The only guide you need to integrate your payment terminals with FUNDiMO. It goes from a sandbox key to terminals that take payments for real merchants. It is for a terminal fleet vendor's developer.

The stages

Stage	What it is	What you need
1. Integrate	Section 1, in the sandbox. Build your terminal's app until a conformance run is green	A sandbox key: one call, with no sign-up Help: an email to `support@fundimo.com.au`
2. Enrol	Section 2, on staging, then on production. You register your app and list your terminals. An installer enrols each terminal	A first pairing code from FUNDiMO, and your device models allowed by FUNDiMO. You cannot enrol alone Help: your Slack Connect channel
3. Link	Section 3, at the counter. A register links to your terminal. The POS vendor does the linking	Your terminal enrolled and connected. The cashier, with your terminal in their hand Help: your Slack Connect channel

The environments

	Sandbox	Staging	Production
Address	`https://mw.sandbox.fundimo.com.au`	`https://mw.staging.fundimo.com.au`	`https://mw.fundimo.com.au`
For	Integrate. Your terminal beside simulated terminals. No card, and no money	Enrol and link, to test. Your real terminals, on test cards only. Never use a real card	Enrol and link, for real merchants
You need	A sandbox key: one call, with no sign-up	Your terminal fleet. Your app registered, and your device models allowed. A certificate that staging issued, for each terminal	Your terminal fleet at your merchants' stores. A certificate that production issued, for each terminal

Contents

The system **3**

1 Integration **5**
1.1 Take a sandbox key 5
1.2 Connect your terminal 6
1.3 Get an instruction 7
1.4 Send the outcome 8
1.5 Check the two sides 10
1.6 Do a refund 11
1.7 Get the result of an unknown payment 12
1.8 Send again after a drop, and respond to a `StatusRequest` 13
1.9 Stop a payment on an `Abort` 15
1.10 Enrol your terminal for its certificate 16
1.11 Run the conformance scenarios, and keep the report 19

2 Enrolment **21**
2.1 Register your app, and get your device models allowed 22
2.2 List each terminal at its store, and get its pairing code 23
2.3 The installer enrols each terminal 23
2.4 Connect on the certificate 24
2.5 Automate the renewal of every certificate 24
2.6 Enrol again with a new pairing code, when a person must 25

3 Link **26**
3.1 Show the link words 26
3.2 Monitor the links at your stores 27

What green conformance means for your terminal software **28**
What your terminal can send 28

The sandbox's MCP server **29**
What it offers 29
An example 30

Get help **32**

The pages **33**

Addendum: the ISO 20022 terms in this guide **34**
Fields 34
Codes 35
FUNDiMO's own names 36

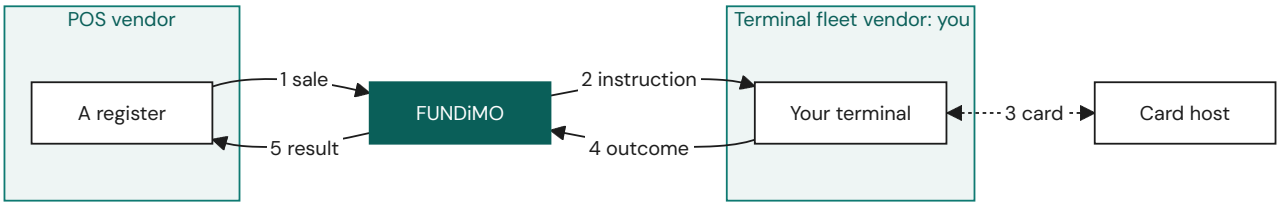
The system

FUNDiMO is the relay between a point of sale (POS) and a payment terminal. Three players take part in each sale:

Player	What it runs	What it holds
The terminal fleet vendor: you	The payment terminals at each store, and your app on each terminal. A terminal holds a WebSocket connection to FUNDiMO while it is on. It takes the card with its own card host.	The card and the payment, between your terminal and its host. The private key of each terminal, in its keystore. You list your terminals at each store. You give the merchant the store's `MerchantIdentification`.
The POS vendor	The POS software at each register. It runs on the vendor's own server, or as an app on a device. Each register sends its sales to FUNDiMO, and reads their results.	The bill, its items and the store's own surcharge. Its own reference for each sale.
FUNDiMO	The relay between the two. Each register calls FUNDiMO over HTTPS. Each terminal holds a WebSocket connection open to FUNDiMO.	Each sale, its status and the references that your terminal sent. Which register is linked to which terminal. A log of every message.

A register is the POS software at one counter of a store, where a cashier takes payments. It is an app on a device, or a name that a POS vendor's server gives. This guide calls it a register everywhere.

How a sale moves. A register sends a sale to FUNDiMO. FUNDiMO sends your terminal an instruction, on the connection that your terminal holds open. Your terminal takes the payment with its own card host. Your terminal then sends an outcome, and FUNDiMO gives it to the register. What your terminal sends passes an allow-list. FUNDiMO removes anything else, and logs the removal. Card, bank and personal data are forbidden in every field.



Your terminal holds one WebSocket connection to FUNDiMO while it is on (2, 4). The register calls FUNDiMO over HTTPS (1, 5). The card goes between your terminal and its own card host (3, dotted).

A certificate for each edge. On staging and production, each part that connects to FUNDiMO identifies itself with a client certificate. FUNDiMO issues each certificate. Each terminal, each POS server and each register on a device has its own certificate. A certificate is valid for 30 days. FUNDiMO issues a new one from day 15, for a new private key. After 30 days, a certificate cannot renew. Its terminal then needs a new pairing code, which a person brings.

What FUNDiMO does with your terminal's certificate, and why it is there. It is your terminal's credential on staging and production, in place of a sandbox key.

- FUNDiMO's own CA issues it at `POST /v1/enrol`. The terminal shows three things. The pairing code proves that an authorised person is at the device now. The serial number is the one that its listing holds. The Android Key Attestation chain proves that the private key is in hardware. It also proves an unmodified device of an allowed model, and the app that you registered.
- The private key is in the Android keystore, and never goes out of it.

- The certificate names the kind of environment, `live` or `sandbox`, the role and your terminal's `POIIdentification`. Staging and production are both `live`. Your terminal presents it at the TLS handshake of its WebSocket upgrade. FUNDiMO identifies the terminal from the certificate only.
- It renews with a new private key and a new attestation. Thus a device proves its state again every 15 days. A device that is rooted after it enrolled loses its identity within one life of a certificate. Staging can mark such a device instead.
- After a renewal, the old certificate stays valid until the first use of the new one, and for one hour after it, by default. FUNDiMO then refuses the old one with `403 certificate_retired`.

Your back office's `admin` server has a certificate too, as a POS server does. It reaches the Operator API of your own vendor: your stores, terminals, servers and app identities.

What FUNDiMO trusts. FUNDiMO loads nothing of yours into its trust store. You register your app's identity: its package and the SHA-256 digest of its signing certificate. FUNDiMO checks each device's attestation chain against Google's attestation roots. These roots are not yours to give.

What your software trusts. FUNDiMO's server certificate is a public certificate. Your app checks it with the device's default trust store, as any HTTPS client does. Never disable that check.

Thus our guidance is: automate the renewal. Your app renews its terminal's certificate automatically, on every terminal. Your back office's server renews its own certificate. Use one renewer for each certificate (step 2.5).

SECTION 1

Integration

The sandbox is FUNDiMO's environment for integration. It is self-help: one call gives you a sandbox key and a test world, and you do not sign up. Nothing in the sandbox is a record. No card is charged, and no money moves. Your test world is valid for 30 days. The sandbox has no card host for your terminal. Your terminal and its own host decide each outcome. Use the host's test environment, or a test host of your own. This section ends when a conformance run is green.

What you need. Your terminal's app, and a shell with `curl` for the sandbox's calls. The pages show an Android app, with Kotlin for each step. The Kotlin uses OkHttp, the Android keystore and Bouncy Castle's PKIX library. FUNDiMO publishes no library of its own.

The address is a setting. Keep the address in one setting from the first line of code. Make every address from that setting, the WebSocket's address too. Never write an address into code. In this section, the commands read the sandbox's address from `$SANDBOX`:

```
SANDBOX=https://mw.sandbox.fundimo.com.au
```

The fields that your terminal uses. FUNDiMO's pages use ISO 20022's names. Your code uses them too. The addendum gives the definition of each term (Addendum).

