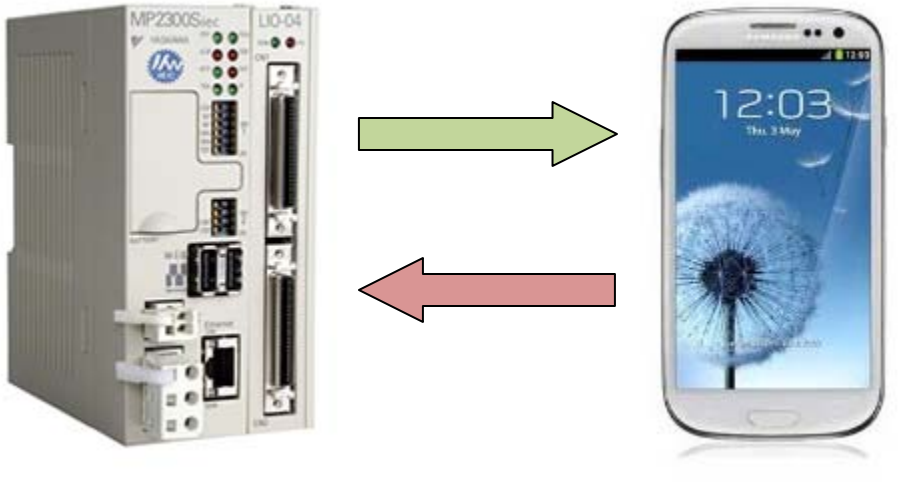
Application Note

Interfacing an MPiec Series Controller with an Android Application

Applicable Product: MPiec Series Controllers



Yaskawa Electric America
2121 Norman Drive South
Waukegan, IL 60085
1-800-927-5292

Subject: Application Note	Product: MPiec	Doc#: AN.MPIEC.04
Title: Interfacing an MPiec Series Controller with an Android Application		

Application Overview

This document describes the process of developing an Android application to interface with an MPiec Series Controller. It covers the following three topics:

1) Setting up the Android development environment (on Windows)
2) Creating a MotionWorks IEC program to accept communication via the YDeviceComm library
3) Establishing and testing communication and developing a basic program.

It also discusses common pitfalls and best practices.

Both the IEC and Android application code is available on **www.yaskawa.com** bu searching for document number **EC.MPIEC.02**.

Products Used:

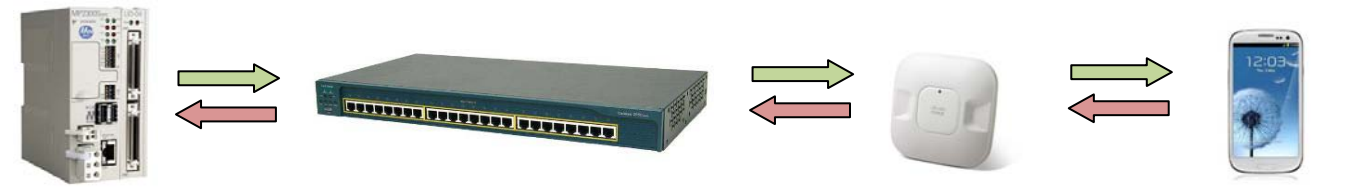
Component	Product and Model Number
Controller	MPiec (Any)
Software	MotionWorks IEC Express or Pro
Third Party Devices	Android Device

Implementation Method of Core Operation

The development of Android Application for communication to an MPiec Series Controller will be discussed in two major parts:

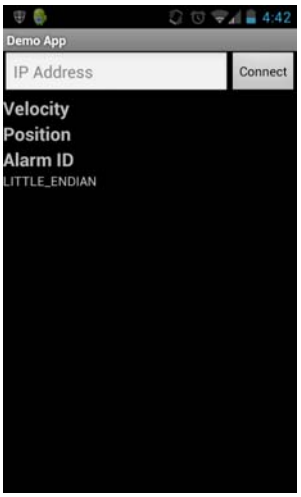
1. Developing the Android Application
2. Developing the MPiec Application

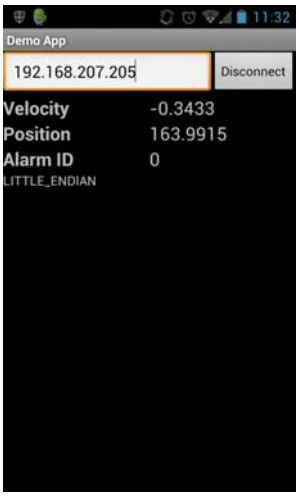
These two parts work together using TCP communications through a network, the most basic network configuration being represented by this diagram:



Of course this is a fairly simplified network diagram but the idea is correct. One could also connect to a controller remotely but that would involve configuring VPN access on the Android phone.

