Heartland

A Global Payments Company

Heartland Android SDK

Heartland

A **Global Payments** Company

Notice

Table of Contents

Version	Author	Date	Revisions
1.0	Shane Logsdon	5/18/21	Initial documentation release
1.1	Shane Logsdon	5/26/21	Update connection config to show non-test way of getting application context
1.2	Phil White	4/20/22	Added option for ClientTxnID to the CreditAuth builder and OTA/firmware update, and update to the latest BBPOS SDK to resolve lost Bluetooth connection scenario.
1.3	Phil White	1/11/23	Added information for using AutoSubstantiation.
1.4	Phil White	5/16/23	Updates to the entire document as the SDK now supports the C2X and Moby 5500 devices.
1.5	Phil White	12/5/23	Updated text in relation to OTA firmware updates as it is now supported for Moby 5500.
1.6	Phil White	1/16/24	Added new section for store and forward (SAF).

1.7	Phil White	9/12/24	Added info for surcharge.Updated cardholder interaction code sample.
1.8	Phil White	11/7/24	Updated surcharge info with pre-tax/post-tax.

SDK Configuration

While the SDK manages the communication with the device and the gateway, neither it nor the device maintains a copy of the merchant account credentials used, so these will need to be set by your application at runtime. Below are code snippets for SDK imports to be used as well as configuring the SDK with credentials:

```
// Used Imports example code

import java.math.BigDecimal;
import com.heartlandpaymentsystems.library.entities.Card;
import com.heartlandpaymentsystems.library.terminals.ConnectionConfig;
import com.heartlandpaymentsystems.library.terminals.DeviceListener;
import com.heartlandpaymentsystems.library.terminals.entities.TerminalResponse;
import com.heartlandpaymentsystems.library.terminals.TransactionListener;
import com.heartlandpaymentsystems.library.terminals.transactions.BaseBuilder;
import com.heartlandpaymentsystems.library.terminals.c2x.C2XDevice;
import com.heartlandpaymentsystems.library.terminals.moby.MobyDevice;
import com.heartlandpaymentsystems.library.terminals.transactions.CreditAuthBuilder;
import com.heartlandpaymentsystems.library.terminals.entities.CardholderInteractionRequest;
import com.heartlandpaymentsystems.library.terminals.entities.CardholderInteractionResult;
import com.heartlandpaymentsystems.library.terminals.enums.Environment;
import com.heartlandpaymentsystems.library.terminals.enums.TransactionStatus;
```

```
// Configuration example code

ConnectionConfig connectionConfig = new ConnectionConfig();
connectionConfig.setUsername("xxxxxxxxxxxx");
connectionConfig.setPassword("xxxxxxxx");
connectionConfig.setSiteId("xxxxx");
connectionConfig.setDeviceId("xxxxxxx");
connectionConfig.setLicenseId("xxxxx");
connectionConfig.setConnectionMode(ConnectionMode.BLUETOOTH);
connectionConfig.setEnvironment(Environment.TEST);
connectionConfig.setSafEnabled(safEnabled);
connectionConfig.setSafExpirationInDays(5);
connectionConfig.setSurchargeEnabled(surchargeEnabled);

//C2X
C2XDevice device = new C2XDevice(
    this.getApplicationContext(), // when called from an activity
    connectionConfig
);

//Moby 5500
MobyDevice device = new MobyDevice(
    this.getApplicationContext(), // when called from an activity
    connectionConfig
);
```

Bluetooth Pairing

Before creating a bluetooth connection with the device, your application requires some additional code in the form of a device listener / delegate object. Here's a brief description of each of the available methods and how they can assist in your application development:

- `onBluetoothDeviceFound` - As the bluetooth scanning process continues, this method will be called by the SDK as soon as a device is found, allowing your application to immediately update its UI to present this information to the user.
- `onBluetoothDeviceList` - Once the bluetooth scanning process completes, this method will be called by the SDK, signaling that no additional devices are currently available. The user should select one of the available devices or check their settings before initiating another bluetooth scan.
- `onConnected` - This method will be called by the SDK once a successful bluetooth connection has been established with the selected device.
- `onDisconnected` - This method will be called by the SDK when the bluetooth connection has been lost.
- `onError` - This method will be called by the SDK if any errors occur during the bluetooth connection process.
- `onTerminalInfoReceived` - This method may be called by the SDK if any information about the device is available.

Your application will need to initiate the bluetooth scanning process by calling the `initialize` method on the `device` object. Any progress along the way will be indicated by the SDK calling the appropriate method on your application's device listener / delegate object.