Field	What it is
`POIIdentification`	Your terminal's id. FUNDiMO issues it when you list the terminal: `poi_` and 32 hex characters. In the sandbox, your terminal's id is `<tester>-yours`.
`TransactionReference`	One sale or refund: `sal_` and 32 hex characters. Keep it. Send it back in the outcome, with no change.
`CashierIdentification`	The register's id for the user who made the sale or the refund, for example a staff number. It is present only when the register sent one. Show it, print it or pass it on, or ignore it. The payment does not change.
`TotalAmount`	The amount to charge, or to refund. It is a string with two decimals, for example `"12.50"`, in `AUD`.
`TransactionStatus`	What your outcome says happened: `ACSP` or `RJCT`.
`AuthorisedAmount`	In an `ACSP` outcome, in `TransactionDetails`: the amount that the host approved. It is the instruction's `TotalAmount`, with no change.
`PaymentReference`	The references in your outcome, for example the authorisation code. They are in groups, by the kind of payment.

1.1 Take a sandbox key

Send one call. It needs no sign-up and no credential:

```
curl -s -X POST "$SANDBOX/sandbox/v1/keys"
```

Response: `201`, and a test world of your own:

- `SandboxKey`: your sandbox key, `sbx_` and 43 more characters. FUNDiMO shows it one time only. Keep it out of logs and repositories.
- `TesterIdentification`, for example `sbx-3f9a1c2e7b10`. It is not a secret. The names of your terminals start with it.
- `Terminals`: nine terminals. `<tester>-yours` is the terminal that your terminal connects as. Nothing in the sandbox holds it. You can operate `<tester>-manual` through the MCP server.

- `MerchantIdentification`, your world's store.
- `ExpiryDateTime`: 30 days later.

Refusals

Status	Code	What to do
429	`sandbox_key_rate`	The sandbox issued all the sandbox keys that it can this hour. Try again in one hour.
503	`sandbox_full`	Sandbox keys become free when they expire. Try again in one hour.
503	`store_unavailable`	FUNDiMO made no sandbox key. Retry with the normal backoff: wait 1 second, and double the wait after each failure. The maximum wait is 30 seconds. Add a small random time to each wait.

Before you write terminal code, your AI assistant can try the frames on `-manual` (The sandbox's MCP server).

1.2 Connect your terminal

Open a WebSocket to your terminal's path. Use the sandbox's address, with its scheme changed to `wss://`. Put your own `TesterIdentification` in the path. Put your sandbox key on the upgrade. On every upgrade, also declare what your terminal does:

- In `Mw-Transaction-Types`, `RFND` when your terminal does refunds (step 1.6).
- In `Mw-Message-Functions`, `SSMQ` when it responds to a `StatusRequest` (step 1.8).
- In `Mw-Message-Functions`, `SSAB` when it responds to an `Abort` (step 1.9).

```
wss://mw.sandbox.fundimo.com.au/v1/terminals/sbx-3f9a1c2e7b10-yours/connect
Authorization: Bearer sbx_...
Mw-Transaction-Types: CRDP, RFND
Mw-Message-Functions: SSAB, SSMQ
```

FUNDiMO keeps what an upgrade declared for that connection only. Thus send the two headers every time. Declare a code only when your terminal does what the code names. A terminal that declares nothing gets only sales. It gets no `StatusRequest` and no `Abort`.

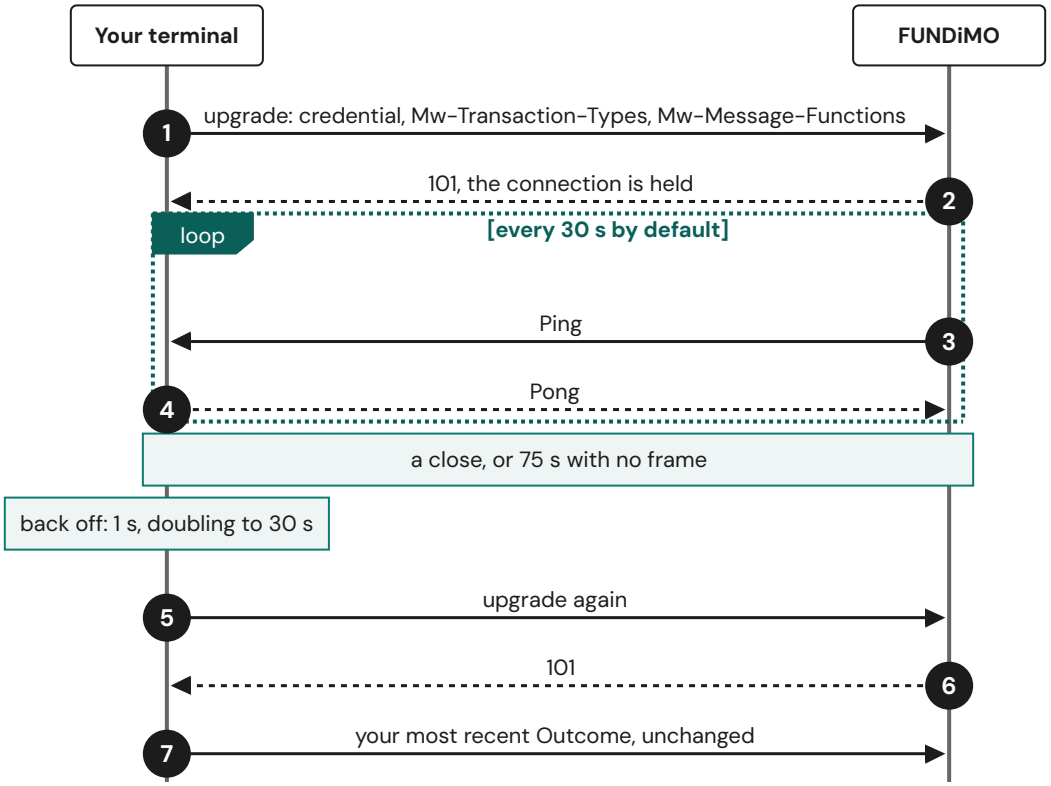
Each message, in the two directions, is a text frame with one JSON object. The maximum size is 65,536 bytes.

Response: `101`: the connection is upgraded. Hold it while your terminal is on. By default, FUNDiMO sends a WebSocket Ping every 30 seconds. Respond to each Ping with a Pong. If no frame comes from FUNDiMO for 75 seconds, close and connect again. Some libraries respond to Pings themselves and never show them, for example OkHttp. With such a library, set its own ping: `pingInterval(30, TimeUnit.SECONDS)`.

After any close, connect again with the normal backoff (step 1.1). Reset the wait only when a connection stays open for 10 seconds. FUNDiMO closes connections itself. Thus your terminal must always connect again.

Refusals

Status	Code	What to do
401	certificate_required	No credential came. Send the sandbox key on the upgrade.
401	sandbox_key_invalid	Check the sandbox key. Wait one hour before the next connection. Keep any outcome that you hold.
403	not_permitted	The sandbox key cannot hold this terminal. Wait one hour.
426	upgrade_required	A GET with no Upgrade header. Correct the client.
503	store_unavailable	FUNDiMO gave no decision on your sandbox key. Connect again with the backoff.



A terminal's connection. Pings keep it open (3, 4). After a close, your terminal waits, connects again (5, 6), and sends its most recent outcome again (7).

1.3 Get an instruction

Send a sale to your terminal, as a register does. Use the same sandbox key, and a new IdempotencyKey each time:

```
curl -s -X POST "$SANDBOX/v1/sales" \
  -H 'Authorization: Bearer sbx_...' \
  -H 'Content-Type: application/json' \
  -d '{"MerchantIdentification": "sbx-3f9a1c2e7b10-M1",
       "POIIdentification": "sbx-3f9a1c2e7b10-yours",
       "SaleReferenceIdentification": "INV-1001",
       "IdempotencyKey": "test-sale-0001",
       "TransactionDetails": {"Currency": "AUD", "TotalAmount": "12.50",
         "SaleItem": [{"ProductDescription": "Flat white", "ProductQuantity": 2,
                       "UnitPrice": "6.25", "ProductAmount": "12.50"}]}}'
```

Response: 202 to the call, with the sale's TransactionReference. Your terminal gets the instruction:

```
{ "MessageType": "Instruction",
  "TransactionReference": "sal_01970f6e7c3a7d3e8a4f2b1c9d0e1f2a",
  "SaleReferenceIdentification": "INV-1001",
  "TransactionDetails": { "Currency": "AUD", "TotalAmount": "12.50",
    "SaleItem": [ { "ProductDescription": "Flat white", "ProductQuantity": 2,
                    "UnitPrice": "6.25", "ProductAmount": "12.50" } ] } }
```

Do these steps:

1. Store its `TransactionReference` first. Store it where a crash or a reboot does not lose it. Keep it for 24 hours or more (step 1.8).
2. Charge `TotalAmount`. Show it as the total, whatever the items say.
3. Show the items if your terminal can. Ignore them if it cannot.