The purpose of the demo application in this Application Note is to display the Actual Position, Actual Velocity and Axis Alarm ID from a single axis in a Yaskawa MP2300Siec Demo Unit. The final Android application will look like this:





Programming: IEC Application

The core of the IEC application depends on the YDeviceComm firmware library that **must be included** in your project. The YDeviceComm firmware library includes a number of function blocks that provide TCP, UDP and serial communications. For this Application we will be using the TCP function blocks to:

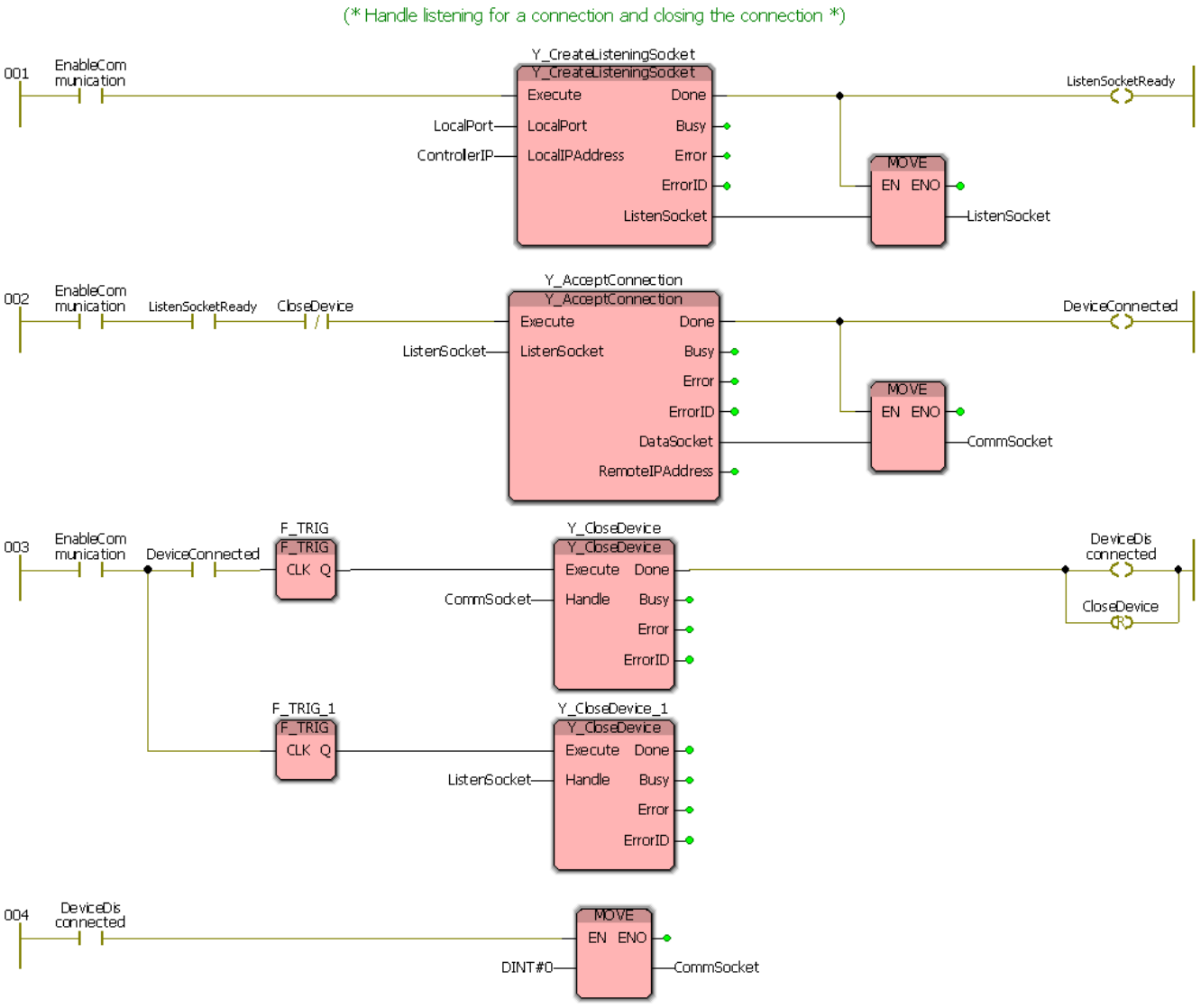
- Create a listening socket *(Y_CreateListeningSocket)*
- Accept a connection from the listening socket *(Y_AcceptConnection)*
- Read from the new data socket *(Y_ReadDevice)*
- Write to the new data socket *(Y_WriteDevice)*
- Close the new data socket *(Y_CloseDevice)*

Making sure the data socket closes is important. If a socket is opened but not closed, then after a number of lost connections the controller will not accept any more connections and the sockets will either have to be closed manually in the program (not covered in this application note) or the controller must be stopped and warm started.

The IEC application consists of 5 POUs but most of the application resides in the **Communication** POU. The other POUs and their functions are:

Initialize	Configure communication port, controller IP and *AxisRef.AxisNum*
IO	Read actual Position/Velocity and Axis Status
HMI	A basic “HMI” that uses contacts to enable communication, servo power and servo movement
Main	Contains axis /motion related function blocks

Here is the basic layout for this process used in the example application:

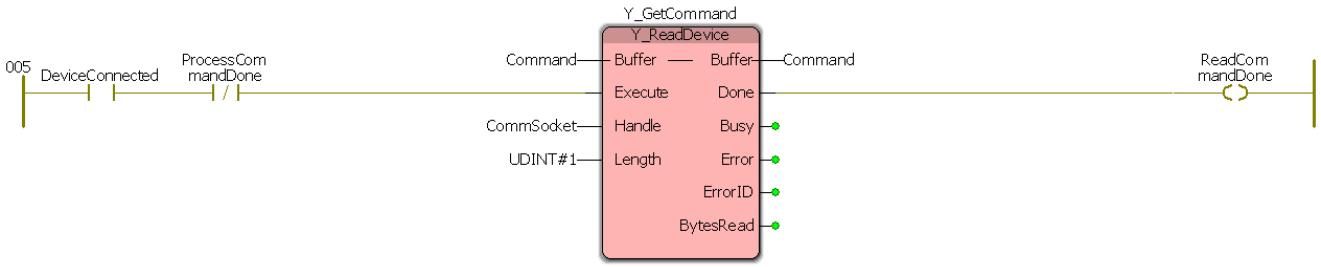


This methodology allows a single connection to be opened to the controller from a requester over port 23. It is important to note that the port number is independent of the application – you can choose the port number your controller and Android device communicate over, however it is suggested to avoid common port numbers associated with services like FTP, SMTP, etc. The above IEC code also handles a commanded disconnect that

ensures the current socket is closed properly. The remainder of the POU focuses on implementing:

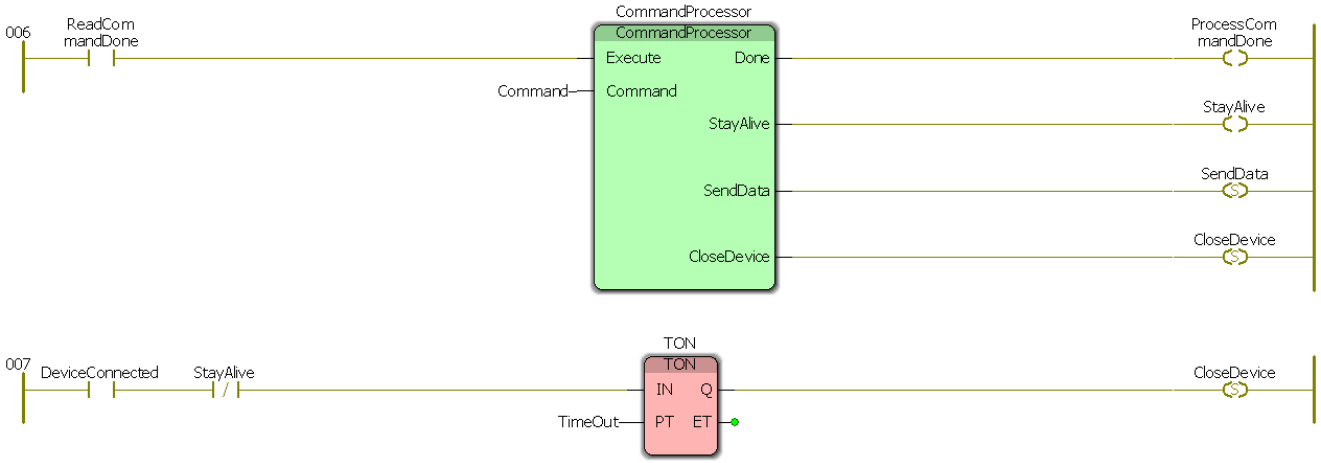
- Read a command from the connected device
- Process a received command
- Send a data packet to the connected device

Reading from the Android device is as simple as using Y_ReadDevice, setting the *Buffer* to a BYTE variable and setting *Length* to UDINT#1. Assuming there is data to collect, each scan this block will copy one byte from the input stream to the *Command* variable. Each time this occurs, *ReadCommandDone* becomes true.



Now that the controller has received a command, it can process it. The suggested method of performing this is to write a custom function block (in Structure Text) that takes the input command and triggers events using some form of output. Because the number of commands in this application is so limited, the command processor block has a BOOL output dedicated to each action. In more complicated input/action scenarios, it may be useful to utilize a VAR_IN_OUT STRUCT that contains variables associated with actions. You should **not** use global variables inside of your custom function block.

Another important concept used with this block is the type of coil used on the SendData and CloseDevice outputs – SET coils. These coils are set TRUE by the **CommandProcessor** but will not go FALSE when the block turns off. These coils must be reset by a RESET coil somewhere else in the POU.



