HCD POS SDK Interface Description

version information

timing	version number	element	author
2020-06-23	V1.0.1	first version	Conlin
2020-08-29	V1.0.2	Add serial port read/write, modify package name prefix to com.hcd.hcdpos	Conlin
2022-06-06	V1.0.3	Add the interface call description for the built-in scanning module	Conlin
2022-07-03	V1.0.4	Add the interface call description of the built-in hard-decode scanning module	Conlin
2022-08-15	V1.0.5	Optimize the probability of garbled code when the built-in scanning hard decoding module starts up; add support for meter counter interface calls	Conlin
2022-09-18	V1.0.6	Adaptation to new ways of hard decoding interfaces	Conlin
2022-12-17	V1.0.7	Added support for TX60 electronic scales Add support for different voltages to control the money box interface (if supported by the hardware motherboard)	Conlin
2023-07-11	V1.0.8	Fix bugs in TX60 electronic scale Support the control of the money box pulse output of different voltage interfaces (if the hardware motherboard support) electronic scales part of the heartbeat monitoring report timeout from 3s to 9s, for compatibility with the Dahua weighing Built-in soft solution scanning module to add support for controlling the scanning light (if supported by the hardware) Documentation to add instructions for selecting the input source of the electronic scale	Conlin

elaborate (on a topic)

HCD POS SDK is a layer of calling interface packaged for application development, which is convenient for calling & controlling hardware related devices.

The SDK file is hcdpos.jar, which should be imported into the developer's code. The current version of hcdpos.jar is **V1.0.22**.

connector

money box

```
// No voltage selection hardware board
// Import the class
import com.hcd.hcdpos.cashbox.
//  Controls the opening of the money box (outputs money box pulses)
Cashbox.doOpenCashBox();
```

```
// Hardware supports different voltage outputs for money box pulses
// Import class
import com.hcd.hcdpos.cashbox.
// Controls the opening of the cash box (output pulses correspond to the options configured in the system setup)
Cashbox.doOpenCashBoxVol();
```

electronic balance

Electronic scales currently support Dahua protocol, top protocol (ACLAS), Taiwan Scale protocol (TSCALE), XK3118T1, TX60;

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

// Initialize, e.g. in onCreate.
mElecScale = new ElecScale(this);

//  Turn on the electronic scale with listening
mElecScale.startGetElecScaleData(new ElecScale.ElecScaleListener() {
    @Override
    public void onDeviceConnect(int status) {
        if (ElecScale.CONNECT_STATUS_CONNECT == status) {
            // Electronic scale connected
        } else if (ElecScale.CONNECT_STATUS_DISCONNECT == status) {
            //  Electronic scale disconnected (USB terminal disconnected)
        } else if (ElecScale.CONNECT_STATUS_CONNECT_NO_DATA == status) {
            //  no data from the electronic scale (USB OK, DB9 of the electronic scale may be faulty
            or no data from the electronic scale)
        } else {
            // Unknown connection status
        }
    }
    @Override
    public void onDataReport(String value) {
        //  Get the current weight value
        String weightValue = mElecScale.getWeightFromReportString(value);
        //  Get whether the current electronic scale is stable or not [S->Stable; U->Unstable; F-
        >Abnormal electronic scale].
        String status = mElecScale.getStableStatusFromReportString(value);
    }
    @Override
    public void onDeviceElecScaleType(String type) {
        //  type is the type of scale to get (can be ignored).
        // [COMMON -> generic type; ACLAS -> top electronic scale; F -> electronic scale
        exception]
    }
}).

// Controlling the zeroing of the electronic scale
mElecScale.doElecScaleZero();
```

```
// Controlled tare of electronic scales
mElecScale.doElecScalePeel();

// Turn off the electronic scale (to be called after use)
mElecScale.stopGetElecScaleData();
mElecScale = null; //recommended to be released
```

The above scale interfaces are the "default" startup interfaces, when the POS hardware is of a different type, the "default" scale data source rules are:

1. When the POS device does not have an exposed DB9 serial port, then the system ROM will not be configured accordingly, and the "default" data source for the electronic scale will be the U-to-serial line;
2. When the POS device has an exposed DB9 serial port, then the system ROM will be configured accordingly, and the "default" electronic scale data source will be the DB9 serial port;

At the same time, there is an additional "startup" interface in the SDK that can be used to force the data source:

```
// Open the interface to the electronic scale as described above
startGetElecScaleData(ElecScaleListener listener)
// ------> Replace
startGetElecScaleDataInterface(ElecScaleListener listener, int interfaceType)
// where listener is called with the same logic as the original function above.
//       interfaceType denotes the input source of the electronic scale, 0 denotes U to serial line, 1 denotes DB9 port.
// If the application layer needs to support both exposed DB9 and U-string, you need to add a UI level to configure the interaction, mapped to interfaceType.
passed into the SDK.
```

digital tube

```
// Note: This API is currently limited to digital tubes connected via USB (->POS) to serial cable (->Digital Tube);
// Import class
import com.hcd.hcdpos.digitaldisplay.
import com.hcd.hcdpos.digitaldisplay.DigitalDisplay.DigitalCmdType;
import com.hcd.hcdpos.digitaldisplay. DigitalConnectStatus; import com.hcd.hcdpos.digitaldisplay.