Ignore any key, and any frame, that you do not know.

Add nothing to `TotalAmount`. Add no surcharge, tip or fee. This includes your terminal's own, and any that its merchant settings would apply. Thus disable those settings for a payment that FUNDiMO instructs. A store's own surcharge on the bill, for example 15% on a public holiday, is already in `TotalAmount`. The register added it before it sent the sale. Never add it again.

FUNDiMO sends each instruction one time. If the connection drops, FUNDiMO does not send it again. On a later connection, FUNDiMO asks about it with a `StatusRequest` (step 1.8). An instruction can come again with a `TransactionReference` that your terminal has already started. Do not charge again. Send its stored outcome.

One payment at a time. Sometimes an `Instruction` comes while your terminal is in a payment. Do not start it, and do not put it in a queue. Store `RJCT` with `AB11` as the outcome of its `TransactionReference`, and send that outcome. No payment was taken for it. A payment that your terminal cannot resolve yet keeps the terminal busy (step 1.7).

Refusals

Status	Code	What to do
409	`terminal_offline`	Your terminal is not connected to `<tester>-yours`. Connect it. Then send the same sale with the same idempotency key.

1.4 Send the outcome

When the payment ends, make the `Outcome` with the instruction's `TransactionReference`. Store the outcome. Then send it:

```
{ "MessageType": "Outcome",
  "TransactionReference": "sal_01970f6e7c3a7d3e8a4f2b1c9d0e1f2a",
  "TransactionDateTime": "2026-11-06T15:12:40+11:00",
  "TransactionStatus": "ACSP",
  "StatusReasonInformation": { "ResponseCode": "00", "AdditionalInformation": "APPROVED" },
  "TransactionDetails": { "AuthorisedAmount": "12.50" },
  "PaymentReference": { "Card": { "RetrievalReferenceNumber": "631012345678",
    "SystemTraceAuditNumber": "000123", "AuthorisationCode": "A1B2C3",
    "AcquirerPOIIdentification": "12345678", "AcquirerMerchantIdentification": "987654321",
    "RefundReference": "klujhadsc9ph87asd" } } }
```

Give the amount on every `ACSP`. Send `TransactionDetails.AuthorisedAmount`, with no other field in that group. It is the amount that the host approved. It is the instruction's `TotalAmount`, with no change, because your terminal adds nothing to it. Give the same amount each time that you send that outcome again. Do not put `TotalAmount` in the group: FUNDiMO gives the total. With `RJCT`, send no `AuthorisedAmount`. No payment was taken, and FUNDiMO removes one that you send. The register records `AuthorisedAmount` as

the amount charged. If it is different from `TotalAmount`, FUNDiMO logs that your terminal did not obey the contract.

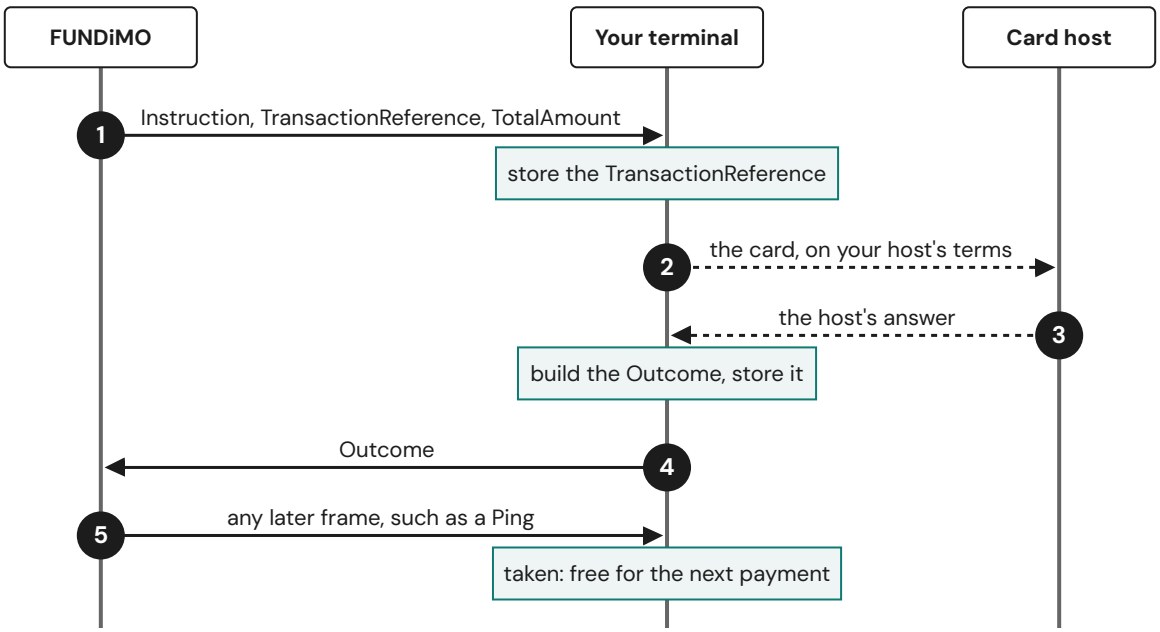
What happened	Send
The host approved the full `TotalAmount`	`ACSP`, with `AuthorisedAmount`
The card or the host declined	`RJCT`, `Reason AG07`
The customer cancelled at the terminal before anything got to the host. Or the terminal stopped before anything got to the host	`RJCT`, `Reason AB11`
An `Abort` came before the payment got to the host, and the terminal stopped the payment (step 1.9)	`RJCT`, `Reason DS02`
An `Abort` came when the payment was already with the host	What the host gives, by the other rows. Never `DS02`
The host did not respond, and a reversal confirmed no charge	`RJCT`, `Reason AB11`
The host approved, and the terminal reversed the approval. Or the host approved only part, and the terminal reversed that part	`RJCT`, `Reason AB11`
Your terminal does not know if a payment was taken	**Nothing yet** (step 1.7)

Response: nothing. FUNDiMO does not acknowledge an outcome. The outcome is received when a later frame comes on the same connection, with no `Refusal` before it. A Close does not count. A Ping is an example of a later frame. If your library never shows a Ping, the later frame is the next text frame. Your terminal is then free for its next payment.

If a `Refusal` frame comes

`ResponseReason`	What to do
`unparseable_frame`, `binary_frame`, `unknown_frame_type`, `missing_field`, `invalid_result`, `invalid_reference`, `other_terminal`	Your terminal's code has a bug. Keep the frame, and do not send it again. Tell a person.
`not_held`	The sale is past the retention time. Stop, and keep your own record. FUNDiMO's people reconcile it from the log.
`terminal_not_permitted`	Keep the outcome. Close the connection, connect again, and send the outcome again.

A `Refusal` names no frame. It is about the last frame that you sent. Thus send one frame at a time. Read any `Refusal` before you send the next frame.



An instruction and its outcome. Store the reference before the card (1). Store the outcome before you send it (4). A later frame from FUNDiMO shows that FUNDiMO received the outcome (5).

1.5 Check the two sides

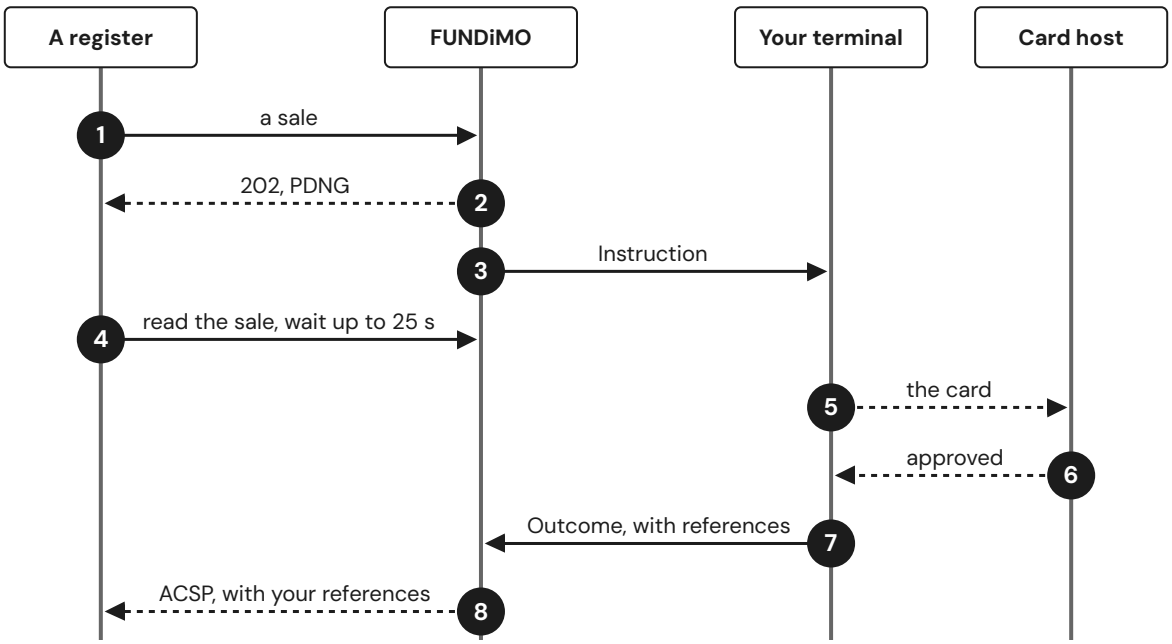
Read the sale as the register does, with your sandbox key:

```
curl -s -H 'Authorization: Bearer sbx_...' \
  "$SANDBOX/v1/sales/sal_01970f6e7c3a7d3e8a4f2b1c9d0e1f2a?wait_seconds=25"
```

Response: `200`, with your `TransactionStatus`. `TransactionDetails.AuthorisedAmount` shows the amount that you gave, next to `TotalAmount`. `PaymentReference` shows your references, in the same group and key that you sent.

