Watch Payment SDK - HarmonyOS Lite

HarmonyOS Lite Pay SDK Introduction

Basic abbreviations and definitions

Term	Description
APDU	Application Protocol Data Unit exchanged between a payment terminal and the HCE service.
CDCVM	Consumer Device Cardholder Verification Method.
EMV	Payment-card specifications used by terminals and cards.
HCE	Host Card Emulation, enabling the application to answer terminal APDUs.
HUKS	HarmonyOS Universal KeyStore, used by the SDK for key operations.
Payment Instrument	A host application token/card linked to an SDK card ID.
POS	Point of Sale terminal.
Transaction credential	A local credential used by the payment engine during a transaction.

What is HarmonyOS Lite Pay SDK?

HarmonyOS Lite Pay SDK is a client-side payment component for HarmonyOS Lite wallet applications. It manages local card profiles and transaction credentials, establishes pairing and provisioning sessions, selects payment cards, and processes contactless payments.

The SDK is not a complete wallet application. The integrator owns the UI, HCE lifecycle, backend or companion-application transport, payment-instrument store, and HarmonyOS platform adapters.

How Huawei Pay SDK works

The SDK provides five public components:

- Crypto - adapts HUKS and CryptoFramework.
- StatusManager - SDK version and device pairing.
- SyncManager - secure session, profiles, credentials, and deletion.
- PaymentInstance - payment dependencies and application callbacks.
- PaymentManager - card selection and APDU processing.

HarmonyOS Lite page-size limitation

For HarmonyOS Lite, divide SDK bootstrap work across dedicated pages because the platform constrains the size of an individual page. Do not concentrate all SDK setup in one startup page: prepare Crypto, PaymentInstance, PaymentManager, SyncManager, and StatusManager in a sequential page chain.

This is a recommended HarmonyOS Lite integration architecture, not a public SDK limitation. You may use different page names or UI structure, provided that the SDK dependency and initialisation order are retained.

Versioning and backward compatibility

The runtime version is available through StatusManager.getSdkVersion().

Adopt semantic versioning for released artifacts:

Version part	Meaning
Major	Public API or behaviour change requiring integration changes.
Minor	Backward-compatible functionality, such as an optional method or result field.
Patch	Internal or defect change not requiring application-code changes.

Technical overview

SDK basic configuration

The host application must provide:

- A file-system adapter compatible with @system.file.
- HUKS through @ohos.security.huks.
- CryptoFramework through @ohos.security.cryptoFramework.
- An NFC HCE service that receives terminal APDUs and transmits SDK responses.
- An authorised backend or companion application for pairing and provisioning.
- Optional non-sensitive logging adapters.

The SDK is delivered as five ES modules: `SdkDependencies`, `SdkInit`, `SdkPayment`, `SdkStatus`, and `SdkSync`. Each module exposes one public SDK component.

Application architecture and responsibilities

Area	SDK responsibility	Integrator responsibility
Key management	Uses supplied HUKS/CryptoFramework services.	Supplies genuine platform services and protects the application environment.
Local data	Stores and deletes encrypted card-related data.	Supplies file-system access and storage lifecycle.
Pairing	Creates pairing data and imports the encrypted result.	Authenticates the remote party and transports pairing messages.
Provisioning	Consumes encrypted profile and credentials payloads.	Obtains, authorises, and routes payloads.
Contactless payment	Selects cards and processes supported APDUs.	Runs HCE, forwards APDUs unchanged, and transmits responses.
UX and operations	Emits transaction callbacks and errors.	Shows user states, retries, reporting, card management, and support.

SDK setup

For HarmonyOS Lite, organise SDK setup as five lightweight pages to stay within per-page size constraints. Each page configures one SDK component and then navigates to the next page. Store the configured SDK instances in shared application state: `crypto`, `paymentInstance`, `paymentManager`, `syncManager`, and `statusManager`.