// Some values already defined in the SDK
/**********************************************************************/
// USB connection status
public static class DigitalConnectStatus {
    public static final int CONNECT = 0; // means connected
    public static final int DISCONNECT = 1; // Indicates a broken connection
}
// Type value underneath the digital tube
public static class DigitalCmdType {
    public static final int NULL = 0; // Clear the type value public static final int UNITPRICE = 1; // Highlight 'Unit Price' public static final int TOTALPRICE = 2; // Highlight 'Total Price' public static public static final int COLLECTION = 3; // highlight 'Cashier' public static final int CHANGE = 4; // highlight 'Change'
}
/**********************************************************************/

// Define the digital tube class
private DigitalDisplay mDigitalDisplay;

// Initialize, e.g. in onCreate.
mDigitalDisplay = new DigitalDisplay(this);
```

```
// turn on the digital tube, result [boolean]: true-> turn on
successfully, false-> turn on failed boolean result =
mDigitalDisplay.startDigitalDisplayUsb(new
DigitalDisplay.DigitalDisplayListener() {
    @Override
    public void onDeviceConnect(int status) {
        //  Callback status [int] for current connection status, see definition
        'DigitalConnectStatus'.
    }).

// Clear all displays on the digital pipe, result [boolean]: true->clear
successfully, false->clear failed boolean result =
mDigitalDisplay.sendClearData();

// Restore the digital tube to the powered-up state, result [boolean]: true-
>Restore successfully, false->Restore failed boolean result =
mDigitalDisplay.sendResetData();

//  Lights up/clears the type indication below the digital tube, type [int], please refer to
'DigitalCmdType' for exact value.
//  result [boolean]:  true->successful send, false->failed send
boolean result = mDigitalDisplay.sendTypeData(type);

//  Display the numeric value of the digital pipe, value [String] is only '0-9' and '.' (decimal
places), and
//  The  maximum length of a string without a decimal point is 8; with a decimal point, the maximum
length of a string with a decimal point is 9.
//  Decimal point can't be in the first place of the string.
//  If the above rule is not met, it will fail to be sent; // If the above rule is not met, it will fail to be
sent.
//  result [boolean]:  true->successful send, false->failed send
boolean result = mDigitalDisplay.sendValueData(value));

// Turn off the digital display (to be called after use)
mDigitalDisplay.stopDigitalDisplay();
mDigitalDisplay.stopDigitalDisplay();
mDigitalDisplay.stopDigitalDisplay()
mDigitalDisplay = null; //recommended to be released
```

serial port (computing)

```
// Note: Use of this interface requires confirmation that the corresponding serial port exists in the hardware.
// Import class
import com.hcd.hcdpos.uartmanager.UartManager;

// Define the Scale class
private UartManager mUartManager.

// Initialize, e.g. in onCreate.
mUartManager = new UartManager(this);

//  Initialize the serial port with two interfaces
// 1. public boolean initUart(String sPath, int iBaudrate)
// 2. public boolean initUart(String sPath, int iBaudrate, int iParity, int iStop, int iBits)
// where sPath is the actual serial port node, e.g. "/dev/ttyS0"
//iBaudrate   iBaudrate baud rate contains 1200/2400/4800/9600/19200/38400/57600/115200
//        iParity is parity, 0 means no parity, 1 means odd parity, 2 means even parity.
        //iStop is the stop bit, 1 means 1 stop bit, 2 means 2 stop bits
//        iBits is the number of bits, 7 means 7bits, 8 means 8bits.
// Method 1 initUart Default iParity = 0, iStop = 1, iBits = 8
// The return value true means the initialization succeeded, false means it failed; // The return value true means the initialization succeeded, false means it failed.
// 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 a serial message as a byte array.
// The return value true indicates a successful send, false indicates a failure.
// The reference code is as follows.
if (null != mUartManager) {
    byte[] sendData = {(byte)0x12, (byte)0x34, (byte)0x56};
    boolean result = mUartManager.sendUartData(sendData);
}

// read the serial port information, this function should be called in a separate sub-thread, the reading process that is waiting for the serial port data will be stalled waiting; // read the serial port information, this function should be called in a separate sub-thread, the reading process that is waiting for the serial port data will be stalled waiting.
// Returns an array of bytes, or null if reading fails.
// The reference code is as follows.
if (null != mUartManager) {
    byte[] buffer = mUartManager.receiveUartData();
}

// Release the serial port (to be called after use), no return value
if (null != mUartManager) {
    mUartManager.deinitUart();
    mUartManager = null;
    //recommended to release the
}
```

Built-in scanning module (soft decoding)

```
// Note: Use of this interface requires confirmation that the current hardware has built-in scanning.
// Import class
import com.hcd.hcdpos.scannermanager.scannerdecode.