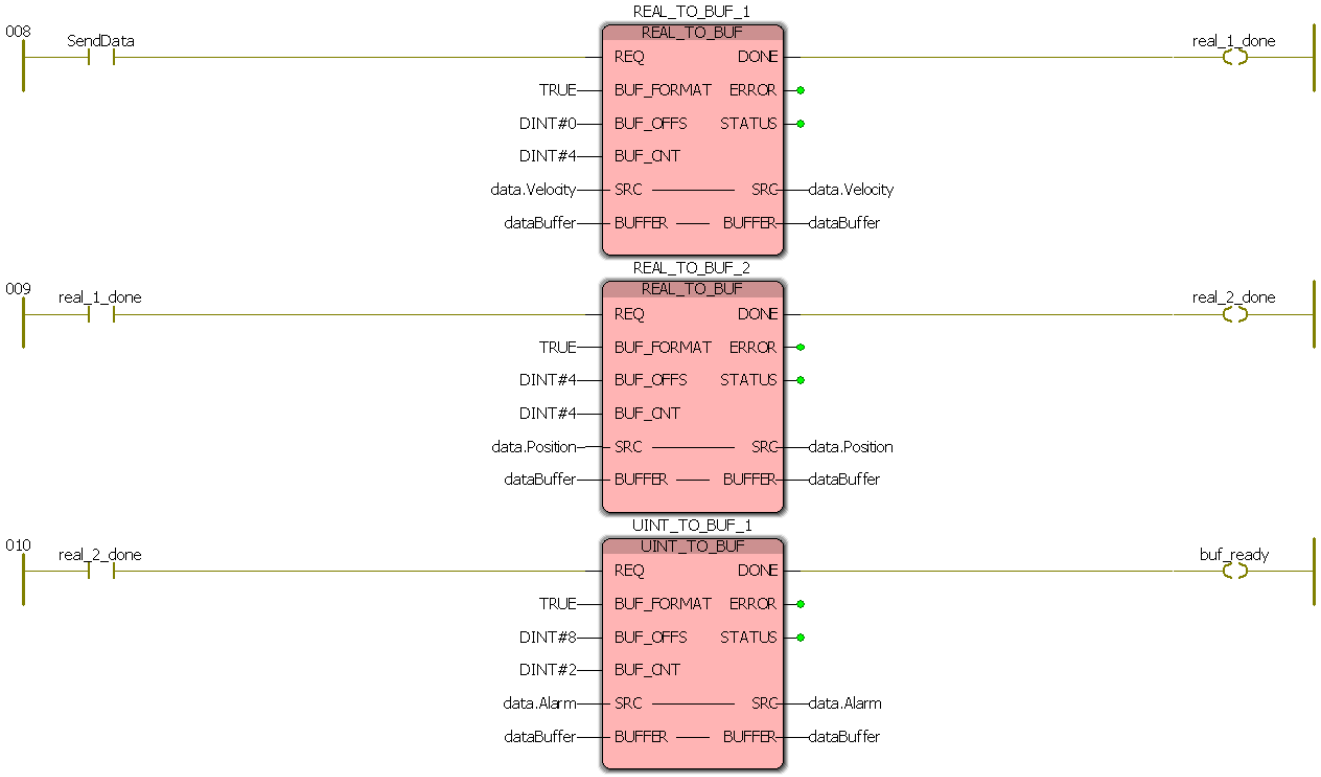
Right below the **CommandProcessor** you can see the controller side stay-alive functionality. This **TimerON** function block counts up to the specified *TimeOut* unless it is reset by *StayAlive* becoming TRUE. If *StayAlive* is not set TRUE within the *TimeOut* then *CloseDevice* is set TRUE and the controller disconnects the data socket and starts listening for a new connection. It is worth noting that the stay-alive functionality is **optional** to the functionality of this application; however, it is a suggested "best practice" that helps ensure connections are closed properly (for example, the Android application could crash or disconnect before issuing a Disconnect command and the stay-alive functionality would ensure the MPiec controller closes that connection).

Once the Android device sends a command, it will expect a packet in response. Before this packet can be sent by the controller, it must be prepared in some manner. This Application Note suggests putting all output values in a structure and then packing that structure in to an array of BYTEs. This is performed by using the **PROCONOS** firmware library function blocks ***_TO_BUF**. As mentioned in the Endian section below, when communicating with a device using Java (or any device in which the target Endianness is not known) the ***_TO_BUF** *BUF_FORMAT* must be set TRUE to pack the bytes in Big Endian format (a.k.a. Network Byte Order).

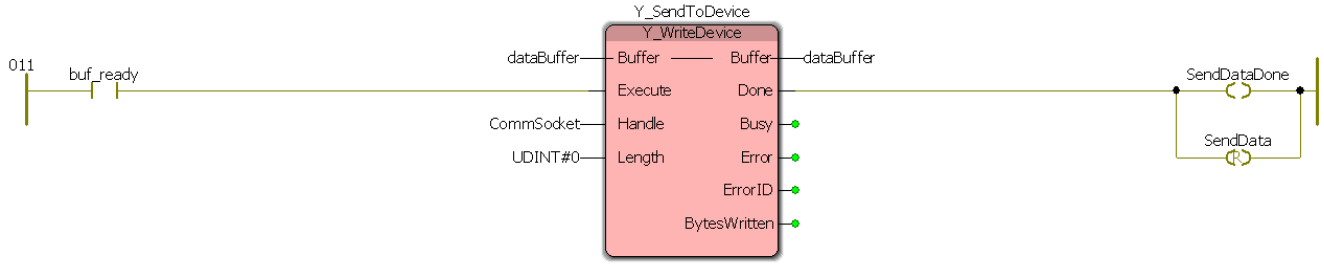
For this application, three output values are sent:

- Actual Position *(REAL)*
- Actual Velocity *(REAL)*
- Alarm ID *(UINT)*

Thus, two **REAL_TO_BUF** blocks and one **UINT_TO_BUF** block is required. Once all three values have been packed in to the *dataBuffer*, *buf_ready* is set TRUE and the packet can be sent.



Finally, the controller is ready to send the data packet to the Android device. This is done using the **Y_WriteDevice** function block. Set the *Buffer* to *dataBuffer* and *Length* to UDINT#0 which will cause the block to send the entire length of the *dataBuffer*.



