INDUSTRIAL CONTROL COMMUNICATIONS, INC.

NetLink

Invoking NetLink Application Components on Android Devices

July 12, 2013

TABLE OF CONTENTS

1. Introduction

This document describes how to access the various components of the NetLink application and its plugins through standard Android API's for accessing content providers using URI's, invoking activities and services using intents, and passing messages to running services.

The ability to use other Android application components in your own application is built into the Android framework. As long as an application is installed on the device, and the application publishes its components' availability to the Android system, it can be used by any other application. For example, if an application wishes to show a map of where a city is located in the US, it can simply invoke the Google Maps application by passing an appropriate intent. The map then opens as if it were a part of your own application. If that application then wanted to show all the user's contacts located in that city, it can access the Contacts application's content provider to get a list of contacts whose addresses are in that city.

In this same way, the components of the NetLink application and its plugins can be accessed by your applications as long as the NetLink application or plugin application that hosts the components used is installed on the device. In general, an application will interact with the content provider for the protocol of choice to configure objects and monitor or command values. Objects can also be configured by invoking the object settings activity of the particular protocol in the same way as the Google Maps example above. Once the objects are configured, an application can bind to that protocol's service to begin communication on the network. When the service is started, it will automatically begin communicating on the network. The service will continually update values in the content provider by making requests on the network. When a value in the content provider changes and your application sets the write flag, the service will write this new value to the network.

2. NetLink Components

The table below lists the various components of the NetLink application and its plugin applications. It also describes which components are contained in each package. Note that the NetLink application is not required to be installed if only using components supplied by a plugin application.

Application	Package	Component
NetLink	com.iccdesigns.netlink	NetLink App (Object List)
		NetLink Preferences
	com.iccdesigns.netlink.graphing	Graphs
	com.iccdesigns.netlink.grouping	Groups
	com.iccdesigns.netlink.discovery	Object Discovery
NetLink – Demo Plugin (Packaged with NetLink app)	com.iccdesigns.netlink.demo	Demo Content Provider
		Demo Object Settings
		Demo Preferences
	com.iccdesigns.netlink.demo.service	Demo Protocol Service
NetLink – Modbus/TCP Plugin	com.iccdesigns.netlink.modbus	Modbus Content Provider
		Modbus Object Settings
		Modbus Preferences
	com.iccdesigns.netlink.modbus.service	Modbus Protocol Service
NetLink – BACnet/IP Plugin	com.iccdesigns.netlink.bacnet	BACnet Content Provider
		BACnet Object Settings
		BACnet Preferences
	com.iccdesigns.netlink.bacnet.service	BACnet Protocol Service

2.1 Component Overview

This section describes what each NetLink component does and how it is used by other components.

