AN701 – Axiomatic Proprietary SSP Message Format

Message Transmission

The communications between a smart device such as a phone or a tablet and the CAN2BT device is based on Bluetooth Serial Port Profile (SPP). By default, the CAN2BT firmware declares itself as a SPP device (UUID: 00001101-0000-1000-8000-00805F9B34FB)

The messages are transferred in binary format, least significant byte first. The list of supported proprietary messages is shown below. Both versions of the CAN to Bluetooth device, AX141100 and AX141150 support this message format. However, not all messages described below will work in all devices, the biggest differences being in the Flash data handling and RTC commands.

Most of this material is also available in the user manuals of the AX141100 and AX141150 devices.

Overall message format

Byte 0	Byte 1	Byte 2	Byte 3	…	Byte n+2	Byte n+3	Byte n+4	Byte n+5	Byte n+6
<msg type>	<length>	<payload 0>	<payload 1>	…	<payload length-1>	<crc32>	<crc32>	<crc32>	<crc32>

In the above description the *<msg type>* is as listed in the table below. *<length>* is the full message length without the four CRC32 bytes. CRC32 is selected because the support for it is readily available in Android (the Axiomatic CAN2Bluetooth Configuration app uses **Android's built-in CRC32 function**).

All data that is expressed as Byte 0, Byte 1, … in the message descriptions below, is expected to be either 16 bits or 32 bits wide data, broken down to bytes (8 bits) least significant byte first. The only exception is the PIN code data, that is expected to be formatted one digit per byte. The PIN codes are hard formatted to have 6 digits.

There is an ack response sent by the CAN2BT device after receiving the configuration messages. The ack response contains status info: **0x00 – OK, 0x01 – Failed**.

Available CAN baudrate options include: 1 – 10k, 2 – 20k, 3 – 50k, 4 – 100k, 5 – 125k, 6 – 250k, 7 – 500k, 9 – 1M. In case CAN baudrate is configured as 0 (or 8), it is considered unconfigured and will default to 250k.

Message types

Message type	Direction	<msg type> byte
New connection notification	CAN2BT -> SD[*]	0x00
CAN data with StdID	SD[*] -> CAN2BT	0x12
CAN data with ExtID	SD[*] -> CAN2BT	0x13
CAN data with StdID	CAN2BT -> SD[*]	0x21
CAN data with ExtID	CAN2BT -> SD[*]	0x31
MAP write	SD[*] -> CAN2BT	0x34
MAP write response	CAN2BT -> SD[*]	0x43
MAP read	SD[*] -> CAN2BT	0x45
MAP read response	CAN2BT -> SD[*]	0x54
Download Flash contents	SD[*] -> CAN2BT	0x56
Download Flash contents response	CAN2BT -> SD[*]	0x65
Download Flash contents ACK	SD[*] -> CAN2BT	0x57
Download Flash contents ACK response	CAN2BT -> SD[*]	0x75
Change configuration	SD[*] -> CAN2BT	0x67
Change configuration response	CAN2BT -> SD[*]	0x76
LED state	SD[*] -> CAN2BT	0x78
LED state response	CAN2BT -> SD[*]	0x87

[*] SD = Smart Device, a phone or a tablet

New connection notification

Connected to: <configured node name> (default: CAN-BLE)						
<'C'>	<'A'>	<'N'>	<'-'>	<'B'>	<'L'>	<'E'>

CAN messages

CAN frame format, StdID (ID bit 11 = RTR flag, bits 10 to 0, StdID)						
<ID 0>	<ID 1>	<len>	<D0>	<D1>	…	<D len-1>

CAN frame format, ExtID (ID bit 31 = RTR flag, bits 28 to 0, ExtID)								
<ID 0>	<ID 1>	<ID 2>	<ID 3>	<len>	<D0>	<D1>	…	<D len-1>

MAP access

J1939 MAP access (data types: 1=u8, 2=u16, 4=u32, 5=f32)								
<Remote node addr >	<SP A0>	<SP A1>	<SP A2>	<data type>	<SP D0>	<SP D1>	<SP D2>	<SP D3>

Configuration messages

Configuration mode (d0 … d5 are single digits of the PIN code set using command 0x21)							
Enter config mode	**0xC0**	<d0>	<d1>	<d2>	<d3>	<d4>	<d5>
Exit config mode	**0xC1**						

