forceClose is FALSE.
- Makes the socket unable to send data or receive replies.
- Forces reconfiguration of the socket before attempting additional communication.

AcceptNextSession (Method)

Accept the next new inbound data session. This does not change the active session.

Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SESSION_HANDLE</td>
<td>The descriptor to allow further communication through this session.</td>
</tr>
</tbody>
</table>

Processing

The `AcceptNextSession()` method does the following:

- Does nothing if the socket is not open for listening.
- Accepts the next outstanding client session to as many as `maxSessions`.
- Updates `NumSessions` based on the sessions accepted.
- Returns the value SysSocket.RTS_INVALID_HANDLE if there are no outstanding sessions.
- Returns the handle to the accepted session.
SetSession (Method)

Select the session from which the class reads data from and replies to based on a previously received identifier.

### Inputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sessionID</td>
<td>SESSION_HANDLE</td>
<td>The identifier of the desired session as returned by AcceptNextSession().</td>
</tr>
</tbody>
</table>

### Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE if the session exists.</td>
</tr>
</tbody>
</table>

### Processing

The SetSession() method does the following:

- Selects the read and write session tied to sessionID.
- Sets DestIPAddr and DestPort to the values for the selected session.
- Returns false and sets DestIPAddr to 0.0.0.0, DestPort to 0, and SessionState to closed if sessionID does not represent an active session.

GetSessionInfo (Method)

Get the IP address and port number related to a session handle.

### Inputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sessionID</td>
<td>SESSION_HANDLE</td>
<td>The identifier of a session as returned by AcceptNextSession().</td>
</tr>
</tbody>
</table>

### Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sessionIPAddr</td>
<td>INADDR</td>
<td>The IP address attached to sessionID.</td>
</tr>
<tr>
<td>sessionPort</td>
<td>UINT</td>
<td>The port number attached to sessionID.</td>
</tr>
</tbody>
</table>

### Processing

The GetSessionInfo() method does the following:

- Outputs the IP address and port number tied to sessionID.
- Sets `sessionIPAddr` to 0.0.0.0 and `sessionPort` to 0, if `sessionID` does not represent an active session.

**CloseSession (Method)**

Close the selected session and clear any pending data.

**Inputs**

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>forceClose</td>
<td>BOOL</td>
<td>Close the session without waiting for confirmation.</td>
</tr>
</tbody>
</table>

**Processing**

The `CloseSession()` method does the following:

- Closes the active session.
- Sets `DestIPAddr` and `DestPort` to zero.
- Discards any unread data.
- Allows client to retain session until all data are read, if `forceClose` is FALSE.
- Decrements `NumSessions`.

**SendData (Method)**

Send a block of data to the socket.

**Inputs**

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pt_data</td>
<td>POINTER TO BYTE</td>
<td>The data to send.</td>
</tr>
<tr>
<td>numBytes</td>
<td>DINT</td>
<td>The quantity of data in bytes.</td>
</tr>
</tbody>
</table>

**Return Value**

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DINT</td>
<td>The number of bytes sent.</td>
</tr>
</tbody>
</table>

**Processing**

The `SendData()` method does the following:

- Does nothing if no valid session has been selected.
- Validates access to the pointer provided.
- Limits `numBytes` to a positive number.
- Sends all provided data to the socket.
Returns the number of bytes successfully sent.

**SendQueuedData (Method)**

Send a block of data to the socket from the front of a queue.

**Inputs/Outputs**

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue</td>
<td>class_ByteDeque</td>
<td>The data to send.</td>
</tr>
</tbody>
</table>

**Return Value**

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DINT</td>
<td>The number of bytes sent.</td>
</tr>
</tbody>
</table>

**Processing**

The `SendQueuedData()` method does the following:

- Does nothing if the socket is not open.
- Sends all provided data, starting at the front of the queue, to the socket.
- Returns the number of bytes successfully sent.
- Removes bytes sent from the front of `queue`.

**ReceiveData (Method)**

Receive a block of data from the active session.