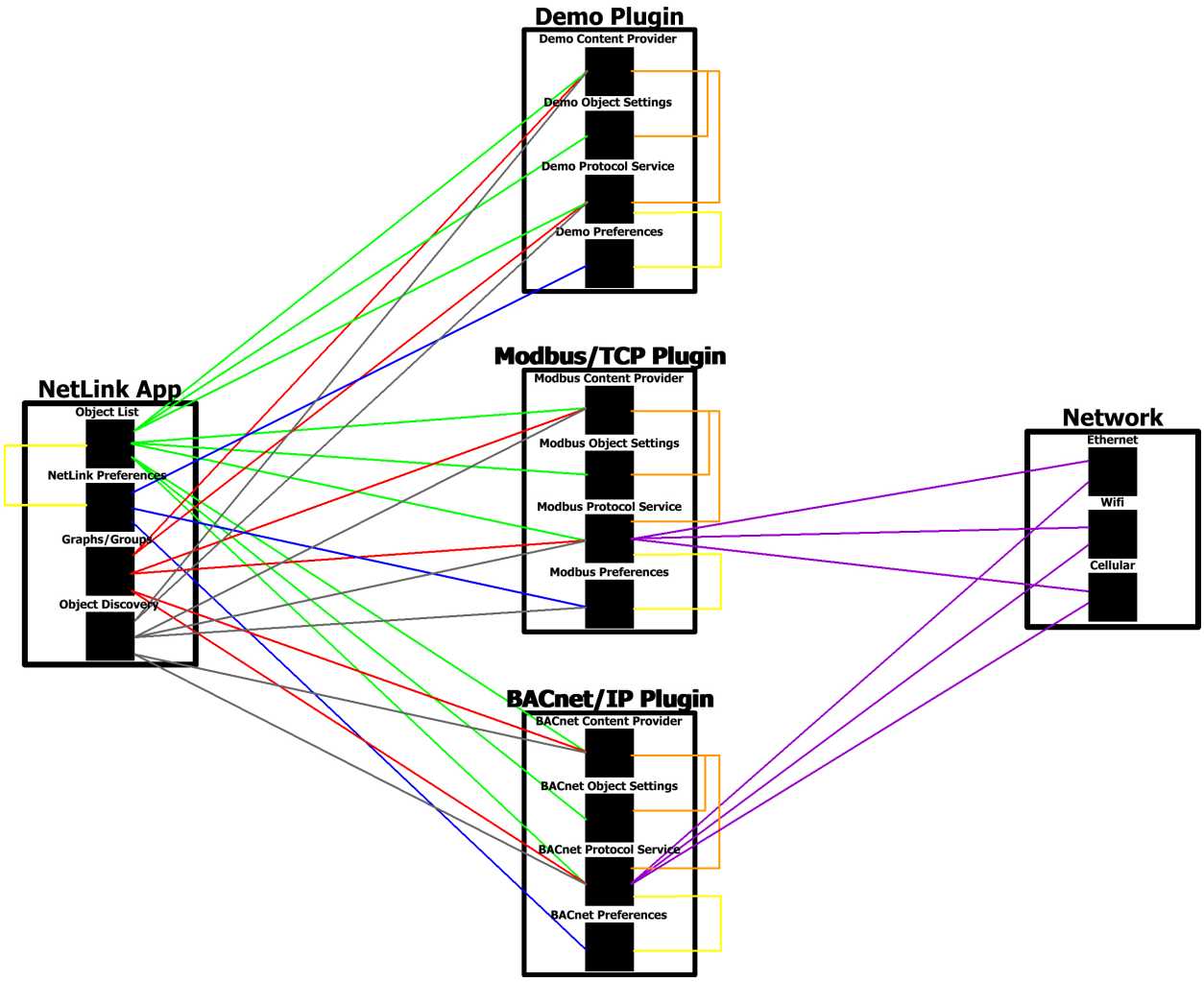
- **NetLink App**
  - *Object List*
    - The Object List activity displays all configured objects in the Content Provider for the protocol specified in the intent which invoked the activity.
  - *NetLink Preferences*
    - The NetLink Preferences activity provides settings for the NetLink application for a user to configure.
    - The Object List activity invokes the NetLink Preferences activity for a user to configure setting for NetLink.
  - *Graphs*
    - The Graphs activity displays graphs of object values over time for the protocol specified in the intent which invoked the activity.
  - *Groups*
    - The Groups activity displays groups of objects for the protocol specified in the intent which invoked the activity.
  - *Object Discovery*
    - The Object Discovery activity is used to browse for objects on the network for the protocol specified in the intent which invoked the activity. The Object Discovery activity can then add discovered objects to the protocol's Content Provider.

- **Protocol Plugin**
  - *Content Provider*
    - The Content Provider stores the configured objects for the protocol.
    - The Object Settings activity accesses the Content Provider to create and update protocol objects.
    - The Protocol Service accesses the Content Provider to make requests on the network and update object values.
    - The Object List, Groups, and Graphs activities in the NetLink App access the protocol's Content Provider to read and write properties of protocol objects.
    - The Object Discovery activity in the NetLink App accesses the protocol's Content Provider to create discovered objects.
  - *Object Settings*
    - The Object Settings activity is used to configure a new protocol object or edit an existing object.
    - The Object List in the NetLink App invokes the Object Settings activity to create and edit protocol objects.
  - *Protocol Service*
    - The Protocol Service makes requests on the network and updates values in the Content Provider.
    - The Object List, Groups, Graphs, and Object Discovery activities in the NetLink App bind to the Protocol Service and send messages to the service so that requests are made on the network and values are updated in the Content Provider while the activity is running.
  - *Preferences*
    - The Preferences activity provides protocol-specific settings for a user to configure.
    - The Protocol Service accesses the settings configured by the Preferences activity to define the network behavior of the service.
    - The NetLink Preferences activity in the NetLink App invokes the protocol's Preferences activity for a user to configure protocol-specific settings.
    - The Object Discovery activity in the NetLink App invokes the Modbus Preferences activity for a user to configure discovery settings for Modbus.