This same sort of communication methodology can be expanded and enhanced to receive and send significantly more complicated packet structures allowing large amounts of control and data gathering with a fairly simple set of function blocks and a custom command processing function block.

Testing IEC Application

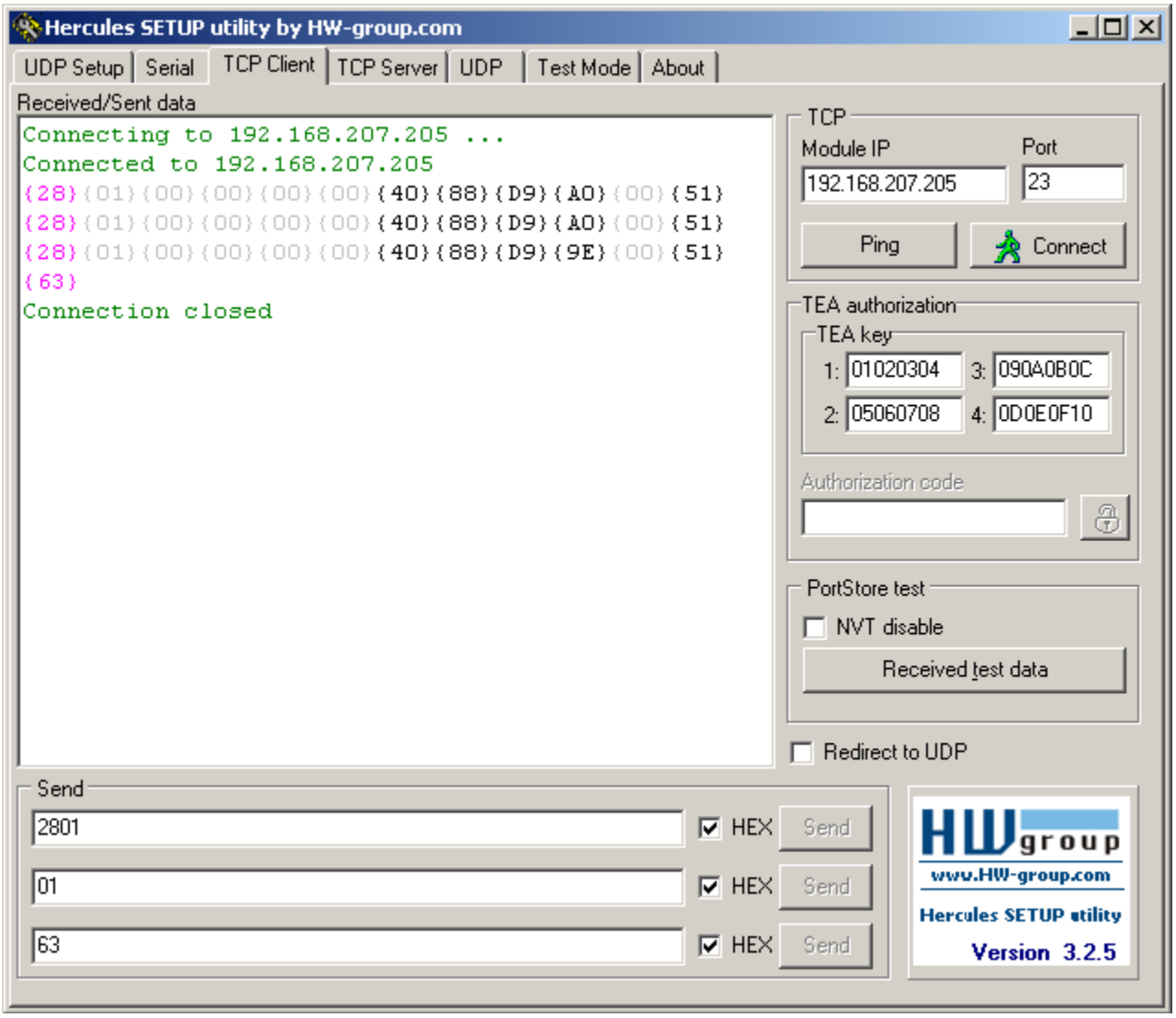
Before continuing on to write the Android application it is a good idea to test the IEC app functionality. This can be done by using any program capable of acting as a TCP client. The example below was performed using Hercules, a HW Group program freely available:

http://www.hw-group.com/products/hercules/index_en.html

There are only three commands in this application, Stay Alive (0x01), Get Data (0x28) and Close Connection (0x63). To test your application:

1. Enter the IP address of a controller on which it is running (with port set to 23)
2. Make sure that communications are enabled in the HMI and that the servo is enabled
3. Click connect
4. Send 0x2801 (for Get Data, Stay Alive) a few times and then send 0x63 to close the connection

The resulting Hercules output should look like:



Breaking down one line of this communication you can see:

{28}{01}	{00}{00}{00}{00}	{40}{88}{D9}{A0}	{00}{51}
Get Data, Stay Alive	Velocity	Position	Drive Alarm

Checking the web server for our controller, there is an A.51 alarm (axis overspeed) which matches the output from the controller.

