

Quick Start Guide:

Kubota USB-Link™ 3 Mobile RP1210 Drivers for Android and iOS

Document Overview

This document describes the basic setup and operation for using the Kubota USB-Link™ 3 Mobile RP1210 API on Android and iOS. The document is divided into the following sections:

Android Overview

- Step 1: Android Load Library from Java
- Step 2: Android RP1210 API Setup
- Step 3: Copy INI Files to Data Path
- Step 4: Set Data Path in RP1210 Drivers IOCTL 0x102
- Step 5: Android Discovery IOCTL 0x100
- Step 6: Bluetooth Connect on Android
- Step 7: Wi-Fi Connect on Android

iOS Overview

- Step 1: iOS RP1210 API Setup
- Step 2: iOS Discovery IOCTL 0x100
- Step 3: Bluetooth Connect on iOS
- Step 4: Wi-Fi Connect on iOS

RP1210 Logging Overview

Android Overview

Android requires the Java application to load the library, `kuln3r32.so`, from Java. Then uncompress and copy the INI files from the package to a data directory accessible by the RP1210 drivers. After that find Bluetooth or Wi-Fi devices to connect with and make a connection.

NOTE: The Android drivers require that the `libc++_shared.so` library that is included in the SDK be installed on the device in the same folder as the `kuln3r32.so` library.

In practice, the RP1210 program accessing RP1210 should not run on the main thread (GUI thread). Otherwise the responsiveness of the application will suffer and Android may decide to kill the application.

Step 1: Android Load Library from Java

The following call must be made from Java:

```
System.loadLibrary("kuln3r32");
```

so that the `Java_OnLoad()` function is called and the Java virtual machine pointer is copied in the `kuln3r32.so` RP1210 library: Otherwise, Bluetooth® will not function.

Step 2: Android RP1210 API Setup

The RP1210 API is resolved from the dynamic library using the following code sample. The API functions are typedefed in `rp1210_base.h` and in `Rp1210Test.cpp`.

Two additional API calls, `NexiqSetBluetoothAddress` and `NexiqSetDataPath`, are available to set the Bluetooth MAC address and to set the uncompressed INI files location respectively. Alternatively, `SELECT_DEVICE_IOCTL 0x103` and `SET_DATA_PATH_IOCTL 0x102` can be used.

```
RP1210ConnectProc RP1210_ClientConnect = 0;
RP1210DisconnectProc RP1210_ClientDisconnect = 0;
RP1210ReadMessageProc RP1210_ReadMessage = 0;
RP1210SendMessageProc RP1210_SendMessage = 0;
RP1210SendCommandProc RP1210_SendCommand = 0;
RP1210ReadDetailedVersionProc RP1210_ReadDetailedVersion = 0;
RP1210ReadSerialNumberProc RP1210_ReadSerialNumber = 0;
RP1210GetErrorMsgProc RP1210_GetErrorMsg = 0;
RP1210GetLastErrorMsgProc RP1210_GetLastErrorMsg = 0;
RP1210GetHardwareStatusProc RP1210_GetHardwareStatus = 0;
RP1210IoctlProc RP1210_Ioctl = 0;
RP1210ReadVersionProc RP1210_ReadVersion = 0;

void kuln3r32_SetupAPI(void)
{
    void * g_dll_handle = dlopen("libnuln2r32.so", RTLD_GLOBAL|RTLD_NOW);

    RP1210_ClientConnect = (RP1210ConnectProc) dlsym(g_dll_handle, "RP1210_ClientConnect");
    RP1210_ClientDisconnect = (RP1210DisconnectProc) dlsym(g_dll_handle, "RP1210_ClientDisconnect");
    RP1210_ReadMessage = (RP1210ReadMessageProc) dlsym(g_dll_handle, "RP1210_ReadMessage");
    RP1210_SendMessage = (RP1210SendMessageProc) dlsym(g_dll_handle, "RP1210_SendMessage");
    RP1210_SendCommand = (RP1210SendCommandProc) dlsym(g_dll_handle, "RP1210_SendCommand");
    RP1210_ReadDetailedVersion = (RP1210ReadDetailedVersionProc)
        dlsym(g_dll_handle, "RP1210_ReadDetailedVersion");
    RP1210_ReadSerialNumber = (RP1210ReadSerialNumberProc)
        dlsym(g_dll_handle, "RP1210_ReadSerialNumber");
    RP1210_GetErrorMsg = (RP1210GetErrorMsgProc) dlsym(g_dll_handle, "RP1210_GetErrorMsg");
    RP1210_GetLastErrorMsg = (RP1210GetLastErrorMsgProc)
        dlsym(g_dll_handle, "RP1210_GetLastErrorMsg");
```

```
RP1210_GetHardwareStatus = (RP1210GetHardwareStatusProc)
    dlsym(g_dll_handle, "RP1210_GetHardwareStatus");
RP1210_Ioct1 = (RP1210IoctlProc) dlsym(g_dll_handle, "RP1210_Ioct1");
RP1210_ReadVersion = (RP1210ReadVersionProc) dlsym(g_dll_handle, "RP1210_ReadVersion");
}
```

