

# POS SDK Interface description

## Version information

Time	Version No	Content	Author
2020-06-23	V1.0.1	First edition	Conlin
2020-08-29	V1.0.2	Add serial port read/write, and modify the package name prefix to com.hcd.hcdpos	Conlin
2022-06-06	V1.0.3	Add interface call description of built-in scanning module	Conlin
2022-07-03	V1.0.4	Add interface call description of built-in hard decoding scanning module	Conlin
2022-08-15	V1.0.5	Optimize the probability of garbled code when the built-in scanning hard decoding module starts; Add interface call supporting meter counter	Conlin
2022-09-18	V1.0.6	Adapt new mode hard decoding interface	Conlin

## Elaborate

Explain that POS SDK is a call interface packaged for application development, which is convenient for calling&controlling hardware related devices.

The SDK file is uniformly hcdpos.jar, which needs to be imported into the developer's code for use.

Current hcdpos The jar version is V1.0.16.

## Interface

### Cashbox

```
// import class
import com.hcd.hcdpos.cashbox.Cashbox;
// Control cashbox opening (output cashbox pulse)
Cashbox.doopenCashBox();
```

# Elecscale

```
// import class
import com.hcd.hcdpos.elecscale.ElecScale;
// Define ElecScale class
private ElecScale mElecScale;

// Initialization, if it can be placed in onCreate
mElecScale = new ElecScale(this);

// Open the ElecScale with monitoring
mElecScale.startGetElecScaleData(new ElecScale.ElecScaleListener()
    { @Override
        public void onDeviceConnect(int status) {
            if (ElecScale.CONNECT_STATUS_CONNECT == status) {

                // Elecscale connected
            } else if (ElecScale.CONNECT_STATUS_DISCONNECT == status) {
                // Elecscale disconnected (USB terminal disconnected)
            } else if (ElecScale.CONNECT_STATUS_CONNECT_NO_DATA == status) {
                // The Elecscale has no data (the USB terminal is OK, the DB9 terminal of the
                // electronic scale may have a problem or the electronic scale has no data to send)
            } else {
                // Unknown connection status
            }
        }
    })
@Override
public void onDataReport(String value) {
    // Get the current weight value
    String weightValue = mElecScale.getWeightFromReportString(value);
    // Obtain whether the current elecscale is stable [S ->Stable; U ->Unstable; F -
    >Abnormal]
    String status = mElecScale.getStableStatusFromReportString(value);
}
@Override
public void onDeviceElecScaleType(String type) {
    // type To obtain the elecscale type (negligible);
    // [COMMON ->general type; ACLAS ->top elecscale; F ->abnormal elecscale]
}
});

// Clear the control elecscale
mElecScale.doElecScaleZero();
// Control elecscale peeling
mElecScale.doElecScalePeel();
// Close the elecscale (call it after
use)
mElecScale.stopGetElecScaleData();
mElecScale = null; //Suggested
release
```

## Nixie tube

```
// Note: The API is currently limited to nixie tube connection via USB (->POS) to serial port
line (->nixie tube);
// import class
import com.hcd.hcdpos.digitaldisplay.DigitalDisplay;
import com.hcd.hcdpos.digitaldisplay.DigitalDisplay.DigitalCmdType;
import com.hcd.hcdpos.digitaldisplay.DigitalDisplay.DigitalConnectStatus;

// Some values defined in SDK
/*****
//USB connection status
public static class DigitalConnectStatus {
    public static final int CONNECT = 0; // Indicates connected
    public static final int DISCONNECT = 1; // Indicates disconnection
}
// Type value below nixie tube
public static class DigitalCmdType {
    public static final int NULL = 0; //Empty Type
    valuepublic static final int UNITPRICE = 1; // Light
up 'unit price' public static final int TOTALPRICE =
2; // Light up 'total price' public static final int
COLLECTION = 3; // Light up 'Cashier' public static
final int CHANGE = 4; // Turn on 'change'
}
```

```

// Define nixie tube class
private DigitalDisplay mDigitalDisplay;

// Initialization, if it can be placed in onCreate
mDigitalDisplay = new DigitalDisplay(this);

// Open the nixie tube, result [boolean]: true ->open successfully,
false->openfailed
boolean result = mDigitalDisplay.startDigitalDisplayusb(new
DigitalDisplay.DigitalDisplayListener() {
    @Override
    public void onDeviceConnect(int status) {
        // Callback status [int] indicates the current connection status. Please refer
        to the definition for details 'DigitalConnectStatus'
    }
});

//Clear all displays on the nixie tube, result [boolean] : true->Clearing
succeeded, false->Clearing failed
boolean result = mDigitalDisplay.sendClearData();

//Restore the nixie tube to the power on state, result [boolean] : true-
>recovery was successful, false->restore failed
boolean result = mDigitalDisplay.sendResetData();

// Illuminate/clear the type indicator below the nixie tube, type [int], For specific values,
please refer to'DigitalCmdType'
// result [boolean] : true->Sending succeeded, false-
>fail in send
boolean result = mDigitalDisplay.sendTypeData(type);

// Display the nixie tube value. The value [String] can only be '0-9' and '.' (decimal point),
// When there is no decimal point, the maximum length of the string is 8; When there is a decimal
point, the maximum length of the string containing the decimal point is 9;
// The decimal point cannot be at the first place of the string;
// If the above rules are not met, the sending will fail;
// result [boolean] : true->Sending succeeded, false->fail
in send
boolean result = mDigitalDisplay.sendValueData(value));

//Close nixie tube (call it after
use)mDigitalDisplay.stopDigitalDisplay
()); mDigitalDisplay = null;
//Suggested release