**Inputs**

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numBytes</td>
<td>DINT</td>
<td>The number of bytes requested.</td>
</tr>
</tbody>
</table>

**Inputs/Outputs**

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>class_ByteVector</td>
<td>The container to which the data are written.</td>
</tr>
</tbody>
</table>

**Return Value**

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DINT</td>
<td>The number of bytes loaded into <code>data</code>.</td>
</tr>
</tbody>
</table>
### Processing

The `ReceiveData()` method does the following:

- Does nothing if no valid session has been selected.
- Checks for available data in the selected session.
- Requests data from the socket until there are no more or it reaches `numBytes`, whichever happens first.
- Appends all data retrieved to `data`.
- Returns the number of bytes appended.

### ReceiveToQueue (Method)

Pushes a block of data from the socket to the back of `queue`.

### Inputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numBytes</td>
<td>DINT</td>
<td>The number of bytes requested.</td>
</tr>
</tbody>
</table>

### Inputs/Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue</td>
<td>class.ByteDeque</td>
<td>The container to which the data are written.</td>
</tr>
</tbody>
</table>

### Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DINT</td>
<td>The number of bytes loaded.</td>
</tr>
</tbody>
</table>

### Processing

The `ReceiveToQueue()` method does the following:

- Does nothing if the socket is not open.
- Requests data from the socket until there are no more or it reaches `numBytes`, whichever happens first.
- Pushes all data retrieved to the back of `queue`.
- Returns the number of bytes pushed.
Benchmarks

Benchmark Platforms

The benchmarking tests recorded for this library are performed on the following platforms.

- SEL-3555
  - Dual-core Intel i7-3555LE processor
  - 4 GB ECC RAM
  - R134-V0 firmware
- SEL-3530
  - R134-V0 firmware
- SEL-3505
  - R134-V0 firmware

Benchmark Test Descriptions

fun_StringToInaddr

The posted time is the average execution time of 100 consecutive calls for the string “192.168.100.100”.

fun_InaddrToString

The posted time is the average execution time of 100 consecutive calls for the IP address 192.168.100.100.

class_UdpSocket.Open

The posted time is the average execution time of 100 successful method calls to open a socket.

class_UdpSocket.Close

The posted time is the average execution time of 100 successful method calls to close a socket.

class_UdpSocket.SendData

The posted time is the average execution time of 100 consecutive calls when sending 504 bytes of data, resulting in a 512-byte total packet size.
**class_UdpSocket.SendQueuedData**

The posted time is the average execution time of 100 consecutive calls when sending 504 bytes of data, resulting in a 512-byte total packet size.

**class_UdpSocket.ReceiveFrom**

The posted time is the average execution time of 100 consecutive calls when receiving 504 bytes of data, resulting in a 512-byte total packet size.

**class_UdpSocket.ReceiveToQueueFrom**

The posted time is the average execution time of 100 consecutive calls when receiving 504 bytes of data, resulting in a 512-byte total packet size.

**class_TcpClient.SetIP**

The posted time is the average execution time of 100 consecutive calls.

**class_TcpClient.Open**

The posted time is the average execution time of 100 successful method calls to open a socket.

**class_TcpClient.Close**

The posted time is the average execution time of 100 successful method calls to close a socket.

**class_TcpClient.SendData**

The posted time is the average execution time of 100 consecutive calls when sending 1400 bytes of data.

**class_TcpClient.SendQueuedData**

The posted time is the average execution time of 100 consecutive calls when sending 1400 bytes of data.

**class_TcpClient.ReceiveData**

The posted time is the average execution time of 100 consecutive calls when receiving 1400 bytes of data.
class TcpClient.ReceiveToQueue

The posted time is the average execution time of 100 consecutive calls when receiving 1400 bytes of data.

class TcpServer.Open

The posted time is the average execution time of 100 successful method calls to open a socket.

class TcpServer.Close

The posted time is the average execution time of 100 successful method calls to close a socket.

class TcpServer.AcceptNextSession

The posted time is the average execution time of 100 successful method calls when there is another session to accept.

class TcpServer.SetSession

The posted time is the average execution time of 100 consecutive calls.

class TcpServer.GetSessionInfo

The posted time is the average execution time of 100 consecutive calls.

class TcpServer.CloseSession

The posted time is the average execution time of 100 successful method calls to close a session.

class TcpServer.SendData

The posted time is the average execution time of 100 consecutive calls when sending 1400 bytes of data.

class TcpServer.SendQueuedData

The posted time is the average execution time of 100 consecutive calls when sending 1400 bytes of data.
**class_TcpServer.ReceiveData**

The posted time is the average execution time of 100 consecutive calls when receiving 1400 bytes of data.