- **Network**
  - *Ethernet*
    - The Ethernet interface is used to communicate via a wired connection to other devices.
    - The protocol's Protocol Service connects to this interface to communicate with devices on the network.
  - *Wifi*
    - The Wifi interface is used to communicate via a wireless connection to other devices.
    - The protocol's Protocol Service connects to this interface to communicate with devices on the network.
  - *Cellular*
    - The Cellular interface is used to communicate via a mobile data connection to other devices.
    - The protocol's Protocol Service connects to this interface to communicate with devices on the network.

2.2 Component Interaction

The graphic below displays the interaction between various NetLink components described in the previous section. A line drawn between components indicates that one component uses the other component.



3. Content Providers

The table below lists the content providers of the NetLink application and its plugin applications.

Provider	URI	Available Columns
Demo	“content://com.iccdesigns.provider.netlink.demo/demo”	“ _count”, “_id”, “desc” , “ipAddr”, “dest”, “type”, “start”, “num”, “value”, “minVal”, “maxVal”, “mult”, “offset”, “units”, “writeFlag”, “readOnly”, “status”, “radix”
Modbus	“content://com.iccdesigns.provider.netlink.modbus/modbus”	“ _count” , “_id”, “desc”, “ipAddr”, “dest”, “type”, “start”, “num”, “value”, “minVal”, “maxVal”, “mult”, “offset”, “units”, “writeFlag”, “readOnly”, “status”, “radix”, “unitId”, “_32bit”, “float”, “wordSwap”, “wordSize”, “wrSingle”
BACnet	“content://com.iccdesigns.provider.netlink.bacnet/bacnet”	“ _count” , “_id”, “desc” , “ipAddr”, “dest”, “type”, “start”, “num”, “value”, “minVal”, “maxVal”, “mult”, “offset”, “units”, “writeFlag”, “readOnly”, “status”, “radix”, “priority”

3.1 Column Description

3.1.1 Common Columns

3.1.1.1 _count

The count of rows in a directory.
Type: INTEGER

3.1.1.2 _id

The unique ID for a row.
Type: INTEGER (long)

3.1.1.3 desc

The description of the object.
Type: TEXT

3.1.1.4 ipAddr

The IP Address of the target in the object. Note that not all protocols use this column.
Type: TEXT

3.1.1.5 dest

The destination device instance to access. Note that not all protocols use this column.
Type: INTEGER (int)

3.1.1.6 type

The object type of the object to access. The available types are listed below.
Type: INTEGER (int)

Protocol	Type	Value
Demo	Analog Input	0
	Analog Output	1
	Digital Input	2
	Digital Output	3
Modbus	Holding Register	0
	Input Register	1
	Coil Status	2
	Input Status	3
BACnet	Analog Input	0
	Analog Output	1
	Analog Value	2
	Binary Input	3
	Binary Output	4
	Binary Value	5
	Multi-state Input	13
	Multi-state Output	14
	Multi-state Value	19

3.1.1.7 start

The starting instance number of the object to access. Note that because only one instance number is currently supported per object, this field is simply the instance number of the object to access.
Type: INTEGER (int)

3.1.1.8 num

The number of instances to access. Note that because only one instance number is currently supported per object, this field must be set to a value of 1.
Type: INTEGER (int)

3.1.1.9 value

The value of the object.
Type: REAL (double)

3.1.1.10 minVal

The maximum value of the object.
Type: REAL (double)

3.1.1.11 maxVal

The minimum value of the object.
Type: REAL (double)

3.1.1.12 mult

The multiplier used in scaling the value.
Type: REAL (double)

3.1.1.13 offset

The offset used in scaling the value.
Type: REAL (double)

3.1.1.14 units

The displayed units for the object.
Type: TEXT

3.1.1.15 writeFlag

Set this to indicate to the service whether or not there is an outstanding write for this object. For BACnet, to relinquish an object's value at the configured priority, set the write flag to a value of 2.
Type: INTEGER (int)

0 ..... FALSE (There is no outstanding write for this object)
1 ..... TRUE (There is an outstanding write for this object)
2 ..... Relinquish Value (*BACnet only*: there is an outstanding write for this object and that write should relinquish any previously-written value)

3.1.1.16 readOnly

Indicates whether or not this object is read only.
Type: INTEGER (boolean)

3.1.1.17 status

The connection status of the object.
Type: INTEGER (int)