Step 3: Copy INI Files to Data Path

The files `usb13map.ini`, `kuln3r32_internal.ini`, and `kuln3r32.ini` should be placed in the `assets/Files` folder of the Android project. Then the files need to be uncompressed and copied to the data path so that the native C++ libraries can access them with standard file io calls.

Use the following code from a fragment that has access to the Context via `getActivity()`:

```
Context ctx = getActivity();
try {
    String[] files = ctx.getAssets().list("Files");

    for(int i = 0; i < files.length; i++)
    {
        CopyFileToDataPath(files[i]);
    }
} catch (IOException e) {
    e.printStackTrace();
}

private void CopyFileToDataPath(String fname)
{
    // copy compressed file from apk assets folder to private
    // data folder for read/write in C/C++
    InputStream input;
    try {
        String path_fname = "Files/" + fname;
        input = ctx.getAssets().open(path_fname);

        int size = input.available();
        byte[] buffer = new byte[size];
        input.read(buffer);
        input.close();

        FileOutputStream outputStream;
        try{
            outputStream = ctx.openFileOutput(fname, Context.MODE_PRIVATE);
            outputStream.write(buffer);
            outputStream.close();
        }
        catch (Exception e){
            e.printStackTrace();
        }
    } catch (IOException e)
    {
        e.printStackTrace();
    }
}
```

Step 4: Set Data Path in RP1210 Drivers IOCTL 0x102

From a Java Context get the data path as:

```
Context ctx = getActivity();
String data_path_str = ctx.getFilesDir().getPath()
```

Pass this Java string to native code via a native call with parameter `jstring str_path`. Then convert this string to a UTF8 string, fill in the `SDATA_PATH` struct and then call `RP1210_ioctl()`

```
typedef struct _SDATA_PATH
{
    unsigned char *DataPath;
} SDATA_PATH;
```

...

```

SDATA_PATH s_data_path;

const char* temp_path = pEnv->GetStringUTFChars(str_path, NULL);
strcpy(log_path,temp_path);

strcpy((char*)data_path,temp_path);
s_data_path.DataPath = data_path;

RP1210_Ioctl(0,IOCTL_SET_DATA_PATH,&s_data_path,0);

```

After the RP1210 drivers have access to the uncompressed INI files and know their location, the application can make calls using the RP1210 API. Generally the RP1210_Ioctl with IOCTL_DEVICE_DISCOVERY will be used first. Afterwards, the RP1210_ClientConnect call will be made after the MAC address has been set for Bluetooth.

Step 5: Android Discovery IOCTL 0x100

```

#define PACKED __attribute__((packed, aligned (1)))

#define RP1210_DISCOVERY_MAX_ENTRY_LENGTH (256)

struct Rp1210Discovery
{
    unsigned int    scan_interval_sec;
    unsigned int    max_entries;
} PACKED;
typedef struct Rp1210Discovery RP1210_DISCOVERY;

struct RP1210DiscoveryResult
{
    unsigned int    num_entries_found;
    char            buf[0];          // allocated buffer start
} PACKED;
typedef struct RP1210DiscoveryResult RP1210_DISCOVERY_RESULT;

void TestDiscovery(void)
{
    RP1210_DISCOVERY discovery;
    discovery.scan_interval_sec = 5;      // 5 second Wi-Fi scan interval
    discovery.max_entries = 10;          // scan for up to 10 devices

    // allocate space for returned resulting structure
    RP1210_DISCOVERY_RESULT *discovery_result = (RP1210_DISCOVERY_RESULT *)
                                                malloc(sizeof(unsigned int) +
                                                RP1210_DISCOVERY_MAX_ENTRY_LENGTH * discovery.max_entries);

    ret_val = RP1210_Ioctl(0, IOCTL_DEVICE_DISCOVERY, &discovery, discovery_result);

    for (int k = 0; k < discovery_result->num_entries_found; k++)
    {
        char *p = discovery_result->buf;
        p += k * RP1210_DISCOVERY_MAX_ENTRY_LENGTH;
#ifdef ANDROID
        __android_log_print(ANDROID_LOG_INFO,"Example_App","%d. = %s",k, p);
#else
        printf("%d. %s\n", k, p);
#endif
    }
}