Page 1 - init_payment_dependencies

```
import huks from '@ohos.security.huks';
import cryptoFramework from '@ohos.security.cryptoFramework';

export default {
  onInit() {
    const paymentKit = getApp().data.paymentKit;

    paymentKit.crypto = new Crypto()
      .setHuks(huks)
      .setCryptoFramework(cryptoFramework)
      .initialize();

    navigate('init_payment_instance');
  }
};
```

Page 2 - init_payment_instance

```
import File from '@system.file';

export default {
  onInit() {
    const paymentKit = getApp().data.paymentKit;

    paymentKit.paymentInstance = new PaymentInstance()
      .setFileSystem(File)
      .setTransactionEventListener(paymentListener)
      .setCrypto(paymentKit.crypto);

    navigate('init_payment_manager');
  }
};
```

paymentListener must implement onContactlessPaymentCompleted, onContactlessPaymentAborted, and onTransactionStopped.

Page 3 - init_payment_manager

```
export default {
  onInit() {
    getApp().data.paymentKit.paymentManager = new PaymentManager();

    navigate('init_payment_sync_manager');
  }
};
```

Page 4 - init_payment_sync_manager

```
import File from '@system.file';

export default {
  onInit() {
    const paymentKit = getApp().data.paymentKit;

    paymentKit.syncManager = new SyncManager()
      .setFileSystem(File)
      .setCrypto(paymentKit.crypto);

    navigate('init_payment_status_manager');
  }
};
```

Page 5 - init_payment_status_manager

```
export default {
  onInit() {
    const paymentKit = getApp().data.paymentKit;

    paymentKit.statusManager = new StatusManager()
      .setCrypto(paymentKit.crypto)
      .initialize();

    if (paymentKit.statusManager.isDevicePaired()) {
      paymentKit.paymentInstance.initialize();
```

```
      paymentKit.paymentManager
        .setInitializedPaymentInstance(paymentKit.paymentInstance)
        .initialize();
      paymentKit.syncManager.initialize();
    }

    navigate('next_application_page');
  }
};
```

When the device is not paired, run the pairing flow first. After StatusManager.pairDevice(...) succeeds, initialise PaymentInstance, PaymentManager, and SyncManager using the same sequence shown in Page 5.

Configuration methods return the same instance, so they can be called in a chain. Operational methods throw an error when invoked before the component has been initialized.

HCE integration

The host application owns the HCE service. It selects a card before terminal exchange and forwards every received APDU to PaymentManager.

```
const result = paymentManager.selectForPayment(cardId);
if (result.unusedCredentialsCount === 0) {
  // Trigger authorised credential replenishment instead of local payment.
}

function onHceCommand(apduBytes) {
  const request = new Int8Array(apduBytes.length);
  request.set(apduBytes);

  const response = paymentManager.processApdu(request);
  if (response && response.length > 0) {
    hceService.transmit(Array.from(response));
  }
}
```

The SDK dispatches SELECT, READ RECORD, GET PROCESSING OPTIONS, and GENERATE AC payment APDUs. Do not modify request or response bytes.

HCE configuration

HCE is configured by the HarmonyOS Lite host application. It is required to deliver contactless APDUs to PaymentManager, but it is not configured by the SDK itself.

Required HarmonyOS permissions

HCE configuration consists of the ability declaration in *config.json* and the runtime application name and AID list. Keep the AID values identical in both places.

```
{
  "app": {
    "bundleName": "[APP_PACKAGE_NAME]"
  },
  "module": {
    "deviceType": [
      "liteWearable"
    ],
    "reqPermissions": [
      {
        "name": "ohos.permission.NFC_TAG",
        "reason": "Card payments",
        "usedScene": {
          "ability": [
            "MainAbility"
          ],
          "when": "always"
        }
      },
      {
        "name": "ohos.permission.NFC_CARD_EMULATION",
        "reason": "Card payments",
        "usedScene": {
          "ability": [
            "MainAbility"
          ],
          "when": "always"
```

```
    }
  }
],
"abilities": [
  {
    "skills": [
      {
        "entities": [
          "ohos.nfc.cardemulation.action.HOST_APDU_SERVICE"
        ],
        "actions": [
          "ohos.nfc.cardemulation.action.HOST_APDU_SERVICE"
        ]
      }
    ],
    "metaData": {
      "customizeData": [
        {
          "name": "paymentAid",
          "value": "325041592E5359532E4444463031"
        },
        {
          "name": "paymentAid",
          "value": "A0000000041010"
        },
        {
          "name": "otherAid",
          "value": "A0000000042203"
        },
        {
          "name": "otherAid",
          "value": "A0000000043060"
        },
        {
          "name": "otherAid",
          "value": "A0000000049100"
        }
      ]
    },
```

```
        "name": ".MainAbility",
        "srcLanguage": "js",
        "srcPath": "MainAbility",
        "type": "page"
      }
    ]
  }
}
```