Configuring the Android Development Environment

There are two main methods of developing Android Applications in a Windows environment: command line and Integrated Development Environment (IDE). The IDE method is an all-in-one solution that allows you to write, build and test code in a single application while the command line method involves writing the code in one application and building the code using command line tools. This Application Note will demonstrate the command line methodology; however, using the Eclipse IDE may be easier to those familiar developing with an IDE or with little to no experience in command line development.

Configuring the command line: **http://developer.android.com/tools/projects/projects-cmdline.html**
Configuring Eclipse IDE: **http://developer.android.com/sdk/index.html#ExistingIDE**
Using Eclipse: **http://coding.smashingmagazine.com/2011/11/04/getting-the-best-out-of-eclipse-for-android-development/**

Both the IEC and Android application code is available on **www.yaskawa.com** by searching for document number **EC.MPIEC.02**.

To start, the developer needs to acquire:

- Java JDK
- Android SDK
- Apache Ant
- At least 1 Android Target
- All of the above configured to run on his/her system

Configuring the command line development environment is outside of the scope of this Application Note but there a number of tutorials online describing how to do so for your system configuration.

Once the environment is configured the developer can move on to creating his/her first project. From here most steps will be either performed in the Windows DOS prompt (“command line”) or in a text editor (“editor” – notepad, wordpad, Notepad++, etc.). The first step is to create the project using this command:

```
android create project --target 1 --name DemoApp --path DemoApp --activity demo_app
--package com.yaskawa.demoapp
```

where target is the ID associated with your installed Android target. To get a list of targets you can type the command:

```
android list targets
```

which should output a list or targets, a sample of which looks like:

```
Available Android targets:
----------
id: 1 or "android-16"
     Name: Android 4.1
     Type: Platform
     API level: 16
     Revision: 2
     Skins: HVGA, QVGA, WQVGA400, WQVGA432, WSVGA, WVGA800 (default), WVGA854, WXGA720,
WXGA800, WXGA800-7in
     ABIs : armeabi-v7a
```

The **name** is the name of your application as it will be viewed on your device, the path is the location in your file system where the project will be stored (in this example, the path is a folder named “DemoApp” which means that prior to typing this command the `cd` command was used to change directory to the directory containing “DemoApp”), the activity is the name of the main activity that will run in your application and the package is a namespace for your application within the Java environment (so you can change “yaskawa” to “yourcompany” or whatever you like).

Now that the project is created, there are a number of new files in the directory you specified which must be modified to created the demo application for this Application Note. The first file that needs to be modified is “AndroidManifest.xml” which is in the top level of the application folder. Only one line must be added:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

Place it below the `<application>` close tag. This tag allows your app to utilize the internet capabilities of the device which is necessary to connect to an MPiec controller over Wi-Fi.

Programming: Android Application

To load the application on to your device at any point, follow these steps:

1. Make sure that you changed your working directory in the command prompt to the “DemoApp” folder (or whatever you named it) or else the following command will fail:

   ```
   ant debug
   ```

2. Once that command completes and says `BUILD COMPLETE` your application is ready to load on to your phone. If you will be loading applications using USB, make sure that your phone has “USB Debugging” enabled and is plugged in to your computer. If this is the only Android device attached to your computer, then you are ready to issue this command:

   ```
   adb install -r bin/demoapp-debug.apk
   ```

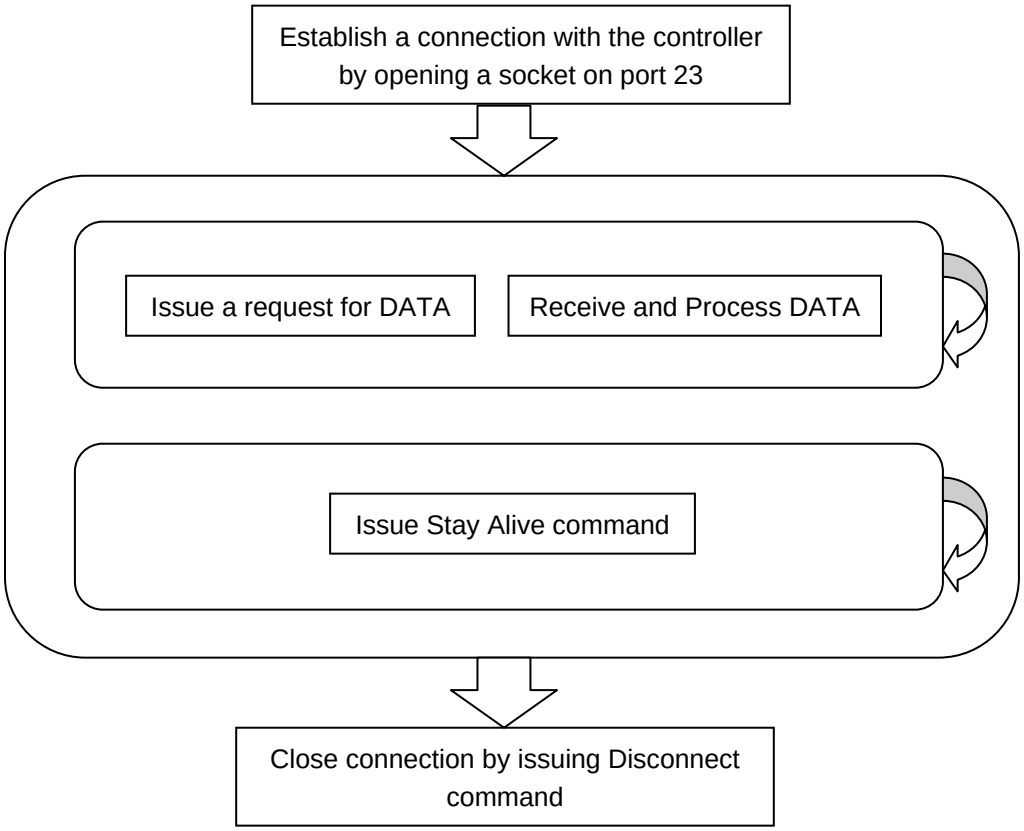
3. If you named the application something different than “DemoApp” you can see what file you need to load by changing to the “bin/” directory and issuing the `dir` command looking for the file named similar to the one above (i.e. “[application_name]-debug.apk”).