Then read the sale's log with the `read_log` tool of the MCP server. The log has every message of the sale, in the two directions. A `fields_dropped` line names each field that FUNDiMO removed from your outcome, by its path. Thus you can make sure that your terminal sends nothing extra. A sale to `-approve`, a simulated terminal, is a full sale that you can compare with yours.

If a value does not match its pattern, FUNDiMO removes that value only. The outcome still applies. The log names the value by its path, and writes the value nowhere.



A sale, from start to end. The register's read (4) responds when your outcome comes (7, 8). The register prints your references on its receipt.

1.6 Do a refund

Declare refunds on every upgrade, `Mw-Transaction-Types: CRDP, RFND`. Do this only when your terminal reads `TransactionType`. A terminal that ignores it would charge a refund as a sale. Approve a sale on `-yours`. Then send a refund of that sale to `-yours`:

```
curl -s -X POST "$SANDBOX/v1/sales" \
  -H 'Authorization: Bearer sbx_...' \
  -H 'Content-Type: application/json' \
  -d '{"TransactionType": "RFND",
       "OriginalTransaction": {"TransactionReference": "sal_01970f6e7c3a7d3e8a4f2b1c9d0e1f2a"},
       "MerchantIdentification": "sbx-3f9a1c2e7b10-M1",
       "POIIdentification": "sbx-3f9a1c2e7b10-yours",
       "SaleReferenceIdentification": "CN-1001",
       "IdempotencyKey": "0b6e2f4a-9c1d-4e7f-8a3b-5c6d7e8f9a0b",
       "TransactionDetails": {"Currency": "AUD", "TotalAmount": "6.25"}}'
```

Response: your terminal gets an `Instruction` with `TransactionType` `RFND` and its own `TransactionReference`. `OriginalTransaction` holds the sale, with the references that the sale's approval had.

Do these steps:

1. Read `TransactionType` first.
2. Store the refund's reference.
3. Refund `TotalAmount` against that sale. Use the sale's references if your host can. If your host cannot, the customer presents the card.
4. Report the refund in an `Outcome`, with the refund's own reference. Never use the sale's reference.

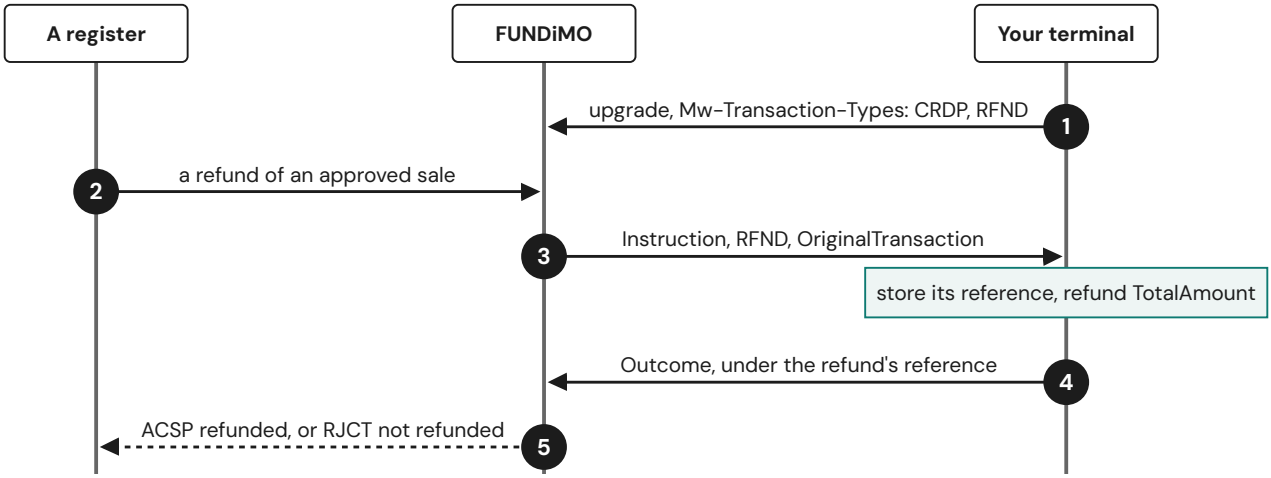
Send one of these results:

- `ACSP`, when the full amount was refunded. `AuthorisedAmount` is the amount refunded, the refund's `TotalAmount`.
- `RJCT` with `AG07`, when the host declined.
- `RJCT` with `DS02`, when your terminal stopped the refund before it got to the host, after an `Abort` (step 1.9).
- `RJCT` with `AB11`, when no refund was made for a different cause.

FUNDiMO keeps no total of the refunds of a sale. A host that has a limit declines the refund that goes over the limit. The other cases of a refund are the same as a sale's. For a refund with an unknown result, get the result as in step 1.7. For a refund that your terminal did not receive, FUNDiMO asks as in step 1.8.

Refusals

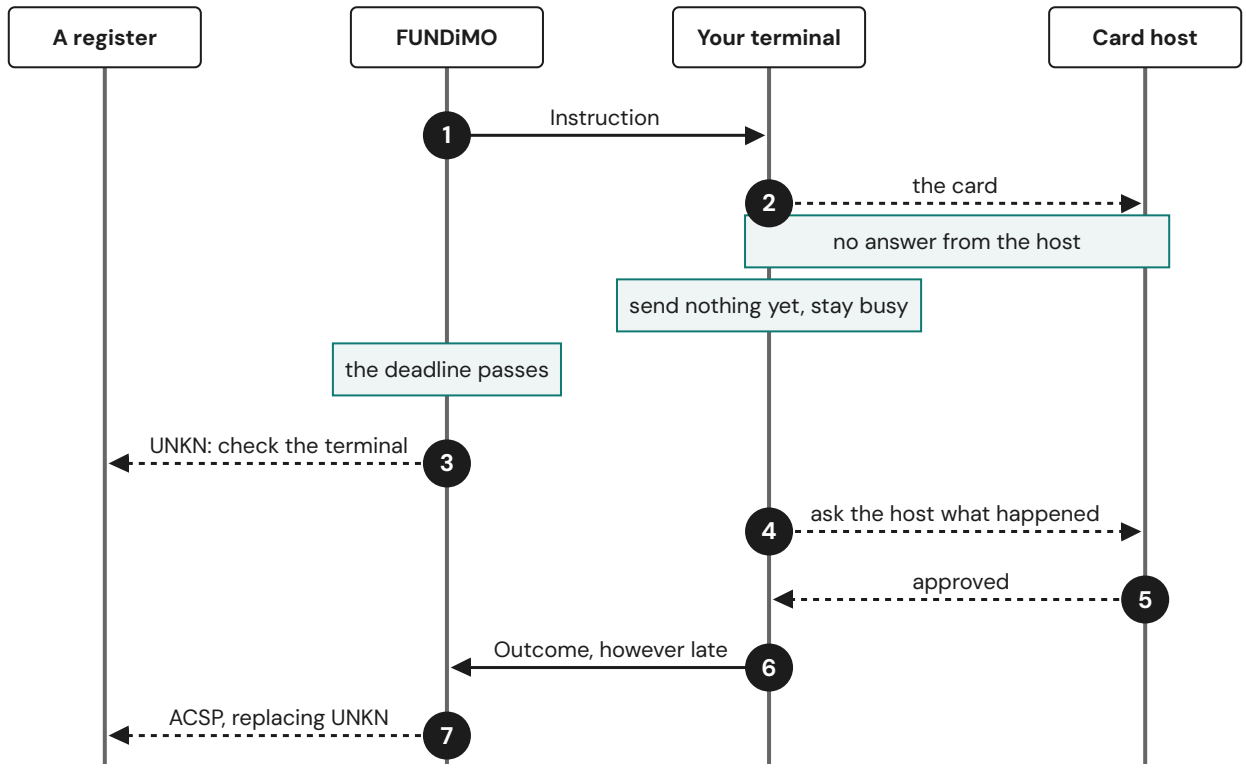
Status	Code	What to do
409	`terminal_cannot_refund`	Your terminal's connection did not declare `RFND`. Connect again with the header.