Once the user selects their correct bluetooth device, your application will need to update the SDK by calling the `connect` method on the `device` object, and this method will require an object obtained by either the `onBluetoothDeviceFound` method or the `onBluetoothDeviceList` method.

```
// Device Bluetooth Connection example code

device.setDeviceListener(new DeviceListener() {
    @Override
    public void onBluetoothDeviceFound(BluetoothDevice bluetoothDevice) {
        // receive available bluetooth devices as they are found
    }

    @Override
    public void onBluetoothDeviceList(HashSet<BluetoothDevice> deviceList) {
        // receive list of available bluetooth devices
        // present to user for selection/confirmation

        if (deviceList == null || deviceList.isEmpty()) {
            return;
        }

        // connect to the selected device
        device.connect(deviceList.iterator().next());
    }

    @Override
    public void onConnected(TerminalInfo terminalInfo) {
        // device is connected
        // allow processing
    }

    @Override
    public void onDisconnected() {
        // device has disconnected
    }

    @Override
    public void onError(Error Error, ErrorType errorType) {
        // handle errors
    }

    @Override
    public void onTerminalInfoReceived(TerminalInfo terminalInfo) {
        // handle received device information
    }
});

// initiate a bluetooth scan
device.initialize();
```

USB (Moby 5500)

The Moby 5500 supports USB communication which can be used instead of bluetooth. Just like with bluetooth, you will need to set up the same `DeviceListener` shown above in the “Bluetooth Pairing” section.

Your application will initiate the USB connection process by calling the `initialize` method on the `device` object. The `onConnected` method will be called when the connection is established.

Transaction Listener

Before starting a transaction request, your application requires some additional code in the form of a transaction listener / delegate object. Here's a brief description of each of the available methods and how they can assist in your application development:

- `onStatusUpdate` - As the EMV process occurs, this method will be called by the SDK when the device may need to prompt the user with information about the current status. Any status update received by this method should be displayed to the user.
- `onCardholderInteractionRequested` - As the EMV process occurs, this method will be called by the SDK when the card holder needs to take action. This is currently limited to EMV application selection and confirmation of the final authorization amount. **NOTE**: Version 1.4.0 and above changes this function to return a boolean instead of void. Return true if you handled the interaction and false if you did not.
- `onTransactionCompleted` - Once the authorization request has been submitted to the gateway, this method will be called by the SDK, passing the authorization response to your application.
- `onError` - This method will be called by the SDK if any errors occur during the transaction process.

```
// Transaction Response Handling example code

device.setTransactionListener(new TransactionListener() {
    @Override
    public void onStatusUpdate(TransactionStatus transactionStatus) {
        // inform user of status updates
    }

    @Override
    public boolean onCardholderInteractionRequested(
        CardholderInteractionRequest cardholderInteractionRequest) {
        // prompt user for action
        CardholderInteractionResult result;
        switch (cardholderInteractionRequest.getCardholderInteractionType()) {
            case EMV_APPLICATION_SELECTION:
                String[] applications =
cardholderInteractionRequest.getSupportedApplications();

                // prompt user to select desired application

                // send result
                result = new CardholderInteractionResult(
                    cardholderInteractionRequest.getCardholderInteractionType()
                );
```

```
                    result.setSelectedAidIndex(0);
                    device.sendCardholderInteractionResult(result);
                    return true;
                case FINAL_AMOUNT_CONFIRMATION:
                    // prompt user to confirm final amount
                    result = new CardholderInteractionResult(
                        cardholderInteractionRequest.getCardholderInteractionType()
                    );
                    result.setFinalAmountConfirmed(true);
                    device.sendCardholderInteractionResult(result);
                    return true;
               case SURCHARGE_REQUESTED:
                    // prompt user to approve/decline the surcharge amount
                    result = new CardholderInteractionResult(
                                CardholderInteractionType.CARDHOLDER_SURCHARGE_CONFIRMATION);
                    result.setFinalAmountConfirmed(isApproved);
                    device.sendCardholderInteractionResult(result);
                    return true;
                default:
                    break;
            }
            return false;
        }

        @Override
        public void onTransactionComplete(TerminalResponse response) {
            // handle response
        }

        @Override
        public void onError(Error error, ErrorType errorType) {
            // handle errors
        }
    });
```