0 ..... DISCONNECTED (A connection has not been established with target device, or a timeout is occuring on the object.)
1 ..... WARNING (A protocol exception was received, or some other warning condition occurred)
2 ..... ERROR (A connection error has occurred, such as a socket error or reception error)
3 ..... CONNECTED (A connection has been established and the object's value is reliable)

3.1.1.18 radix

The radix used for the value of the object.
Type: INTEGER (int)

0 ..... DECIMAL
1 ..... HEXADECIMAL

3.1.2 Modbus Columns

3.1.2.1 unitId

The Unit ID for the Modbus object.
Type: INTEGER (int)

3.1.2.2 _32bit

Whether the 32-bit Modbus extension should be used.
Type: INTEGER (boolean)

3.1.2.3 float

Whether the 32-bit value is a floating point value.
Type: INTEGER (boolean)

3.1.2.4 wordSwap

Whether the 32-bit value should be word swapped.
Type: INTEGER (boolean)

3.1.2.5 wordSize

Whether the 32-bit value is addressed by word size registers
Type: INTEGER (boolean)

3.1.2.6 wrSingle

Whether to use the Write Single or Write Multiple Modbus function code. Note that this is only valid for Holding Register and Coil types.
Type: INTEGER (boolean)

3.1.3 BACnet Columns

3.1.3.1 priority

The priority used for writes. A value of 0 indicates a priority of “None”, which is equivalent to priority 16 (the lowest priority). Note that this is only valid for commandable object types.
Type: INTEGER (int)

3.2 Content Provider Code Examples

This section contains code examples for various ways of accessing the content provider of a NetLink plugin.

3.2.1 Quering a Content Provider

3.2.1.1 Read All Objects

```
Cursor cursor = getActivity().getContentResolver().query(Uri.parse(
       "content://com.iccdesigns.provider.netlink.bacnet/bacnet"),
       null, null, null, null);
```

3.2.1.2 Read One Row

```
int rowId = getObjectId();

Cursor cursor = getActivity().getContentResolver().query(ContentUris.withAppendedId(
       Uri.parse("content://com.iccdesigns.provider.netlink.bacnet/bacnet"),
       rowId), null, null, null, null);
```

3.2.1.3 Read Certain Rows

```
int destination;

destination = getSelectedDeviceInstance();

Cursor cursor = getActivity().getContentResolver().query(Uri.parse(
       "content://com.iccdesigns.provider.netlink.bacnet/bacnet"),
       null, "dest=?", new String[] { Integer.toString(destination) }, null);
```

3.2.1.4 Read Certain Columns of All Rows

```
String[] projection = new String[] { "_id", "desc", "type", "value" };

Cursor cursor = getActivity().getContentResolver().query(Uri.parse(
       "content://com.iccdesigns.provider.netlink.modbus/modbus"),
       projection, null, null, null);
```

3.2.1.5 Read All Rows, Sorting by a Certain Column

```
Cursor cursor = getActivity().getContentResolver().query(Uri.parse(
       "content://com.iccdesigns.provider.netlink.modbus/modbus"),
       null, null, null, "ipAddr ASC");
```

3.2.2 Inserting a Row into a Content Provider

3.2.2.1 Add a Row

```
ContentValues values = new ContentValues();

// Create an Analog Value named My Object at instance 100 on device 1,
// with write priority = 8
values.put("desc", "My Object");
values.put("dest", 1);
values.put("type", 2);          // BACnet Object Type – 2 = AV
values.put("start", 100);
```

```
values.put("num", 1);            // Only 1 instance is currently supported,
values.put("mult", 1);           // so this must be set to 1
values.put("offset", 0);
values.put("units", "Hz");
values.put("radix", 0);          // Set to Decimal Radix
values.put("priority", 8);       // Use a value of 8 for the priority

getActivity().getContentResolver().insert(Uri.parse(
      "content://com.iccdesigns.provider.netlink.bacnet/bacnet"), values);
```

3.2.3 Updating a Row in a Content Provider

3.2.3.1 Update All Rows

```
ContentValues values = new ContentValues();
int destination;

destination = getSelectedDeviceInstance();
values.put("dest", destination);

getActivity().getContentResolver().update(Uri.parse(
      "content://com.iccdesigns.provider.netlink.bacnet/bacnet"),
      values, null, null);
```