```

# Serial port

```
// Note: To use this interface, it is necessary to confirm that there is a
// corresponding serial port in the hardware.
// Import class
import com.hcd.hcdpos._uartmanager.UartManager;

// Define ElecScale Class
private UartManager mUartManager;

// Initialization, if it can be placed in onCreate
mUartManager = new UartManager(this);

// Initialize the serial port. There are two interfaces
// 1. public boolean initUart(String sPath, int iBaudrate)
// 2. public boolean initUart(String sPath, int iBaudrate, int iParity, int
iStop, int iBits)
// sPath is the actual serial port node, such as "/dev/ttyS0"
// iBaudrate Baudrate includes 1200/2400/4800/9600/19200/38400/57600/115200
// iParity is Parity check, 0 represents no parity, 1 represents odd parity,
and 2 represents even parity.
// iStop is stop bit, 1 represents 1 stop bit, 2 represents 2 stop bits
// iBits is Bitdigits, 7 represents 7bits, 8 represents 8bits

// method 1 initUart default iParity = 0, iStop = 1, iBits = 8
// Return value true indicates successful initialization, false indicates
failure;
// The reference code is as follows:
if (null != mUartManager) {
    boolean result = mUartManager.initUart("/dev/ttyS0", 115200);
    //boolean result = mUartManager.initUart("/dev/ttyS0", 115200, 0, 1,8);
}

// Send serial port information. The content sent is a byte array
// Return value True indicates successful sending, false indicates failure;
// The reference code is as follows:
if (null != mUartManager) {
    byte[] sendData = {(byte)0x12, (byte)0x34, (byte)0x56};
    boolean result =mUartManager.sendUartData(sendData);
}

// To read serial port information, this function should be called in a single sub thread. The
reading process will stop waiting while waiting for serial port data;
// Return value is a byte array,if the reading fails, null is returned
// The reference code is as follows:
if (null != mUartManager) {
    byte[] buffer = mUartManager.receiveUartData();
}

// Release the serial port (call it after use), no return value
if (null != mUartManager)
{ mUartManager.deinitUart();
mUartManager = null;
//Suggested release
```