HCE service configuration

Configure the host HCE service with the application name and AID list from the HCE configuration above. The complete example below creates the service, starts it, handles the hceCmd event, forwards APDUs to PaymentManager, transmits the response, and stops the service.

config.js

```
export const appName = '[APP_PACKAGE_NAME]';

export const aidList = [
  '325041592E5359532E4444463031', // PPSE: 2PAY.SYS.DDF01
  'A0000000041010',
  'A0000000042203',
  'A0000000043060',
  'A0000000049100'
];
```

```
import cardEmulation from '@ohos.nfc.cardEmulation';
import { appName, aidList } from './config';



// Create one service instance for the application lifecycle.
const hceService = cardEmulation
  ? new cardEmulation.HceService()
  : undefined;

export function startHce(paymentManager) {
  if (!hceService) {
    throw new Error('HCE service is not available on this device.');
  }
```

```
  // Register this application and all supported AIDs for HCE.
  hceService.start(appName, aidList);

  // Receive an APDU command from the payment terminal.
  hceService.on('hceCmd', apduBytes => {
    const apdu = new Int8Array(apduBytes.length);
    apdu.set(apduBytes);

    // PaymentManager must already be initialised and a card selected.
    const response = paymentManager.processApdu(apdu);
    if (!response || response.length === 0) {
      return;
    }

    // Send the SDK response APDU back to the terminal.
    hceService.transmit(Array.from(response), error => {
      if (error) {
        // Apply non-sensitive application error handling.
      }
    });
  });
}

export function stopHce() {
  if (hceService) {
    hceService.stop(appName);
  }
}
```

Call startHce(paymentManager) during the application lifecycle after PaymentManager is available. Call stopHce() when the host application is destroyed.

SDK usage

Domains

Domain	Component	Main responsibilities
Dependencies	Crypto	Platform cryptography setup.
Status	StatusManager	SDK version, pairing state, pairing data, and pairing completion.
Synchronisation	SyncManager	Secure session, profile/credentials synchronisation, status, removal, and reset.
Payment	PaymentInstance, PaymentManager	Payment setup, callbacks, card selection, and APDU processing.

Error handling

SDK methods throw JavaScript Error objects. Catch errors at the application boundary, present safe user states, and log only a non-sensitive code or correlation ID.

Code	Condition	Recommended response
e56	Required dependency missing during initialisation.	Treat as an integration/configuration failure; stop the flow.
e74	A method requiring initialisation was called too early.	Complete the setup lifecycle before retrying.
e73	Pairing was requested without required local pairing material.	Restart pairing from getDevicePairingData().
e13	A credentials payload does not contain a card ID.	Reject the response and obtain a valid payload.

File-system, cryptography, validation, and APDU layers can also surface underlying errors. Do not expose raw exception text or sensitive values to users.

Dependencies domain

Crypto

Method	Input	Result	Description
setHuks(huks)	HUKS adapter	this	Sets the required HUKS service.

setCryptoFramework(cryptoFramework)	CryptoFramework adapter	this	Sets the required CryptoFramework service.
initialize()	-	this	Creates the cryptographic adapter; throws e56 when required dependency is missing.

Status domain

StatusManager

Method	Input	Result	Description
setCrypto(crypto)	Configured Crypto	this	Sets the required cryptography provider. The supplied Crypto instance must already be initialised.
initialize()	-	this	Activates pairing operations. Throws e56 when no cryptography provider was supplied; may propagate e74 when the supplied Crypto instance is not initialised.
getSdkVersion()	-	string	Returns runtime version. May be called before initialisation.
isDevicePaired()	-	boolean	Returns whether required local pairing material is available.
getDevicePairingData()	-	Base64 string	Creates missing pairing material when needed and returns opaque data for the remote party.
pairDevice(encryptedKeyBase)	Opaque Base64 string	void	Imports encrypted pairing result; throws e73 if flow was not prepared locally.