```

```

    }
}

```

Results for Android:

```

0. KUL3_1,1,0,00:07:80:1D:33:FD
1. KUL3_202,202,0,00:07:80:1D:22:45
2. KUL3_482,482,0,192.168.10.100
3. KUL3_59,59,1,192.168.10.12

```

Results are formatted as "*unique_id,status_byte,discovery_string*"

- *unique_id* = unique identifier for product.
- *serial_number* = manufacturer assigned serial number for the product.
- *status byte* = 0 if available or 1 if busy.
- *discovery_string* = parameter passed in to protocol connect string to connect to Kubota USB-Link™ 3.

Step 6: Bluetooth Connect on Android:

The Bluetooth results above will only be for previously paired devices. To pair with a device, go to the Android Bluetooth settings screen and select a discovered Kubota USB-Link™ 3 to pair to it.

```

short n_ret;
n_ret = RP1210_ClientConnect(0, 2, "J1708:DiscoveryString=00:07:80:1D:22:45", 32768, 32768, 0);

```

An alternative method for connecting to a Kubota USB-Link™ 3 using Bluetooth on Android is to specify the *DiscoveryString* in the `IOCTL_SELECT_DEVICE` `IOCTL 0x103` call before calling `RP1210_ClientConnect`.

```

typedef struct _SDEVICE_NAME
{
    unsigned char *DeviceName;
} SDEVICE_NAME;

short n_ret;
SDEVICE_NAME s_device_name;
char mac_address[18];
strcpy(mac_address, "00:07:80:1D:22:45");
s_device_name.DeviceName = (unsigned char*)mac_address;

n_ret = RP1210_Ioctl(0, IOCTL_SELECT_DEVICE, &s_device_name, NULL);
n_ret = RP1210_ClientConnect(0, 2, "J1708", 32768, 32768, 0);

```

Step 7: Wi-Fi Connect on Android:

Use the IP address in the *DiscoveryString* protocol connect parameter to connect to a Wi-Fi device on the currently connected network.

```

short n_ret;

```

```
n_ret = RP1210_ClientConnect(0, 3, "J1708:DiscoveryString=192.168.10.100", 32768, 32768, 0);
```

After this RP1210 API calls can be used as if the development environment is a Windows machine.

iOS Overview

Unlike Android, iOS does not compress the INI files; therefore, the drivers just need a copy of them. This is handled in the drivers as long as the INI files are located in the package as shown in the example application in the "Supporting Files" folder.

Libraries `libnexiq_rp1210.a` (iOS RP1210 library) and library `libc++.tbd` (c++ standard library) need to be linked with the application. `libnexiq_rp1210.a` is built against `libc++`.

The application accessing RP1210 library functions must not run on the main thread (GUI thread), otherwise there will be a GUI responsiveness penalty.

The iOS RP1210API is setup following the example code given later on. After that, Bluetooth and Wi-Fi devices to connect with can be discovered and a connection to a Kubota USB-Link™ 3 can be made. In the Bluetooth case, if no `DiscoveryString` is given, the first available connected Kubota USB-Link™ 3 will be the one connected to.

Step 1: iOS RP1210 API Setup

The RP1210 API is resolved from the statically linked library using the following code sample. The API function prototypes and typedefs are in `rp1210_base.hpp`.

```
RP1210ConnectProc RP1210_ClientConnect = 0;
RP1210DisconnectProc RP1210_ClientDisconnect = 0;
RP1210ReadMessageProc RP1210_ReadMessage = 0;
RP1210SendMessageProc RP1210_SendMessage = 0;
RP1210SendCommandProc RP1210_SendCommand = 0;
RP1210ReadDetailedVersionProc RP1210_ReadDetailedVersion = 0;
RP1210ReadSerialNumberProc RP1210_ReadSerialNumber = 0;
RP1210GetErrorMsgProc RP1210_GetErrorMsg = 0;
RP1210GetLastErrorMsgProc RP1210_GetLastErrorMsg = 0;
RP1210GetHardwareStatusProc RP1210_GetHardwareStatus = 0;
RP1210IoctlProc RP1210_Ioctl = 0;
RP1210ReadVersionProc RP1210_ReadVersion = 0;

void kuln3r32_SetupAPI(void)
{
    RP1210_ClientConnect = kuln3r32_RP1210_ClientConnect;
    RP1210_ClientDisconnect = kuln3r32_RP1210_ClientDisconnect;
    RP1210_ReadMessage = kuln3r32_RP1210_ReadMessage;
    RP1210_SendMessage = kuln3r32_RP1210_SendMessage;
    RP1210_SendCommand = kuln3r32_RP1210_SendCommand;
    RP1210_ReadDetailedVersion = kuln3r32_RP1210_ReadDetailedVersion;
    RP1210_ReadSerialNumber = kuln3r32_RP1210_ReadSerialNumber;
    RP1210_GetErrorMsg = kuln3r32_RP1210_GetErrorMsg;
    RP1210_GetLastErrorMsg = kuln3r32_RP1210_GetLastErrorMsg;
    RP1210_GetHardwareStatus = kuln3r32_RP1210_GetHardwareStatus;
    RP1210_Ioctl = kuln3r32_RP1210_Ioctl;
    RP1210_ReadVersion = kuln3r32_RP1210_ReadVersion;
}
```