CAN Rx Filters								
Add CAN Filter	**0x01**	<filter b0>	<filter b1>	<filter b2>	<filter b3>	<mask b0>	<mask b1>	<mask b2>	<mask b3>
Remove CAN Filter	**0x02**	<filter b0>	<filter b1>	<filter b2>	<filter b3>				
Enter promisc. mode	**0x03**								
Exit promisc. mode	**0x04**								

CAN bus configuration (br index = 1-10k, 2-20k, 3-50k, 4-100k, 5-125k, 6-250k, 7-500k, 9-1M, state = 0-off, 1-on)		
CAN bus baudrate	**0x05**	<br index>
Set BT CAN TX echo	**0x06**	<state>

Connection functions		
Scan available devices	**0x10**	
Connect to remote device	**0x11**	<scan index>
Disconnect from remote device	**0x12**	
Set autoconnect	**0x13**	<scan index>
Define accepted BD ADDR	**0x14**	<scan index, 0 = currently connected device>
Disable autoconnect	**0x15**	

PIN Codes (o0…o5 are single digits of the *OLD* PIN code and n0…n5 are single digits of the *NEW* PIN)

Set Pairing PIN Code	**0x20**	\<o0\>	\<o1\>	\<o2\>	\<o3\>	\<o4\>	\<o5\>	\<n0\>	\<n1\>	\<n2\>	\<n3\>	\<n4\>	\<n5\>
Set Config PIN Code	**0x21**	\<o0\>	\<o1\>	\<o2\>	\<o3\>	\<o4\>	\<o5\>	\<n0\>	\<n1\>	\<n2\>	\<n3\>	\<n4\>	\<n5\>
Set Rem.Acc. PIN Code	**0x22**	\<o0\>	\<o1\>	\<o2\>	\<o3\>	\<o4\>	\<o5\>	\<n0\>	\<n1\>	\<n2\>	\<n3\>	\<n4\>	\<n5\>

Data logging (when defining ID, bit 31 = ExtID, bit 30 = RTR, bits 29-0 define the ID)

Define PGN to log	**0x30**	\<PGN 0\>	\<PGN 1\>	\<PGN 2\>	\<PGN 3\>
Define SrcAddress to log	**0x31**	\<Src Address\>			
Define ID to log	**0x32**	\<ID 0\>	\<ID 1\>	\<ID 2\>	\<ID 3)
List current rules	**0x33**				
Erase all rules	**0x34**				

Flash functions

Erase all flash data	**0x40**				
Erase specific flash blocks	**0x41**	\<start blk b0\>	\<start blk b1\>	\<end blk b0\>	\<end blk b1\>
Set flash address	**0x42**	\<block b0\>	\<block b1\>	\<page b0\>	\<page b1\>

RTC (hour=0…23, min&sec=0…59, day=1…31, weekday=1(mon)…7(sun), month=1…12, year=16… (16=2016)

Get time	**0x50**							
Set time	**0x51**	\<hour\>	\<min\>	\<sec\>	\<day\>	\<weekday\>	\<month\>	\<year\>

LEDs (mode: 0 – default, 1 – use commands, state: 0 – off, 1 - on)

Set LED mode	**0x52**	\<mode\>						

Misc. functions

SW reset	**0xF0**	\<'r'\>	\<'e'\>	\<'s'\>	\<'e'\>	\<'t'\>			
Start bootloader	**0xF1**	\<'b'\>	\<'l'\>	\<'o'\>	\<'a'\>	\<'d'\>	\<'e'\>	\<'r'\>	
Default settings	**0xF2**	\<'d'\>	\<'e'\>	\<'f'\>	\<'a'\>	\<'u'\>	\<'l'\>	\<'t'\>	\<'s'\>
Bluetooth ID	**0x60**	\<chr 1\>	…	\<chr n\>					

Download flash contents

Download flash contents (if start blk=0, end blk=65535, downloading all data)							
<start blk 0>	<start blk 1>	<end blk 0>	<end blk 1>	<start pg 0>	<start pg 1>	<end pg 0>	<end pg 1>

Download flash contents response							
<chunk no#>	<data 0>	<data 1>	…	<data n>			

Download flash contents ACK							
<chunk no#>							

Download flash contents ACK response (status 0:OK, 1:no more data available)							
<status>							

LED commands

LED state (LED #0: Green, LED #1: Red. LED state: 0=off, 1=on)							
<LED no#>	<LED state>						