3.2.3.2 Update One Row

```
ContentValues values = new ContentValues();
int curRowId = getObjectId();
int newValue = getObjectValue();

values.put("value", newValue);   // We must set the write flag when updating
values.put("writeFlag", 1);      // an object's value

getActivity().getContentResolver().update(ContentUris.withAppendedId(Uri.parse(
      "content://com.iccdesigns.provider.netlink.modbus/modbus"), curRowId),
      values, null, null);
```

3.2.3.3 Update Certain Rows

```
ContentValues values = new ContentValues();
int destination = getSelectedDeviceInstance();
int newValue = getValueForAllObjects();

values.put("value", newValue);   // We must set the write flag when updating
values.put("writeFlag", 1);      // an object's value

getActivity().getContentResolver().update(Uri.parse(
      "content://com.iccdesigns.provider.netlink.bacnet/bacnet"),
      values, "dest=?", new String[] { Integer.toString(destination) });
```

3.2.4 Deleting a Row or Rows from a Content Provider

3.2.4.1 Delete All Rows

```
getActivity().getContentResolver().delete(Uri.parse(
      "content://com.iccdesigns.provider.netlink.modbus/modbus"), null, null);
```

3.2.4.2 Delete One Row

```
int curRowId = getObjectId();

getActivity().getContentResolver().delete(ContentUris.withAppendedId(Uri.parse(
      "content://com.iccdesigns.provider.netlink.modbus/modbus"), curRowId),
      null, null);
```

3.2.4.3 Delete Certain Rows

```
int destination;

destination = getSelectedDeviceInstance();

getActivity().getContentResolver().delete(Uri.parse(
      "content://com.iccdesigns.provider.netlink.bacnet/bacnet"), "dest=?",
      new String[] { Integer.toString(destination) });
```

4. Intents List

The table below lists the intents that your application can send in order to invoke NetLink components on Android devices in certain ways. For each action/uri pair, the table describes how the receiving NetLink component handles the intent.

Target Component	Intent URI	Intent Category	Intent Action	Result
Protocol Service (Demo Service) (Modbus Service) (BACnet Service)	“content://com.iccdesigns.provider.netlink.demo/demo” “content://com.iccdesigns.provider.netlink.modbus/modbus” “content://com.iccdesigns.provider.netlink.bacnet/bacnet”		"android.intent.action.RUN" "android.intent.action.SYNC"	Pass this intent to the bindService() function to start the protocol service. The service will interact with objects configured in the content provider.
NetLink App (Demo Objects) (Modbus Objects) (BACnet Objects)	“content://com.iccdesigns.provider.netlink.demo/demo” “content://com.iccdesigns.provider.netlink.modbus/modbus” “content://com.iccdesigns.provider.netlink.bacnet/bacnet”		“android.intent.action.VIEW“ “android.intent.action.EDIT”	Opens the NetLink application, showing the object list of all configured objects in the content provider and allowing a user to add and edit objects.
Object Settings (Create Demo Object) (Create Modbus Object) (Create BACnet Object) (Edit Demo Object) (Edit Modbus Object) (Edit BACnet Object)	“content://com.iccdesigns.provider.netlink.demo/demo” “content://com.iccdesigns.provider.netlink.modbus/modbus” “content://com.iccdesigns.provider.netlink.bacnet/bacnet” “content://com.iccdesigns.provider.netlink.demo/demo/_ID” “content://com.iccdesigns.provider.netlink.modbus/modbus/_ID” “content://com.iccdesigns.provider.netlink.bacnet/bacnet/_ID”		"android.intent.action.INSERT" "android.intent.action.EDIT"	Opens the object settings activity to create a new object or edit an existing object. Note that to edit an object, append the object id (content provider _id field) to the URI.
Graphs (Demo Graphs) (Modbus Graphs) (BACnet Graphs)	“content://com.iccdesigns.provider.netlink.demo/demo” “content://com.iccdesigns.provider.netlink.modbus/modbus” “content://com.iccdesigns.provider.netlink.bacnet/bacnet”		“com.iccdesigns.netlink.graphing.VIEW”	Opens the NetLink graphs activity which shows all configured graphs.