A refund. FUNDiMO sends a refund only on a connection that declared `RFND` (1, 3). Its outcome has the refund's own reference (4).

1.7 Get the result of an unknown payment

Sometimes your terminal does not know if a payment was taken, for example because the host did not respond. Your terminal must then get the result. It must never stop with no result.



An unknown outcome and its recovery. The host does not respond (2). Your terminal sends nothing, and the register reads `UNKN` at the deadline (3). Your terminal asks the host (4, 5). Its late outcome replaces `UNKN` (6, 7).

1. Send nothing for that sale yet. Your terminal is still busy. Start no other payment.
2. Ask the host for the payment, with the reference that you stored before the card. If the host approved, send `ACSP`. If the host declined, send `RJCT` with `AG07`. Send `RJCT` with `AB11` for "never received" only if the host documents that response as final.
3. Sometimes the host cannot respond to the enquiry, or has no enquiry. Sometimes it gives a "not found" that it does not document as final. In these cases, reverse the payment. Send `RJCT` with `AB11` only after the host confirms the reversal. A reversal in a queue can fail, and then the customer pays two times.
4. Try the enquiry and the reversal again until one of them gives a result. Send the result immediately, however late. It replaces the `UNKN` that the register reads after the deadline. At the default settings, a host that responds lets your terminal report before the hold ends. The hold ends 300 seconds after the instruction. A terminal that was off sends the result on its first connection after that.

Do the same for a refund. After a reversal, send `RJCT` with `AB11` only after the host confirms it.

The sandbox's deadline is 30 seconds. The contract's default is 120 seconds. Some sales take more time, for example a card with PIN entry. Such a sale reads `UNKN` at 30 seconds. Your outcome then replaces `UNKN`.

1.8 Send again after a drop, and respond to a `StatusRequest`

After each new connection, send your most recent outcome again, with no change. Do this whether or not you think that it arrived. Stop in these conditions:

- A new `Instruction` arrives.
- A `Refusal` other than `terminal_not_permitted` comes for the outcome.
- 23 hours pass after you first sent it.

FUNDiMO sends each instruction one time. Thus FUNDiMO asks about an instruction that a drop possibly lost. When your terminal responds to a `StatusRequest`, declare `SSMQ` on every upgrade (step 1.2). FUNDiMO

starts at the connection's first Ping, 30 seconds after the connection opens, by default. FUNDiMO asks about each sale and refund of your terminal with these conditions:

- FUNDiMO still holds it open, `PDNG` or `UNKN`.
- It was made in the last 23 hours, while an earlier connection held your terminal.

FUNDiMO asks about the oldest first. It asks about a maximum of 4 at each Ping. It asks about each one a maximum of one time on a connection.

```
{ "MessageType": "StatusRequest",
  "TransactionReference": "sal_01970f6e7c3a7d3e8a4f2b1c9d0e1f2a",
  "SecondsSinceInstruction": 34 }
```

`SecondsSinceInstruction` is the time in whole seconds, from the creation of the sale to this frame, by FUNDiMO's clock. Respond with a usual `Outcome`, or with nothing. What you send depends on what your terminal holds for the `TransactionReference`:

What your terminal holds	What it does
An outcome	It sends that outcome, if it did not send it already on this connection
A payment that it has not finished	It sends nothing until it knows the result. Then it sends that outcome, one time (step 1.7)
No record, and its store of references goes back more than `SecondsSinceInstruction`	It stores `RJCT` with `AB11` as the outcome of that reference, and sends it. It never starts a later `Instruction` with that reference. It sends that outcome for it, if it did not send it already on that connection
No record, and its store does not go back that far	It sends nothing. It cannot prove that it did not receive the `Instruction`

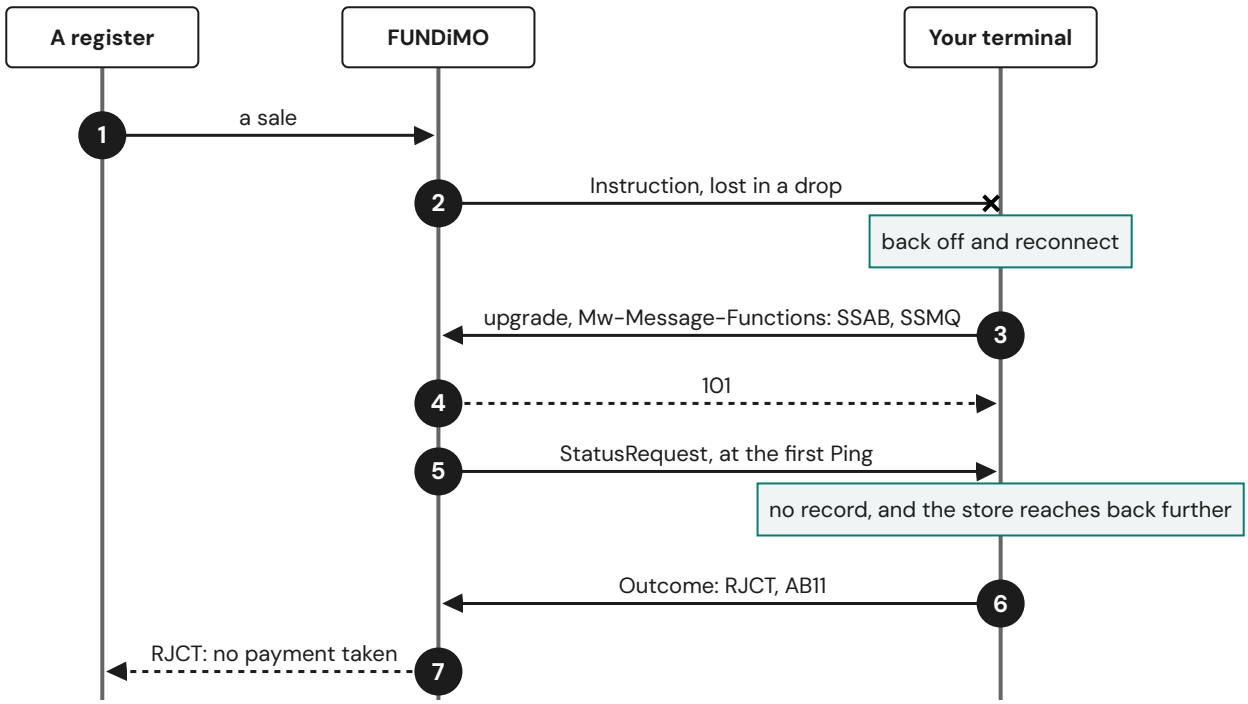
Keep every reference for 24 hours. Your terminal keeps every `TransactionReference` that it received, with its outcome if it has one. It keeps them for 24 hours or more. It knows how far back its store goes. This is the latest of these times:

- the time when the store started;
- the time when it stored the newest reference that it later deleted;
- the time when it lost any reference in a different way, for example one that it received and could not store.

A reset that clears the store starts it again. A restart also starts it again, if the full store and those times do not stay. Measure the age of those times with a clock that never jumps, for example a monotonic clock. Never use the time of day. Thus your terminal never sends `RJCT` with `AB11` for a reference that it possibly received.

One outcome for each reference on a connection. FUNDiMO applies a response as any outcome. A late outcome replaces `UNKN`, and a repeat changes nothing. A different result is a contradiction, and the sale then reads `UNKN` permanently. Send the outcome of a reference a maximum of one time on one connection. This applies whether a `StatusRequest`, an `Abort` or nothing asked for it. Never send it on a timer.

To see a `StatusRequest`, send a sale to `-yours`. Close your terminal's connection before it responds. Connect again with `SSMQ` declared. At the first Ping, FUNDiMO asks about the sale. A conformance run makes a sale that your terminal never receives (step 1.11).



An instruction that a drop lost (2). Your terminal connects again, with `SSMQ` declared (3, 4). At the first Ping, FUNDiMO asks about the sale (5). Your terminal holds no record of it, and its store goes back more than `SecondsSinceInstruction`. Thus it sends `RJCT` with `AB11` (6). The register reads that no payment was taken (7).