LED state response (status 0:OK, 1:failed)							
<status>							

Poll PGN message

Poll J1939 PGN (Destination Address and Source Address bytes are optional)					
<PGN b0>	<PGN b1>	<PGN b2>	<PGN b3>	<DA>	<SA>

Example message breakdown

The *Enter Configuration Mode* command is used as an example:

0x67 0x09 0xc0 0x30 0x30 0x30 0x30 0x30 0x30 0x26 0xd8 0xdd 0x85

The message consists of the following bytes:

0x67 = change configuration

0x09 = message length (first byte included = 0x67, last byte included = 0x30)

0xc0 = enter configuration mode

0x30 0x30 0x30 0x30 0x30 0x30 = pin code (000000)

0x26 0xd8 0xdd 0x85 = CRC32 checksum

Other example messages

These example messages are built using the message format described above and could be sent as is to an AX14100 (or AX141150) device.

Scan remote nodes:	**0x67 0x03 0x10 0x10 0x47 0x43 0x84**
Use bridge mode:	**0x67 0x03 0x03 0xce 0x06 0xfd**
Get time:	**0x67 0x03 0x50 0x80 0x06 0x9f 0xf2**
Start bootloader:	**0x67 0x0a 0xf1 0x62 0x6c 0x6f 0x61 0x64 0x65 0x72 0x2c 0xe7 0xe6 0x64**

Send Ext.ID CAN Frame (*ID=0x18FF0002, len=8, data: 0x11 0xff 0x34 0x22 0x56 0xee 0xab 0x09*):

0x13 0x0f 0x02 0x00 0xff 0x18 0x08 0x11 0xff 0x34 0x22 0x56 0xee 0xab 0x09 0xca 0x35 0x66 0x5e

Send Std.ID CAN Frame (*ID=0x1F0, len=8, data: 0x11 0xff 0x34 0x22 0x56 0xee 0xab 0x09*):

0x12 0x0d 0xf0 0x01 0x08 0x11 0xff 0x34 0x22 0x56 0xee 0xab 0x09 0xbf 0xc8 0xa1 0xe7

Use commands for LED driving:	**0x67 0x04 0x52 0x01 0x3a 0xb7 0x36 0x7f**
Turn on Red LED:	**0x78 0x04 0x01 0x01 0x64 0xfe 0xda 0xe6**
Exit configuration mode:	**0x67 0x03 0xc1 0x52 0xa5 0x97 0x75**

Example messages, from AX141100 to another Bluetooth device

These example messages are generated by an AX141100 device.

New connection notification (connected to CAN-BLE):

0x00 0x09 0x43 0x41 0x4e 0x2d 0x42 0x4c 0x45 0xdb 0xc4 0x5e 0xd4

Get time (12:18:02, 16.Aug.2016):

0x76 0x0a 0x50 0x0c 0x12 0x02 0x10 0x02 0x08 0x10 0xe8 0x67 0x58 0xad

Received 29bit id frame, no special CAN RX filter config needed for PDU2 (id: 0x18ff01f8, len: 8, data 0x 11 22 34 aa bb cc 01 02)

0x31 0x0f 0xf8 0x01 0xff 0x18 0x08 0x11 0x22 0x34 0xaa 0xbb 0xcc 0x01 0x02 0x49 0xd6 0x25 0x52

Received 11bit id data. Note, this needs brigde mode / special CAN RX filter config to work! (id: 0x187, len: 8, data 0x 11 22 34 aa bb cc 01 02)

0x21 0x0d 0x87 0x01 0x08 0x11 0x22 0x34 0xaa 0xbb 0xcc 0x01 0x02 0x78 0x11 0x6f 0x1d