**class_TcpServer.ReceiveToQueue**

The posted time is the average execution time of 100 consecutive calls when receiving 1400 bytes of data.

**Benchmark Results**

<table>
<thead>
<tr>
<th>Operation Tested</th>
<th>SEL-3555</th>
<th>SEL-3530</th>
<th>SEL-3505</th>
</tr>
</thead>
<tbody>
<tr>
<td>fun_StringToInaddr</td>
<td>2</td>
<td>12</td>
<td>18</td>
</tr>
<tr>
<td>fun_InaddrToString</td>
<td>6</td>
<td>40</td>
<td>63</td>
</tr>
<tr>
<td>class_UdpSocket.Open</td>
<td>25</td>
<td>150</td>
<td>330</td>
</tr>
<tr>
<td>class_UdpSocket.Close</td>
<td>13</td>
<td>65</td>
<td>117</td>
</tr>
<tr>
<td>class_UdpSocket.SendData</td>
<td>26</td>
<td>150</td>
<td>380</td>
</tr>
<tr>
<td>class_UdpSocket.SendQueuedData</td>
<td>25</td>
<td>160</td>
<td>380</td>
</tr>
<tr>
<td>class_UdpSocket.ReceiveFrom</td>
<td>12</td>
<td>90</td>
<td>200</td>
</tr>
<tr>
<td>class_UdpSocket.ReceiveToQueueFrom</td>
<td>12</td>
<td>100</td>
<td>220</td>
</tr>
<tr>
<td>class_TcpClient.SetIP</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>class_TcpClient.Open</td>
<td>67</td>
<td>740</td>
<td>1200</td>
</tr>
<tr>
<td>class_TcpClient.Close</td>
<td>41</td>
<td>310</td>
<td>660</td>
</tr>
<tr>
<td>class_TcpClient.SendData</td>
<td>1</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>class_TcpClient.SendQueuedData</td>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>class_TcpClient.ReceiveData</td>
<td>8</td>
<td>80</td>
<td>110</td>
</tr>
<tr>
<td>class_TcpClient.ReceiveToQueue</td>
<td>7</td>
<td>80</td>
<td>110</td>
</tr>
<tr>
<td>class_TcpServer.Open</td>
<td>36</td>
<td>200</td>
<td>400</td>
</tr>
<tr>
<td>class_TcpServer.Close</td>
<td>20</td>
<td>100</td>
<td>140</td>
</tr>
<tr>
<td>class_TcpServer.AcceptNextSession</td>
<td>20</td>
<td>97</td>
<td>130</td>
</tr>
<tr>
<td>class_TcpServer.SetSession</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>class_TcpServer.SessionInfo</td>
<td>2</td>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>class_TcpServer.CloseSession</td>
<td>33</td>
<td>320</td>
<td>690</td>
</tr>
<tr>
<td>class_TcpServer.SendData</td>
<td>1</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>class_TcpServer.SendQueuedData</td>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>class_TcpServer.ReceiveData</td>
<td>9</td>
<td>80</td>
<td>110</td>
</tr>
<tr>
<td>class_TcpServer.ReceiveToQueue</td>
<td>8</td>
<td>75</td>
<td>110</td>
</tr>
</tbody>
</table>

**Examples**

*These examples demonstrate the capabilities of this library. Do not mistake them as suggestions or recommendations from SEL.*
Implement the best practices of your organization when using these libraries. As the user of this library, you are responsible for ensuring correct implementation and verifying that the project using these libraries performs as expected.

Sending Data Out a UDP Socket

Objective

A user has an array of bytes formatted to place on the network and needs to send it to a specific IP address and port via UDP.