Target Component	Intent URI	Intent Category	Intent Action	Result
Groups (Demo Groups) (Modbus Groups) (BACnet Groups)	"content://com.iccdesigns.provider.netlink.demo/demo" "content://com.iccdesigns.provider.netlink.modbus/modbus" "content://com.iccdesigns.provider.netlink.bacnet/bacnet"		"com.iccdesigns.netlink.grouping.VIEW"	Opens the NetLink groups activity which shows all configured groups.
Object Discovery (Discover Demo Objects) (Discover Modbus Objects) (Discover BACnet Objects)	"content://com.iccdesigns.provider.netlink.demo/demo" "content://com.iccdesigns.provider.netlink.modbus/modbus" "content://com.iccdesigns.provider.netlink.bacnet/bacnet"		"com.iccdesigns.netlink.discovery.VIEW"	Opens the NetLink object discovery activity which can browse for objects on the network and add them to the content provider.
NetLink Preferences	"content://com.iccdesigns.provider.netlink.demo/demo" "content://com.iccdesigns.provider.netlink.modbus/modbus" "content://com.iccdesigns.provider.netlink.bacnet/bacnet"	"com.iccdesigns.netlink.MAIN_PREFERENCES"	"com.iccdesigns.netlink.PREFERENCES"	Opens the NetLink main preferences activity.
Modbus Preferences	"content://com.iccdesigns.provider.netlink.modbus/modbus"	*"android.intent.category.PREFERENCE"*	*"com.iccdesigns.netlink.PREFERENCES"*	Opens the Modbus preferences activity to configure Modbus-specific preferences.
BACnet Preferences	"content://com.iccdesigns.provider.netlink.bacnet/bacnet"	*"android.intent.category.PREFERENCE"*	*"com.iccdesigns.netlink.PREFERENCES"*	Opens the BACnet preferences activity to configure BACnet-specific preferences.

4.1 Intents Code Examples

This section contains code examples of how to start a protocol service or a NetLink activity using an intent.

4.1.1 Starting a Protocol Service

```
/** Messenger for communicating with the service. */
private Messenger mService = null;

/** Flag indicating whether we have called bind on the service. */
private boolean mBound = false;

/** Class for interacting with the main interface of the service. */
private ServiceConnection mConnection = new ServiceConnection() {
	public void onServiceConnected(ComponentName className, IBinder service) {

		// This is called when the connection with the service has been
		// established, giving us the object we can use to
		// interact with the service. We are communicating with the
		// service using a Messenger, so here we get a client-side
		// representation of that from the raw IBinder object.
		mService = new Messenger(service);
		mBound = true;
	}

	public void onServiceDisconnected(ComponentName className) {
		// This is called when the connection with the service has been
		// unexpectedly disconnected -- that is, its process crashed.
		mService = null;
		mBound = false;
	}
};

@Override
public void onStart() {
	super.onStart();

	Intent intent = new Intent(Intent.ACTION_RUN, Uri.parse(
			"content://com.iccdesigns.provider.netlink.bacnet/bacnet"));
	getActivity().bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
}

@Override
public void onStop() {
	super.onStop();

	// Unbind from the service
	if (mBound) {
		getActivity().unbindService(mConnection);
		mBound = false;
	}
}
```

4.1.2 Starting a NetLink Activity

```
Intent intent = new Intent("com.iccdesigns.netlink.graphing.VIEW",
      Uri.parse("content://com.iccdesigns.provider.netlink.modbus/modbus"));

try {
      startActivity(intent);
} catch (ActivityNotFoundException e) {
      Log.e("MyActivity", "Could not start Activity", e);
}
```

4.1.3 Starting a Protocol's Settings Activity

```
Intent intent = new Intent("com.iccdesigns.netlink.PREFERENCES",
      Uri.parse("content://com.iccdesigns.provider.netlink.bacnet/bacnet"));
intent.addCategory(Intent.CATEGORY_PREFERENCE);

try {
      startActivity(intent);
} catch (ActivityNotFoundException e) {
      Log.e("MyActivity", "Could not start Activity", e);
}
```

5. Protocol Service Messaging

The table below shows a list of supported messages which can be sent to or received from each protocol service. Note that the Restart Service message should be sent after sending messages to set protocol configuration items such as port, device name, device instance, etc. The service does not need to be restarted when setting the scan rate and timeout. These changes take effect immediately.