Transaction Processing

Your application will need to initiate the transaction request process by creating the correct request builder, preparing any required data / information, and executing the builder. Any progress along the way will be indicated by the SDK calling the appropriate method on your application's transaction listener / delegate object.

```
// Initiate Transaction - CreditAuth - Manual Entry example code

Card card = new Card();
card.setNumber("424242******4242");
card.setExpMonth(12);
card.setExpYear(2025);
card.setCvv("123");

CreditAuthBuilder builder = new CreditAuthBuilder(device);
```

```
builder.setAmount(new BigDecimal("10.00"));
builder.setCreditCard(card);
builder.execute();

// Initiate Transaction - CreditAuth - Tap/Insert/Swipe example code

CreditAuthBuilder builder = new CreditAuthBuilder(device);
builder.setAmount(new BigDecimal("10.00"));
builder.execute();

// Initiate Transaction - CreditAddToBatch - example code

CreditCaptureBuilder builder = new CreditCaptureBuilder(device);
builder.setTransactionId(transactionId);
builder.execute();

// Initiate Transaction - CreditSale - Manual Entry example code

Card card = new Card();
card.setNumber("424242******4242");
card.setExpMonth(12);
card.setExpYear(2025);
card.setCvv("123");

CreditSaleBuilder builder = new CreditSaleBuilder(device);
builder.setAmount(new BigDecimal("10.00"));
builder.setCreditCard(card);
builder.execute();

// Initiate Transaction - CreditSale - Dip/Swipe example code

CreditSaleBuilder builder = new CreditSaleBuilder(device);
builder.setAmount(new BigDecimal("10.00"));
builder.execute();

// Initiate Transaction - Tip Adjust - example code

CreditAdjustBuilder builder = new CreditAdjustBuilder(device);
builder.setTransactionId(transactionId);
builder.setAmount(new BigDecimal("12.00"));
builder.setGratuity(new BigDecimal("2.00"));
builder.execute();

// Initiate Transaction - CreditReturn by Transaction ID - example code

CreditReturnBuilder builder = new CreditReturnBuilder(device);
builder.setTransactionId(transactionId);
builder.setAmount(new BigDecimal("10.00"));
builder.execute();

// Initiate Transaction - CreditVoid - example code

CreditVoidBuilder builder = new CreditVoidBuilder(device);
builder.setTransactionId(transactionId);
```

```
    builder.execute();

    // Cancel a transaction (no effect if there is no transaction active)

    device.cancelTransaction();
```

Getting gift card PAN (Moby 5500)

You can get the SVA PAN data from a gift card when using a Moby 5500. Use the function doSvaStartCard() to start the card reading process and then receive the data on the callback to onTransactionComplete().

Custom Client Transaction ID

To use your own Client Transaction ID, set the value that you want by using the `setReferenceNumber` method of the builder, as shown below. If you do not set your own reference number, a transaction ID will be generated automatically.

```
//Set the reference number
builder.setReferenceNumber(yourClientTxnId);
```

Once the transaction is complete, you will see the same value returned inside the `TerminalResponse` object parameter of `onTransactionComplete`.

```
//Get the transaction ID
transaction.getTransactionId();
```

Auto-Substantiation

Auto-substantiation is used in the healthcare industry as a result of IRS Notice 2006-69 for consumers to use flexible spending account (FSA/HRA) debit cards where the transaction is automatically substantiated at the POS. For merchants who support auto-substantiation at the POS, consumers no longer need to file a separate claim for benefits

To take advantage of auto-substantiation, the merchant must use an Inventory Information Approval System (IIAS). The IIAS identifies the qualified healthcare products being purchased by the cardholder at the POS. The IIAS must identify the FSA and HRA cards, automatically differentiate between qualified and non-qualified products at the POS, flag the items on the customer receipt, subtotal the qualified healthcare products amount including tax and discounts, and accommodate split-tender capability for non-qualified products.

Here is an example for how you can set specific amounts for dental, prescription, clinic, and vision. These amounts must not add up to more than the main transaction amount or an exception will be thrown.

```
//Set the healthcare values
```

```
creditSaleBuilder.setClinicSubTotal(new BigDecimal(5));
creditSaleBuilder.setDentalSubTotal(new BigDecimal(5));
creditSaleBuilder.setPrescriptionSubTotal(new BigDecimal(5));
creditSaleBuilder.setVisionSubTotal(new BigDecimal(5));
```

OTA Updating

The SDK supports OTA (over-the-air) updates for the C2X/C3X and Moby 5500 firmware and configuration updates for C2X/C3X. You will need to establish a bluetooth connection with the device to check for and install updates.