Solution

The user can create the program shown in Code Snippet 1 to send the byte array out each task cycle.

```
PROGRAM prg_UdpOut
VAR
  // Configuration Information - Uses any available interface.
  LocalIPAddress : STRING(15) := '0.0.0.0';
  LocalPortNumber : UINT := 5000;
  DestinationIPAddress : STRING(15) := '10.10.10.10';
  DestinationPortNumber : UINT := 5000;

  // Data to send each task cycle.
  Data : ARRAY [1 .. 1000] OF BYTE;

  // Socket to send data on.
  UdpSocket : class_UdpSocket(maxPacketSize := 1024);

  // Initialization variables.
  SocketInitialized : BOOL := FALSE;
  LocalIP : SELEthernetController.INADDR;
  DestIP : SELEthernetController.INADDR;
END_VAR

IF NOT SocketInitialized THEN
  fun_StringToInaddr(LocalIPAddress, ipAddr => LocalIP);
  fun_StringToInaddr(DestinationIPAddress, ipAddr => DestIP);
  UdpSocket.bootstrap_SetLocalIP(LocalPortNumber, LocalIP);
  UdpSocket.Open();
  SocketInitialized := TRUE;
ELSE
  UdpSocket.SendData(ADR(Data[1]), SIZEOF(Data),
                    DestIP, DestinationPortNumber);
END_IF
```
Creating a UDP Server

Objective

A user would like the RTAC to be able to receive data from multiple clients over the UDP protocol.

After some internal validation of a received packet, the server will reply with OK in ASCII if the packet was correctly formatted.

Assumptions

This solution assumes that the AcRTAC project containing the provided code includes an Access Point opening at the desired port and that the inbound packets are not larger than 1024 bytes.

Solution

The user can create the program shown in Code Snippet 2 to receive one inbound data packet each task cycle.
Creating a TCP Client

Objective

A user would like the RTAC to be able to send data to a TCP server for modification.

Assumptions

For this use case, we assume that the server doubles any data that are sent to it.

Solution

The user can create the program defined in Code Snippet 3 to send packets to the remote server and receive data in reply.
Code Snippet 3  prg_TcpClient

PROGRAM prg_TcpClient
VAR
// Configuration Information - Uses any available interface.
LocalIPAddress : STRING(15) := '0.0.0.0';
LocalPortNumber : UINT := 2442;
DestinationIPAddress : STRING(15) := '10.10.10.10';
DestinationPortNumber : UINT := 2442;

// Storage for data being sent and received.
DataOut : ARRAY [1 .. 200] OF BYTE;
DataIn : SELEthernetController.classByteVector;

// The socket and its initialization state.
TcpClient : class_TcpClient;
SocketInitialized : BOOL := FALSE;

// IP address variables.
LocalIP : SELEthernetController.INADDR;
DestIP : SELEthernetController.INADDR;

// Flags used in manipulating the state of the function.
IsSending : BOOL;
DataSent : DINT;
DataReceived : DINT;

END_VAR

IF NOT SocketInitialized THEN
  fun_StringToInaddr(LocalIPAddress, ipAddr => LocalIP);
  fun_StringToInaddr(DestinationIPAddress, ipAddr => DestIP);
  TcpClient.bootstrap_SetLocalIP(LocalPortNumber, LocalIP);
  TcpClient.SetIP(DestIP, DestinationPortNumber);
  TcpClient.Open();
  SocketInitialized := TRUE;
  IsSending := TRUE;
ELSE
  IF IsSending THEN
    DataSent := TcpClient.SendData(ADR(DataOut[1]), 200);
    DataIn.Recycle();
    IsSending := FALSE;
    DataReceived := 0;
  ELSE
    // Here we request the total data expected minus anything
    // already received and add it to anything already received.
    // This allows the reception of data to cross multiple cycles.
    DataReceived := DataReceived +
      TcpClient.ReceiveData((DataSent * 2) - DataReceived, DataIn);
    IF DataReceived = (2 * DataSent) THEN
      // Do some work based the server’s reply.
      IsSending := TRUE;
    END_IF
  END_IF
END_IF
END_IF

Date Code 20180928 Instruction Manual SELEthernetController
Parsing Network Traffic With a Deque

This example demonstrates using a deque for network communication.

**Objective**

Parse the data stream of information sent from a TCP server to the TCP client on the RTAC. Increment a counter every time the characters “SEL” are seen in the stream.