Protocol	Message	Value	Data	Description
All	Register Messenger	0		Send this message to the service to register a Messenger object as a listener of messages sent from the service.
	Get Display Name	1	Type: Bundle Key: “DisplayName” Value: String	Send this message with no data to receive a message back which includes the display name of the service as a Bundle type in the data of the message. Note that the replyTo field must be populated with the Messenger you wish the service to reply to. This message will set the registered listener to the Messenger populated in the replyTo field.
	Set Scan Rate	2	Type: arg1 (int)	Sets the scan rate (in ms) for the service from the value passed in the arg1 field of the message.
	Set Timeout	3	Type: arg1 (int)	Sets the timeout (in ms) for the service from the value passed in the arg1 field of the message.
	Restart Service	4		Restarts the service.
	Invalid License	5	Type: Bundle Key: “ErrorCode” Value: String	This message is sent from the service to indicate that either the service is not licensed (no data) or there has been a licensing error (error code name is passed in data). Note that a Messenger must be registered with the service for this message to be sent.
	Initiate Device Discovery	6	Type: arg1 (int) Type: arg2 (int)	This message informs the service to start discovering devices on the network using the device range passed in by arg1 (start device) and arg2 (end device). For Modbus, the arguments are IP addresses in big-endian, integer format. For all other protocols, the arguments are device instance numbers.
	Initiate Object Discovery	7	Type: obj (String) Type: arg1 (int)	This message informs the service to start discovering objects on a particular device. The obj parameter is the string representation of the device’s IP address or hostname. (Note that the obj parameter is only used for Modbus.) The arg1 parameter is the instance number (or unit ID for Modbus) of the device.
	End Discovery	8		This message informs the service to stop its current discovery operation.

Protocol	Message	Value	Data	Description
	Device Discovered	9	Type: Bundle Key: “desc” Value: String Key: “idStr” Value: String Key: “ipAddr” Value: String Key: “dest” Value: int	This message is sent from the service to indicate that a device was discovered. The discovered device is detailed in the Bundle type in the data of the message. The keys are defined as follows: “desc” - The name of the device “idStr” - The identifier string for the device “ipAddr” - The IP address the device is at. (Note that this is only included for Modbus) “dest” - The instance (or unit ID for Modbus) of the device.
	Object Discovered	10	Type: Bundle Key: “desc” Value: String Key: “idStr” Value: String Key: “ipAddr” Value: String Key: “dest” Value: int Key: “type” Value: int Key: “start” Value: int Key: “units” Value: String	This message is sent from the service to indicate that an object was discovered. The discovered object is detailed in the Bundle type in the data of the message. The keys are defined as follows: “desc” - The name of the object “idStr” - The identifier string for the object “ipAddr” - The IP address the device which contains the object is at. (Note that this is only included for Modbus.) “dest” - The instance (or unit ID for Modbus) of the device which contains the object. “type” - The type of the object. See the “type” column in the Content Provider section for the enumerated values. “start” - The instance number of the object. “units” - The units string for the object. (Note that this is only included for BACnet.)
	Discovery Progress	11	Type: arg1 (int)	This message is sent from the service to update the progress of the discovery operation. The current progress, in percent, is passed in the arg1 field.
	Discovery Complete	12		This message is sent from the service to indicate that the discovery operation has completed successfully.
	Discovery Failed	13	Type: Bundle Key: “ErrorCode” Value: String	This message is sent from the service to indicate that the discovery operation has failed. The error code message is included in the Bundle type in the data of the message.
Modbus/TCP	Set Port	100	Type: arg1 (int)	Sets the TCP/IP port for the service to use from the value passed in the arg1 field of the message.
BACnet/IP	Set Port	100	Type: arg1 (int)	Sets the UDP port for the service to use from the value passed in the arg1 field of the message.
	Set Device Name	200	Type: Bundle Key: “DevName” Value: String	Sets the device name for the service to use from the string value passed in a Bundle type.
	Set Device Instance	201	Type: arg1 (int)	Sets the device instance for the service to use from the value passed in the arg1 field of the message.

5.1 Protocol Service Messaging Code Examples

This section contains a full code example of how to send and receive various messages to and from a protocol service.