Message flow examples

Figure 1 shows the message flow when a PC/smart device commands the AX141100 to enter the configuration mode and then exit the configuration mode. The PIN code in this example is the default pin ('000000') and is the correct PIN for entering the configuration mode.
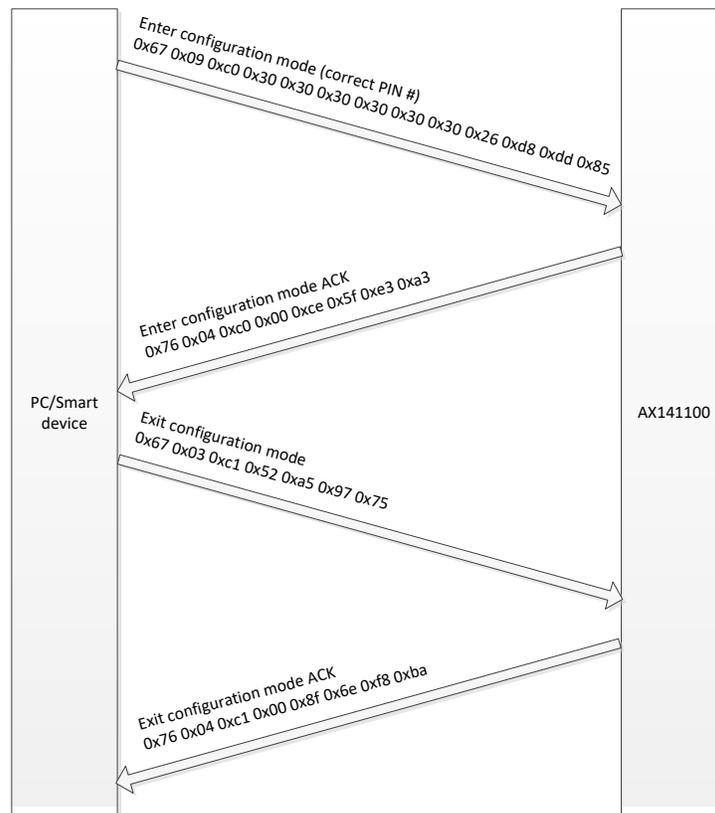

Figure 1 - Enter configuration mode / exit configuration mode

Figure 2 shows the message flow when trying to enter the configuration mode with a wrong PIN code. In this case the AX141100 device sends a NACK.
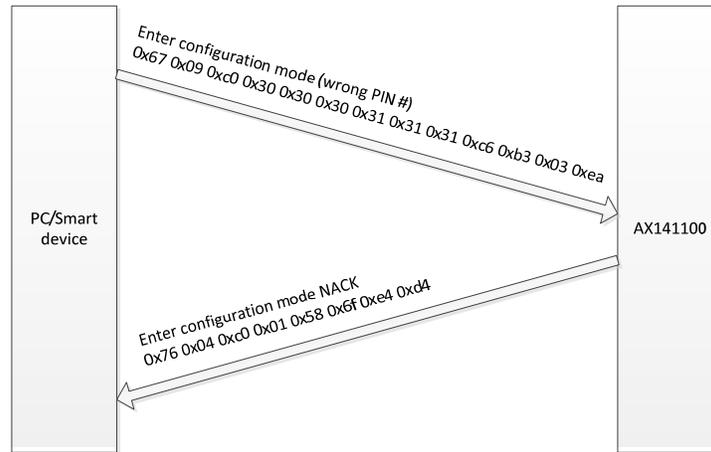

Figure 2 - Entering configuration mode with a wrong PIN code

Figure 3 describes the procedure for configuring the AX141100 device to work as a bridge. First the PC/Smart device configures the AX141100 to enter configuration mode (which is required for setting the bridge mode active). Then the AX141100 is reset for activating the new mode of operation. The reset is needed mostly for reconfiguring the CAN receive filters.