Pairing flow

```
const pairingData = statusManager.getDevicePairingData();
// Send pairingData to the authorised backend or companion application.
// Receive encryptedPairingResult through the authenticated channel.
statusManager.pairDevice(encryptedPairingResult);
```

```
paymentInstance.initialize();
paymentManager.setInitializedPaymentInstance(paymentInstance).initialize();
syncManager.initialize();
```

Treat pairing input and output values as opaque Base64. Never parse, modify, log, or expose them in the UI.

Synchronisation domain

SyncManager

Method	Input	Result	Description
setFileSystem(fs)	File-system adapter	this	Sets required storage adapter.
setCrypto(crypto)	Configured Crypto	this	Sets required cryptography dependency.
initialize()	-	this	Activates secure local data access; throws e56 without file system or crypto.
getSessionKey(rsaComponentsBase64)	Opaque Base64 string	Base64 string	Establishes the current secure synchronisation session and returns an opaque response for the remote party. Call after pairing and before each encrypted synchronisation payload.
syncProfileData(syncProfileDataBase64)	Opaque Base64 string	{ cardId: string }	Decrypts and stores the card profile using the current session, then deletes the session keys.
syncTransactionCredentials(syncTransactionCredentialsBase64)	Opaque Base64 string	{ cardId: string, credentialsCount: number }	Decrypts and replaces the card's credentials using the current session, then deletes the session keys. Throws e13 when the payload has no card ID.
getUsedCredentials()	-	UsedCredentialsResult	Returns the number of unused credentials per card and an opaque Base64 credential-status payload for the remote party.

deleteCard(cardId)	SDK card ID	void	Deletes the selected card's stored profile, credentials, associated cryptographic keys, and local data directory. It does not unpair the device.
clear()	-	void	Deletes every card and its associated keys, then removes wallet storage and internal, session, pairing, and signing keys. Pairing and provisioning must be repeated afterwards.

Secure-session and provisioning flow

```
const profileSessionResponse = syncManager.getSessionKey(profileRsaComponentsBase64);
// Return profileSessionResponse through the authenticated channel.
const { cardId } = syncManager.syncProfileData(encryptedProfileBase64);



const credentialsSessionResponse = syncManager.getSessionKey(
  credentialsRsaComponentsBase64
);
// Return credentialsSessionResponse through the authenticated channel.
const { credentialsCount } = syncManager.syncTransactionCredentials(
  encryptedCredentialsBase64
);
// Persist cardId against the host payment instrument.
```

Synchronise the profile before credentials. Create a new secure session with `getSessionKey()` before each encrypted payload. Treat every provisioning value as opaque Base64 data.

Credentials status, deletion, and reset

```
const status = syncManager.getUsedCredentials();
// status.cardInfos: [{ cardId, credentialsCount }]
// status.usedCredentialsInfo: opaque Base64 status for the remote party

syncManager.deleteCard(cardId); // One card
syncManager.clear(); // Explicit user-confirmed unpair/logout/reset only
```

clear() is destructive. It deletes every stored card profile and credential, then removes local wallet and pairing keys. The device must be paired and provisioned again afterwards.

Payment domain

PaymentInstance

Method	Input	Result	Description
setFileSystem(fs)	File-system adapter	this	Sets required local storage.
setTransactionEventListener(listener)	TransactionEventListener	this	Sets required payment callbacks.
setCrypto(crypto)	Configured Crypto	this	Sets the cryptography dependency. Calls crypto.getCrypto() immediately, so it throws e74 when Crypto has not been initialised.
initialize()	-	this	Builds payment engine; requires file system, listener, and crypto or throws e56.