The first step is for your application to request the available versions for either firmware or config. You will also need to set the proper listener so the SDK can notify for success or failure.

```
//Set the listener for the available terminal versions

device.setAvailableTerminalVersionsListener(new AvailableTerminalVersionsListener() {
    @Override
    public void onAvailableTerminalVersionsReceived(TerminalUpdateType type, List<String>
         versions) {
        //choose a version to install
    }

    @Override
    public void onTerminalVersionInfoError(Error error) {
        //handle error
    }
});

//Request the available terminal firmware versions
device.getAvailableTerminalVersions(TerminalUpdateType.FIRMWARE);

//Request the available terminal config versions
device.getAvailableTerminalVersions(TerminalUpdateType.CONFIG);
```

NOTE: For Moby 5500, the List of versions returned in the onAvailableTerminalVersionsReceived callback will only return a single String stating “Update available” as it does not provide the actual version number. Also, as noted above, the Moby 5500 does not currently support OTA configuration updates.

After getting the available terminal version for the selected `TerminalUpdateType`, you can initiate the download and install to the device. Make sure that the SDK is able to complete the process without interruption. A separate listener is used for the installation and will give progress updates.

```
//Set the listener for terminal updates
c2XDevice.setUpdateTerminalListener(new UpdateTerminalListener() {
    @Override
    public void onProgress(@Nullable Double completionPercentage, @Nullable String
            progressMessage) {
        //Progress updates
    }

    @Override
    public void onTerminalUpdateSuccess() {
        //Update completed successfully
    }

    @Override
    public void onTerminalUpdateError(Error error) {
        //Handle error
    }
});
```

Request the terminal update by using one of the “versions” returned inside the available terminal updates listener method, `onAvailableTerminalVersionsReceived`.

NOTE: As mentioned above, the Moby 5500 callback for available versions does not give an actual version and so the call to updateTerminal will ignore the 2nd parameter for version. This means that you can pass in a null value or any String value you want.

```
//Request the terminal update
device.updateTerminal(TerminalUpdateType.FIRMWARE, versions.get(i));
```

After the update is completed successfully, the device will disconnect and finish completing the process. You will see flashing lights on the terminal indicating this final step. Once the terminal has finished the process, you will be able to connect again as usual.

Permissions

The SDK requires some permissions to properly function. These include permission for `INTERNET` so it can perform the transaction network requests and bluetooth-related permissions so the SDK can bluetooth connect to the terminals. These bluetooth permissions include `ACCESS_FINE_LOCATION` because on Android 11 and lower, a Bluetooth scan could potentially be used to gather information about the location of the user. You will need to request these permissions in your application depending on what functionality you want to use.

The SDK includes the helper class `PermissionHelper`. This class can be used to request the permissions needed. You can see how it is used by referencing the included example-app code.

Store And Forward

Store and forward (SAF) allows transactions to be stored when connection issues prevent processing and then later can manually be processed when the connection is stable again. When SAF is enabled, any supported transaction that fails to process due to server errors or client-side connection issues will be stored on the android device in a secure database. When the user is confident that the server or connection issues are resolved, they can upload all the pending transactions that are stored. This upload process processes each stored transaction one at a time, going oldest to newest.

Setup

To enable SAF, simply add a few lines of code to your `ConnectionConfig` setup. Use `setSafEnabled` to enable the feature and use `setSafExpirationInDays` to set the number of days that a stored transaction can be saved before it is automatically deleted. As with other `ConnectionConfig` settings, these must be set before connecting to the device.

```
connectionConfig.setSafEnabled(true);
connectionConfig.setSafExpirationInDays(5);
```

SAF Listener

To receive callbacks related to SAF, you'll also need to use `setSafListener` on the device object.

```
device.setSafListener(new SafListener() {
    @Override
    public void onProcessingComplete(List<TransactionResponse> responses) {
        //callback for when all previously stored transactions have been attempted
    }

    @Override
    public void onAllSafTransactionsRetrieved(List<SafTransaction> obfuscatedSafTransactions){
       //callback when the requested transactions have been retrieved
    }

    @Override
    public void onError(Error error) {
       //callback for errors
    }

    @Override
    public void onTransactionStored(String id, int totalCount, BigDecimal totalAmount) {
        //callback when a transaction has been stored in the database
    }

    @Override
    public void onStoredTransactionComplete(String id, TransactionResponse
transactionResponse) {
```

```
            //callback when a single transaction processing has been attempted
    }
});
```

Upload SAF

When ready to upload stored transactions, simply call uploadSAF to initiate the upload process, like so:

```
device.uploadSAF();
```

When this process is complete, the onProcessingComplete callback will be called with a full list of results.