After the reset, the AX141100 sends a new connection ack message and then starts forwarding all received CAN messages. Note that in bridge mode all messages, including frames with both 11-bit and 29-bit identifiers are forwarded.
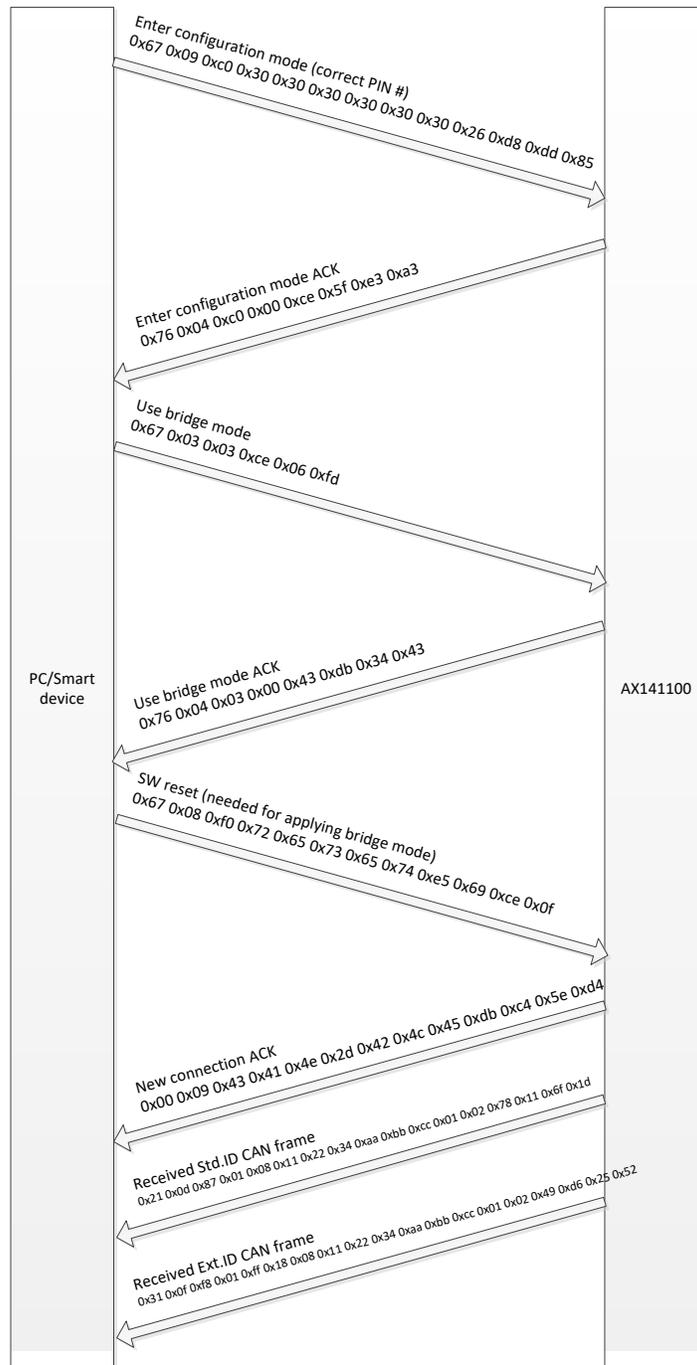