**Assumptions**

This example assumes that there is a server to connect to and streams data through the connection once the connection is established. It also assumes that the library Queue has been inserted in the project.

**Solution**

The deque is used as storage for information received from a TCP socket. The received data are then searched for the string ‘SEL’ and a counter is incremented every time the string is found. As it searches for the string, data are discarded from the front of the deque. This implementation is shown in Code Snippet 4

```
PROGRAM prg_SELSearch
VAR
    // Configuration Information - Uses any available interface.
    LocalIPAddress : STRING(15) := '0.0.0.0';
    LocalPortNumber : UINT := 2442;
    DestinationIPAddress : STRING(15) := '10.10.10.10';
    DestinationPortNumber : UINT := 2442;

    // The socket and its initialization state.
    TcpClient : class_TcpClient;
    SocketInitialized : BOOL := FALSE;
    // IP address variables.
    LocalIP : SELEthernetController.INADDR;
    DestIP : SELEthernetController.INADDR;
    // Deque to hold received data.
    DataIn : Queue.class_ByteDeque(0);
    // The string to search for in the incoming data.
    TargetString : STRING := 'SEL';
    // A temporary string used to find the targetString.
    Peek : STRING(3);
    // A flag for breaking the search loop.
    Searching : BOOL;
    // Counter for the number of times the targetString is found.
    Counter : UDINT := 0;
END_VAR

IF NOT SocketInitialized THEN
    fun_StringToInaddr(LocalIPAddress, ipAddr => LocalIP);
    fun_StringToInaddr(DestinationIPAddress, ipAddr => DestIP);
    TcpClient.bootstrap_SetLocalIP(LocalPortNumber, LocalIP);
```

---

**Code Snippet 4** prg_SELSearch

```
Configuring a Simple TCP Server With Sessions

Objective

Parse all input data from several clients looking for the string “Hello.” When the string is found, that client receives a reply “World$N$R”.

The server handles accepting and tracking all sessions and iterates across them to allow each time to process.

Assumptions

This example assumes that Port 10024 is open through use of an Access Point configured as Ethernet Incoming and set to receive Raw TCP. It also assumes that the client can access that port through any intermediary firewalls. It also assumes that the library Queue has been inserted in the project.

All error checking has been omitted to facilitate the brevity of this implementation.

There must be a structure defined to hold session related data. Code Snippet 5 shows an example for this implementation.
Code Snippet 5  Session Structure