Acknowledgement

When a transaction has been completed with approved or declined, the transaction should be acknowledged so it can be deleted from the database. If the transaction is not acknowledged, it will stay in the database until it is expired. To acknowledge, use this method:

```
device.acknowledgeSAFTransaction(id);
```

This can be done inside the callback for `onStoredTransactionComplete` (see the example-app for examples on all of the items discussed in this document).

Force SAF

Whether for testing or because your connection is unsteady, you may want to force SAF for a period of time. To do so, use this method:

```
device.setForcedSafEnabled(true);
```

This option can be toggled at any time once connected to the device and will automatically store supported transactions, whether or not there is a working internet connection.

Follow-Up Transactions

For handling follow-up transactions like refund or void, special steps must be taken when dealing with a stored and unprocessed transaction. When a transaction is stored, it provides a SAF ID for it. This SAF ID is used in place of a transaction ID when doing a follow-up transaction. So if a sale transaction is stored and you would like to void this transaction, you would use the SAF ID provided as the transaction ID for the void request. When the SAF items are uploaded/forwarded, it will go through them in order of oldest to newest and update the void request with the actual transaction ID after the sale is processed.

Surcharge

With surcharge enabled, a 3% charge will be added to the total for sale/auth transactions that use a credit card. No charge is added if the card type is unable to be detected. To enable surcharge, simply set the option in the ConnectionConfig before connecting to the terminal:

```
connectionConfig.setSurchargeEnabled(surchargeEnabled);
```

Surcharge requires that the cardholder is given the opportunity to approve/decline this charge so the new cardholder interaction type of SURCHARGE_REQUESTED has been added and this interaction request will occur when surcharge is enabled and the card was determined to be a credit card.

```
case SURCHARGE_REQUESTED:
       // prompt user to approve/decline the surcharge amount
        return true; //Only return true if you actually showed a prompt
```

Pre-Tax vs. Post-Tax

The default setting is for the surcharge amount to be calculated using the total amount as-is, but depending on the state you are in, this might need to be calculated without including tax. If you need surcharge to be calculated pre-tax, then use this ConnectionConfig option:

```
connectionConfig.setSurchargePreTax(surchargePreTaxEnabled);
```

Besides setting this ConnectionConfig option, a tax amount must also be provided. This tax amount value is treated the same as gratuity and is considered a part of the total, so a pre-tax surcharge calculation will take the provided total, subtract the tax amount, and then calculate the 3% surcharge. The CreditSaleBuilder and CreditAuthBuilder classes have been updated to include this new tax amount variable.

```
creditSaleBuilder.setTaxAmount(new BigDecimal(taxAmount));
```

Troubleshooting

Device cannot be paired

- Please press the power on button to restart your device.
- Please check to see if you can find the device’s “Serial Number” (Shown on the back of the device) in the “Scanned Device List” of your smartphone or tablet.

Device loses the connection with your smartphone or tablet when the device has gone into auto-off mode

- Please press the power on button to turn on the device again. The device will automatically
- connect with your smartphone or tablet again.
- The device may be at lower battery level, please use the USB cable to recharge it, then retry.
- Please ensure the device or smartphone/tablet is within the reception range.

Device does not work with your phone or tablet

- Please ensure the Bluetooth® function of your smartphone or tablet is turned on.
- Please check if the version of your operating system is supported for this device’s operation.

Device cannot read your card successfully

- Please press the power on button to turn on the device again.
- The device will automatically connect with your smartphone or tablet again.
- The device may be at lower battery level, please use the USB cable to recharge it, then retry.
- Please ensure the device or smartphone/tablet is within the reception range.
- Inserting card
    - Please check if the device has power when operating and ensure devices are connected.
    - Please follow the application instructions to insert or tap the card.
    - Please ensure that there is no obstacle in the card slots.
    - Please check if the chip of the card is facing the right direction when inserting the card.
    - Please ensure that your phone/ tablet is a supported model for this device’s operation.
    - Please insert the card with a more constant speed.
- Tap Card/Mobile Wallet
    - Please check if your card supports NFC payment.

- Please ensure your card is placed within 4 cm range on top of the NFC marking.
- Please take out your NFC payment card from your wallet or purse for payment to avoid any interference.

Device has no response

- Please use a paperclip to press the reset button at the bottom for reboot.