Figure 3 - Setting device to bridge mode for receiving all CAN messages

Example java class for accessing the AX141100 using SPP

```java
1
2    import java.util.zip.CRC32;
3
4    public class bt_example extends MainActivity {
5
6        private static String selectedAddress;
7        // Well known SPP UUID
8        private static final UUID SPP_UUID =
9                UUID.fromString("00001101-0000-1000-8000-00805F9B34FB");
10
11       BluetoothAdapter bluetoothAdapter;
12       BluetoothDevice mBTDev;
13       BluetoothSocket btSocketCAN;
14       OutputStream outStream;
15       InputStream inStream;
16
17       public void initBTConnection(BluetoothAdapter mBluetoothAdapter) {
18
19           // Resetting selectedAddress
20           selectedAddress = EMPTY_BD_ADDR;
21           if (connectionBDADDR != null) connectionBDADDR.setText(selectedAddress);
22
23           bluetoothAdapter = mBluetoothAdapter;
24
25           Log.i(TAG, "BT connection initialized.");
26       }
27
28       void setSelectedAddress(String newAddress) {
29           selectedAddress = newAddress;
30       }
31
32       int startBTConnection(MainActivity activity) {
33           mBTDev = bluetoothAdapter.getRemoteDevice(selectedAddress);
34           Log.i(TAG, "Connecting to " + selectedAddress + "...");
35
36           try {
37               btSocketCAN = mBTDev.createRfcommSocketToServiceRecord(SPP_UUID);
38           } catch (IOException e) {
39               Log.e(TAG, "Error in startBTConnection() and socket create failed: " +
40                   e.getMessage() + ".");
41           }
42
43           // Establish the connection.  This will block until it connects.
44           try {
45               btSocketCAN.connect();
46           } catch (IOException e) {
47               Log.e(TAG, "Something went wrong while connecting - Connection CLOSED!");
48               try {
49                   btSocketCAN.close();
50               } catch (IOException e2) {
51                   Log.e(TAG, "Unable to close locally...");
52               }
53               return -1;
54           }
55           Log.i(TAG, "Connection established and data link opened...");
56
57           // Create a data stream so we can talk to server.
58           try {
59               outStream = btSocketCAN.getOutputStream();
60           } catch (IOException e) {
61               Log.e(TAG, "Error in startBTConnection() and CAN output stream creation
62                   failed:" + e.getMessage() + ".");
63               return -2;
64           }
65           Log.i(TAG, "Output stream opened...");
66
67           try {
68               inStream = btSocketCAN.getInputStream();
69           } catch (IOException e) {
```

```
68              Log.e(TAG, "Error in startBTConnection() and CAN input stream creation
                failed:" + e.getMessage() + ".");
69              return -3;
70          }
71          Log.i(TAG, "Input stream opened...");
72
73          readDataFromRemoteDevice readHandler = new readDataFromRemoteDevice();
74
75          readHandler.setHostMainActivity(activity);
76
77          readHandler.execute();
78
79          return 0;   // OK
80      }
81
82      private class readDataFromRemoteDevice extends AsyncTask<Void, byte[], byte[]> {
83
84          MainActivity hostMainActivity;
85          long lCANFrameCount;
86
87          void setHostMainActivity( MainActivity activity ) { hostMainActivity =
            activity; lCANFrameCount = 0; }
88
89          /**
90           * The system calls this to perform work in a worker thread and
91           * delivers it the parameters given to AsyncTask.execute()
92           */
93          protected byte[] doInBackground(Void... unused) {
94              DataInputStream mmInStream = new DataInputStream(inStream);
95              byte[] buffer = new byte[128];  // buffer store for the stream
96              while (true) {
97                  try {
98                      if( mmInStream.read(buffer) > 0 ) {
99                          publishProgress(buffer);
100                     }
101                 } catch (IOException e) {
102                     Log.v(TAG, "In onResume() and an exception occurred during read: "
                         + e.getMessage());
103                     break;
104                 }
105             }
106
107             return buffer;
108         }
109
110         protected void onProgressUpdate(byte[]... readMsgs) {
111             if (readMsgs == null || readMsgs.length == 0)
112                 return;
113
114             CRC32 crc = new CRC32();
115             int msgLength, msgCRC;
116             byte[] rxMessage;
117
118             // Parsing the received messages
119             msgLength = readMsgs[0][1];
120             if ((msgLength <= 0) || (msgLength > 104)) {
121                 // Illegal message length!
122             }
123             else {
124                 // Length OK, making a local copy. +4 is for the CRC
125                 rxMessage = Arrays.copyOfRange(readMsgs[0], 0, (msgLength + 4));
126
127                 byte[] rawCRC = {rxMessage[msgLength + 3], rxMessage[msgLength + 2],
                    rxMessage[msgLength + 1], rxMessage[msgLength]};
128
129                 ByteBuffer wrapped = ByteBuffer.wrap(rawCRC);
130                 msgCRC = wrapped.getInt();
131
132                 crc.update(rxMessage, 0, msgLength);
```

```
133
134                      if (msgCRC == (int) crc.getValue()) {
135                          // -> CRC OK
136                          /* Handle resposes... */
137                          if (rxMessage[0] == (byte) 0x21) {
138                              // Std.ID CAN data
139                              // ...
140                          } else if (rxMessage[0] == (byte) 0x31) {
141                              // Ext.ID CAN data
142                              // ...
143                          } else if (rxMessage[0] == (byte) 0x43) {
144                              // MAP write response
145                              // ...
146                          } else if (rxMessage[0] == (byte) 0x54) {
147                              // MAP read response
148                              // ...
149                          } else if (rxMessage[0] == (byte) 0x65) {
150                              // Download flash contents response
151                              // ...
152                          } else if (rxMessage[0] == (byte) 0x75) {
153                              // Download flash contents ack response
154                              // ...
155                          } else if (rxMessage[0] == (byte) 0x76) {
156                              // Change configuration response
157                              // ...
158                          } else if (rxMessage[0] == (byte) 0x87) {
159                              // LED state response
160                              // ...
161                          }
162                      }
163                  }
164              }
165
166          /**
167           * The system calls this to perform work in the UI thread and delivers
168           * the result from doInBackground()
169           */
170          protected void onPostExecute(byte[] readMsg) {
171              Log.v(TAG, "...Data reading ended!");
172          }
173      }
174  }
```

Version	Date	Author	Comments
1.00	August 20, 2019	Gustavo Del Valle / Sue Thomas	
1.01	August 26, 2019	Antti Keranen / Sue Thomas	