1.9 Stop a payment on an `Abort`

A register can cancel a sale or a refund in progress. When your terminal responds to an `Abort`, declare `SSAB` on every upgrade (step 1.2). Until it does, FUNDiMO refuses a register's cancel with `409 terminal_cannot_abort`. FUNDiMO then sends nothing to your terminal. On a connection that declared `SSAB`, FUNDiMO sends one `Abort` for each cancel that it accepts. It does this while the sale still holds your terminal. The `Abort` waits behind any frame already in the queue. Thus an `Abort` never arrives before its own `Instruction`:

```
{ "MessageType": "Abort", "TransactionReference": "sal_01970f6e7c3a7d3e8a4f2b1c9d0e1f2a" }
```

To try it, send a sale to `-yours`. Hold its payment, with no card presented. Cancel the sale as a register does, with your sandbox key. Send no body, with `Content-Length: 0`. Over HTTP/1.1, the network edge in front of FUNDiMO refuses a cancel without that header with `411`:

```
curl -s -X POST -H 'Authorization: Bearer sbx_...' -H 'Content-Length: 0' \
  "$SANDBOX/v1/sales/sal_01970f6e7c3a7d3e8a4f2b1c9d0e1f2a/abort"
```

Response: `202` to the call, with the sale as it is now. Your terminal gets the `Abort`. Respond with a usual `Outcome`, or with nothing. What you send depends on what your terminal holds for the `TransactionReference`:

What your terminal holds	What it does
Its current payment, not sent to its host yet	It stops the payment and shows that it is cancelled. It sends `RJCT` with `DS02`
Its current payment, already with its host	It continues. It sends the host's outcome, or gets the result as in step 1.7. Never `DS02`
An outcome, for a payment that ended	It sends that outcome, if it did not send it already on this connection

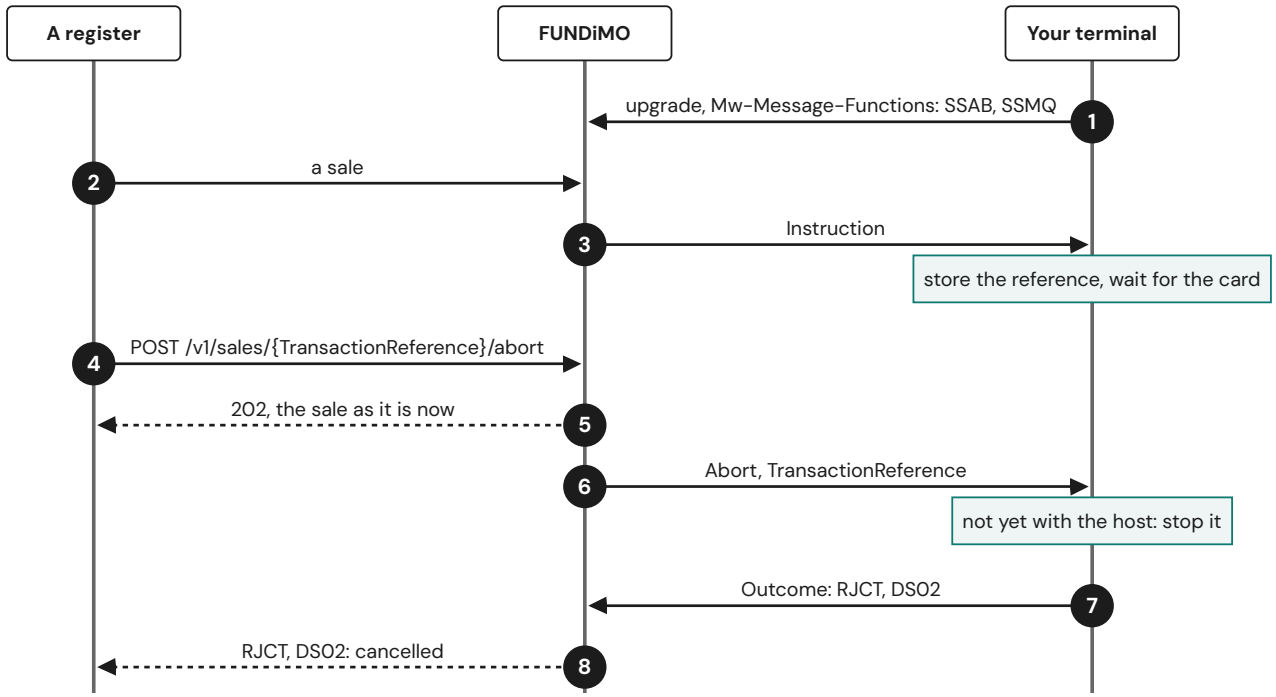
What your terminal holds	What it does
No record	It sends nothing. FUNDiMO asks about an `Instruction` that a drop lost with a `StatusRequest` (step 1.8)

An `Abort` applies only to the payment that it names. An `Abort` can arrive after its sale ended, while your terminal does the payment of a different sale. Do not change that payment. Respond to the `Abort` by the table. Send `DS02` only after an `Abort`. If the customer or the cashier cancels at the terminal, send `AB11`. Many `Abort` frames can name one reference. Send the reference's outcome a maximum of one time on a connection.

An `Abort` does not change a timer. A `DS02` after the deadline is a late outcome. It replaces the `UNKN` that the register reads. Your terminal stays claimed until it sends its outcome or the hold ends. Thus send the `DS02` immediately when the payment stops. Do the same for a refund. Its `DS02` means that no refund was made. A refund that is already with its host continues.

Refusals

Status	Code	What to do
`409`	`terminal_cannot_abort`	Your terminal's connection did not declare `SSAB`. Connect again with it.
`409`	`sale_not_abortable`	The sale has ended, or the hold after its deadline has ended. Read the sale.
`409`	`terminal_offline`	Your terminal is not connected to `<tester>-yours`, or its queue of frames is full. FUNDiMO sent nothing.



A cancel. Your terminal declared SSAB (1). The register cancels the sale while your terminal waits for the card (4). FUNDiMO responds immediately (5). FUNDiMO sends the `Abort` to your terminal (6). Your terminal stops the payment before it gets to the host, and sends `RJCT` with `DS02` (7). The register reads that the sale is cancelled (8).

1.10 Enrol your terminal for its certificate

On staging and production, your terminal presents a client certificate. Its private key is in the Android keystore and never goes out of it. Practise this in the sandbox. The sandbox does not need attestation, so an emulator or a debug build can enrol. `onboarding-device.md`, steps 1 to 8, has the Kotlin for each part.

The pairing code. A pairing code is a one-use secret. An installer types it into your app, and the terminal spends it to enrol for its certificate. It has no relation to linking in section 3.

Register your app with its package and the SHA-256 of the certificate that signs it:

```
curl -s -X POST "$SANDBOX/sandbox/v1/app-identities" \
  -H 'Authorization: Bearer sbx_...' \
  -H 'Content-Type: application/json' \
  -d '{"Role":"terminal","AppPackage":"au.com.example.terminal","SigningDigest":"..."}'
```

Get a pairing code for `-yours`. You can get 6 each hour:

```
curl -s -X POST "$SANDBOX/sandbox/v1/codes" \
  -H 'Authorization: Bearer sbx_...' \
  -H 'Content-Type: application/json' \
  -d '{"Role":"terminal"}'
```

It responds `201`, with `PairingCode` and the `SerialNumber` to enrol with, `<tester>-yours`. The pairing code is five different words of FUNDiMO's word list, in lower case and joined by dots, for example `bronze.farm.loop.prism.tempo`. It is valid for 15 minutes and for one use. A new pairing code re-pairs `-yours`. Re-pair is the pages' word for a new pairing code on a terminal that has enrolled. It has no relation to section 3. FUNDiMO then refuses the terminal's sandbox key on the upgrade until the terminal enrols.

Make the screen where a person enrols the terminal. An installer or a cashier enrols your terminal on your app's enrolment screen. They do this at the first enrolment and at every enrolment after it, with only the device in their hand. Make the screen with these requirements:

- An item of your app's own menu or settings opens it. Never a cable, a developer option or a second device.
- It stays in the menu or settings after the terminal is enrolled.
- It shows the five words before it sends them.
- It shows the device's serial number, as the device reports it. In the sandbox, this is the `SerialNumber` that came with the pairing code, typed into a field that starts empty.
- It shows the environment that it enrols into. It never changes the environment after the terminal is enrolled.
- It shows the state of the terminal.
- After the terminal is enrolled, it accepts a pairing code only behind your app's own supervisor sign-in or PIN.
- It says what to do for each refusal.
- It clears the pairing code's words after use. It keeps them only to send the same enrolment again, and never in storage or in a log.
- It shows no private key.