There should now be an application on your phone named whatever you used as the application name and when you click on its icon an app should open with the default application layout.

The actual Java code will not be discussed in depth in this Application Note as it is outside of the scope. However, the methodology employed in the app will be discussed below.

The Android Application being developed only has one main function which is to collect data from the controller and display it to the user. In order to this, there are three auxiliary functions which must exist for the main function to occur. Two of these functions are related – Connect() and Disconnect(). The third function serves as an assistant to the first function which is KeepAlive(). While the Android Application is connected to the controller, it will issue a KeepAlive() command at a set interval so that the controller knows the Android Application is still connected – otherwise the controller closes the connection (this prevents connections from being left open if something goes wrong on the Android side and it is unable to issue a Disconnect()). As mentioned in the IEC programming section, the KeepAlive() function is **optional** and not necessary for a functioning application. The main function, GetData(), also operates at a fixed interval in which it sends a request for a data packet and processes the response and displays it to the user.

A diagram of this methodology can be seen here:



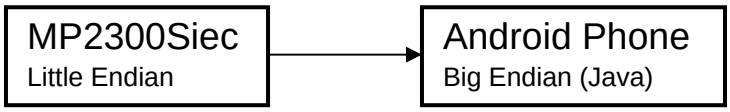
With this simple methodology, much more complex applications can be written simply by expanding the amount and type of the data received and performing additional processing to deliver that content to the user. In addition, the function used to request DATA can also be used to issue a different command (either cyclically or event based) which could then either expect a different response or cause an event on the controller (such as clearing alarms, enabling/jogging servos, restarting the machine, etc.).

The Importance of Endianness

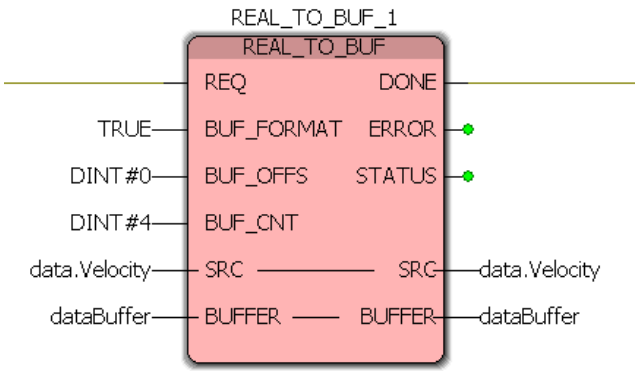
Endianness describes the byte order of multi-byte datatypes (e.g. ints, reals, etc.). There are two types of Endianness that we are concerned about: Little Endian and Big Endian. Big Endian means that the most significant byte is the first byte whereas Little Endian means the least significant byte is the first byte. This is easiest to see with an example. Say you want to store the integer value 1000. In hexadecimal this is 0x03E8. Here is a comparison of how this would be stored in memory of the two different types of Endianness:

Integer Value	0x03E8
Little Endian	0xE8 0x03
Big Endian	0x03 0xE8

So, it's pretty clear what the difference is between the two types. Now, where does this come in to play in IEC to Android communications? The answer is that MP2300iec controllers are Little Endian while Android Applications are Big Endian (note: Android Applications are Big Endian because Java, a Sun product, always assumes data is Big Endian – the actual processor may be Little or Big Endian).



This is very important to know because if you don't account for Endianness in your application then your data may end up backwards. Thus, when you send data from the MPiec controller to the Android Application the byte order (endianness) of the data must be changed. How do you do that? You can specify output byte order by setting the input parameter *BUF_FORMAT* of ***_TO_BUF** to **TRUE**.