```plaintext
TYPE struct_SessionInfo :
STRUCT
   // There is a connected session stored here.
   Active : BOOL := FALSE;
   // The Handle for this session.
   Handle : SESSION_HANDLE := SysSocket.RTS_INVALID_HANDLE;
   // Deque containing incoming data ready for consumption.
   ReceiveDeque : class_ByteDeque(0);
   // Deque containing data ready to be sent to the telnet client.
   SendDeque : class_ByteDeque(0);
END_STRUCT
END_TYPE
```

Solution

Instantiate a class_TcpServer and associated data control objects as seen in Code Snippet 6. Allow this program to run every task scan.

Verification

To verify that this solution is functional, start running the server on the RTAC and then attach with a raw TCP connection on port 10024. Any instance of the word “Hello” should be followed immediately by the echo “World.”

Characters other than new-lines should eventually echo “Usage : Hello.”

The code as written should accept as many as five concurrent sessions with more allowed after a given session disconnects.
CODE SNIPPET 6  prg_SimpleServer

PROGRAM prg_SimpleServer
VAR CONSTANT
  _c_NumSessions : USINT := 5;
END_VAR

VAR
  // Configuration Information
  // Uses any available interface.
  LocalIPAddress : STRING(15) := '0.0.0.0';
  LocalPortNumber : UINT := 10024;
  ListenIPAddr : INADDR;
  SocketInitialized : BOOL := FALSE;
  TcpServer : class_TcpServer(_c_NumSessions);
  Sessions : ARRAY [1 .. _c_NumSessions] of struct_SessionInfo;

  Peek : STRING(5);
  Reply : STRING(7) := 'World$R$N';
  Reply2 : STRING(15) := 'Usage : Hello$R$N';

  tempSession : SESSION_HANDLE;
  sendByteCount : DINT;
  numSend : DINT;
  i : UDINT;
END_VAR

IF NOT SocketInitialized THEN
  IF SELEthernetController.fun_StringToInaddr(LocalIPAddress, ipAddr =>
      ListenIPAddr) THEN
    TcpServer.bootstrap_SetLocalIP(LocalPortNumber, ListenIPAddr);
    IF TcpServer.Open() THEN
      SocketInitialized := TRUE;
    END_IF
  END_IF
ELSE
  // Clean up any session that is in error.
  IF TcpServer.SessionState = ERROR THEN
    TcpServer.CloseSession(TRUE);
  END_IF

  // Accept any new sessions if possible.
  FOR i := 1 TO _c_NumSessions DO
    IF NOT Sessions[i].Active THEN
      // There is a session object available,
      // see if there is a new session pending.
      tempSession := TcpServer.AcceptNextSession();
      IF tempSession <> SysSocket.RTS_INVALID_HANDLE THEN
        Sessions[i].Handle := tempSession;
        Sessions[i].Active := TRUE;
        Sessions[i].SendDeque.Recycle();
        Sessions[i].ReceiveDeque.Recycle();
        EXIT
      END_IF
      //There are no outstanding sessions. Stop asking.
      EXIT;
  END_FOX
// Cycle through all current sessions and process their data.
FOR i := 1 TO _c_NumSessions DO
  IF Sessions[i].Active THEN
    IF TcpServer.SetSession(Sessions[i].Handle) THEN
      // Read any data from the socket.
      TcpServer.ReceiveToQueue(_c_NumBytes, Sessions[i].ReceiveDeque);

      // Here is where meaningful work would be added.
      WHILE Sessions[i].ReceiveDeque.Size >= 5 DO
        // See if the string 'SEL' appears in the front of the deque.
        IF 5 = Sessions[i].ReceiveDeque.Front(ADR(Peek), 5) THEN
          IF peek = 'Hello' THEN
            Sessions[i].SendDeque.PushBack(ADR(Reply), 7);
            Sessions[i].ReceiveDeque.EraseFront(5);
          ELSIF peek[0] <> 10 and peek[0] <> 13 THEN
            // The string wasn't found, erase the first character and continue looking.
            Sessions[i].SendDeque.PushBack(ADR(Reply2), 15);
            Sessions[i].ReceiveDeque.EraseFront(1);
          ELSE
            Sessions[i].ReceiveDeque.EraseFront(1);
          END_IF
        ELSE
          EXIT;
        END_IF
      END_WHILE
      TcpServer.SendQueuedData(Sessions[i].SendDeque);
    IF TcpServer.SessionState = CLOSED THEN
      Sessions[i].Handle := RTS_INVALID_HANDLE;
      Sessions[i].Active := FALSE;
    END_IF
  ELSE
    // If the session closed while doing other work, clean up.
    Sessions[i].Handle := RTS_INVALID_HANDLE;
    Sessions[i].Active := FALSE;
  END_IF
END_IF
END_FOR
END_IF
END_IF
END_FOR
## Release Notes

<table>
<thead>
<tr>
<th>Version</th>
<th>Summary of Revisions</th>
<th>Date Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.1.0</td>
<td>▶ Allows new versions of ACSELERATOR RTAC to compile projects for previous firmware versions without SEL IEC types “Cannot convert” messages. Must be used with R143 firmware or later.</td>
<td>20180921</td>
</tr>
<tr>
<td>3.5.0.6</td>
<td>▶ Allow the class to recover instead of closing the socket when attempting to send large amounts of data all at once.</td>
<td>20160501</td>
</tr>
<tr>
<td>3.5.0.5</td>
<td>▶ Added queues as input and output mechanisms for socket data. ▶ Allow TCP client ports to connect to a server without binding to a specific local port.</td>
<td>20150511</td>
</tr>
<tr>
<td>3.5.0.3</td>
<td>▶ Made TCP sockets not throw error when outgoing data buffer is full.</td>
<td>20141107</td>
</tr>
<tr>
<td>3.5.0.2</td>
<td>▶ Initial release.</td>
<td>20141010</td>
</tr>
</tbody>
</table>