`onboarding-device.md`, "The enrolment screen", gives each requirement. It says if each requirement is a must or a should, and how a person at the device checks it. A conformance run never sees this screen.

Enrol from your app. Your app reads the pairing code as follows:

- The installer types the pairing code's words into your app, or scans its QR code. FUNDiMO makes no QR code. The person who shows the pairing code can make one. The QR code holds the `PairingCode` only, in byte mode, at error correction `M`.
- Put the word list, `/docs/api/words.txt`, in your app. The first four letters of a word identify it, and no word is the start of another word. Thus your app completes each word while the installer types it. FUNDiMO does not do this.
- Your app reads a typed or scanned pairing code as FUNDiMO does. The pairing code has a maximum of 64 characters. Remove spaces, tabs and line ends before the first word and after the last word.

Between two words, accept one or more dots, spaces or hyphens. Accept `a` to `z` in upper or lower case, and make them lower case with `Locale.ROOT`. The result must be five different words of the list. Send them in lower case, joined by dots.

- If the result is different, ask the installer to type or scan the pairing code again. Never open a scanned pairing code as a link.

Then do these steps:

1. Read the serial number. On a device, use the device's own SDK. In the sandbox, use the `SerialNumber` that came with the pairing code.
2. Make a P-256 private key in the keystore, with an attestation challenge. The challenge is the SHA-256 of three items joined with nothing between them: `mw-enrol-v1`, the pairing code as FUNDiMO shows it (never as typed), and the serial number.
3. Make a CSR for the private key.
4. Send `POST /v1/enrol` with no credential, because the pairing code is the credential:

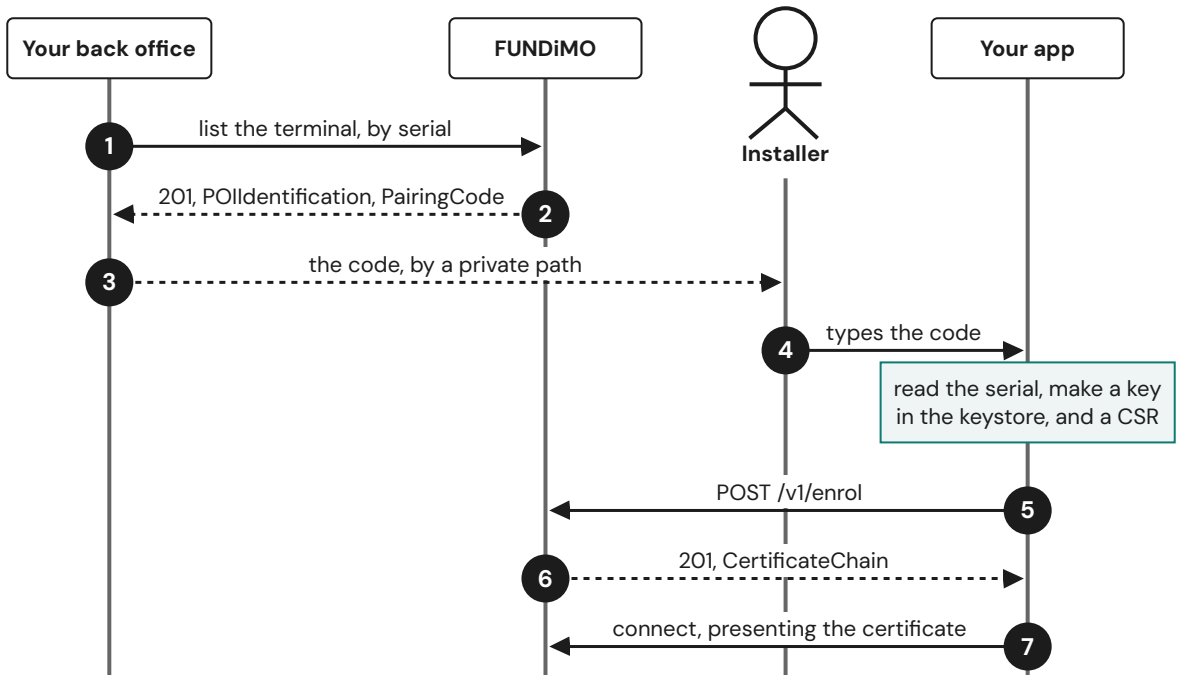
Field	Holds
`PairingCode`	The pairing code, as your app reads it
`CertificateSigningRequest`	The CSR, PEM
`SerialNumber`	The serial number
`AttestationChain`	What `KeyStore.getCertificateChain` gave, PEM, leaf first

Response: `201`, with `CertificateChain`, your terminal's `POIIdentification`, `NotAfter` (30 days later) and `RenewFrom` (day 15). Keep the chain with the private key's alias. Connect immediately on the certificate. Present the private key and the chain at the TLS handshake, and send no `Authorization` header. If you lose the response, send the same body again within the 15 minutes. Never make a new private key to repeat a call that got no response.

Refusals

Status	Code	What to do
`403`	`code_not_valid`	The pairing code is wrong, expired or spent, or the serial number is not the listing's. Check the serial number. Then get a new pairing code.
`403`	`attestation_failed`	The chain failed a check. Check the challenge. Make sure that the keystore made the private key with it. The pairing code stays valid for its 15 minutes.
`403`	`app_identity_unknown`	Register the app's package and signing digest. Then try again.
`400`	`invalid_field`	Correct the field that `Fields` names. For the CSR, make a new private key with a new alias and the same challenge.
`503`	`ca_unavailable`	The pairing code is not spent. Wait, and send the same body again.

After five refusals, a pairing code is unusable. Correct the cause of a refusal before you try again.



Enrolment of a terminal. A listing gives the pairing code one time (1, 2). A person takes it to the device (3, 4). Your app makes its private key and enrols (5, 6). Then it connects on its certificate (7). In the sandbox, FUNDiMO does the listing for you.

1.11 Run the conformance scenarios, and keep the report

A conformance run does the ten terminal scenarios, one step at a time. It judges each step from your world's log. It needs your certificate (step 1.10). The sandbox sends each step's sale as your POS. A person at your terminal, or your test code, operates your terminal. They do what the step's `Instruction` says: present the test card, cancel, or hold the payment with no card. A run of a terminal that works correctly takes less than 4 minutes. You operate the run with the tools of the MCP server (The sandbox's MCP server):

1. Send `conformance_start` with `Side: "terminal"`, `ProductName` and `ProductVersion`. It responds with the `RunIdentification` and the first `Step`, with its `Instruction`.
2. Do what the `Instruction` says, on your terminal and its own host.
3. Send `conformance_status` with the `RunIdentification`, and `WaitSeconds` up to 25. It judges the step and goes to the next step. Each step has 300 seconds.
4. When `Result` is no longer `running`, send `conformance_report`. It gives the signed report. Save the report.

Scenario	What your terminal does
`approved`	Approve the sale. In every `ACSP` that you send for the sale, give `AuthorisedAmount`. It is the instruction's `TotalAmount`, with no change. This includes a resend (`approved.amount`)
`declined`	Decline the sale: `RJCT` with `AG07`
`timed_out`	Stop with no payment: `RJCT` with `AB11`
`unknown`	Send nothing until `Note` says `Now send it.` Then approve
`card_field`	Send the fullest outcome that your terminal makes, with every reference that it has
`duplicate_key`	Not run. FUNDiMO never sends one idempotency key to a terminal two times
`resend`	Hold the outcome for 2 seconds, so that the connection drops first. Then connect again and send it
`restart`	Connect again with the contract's backoff, and send the outcome

Scenario	What your terminal does
`lost_instruction`	Keep your terminal on, with `SSMQ` declared. The sandbox takes your terminal's place for the sale, so its `Instruction` never gets to you. Connect again. When a `StatusRequest` asks about the sale, send `RJCT` with `AB11`
`cancel`	Declare `SSAB`. Accept the sale, and hold its payment with no card. 3 seconds after the instruction, the sandbox cancels the sale. Your terminal gets an `Abort`. Stop the payment, and send `RJCT` with `DS02`

When your terminal completes a step, the step stays open for 3 seconds. Thus a run takes 3 seconds longer for each step that your terminal completes. A second outcome of that sale on the same connection in that time fails the step, on `<scenario>.one_outcome`. In `lost_instruction`, the sandbox can lose its race. This occurs when your terminal connects again before FUNDiMO makes the sale. The sandbox then tries one more time. If it loses again, the step is `undetermined`, never `fail`. `Note` then tells you to run it again.

Response: `Result` `green`, `red`, `undetermined` or `partial`. `Done` gives the result of each scenario. The report has an Ed25519 signature. `sandbox.md` § 6 shows how to check it. A green report is a test result. It is not a certification from a card scheme.