// Define the built-in scan class
private ScannerDecode mScannerDecode;

// Initialize, e.g. in onCreate.
mScannerDecode = new ScannerDecode(this);

// Determine whether the current device is supported, and then use the code-sweeping function if the hardware supports it.
// Choose either soft or hard, the hardware will only support one of them;
// Scanning module with soft decode support
boolean isSupportSW = mScannerDecode.isSupportScanner();

// start & end must occur in pairs, one start, one end, no cross-calls (e.g. start twice);
// start sweep with listen, preview & no preview, only one choice at the same moment ;)
// Start & Close Sweeping because it turns the camera on & off is recommended to be executed in a sub-thread;
// For interfaces with preview, it is recommended to start the code scanning 200ms after the preview form SurfaceView has been drawn.

/* Scanning head without 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, after receiving the scan, you want to restart the scan, perform the following blocking section
            // mScannerDecode.releaseSoftwareScanCode();
            // initScannerDecodeListener(); // 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, after receiving the scan, you want to restart the scan, perform the following
            blocking section
            // mScannerDecode.releaseSoftwareScanCode();
            // initScannerDecodeListener(); // initScannerDecodeListener().
        }
    }, }
    ((SurfaceView) findViewById(R.id.camerascanner_preview)).getHolder());;
}

// Disable the scanning module (to be called after use)
mScannerDecode.stopScanCode()
mScannerDecode = null;
//recommended to release the
```

Built-in scanning module (hard decoding)

```
// Note: Use of this interface requires confirmation that the current hardware has built-in scanning.
// Import class
import com.hcd.hcdpos.scannermanager.scannerdecode.

// Define the built-in scan class
private ScannerDecode mScannerDecode;

// Initialize, e.g. in onCreate.
mScannerDecode = new ScannerDecode(this);

// Determine whether the current device is supported, and then use the code-sweeping function if the hardware supports it.
// Choose one of the following soft & hard solutions, the hardware will only support one of them;
// Sweep module with hard-decode support
boolean isSupportHW = mScannerDecode.isSupportHWScanner();

// start & end must occur in pairs, one start, one end, no cross-calls (e.g. start twice);
// Start scanning, with a listener that receives the scanned content when it is available.
// Continuous scanning process, no need to end + restart, just wait and listen;
// End scanning when you don't need it.

/* Hard-decode scanning head */
private void initHWScannerDecodeListener() {
    mScannerDecode.startScanCode(new ScannerDecode.ScannerValueListener() {
        @Override
        public void onScannerValue(String value) {
            // String value is the result of the scan, you don't need to retrigger here, just keep waiting for the value callback.
            Log.d(TAG, "onScannerValue value: " + value);
        }
```

```
    }).
}

// Close the scanner module (to be called after use)
mScannerDecode.stopScanCode()
mScannerDecode = null; //recommended to be released
```

meter (e.g. for meters)

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

// Initialize, e.g. in onCreate.
mMeterCounter = new MeterCounter(this);

// Initialize the meter, with listener
mMeterCounter.initMeterCounter(new MeterCounter.MeterCounterListener() {
    @Override
    public void onDeviceConnect(int status) {
        if (MeterCounter.CONNECT_STATUS_CONNECT == status) {
            // Meters connected
        } else if (MeterCounter.CONNECT_STATUS_DISCONNECT == status) {
            // Meters disconnected (USB terminal disconnected)
        }
    }

    @Override
    public void onDataReport(String value) {
        // status value mMeterCounter.getStatusCodeFromReportString(value)
        // count type mMeterCounter.getCountTypeFromReportString(value)
        // shift mMeterCounter.getClassIdFromReportString(value)
        //count value mMeterCounter.getCountValueFromReportString(value)
        // speed value mMeterCounter.getSpeedValueFromReportString(value)
        }
    }).
}

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

// Release meter (to be called after use)
if (null ! = mMeterCounter) {
    mMeterCounter.deinitMeterCounter();
}
mMeterCounter = null; //recommended release
```