Step 2: iOS Discovery IOCTL 0x100

Following the example in section "Android Discovery ioctl 0x100", the following will be returned on an iOS device.

Results for iOS:

```
0. KUL3_1,1,0,1
1. KUL3_202,202,0,202
2. KUL3_482,482,0,192.168.10.100
3. KUL3_59,59,1,192.168.10.12
```

The discovery string parameters can be either serial numbers or IP addresses. Therefore, the calling program can differentiate them by looking for a period('.') to determine an IP address.

Step 3: Bluetooth Connect on iOS:

The Bluetooth results above will only be of Kubota USB-Link™ 3 devices that are currently connected to the iOS device via the iOS Bluetooth Settings screen. This is also where pairing takes place for new Kubota USB-Link™ 3s that are unknown to the iOS device.

```
short n_ret;
n_ret = RP1210_ClientConnect(0, 2, "J1708:DiscoveryString=202", 32768, 32768, 0);
```

An alternative method for connecting to a Kubota USB-Link™ 3 using Bluetooth on iOS is to specify the DiscoveryString in the IOCTL_SELECT_DEVICE IOCTL 0x103 call before calling RP1210_ClientConnect.

```
typedef struct _SDEVICE_NAME
{
    unsigned char *DeviceName;
} SDEVICE_NAME;

short n_ret;
SDEVICE_NAME s_device_name;
char serial_num[4];
strcpy(serial_num, "202");
s_device_name.DeviceName = (unsigned char*)serial_num;

n_ret = RP1210_Ioctl(0, IOCTL_SELECT_DEVICE, &s_device_name, NULL);
n_ret = RP1210_ClientConnect(0, 2, "J1708", 32768, 32768, 0);
```

Another alternative method for connecting to an Kubota USB-Link™ 3 using Bluetooth on iOS is to not specify the DiscoveryString. This will connect to the first Kubota USB-Link™ 3 connected via Bluetooth in the iOS Bluetooth Settings screen.

```
short n_ret;
n_ret = RP1210_ClientConnect(0, 2, "J1708", 32768, 32768, 0);
```

Step 4: Wi-Fi Connect on iOS:

Use the IP address in the DiscoveryString protocol connect parameter to connect to a Wi-Fi device on the currently connected network.

```
short n_ret;  
n_ret = RP1210_ClientConnect(0, 3, "J1708:DiscoveryString=192.168.10.100", 32768, 32768, 0);
```

After this RP1210 API calls can be used as if the development environment is a Windows machine.

RP1210 Logging Overview

RP1210 logging can be enabled via the SET LOG LEVEL IOCTL 0x101.

```
typedef struct _SLOG_LEVEL
{
    unsigned long DebugLevel;
    unsigned char *DebugFilePath;
    unsigned long DebugFilePathLength;
    unsigned long DebugMode;
    unsigned long DebugFileSize;
} SLOG_LEVEL;

SLOG_LEVEL slog;

//set up slog
...

short ret_val = RP1210_Ioctl(0, IOCTL_SET_LOG_LEVEL, &slog, NULL);
```

Different logging options can be set up by configuring following the RP1227 specification.

On Android, the logging will be written to the LogCat debug window with the tag “RP1210_LOG.” On iOS the logging will be written to the debug window.

A log file named kuln3r32.log will be written to the DebugFilePath specified.

On Android, the file can be retrieved with Android Studio and Device File Explorer. The log file for the SDK example is stored in /data/data/**application_name**/files/, where **application_name** is com.nexiq.rp1210test for the Android Studio example and com.nexiq.rp1210example for the Xamarin example.

To retrieve the file from an iOS device with MacOS Catalina or later, connect the device to the Mac via USB and click on the device under Locations in Finder. Then select the disclosure triangle to the left of the iOS app name to show the kuln3r32.log file. Copy the file to the Mac to view it.