## Built in code scanning module (soft decoding)

```
// Note: To use this interface, confirm whether the current hardware has built-in scanning
// Import class
import com.hcd.hcdpos.scannermanager.scannerdecode.ScannerDecode;

// Define the built-in code scanning class
private ScannerDecode mScannerDecode;

// Initialization, if it can be placed in onCreate
mScannerDecode = new ScannerDecode(this);

// Judge whether the current device supports it, and use the code scanning function when the
hardware supports it;
// Select one of the software solutions and hard solutions, and the hardware will only support
one of them;
// Code scanning module supporting soft solution
boolean isSupportSW = mScannerDecode.isSupportScanner();

// Start&end must occur in pairs, start once, and end once. Cross calling is not allowed (such as
starting twice);
// Start code scanning, with monitoring and preview&preview is not required, and only one can be
selected at a time;
// Start&close scanning code, because the camera will be opened&closed, it is recommended to
execute in the sub thread;
// The interface with preview is recommended to start scanning code 200 ms after the SurfaceView
of the preview form is drawn;

/* Scan head does not need preview Sample */
private void initScannerDecodeListener()
{ mScannerDecode.startSoftwareScanCode(new
ScannerDecode.ScannerValueListener()
{ @Override
public void onScannerValue(String value) {
```

```

        // String value is the result of the scan;

        // If you want to start scanning again after receiving the scanning content,
        execute the following shielding parts
        // mScannerDecode.releaseSoftwareScanCode();
        // initScannerDecodeListener();
    }
});

/* Need to preview Sample */
<SurfaceView
    android:id="@+id/camerascanner_preview"
    android:layout_width="400dip"
    android:layout_height="300dip"
    android:layout_gravity="center" />

private void initScannerDecodeListener()
{
    mScannerDecode.startScanCode(new
    ScannerDecode.ScannervalueListener() {
        @Override
        public void onScannervalue(String value) {
            // String value is the result of the scan;

            // If you want to start scanning again after receiving the scanning content,
            execute the following shielding parts
            // mScannerDecode.releaseSoftwareScanCode();
            // initScannerDecodeListener();
        }
    },
    ((SurfaceView) findViewById(R.id.camerascanner_preview)).getHolder());
}
//Close the code scanning module (call it after use)
mScannerDecode.stopScanCode()
mScannerDecode = null; //Suggested release

```

## Built in code scanning module (hard decoding)

```
// Note: To use this interface, confirm whether the current hardware has built-in scanning
// Import class
import com.hcd.hcdpos.scannermanager.scannerdecode.ScannerDecode;
// Define the built-in code scanning class
private ScannerDecode mScannerDecode;
// Initialization, if it can be placed in onCreate
mScannerDecode = new ScannerDecode(this);
// Judge whether the current device supports it, and use the code scanning function when the
hardware supports it;
// Select one of the following software solutions and hard solutions, and the hardware will
only support one of them;
// Code scanning module supporting hard solution
boolean isSupportHW = mScannerDecode.isSupportHWScanner();
// Start&end must occur in pairs, start once, and end once. Cross calling is not allowed
(such as starting twice);
// Start code scanning with monitoring. When there is code scanning content, the monitoring
can receive the scanning content;
// The continuous scanning process does not need to end+restart, just wait for monitoring
all the time;
// When the code scanning is not needed, then finished the code scanning
/* Hard Unscanning Head*/
private void initHWScannerDecodeListener() {

mScannerDecode.startScanCode(new ScannerDecode.ScannervalueListener()
{ @Override
public void onScannervalue(String value) {
// String value is the result of the scan. There is no need to re trigger here,
just wait for the value callback
Log.d(TAG, "onScannervalue value: " + value);
}
});
}
// Closed scanning module (to be
called after use)
mScannerDecode.stopScanCode()
mScannerDecode = null;
```



# Metercounter

```
// Import class
import com.hcd.hcdpos.metercounter.MeterCounter;
// Define metercounter class
private MeterCounter mMeterCounter;

// Initialization, if it can be placed in onCreate
mMeterCounter = new MeterCounter(this);

// Initialize metercounter with monitoring
mMeterCounter.initMeterCounter(new MeterCounter.MeterCounterListener()
{ @Override
    public void onDeviceConnect(int status) {
        if (MeterCounter.CONNECT_STATUS_CONNECT == status) {
            // Metercounter connected
        } else if (MeterCounter.CONNECT_STATUS_DISCONNECT == status) {
            // Metercounter disconnected (USB terminal disconnected)
        }
    }
});

@Override
public void onDataReport(String value) {
    //Status value mMeterCounter.getStatusCodeFromReportString(value)
    //Count Type mMeterCounter.getCountTypeFromReportString(value)
    //Class mMeterCounter.getClassIdFromReportString(value)
    //Count value mMeterCounter.getCountValueFromReportString(value)
    //Speed value mMeterCounter.getSpeedValueFromReportString(value)
}
});

}

// Trigger acquisition action
if (null != mMeterCounter)
{ mMeterCounter.doTriggerData();
}

//Release the meter counter (call it after use)
if (null != mMeterCounter)
{ mMeterCounter.deinitMeterCounter();
}
mMeterCounter = null; //Suggested release
```