PaymentManager

Method	Input	Result	Description
setInitializedPaymentInstance(instance)	Initialised PaymentInstance	this	Attaches the payment engine and configures its cryptographic-methods bridge. Required before initialisation; passing an uninitialised instance throws e74
initialize()	-	this	Initialises payment processing. Call after attaching PaymentInstance and before payment operations. Throws e56 when no instance is attached.
selectForPayment(cardId)	SDK card ID	{ unusedCredentialsCount: number }	Selects and prepares the card for the next contactless exchange. Call before processing APDUs. Throws e74 when no PaymentInstance is attached..

processApdu(apdu)	Int8Array	Int8Array or undefined	Processes a terminal APDU for the selected card. Returns undefined when no card is selected; throws e74 when no PaymentInstance is attached.

Transaction event listener

```
const paymentListener = {
  onContactlessPaymentCompleted(card, log) {
    const cardId = card.getCardId();
    const outcome = log.getTransactionOutcome();
  },
  onContactlessPaymentAborted(card, abortReason, error) {
    // Show a safe aborted state; record non-sensitive diagnostics.
  },
  onTransactionStopped() {
    // Clear transient payment UI state.
  }
};
```

Callback	Parameters	Purpose
onContactlessPaymentCompleted	Card, ContactlessLog	Signals completed payment and provides transaction data.
onContactlessPaymentAborted	Card, abort reason, Error	Signals payment abort.
onTransactionStopped	-	Signals that current transaction stopped.

Models

Core result models

```
type SelectForPaymentResult = { unusedCredentialsCount: number };
type SyncProfileResult = { cardId: string };
type SyncCredentialsResult = { cardId: string, credentialsCount: number };
type UsedCredentialsResult = {
  cardInfos: Array<{ cardId: string, credentialsCount: number }>;
```

```
  usedCredentialsInfo: string; // opaque Base64 value
};
```

Payment-event models

```
interface Card { getCardId(): string; }
interface ContactlessLog {
  getTransactionOutcome(): TransactionOutcome;
  getTerminalInformation(): TerminalInformation;
  getTransactionInformation(): TransactionInformation;
  getTransactionId(): Int8Array;
}
interface TerminalInformation {
  getTerminalType(): TerminalType;
  getMerchantAndLocation(): Int8Array;
}
interface TransactionInformation {
  getCurrencyCode(): Int8Array;
  getAuthorizedAmount(): Int8Array;
  getOtherAmount(): Int8Array;
  getTransactionRange(): 'LOW_VALUE' | 'HIGH_VALUE' | 'UNKNOWN';
  getRichTransactionType():
    | 'PURCHASE' | 'REFUND' | 'CASH' | 'TRANSIT'
    | 'PURCHASE_WITH_CASHBACK' | 'UNKNOWN';
  getExpectedUserActionOnPoi():
    | 'NONE' | 'ONLINE_PIN' | 'SIGNATURE'
    | 'ONLINE_PIN_OR_SIGNATURE' | 'UNKNOWN';
  getPurpose(): 'AUTHORIZE' | 'AUTHENTICATE' | 'UNKNOWN';
  getConditionsOfUse(): 'DOMESTIC' | 'INTERNATIONAL' | 'UNKNOWN';
  hasTerminalRequestedCdCvm(): boolean;
}
```

Transaction outcomes: AUTHORIZE_ONLINE, AUTHENTICATE_OFFLINE, DECLINE_BY_TERMINAL, DECLINE_BY_CARD, WALLET_ACTION_REQUIRED.

Abort reasons: WALLET_CANCEL_REQUEST, CARD_ERROR, TERMINAL_ERROR.

TerminalType returns BANK_ATTENDED_, *BANK_UNATTENDED_*, MERCHANT_ATTENDED_, *MERCHANT_UNATTENDED_*, CARDHOLDER_OPERATED_*, or UNKNOWN.

Amounts, currency code, merchant/location, and transaction ID are Int8Array values. Preserve bytes until decoding them according to the applicable EMV/protocol rules.