5.1.1 Sending and Receiving Messages

```
/** Register to receive messages from the service */
public static final int MSG_REGISTER = 0;

/** Get Display Name message code */
public static final int MSG_GET_DISPLAY_NAME = 1;

/** Set Scan Rate message code */
public static final int MSG_SET_SCAN_RATE = 2;

/** Set Timeout message code */
public static final int MSG_SET_TIMEOUT = 3;

/** Reset service message code */
public static final int MSG_RESET_SERVICE = 4;

/** Invalid License message code */
public static final int MSG_INVALID_LICENSE = 5;

/** Set Port message code */
public static final int MSG_PORT = 100;

/** Set Device Name message code */
public static final int MSG_DEV_NAME = 200;

/** Set Device Instance message code */
public static final int MSG_DEV_INST = 201;

/** Messenger for communicating with the service. */
private Messenger mService = null;

/** Flag indicating whether we have called bind on the service. */
private boolean mBound = false;

/** Target we publish for clients to send messages to IncomingHandler. */
private final Messenger mMessenger = new Messenger(new IncomingHandler(this));

/** Class for interacting with the main interface of the service. */
private ServiceConnection mConnection = new ServiceConnection() {
      public void onServiceConnected(ComponentName className, IBinder service) {
            Message msg;
            Bundle bundle;

            // This is called when the connection with the service has been
            // established, giving us the object we can use to
            // interact with the service. We are communicating with the
            // service using a Messenger, so here we get a client-side
            // representation of that from the raw IBinder object.
            mService = new Messenger(service);
            mBound = true;
```

```
        // Register with the service to receive messages,
        // we don't need this if asking for display name
        msg = Message.obtain(null, MSG_REGISTER);
        try {
               msg.replyTo = mMessenger;
               mService.send(msg);
        } catch (RemoteException e) {
               Log.e("MyActivity", "Error sending message", e);
        }

        // Ask for the display name and register our messenger
        msg = Message.obtain(null, MSG_GET_DISPLAY_NAME);
        try {
               msg.replyTo = mMessenger;
               mService.send(msg);
        } catch (RemoteException e) {
               Log.e("MyActivity", "Error sending message", e);
        }

        // Set the Scan Rate to 5s
        msg = Message.obtain(null, MSG_SET_SCAN_RATE, 5000, 0);
        try {
               mService.send(msg);
        } catch (RemoteException e) {
               Log.e("MyActivity", "Error sending message", e);
        }

        // Set the Timeout to 250ms
        msg = Message.obtain(null, MSG_SET_TIMEOUT, 250, 0);
        try {
               mService.send(msg);
        } catch (RemoteException e) {
               Log.e("MyActivity", "Error sending message", e);
        }

        // Set the BACnet Device Name
        msg = Message.obtain(null, MSG_DEV_NAME);
        bundle = new Bundle();
        bundle.putString("DevName", "HCRT");
        msg.setData(bundle);
        try {
               mService.send(msg);
        } catch (RemoteException e) {
               Log.e("MyActivity", "Error sending message", e);
        }

        // Restart the service for changes to take effect
        msg = Message.obtain(null, MSG_RESET_SERVICE);
        try {
               mService.send(msg);
        } catch (RemoteException e) {
               Log.e(TAG, "Error sending message", e);
        }
}
```

```
	public void onServiceDisconnected(ComponentName className) {
		// This is called when the connection with the service has been
		// unexpectedly disconnected -- that is, its process crashed.
		mService = null;
		mBound = false;
	}
};

/**
 * Handler of incoming messages from service.
 */
private static class IncomingHandler extends Handler {

	/** Provides a weak reference to the outer class to avoid memory leaks */
	private final WeakReference<ObjectListActivity> mOuterClass;

	/**
	 * Creates a new IncomingHandler passing a reference to the outer class
	 * @param outerClass The outer class to reference
	 */
	public IncomingHandler(ObjectListActivity outerClass) {
		mOuterClass = new WeakReference<ObjectListActivity>(outerClass);
	}

	@Override
	public void handleMessage(Message msg) {
		Bundle bundle;
		String displayName;
		String errorCode;

		if (mOuterClass.get() != null) {
			switch (msg.what) {
			case MSG_GET_DISPLAY_NAME:
				bundle = msg.getData();

				if (bundle != null && bundle.containsKey("DisplayName") &&
						mOuterClass.get().getActivity() != null) {
					displayName = bundle.getString("DisplayName");
					// Do something with the display name here
				}
				break;
			case MSG_INVALID_LICENSE:
				bundle = msg.getData();

				if (bundle != null && bundle.containsKey("ErrorCode") &&
						mOuterClass.get().getActivity() != null) {
					errorCode = bundle.getString("ErrorCode");
					// Do something with the error code here
				}
				break;
			default:
				super.handleMessage(msg);
			}
		}
	}
}
```

ICC
INDUSTRIAL CONTROL COMMUNICATIONS, INC.

1600 Aspen Commons, Suite 210
Middleton, WI USA 53562-4720
Tel: [608] 831-1255 Fax: [608] 831-2045

http://www.iccdesigns.com

Printed in U.S.A