Setting *BUF_FORMAT* to **TRUE** will ensure that the Android App properly interprets the input

So how does this relate to the Android Application? This concept becomes very important when receiving the information from the Controller. If the Controller's output is configured using the methodology employed above then parsing the packet sent by the Controller should be as easy as:

```
velocity = input_stream.readFloat();
position = input_stream.readFloat();
alarm = input_stream.readShort();
```

The `DataInputStream` and `InputStreamReader` classes in Java expect (like all of Java) Big Endian format which means that the functions above will read, from the InputStream, the number of bytes specified by the data type, Big Endian format. For reference, here are the Java primitive data types and their respective sizes:

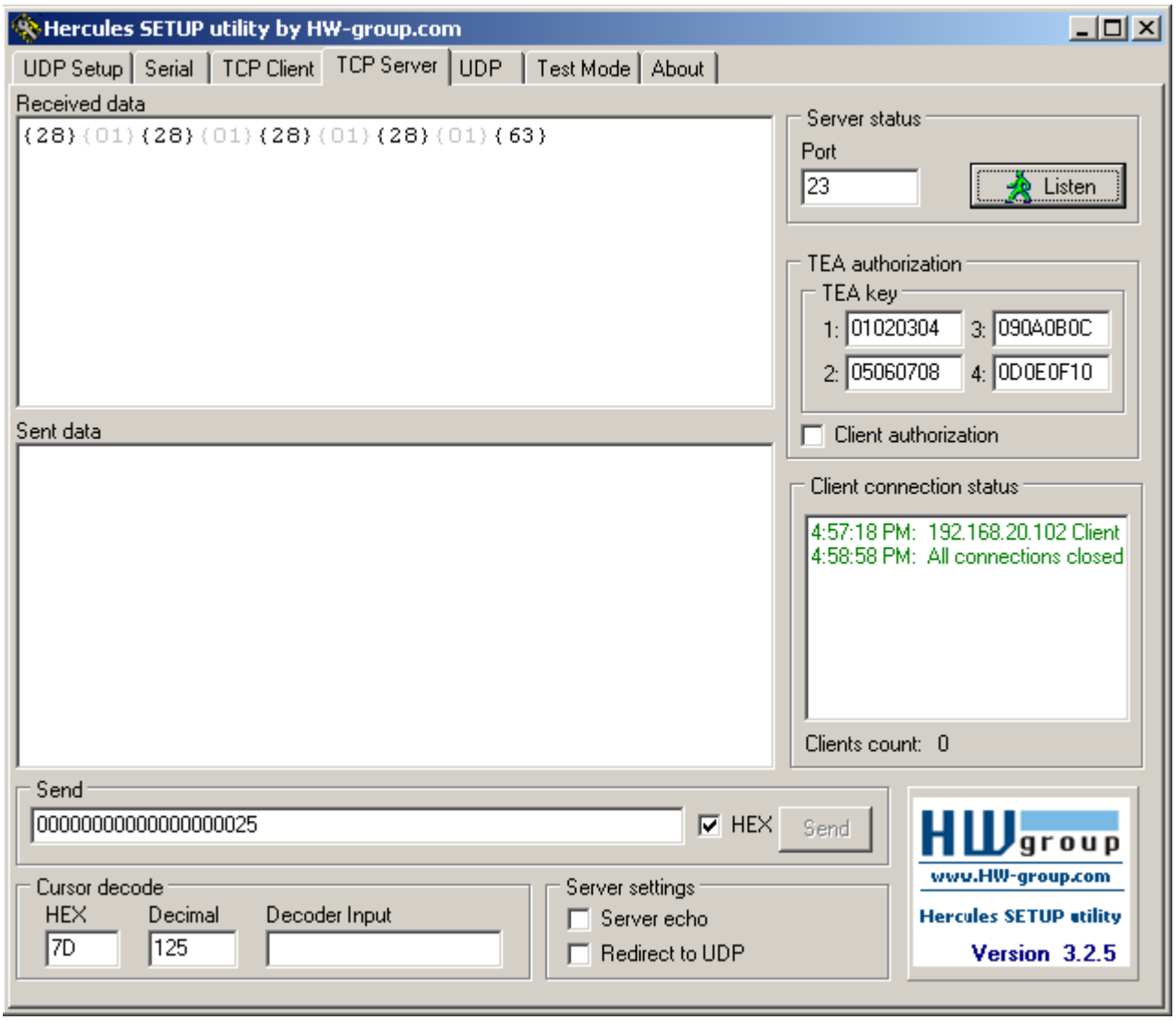
Type	Size
byte	1
short	2
int	4
long	8
float	4
double	8
char	2

Further determination of what Java data type is correct for your input data involves determining sign and if there is any padding in your packet.

Testing the Android Application

Testing the Android application will involve using Hercules TCP server functionality. To perform this test you will need your computer's IP address and connect your Android phone on the network your as the computer. If you don't know your computer's IP address, go to the command line and issue `ipconfig` and look for the IP address associated with whichever method you are connected to your network through (Ethernet or Wireless). Enter this IP address in to the Android app, click "Listen" in Hercules on the TCP Server page and click "connect" in the Android app.

The result in Hercules should look something like this:



If you want to test the receiving portion of the Android application you can use the “Send” box pictured (make sure HEX is checked) and send 10 byes (therefore 20 individual hexadecimal digits). Unless you know how 32-bit floating point numbers are represented in hexadecimal you will either need to use a decimal to hexadecimal online conversion application or just enter all 0’s for the two REAL values. You can, however, use

the last 2 bytes to test different values (since the result is predictable as the input is of type UINT).

Final Testing

The last thing to test is actually connecting the Android application to the MPiec controller. If testing was successful for the individual components then testing should be just as successful. Simply enter the controller IP on the android application and click “connect” and the data fields in the application should be filled (and fluctuating if the servo is moving). If the values appear to be reversed (or the REALs are gigantic numbers) then you may have an endian issue and will want to review that section.