HarmonyOS Lite SDK initialisation pages

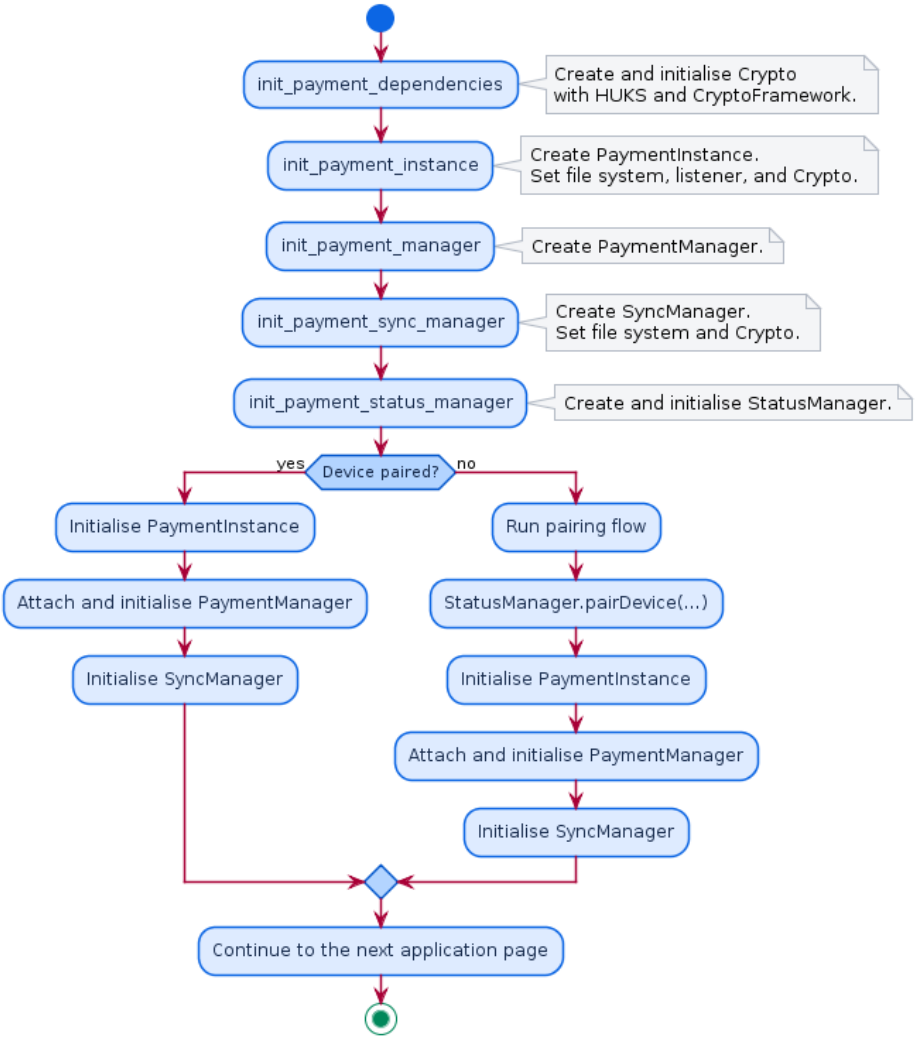
Split the SDK bootstrap flow across dedicated pages to keep individual pages within HarmonyOS Lite page-size constraints and to isolate the setup responsibility of each SDK component.

Only SDK-related pages are documented here. Other application pages and services are host-application concerns and are not part of the SDK contract.

Keep the shared SDK instances in `getApp().data`, rather than in page-local state, so every initialisation page and payment flow uses the same objects. Store `crypto`, `paymentInstance`, `paymentManager`, `syncManager`, and `statusManager` there.

Order	SDK initialisation page	SDK responsibility
1	pages/init_payment_dependencies	Creates and initialises Crypto with HUKS and CryptoFramework.
2	pages/init_payment_instance	Creates PaymentInstance; supplies file system, configured Crypto, and payment event listener.
3	pages/init_payment_manager	Creates PaymentManager.
4	pages/init_payment_sync_manager	Creates SyncManager; supplies file system and configured Crypto.
5	pages/init_payment_status_manager	Creates and initialises StatusManager. If the device is paired, initialises PaymentInstance, attaches and initialises PaymentManager, then initialises SyncManager.

You may use a different page structure or page names, but retain this dependency order:



Revision #1
Created 27 July 2026 12:39:48 by Beata Rzemieniak
Updated 27 July 2026 12:47:19 by Beata Rzemieniak