Refusals

Status	Code	What to do
	`certificate_required`	Enrol first (step 1.10).
	`run_not_finished`	The run has not ended. Read `conformance_status`.

A step that fails names its assertion and what to change. An example is `card_field.no_drop`, when your terminal sent a field that FUNDiMO does not accept. `mw://conformance` holds every assertion.

> **A green run ends this section.** Keep the signed report. Its `SandboxVersion` names the release of the sandbox that ran it. Compare it with staging's release before you start section 2.

SECTION 2

Enrolment

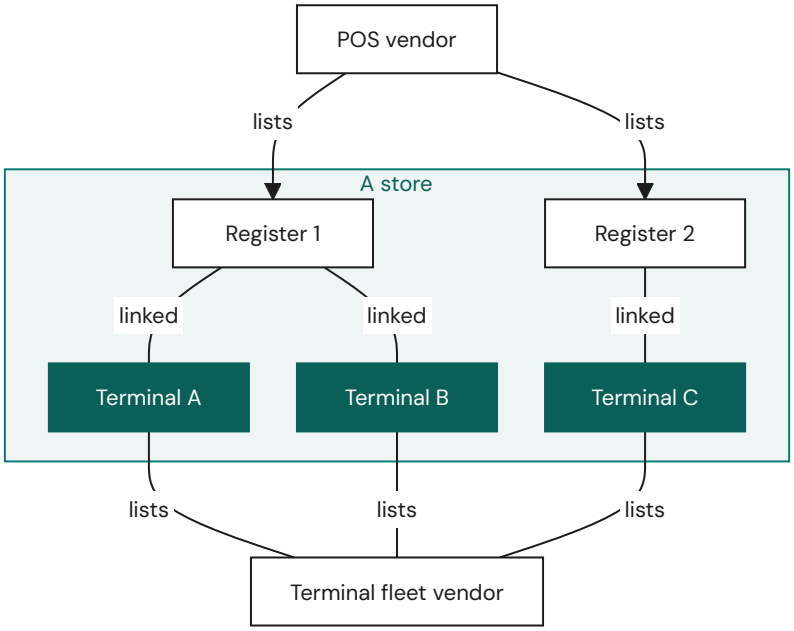
In this section, you register your app and list each terminal at its store. Each terminal then gets its certificate in an environment. Do this section on staging first. Then do it on production. The steps are the same in the two environments. Only the address changes, and with it the pairing codes, the private keys and the certificates. A credential belongs to the environment that issued it. Thus enrol in each environment with its own pairing code. Never use a copied private key.

> **You cannot enrol alone.** Your first server, your `admin` server, needs a pairing code that only FUNDiMO gives, one time only. FUNDiMO also allows your device models (step 2.1). Ask FUNDiMO for the two before the first enrolment, in each environment (Get help). After that, your `admin` server lists each terminal at its store and gets its pairing code (step 2.2). FUNDiMO's onboarding team can also list a terminal and give its first pairing code. The installer at the device then needs that pairing code.

	Staging	Production
Address	`MW=https://mw.staging.fundimo.com.au`	`MW=https://mw.fundimo.com.au`
For	Tests with your real terminals, on test cards only. Never use a real card	Real merchants
You need	Your terminal fleet, at a store that you list	Your terminal fleet, at your merchants' stores
Attestation	Checked. Staging can mark a device that fails, and not refuse it	Checked. FUNDiMO refuses a device that fails, and never marks it

How enrolment fits together:

- You list each terminal at one store.
- A terminal links to one register only.
- A register can link to several terminals of one store.
- A register belongs to one POS vendor only. A store can have registers of more than one POS vendor.
- A register is at no store until it links to a terminal there. When its last link ends, it can link at a different store.



One store, with three terminals that you list. Terminals A and B are linked to register 1, and terminal C to register 2. Each terminal is linked to one register. The POS vendor, or FUNDiMO, lists each register.

Your app selects its environment one time, before its first enrolment. It selects from a list of addresses that it holds apart from its code. It never accepts an address that an installer types. To move a device to a different environment, a person clears the app's data. The app keeps the selected address and its identities in that data. The person then enrols the device in the new environment.

Before staging. Compare staging's release with your report's `SandboxVersion`. Staging's release is `Version` in the response to `GET $MW/v1/health`. The two releases can be different for a time. Select staging in your app before the terminal's first enrolment there. On a device that was in the sandbox, clear the app's data first.

Then production. When your terminals take test payments for staging's registers (section 3), you are ready for real merchants. Do this section and section 3 again, with `MW` set to production's address. A device enrols in production only with production selected in its app before its first enrolment there. If the device was on staging, a person first clears the app's data.

2.1 Register your app, and get your device models allowed

FUNDiMO, or your own `admin` server, registers your app in the environment. Use `POST /v1/vendors/{VendorIdentification}/app-identities`, with the same body as in step 1.10. The digest is of the certificate that signed the app as the device has it. If a store signs your app again, the digest is of the store's certificate. Register each signing certificate. Register a debug build separately. Staging and production refuse a debug build or mark it, as below.

On staging and production, FUNDiMO checks the attestation chain:

- It goes to one of Google's attestation roots.
- The keystore made the private key in its trusted environment or StrongBox.
- The bootloader is locked, and the boot is verified.
- The app is the one that you registered.
- The device is on FUNDiMO's allowed list, by its attested brand and model, its firmware's boot key, or the two.

FUNDiMO refuses a device that fails, or a debug build. Staging can mark it instead. `onboarding-device.md`, "Which devices the middleware admits", gives the details. The allowed list starts empty. Thus ask FUNDiMO to allow your models before the first enrolment (Get help).

2.2 List each terminal at its store, and get its pairing code

Your `admin` server presents its own certificate. FUNDiMO lists your first server, and it enrols as `onboarding-server.md` shows. You cannot get its pairing code in a different way: ask FUNDiMO for it (Get help). Do these steps with your `admin` server:

1. Make each store with `POST /v1/merchant-stores` and the body `{}`. The response gives the store's `StoreIdentification` and its `MerchantIdentification`.
2. Give the merchant the `MerchantIdentification`.
3. Make the store live with `PUT /v1/merchant-stores/{StoreIdentification}/state` and `{"Event":"live"}`. A store takes no sale until it is live.
4. List each terminal at its store, with the serial number on the terminal:

```
curl -s -X POST "$MW/v1/merchant-stores/mst_.../terminals" \
  --cert admin-chain.pem --key admin.key \
  -H 'Content-Type: application/json' \
  -d '{"SerialNumber":"SN-0001"}'
```

Response: `201`, with the terminal's id and its pairing code:

```
{ "POIIdentification": "poi_...", "SerialNumber": "SN-0001", "State": "listed",
  "PairingCode": "bronze.farm.loop.prism.tempo" }
```

The pairing code is five different words of FUNDiMO's word list, in lower case and joined by dots, for example `bronze.farm.loop.prism.tempo`. It is valid for 15 minutes and for one use. FUNDiMO shows it one time only. It is a secret. Give it to the installer by a path that nobody else reads. To get a new pairing code, use `POST /v1/terminals/{POIIdentification}/pairing-code`. Send no body, with `Content-Length: 0`, as for a cancel (step 1.9). On a terminal that has enrolled, a new pairing code re-pairs it, in the pages' word. FUNDiMO then refuses its certificate with `403 not_permitted` until it enrols with the new pairing code. Its link to a register stays.

Refusals

Status	Code	What to do
409	serial_taken	You already list a device with this serial number. Use that terminal. Or remove it, and list the device again.

FUNDiMO's onboarding team can also list a terminal and give its first pairing code.

2.3 The installer enrols each terminal

The installer needs the pairing code from step 2.2. You give it, or FUNDiMO's onboarding team gives the first one. At the device, the installer opens your app's enrolment screen from its menu or settings. The installer types the pairing code. Your app enrols as in step 1.10, at the environment's address. This time, FUNDiMO checks its attestation. A person is necessary only for this, and for a new pairing code later (step 2.6).

Refusals

Status	Code	What to do
403	model_not_allowed	No row of the allowed list accepts the device. Ask FUNDiMO to allow the model (Get help). Then try again within the 15 minutes, or later with a new pairing code.
403	attestation_failed	The chain failed a check. A rooted, unlocked or revoked device does not pass.
503	attestation_status_unavailable	Wait, and send the same body again. The pairing code is not spent.

2.4 Connect on the certificate

Connect to `/v1/terminals/{POIIdentification}/connect` at the environment's address. Use `wss://` on port 443. Present the private key and the chain at the TLS handshake of the upgrade. Send no `Authorization` header. Declare `Mw-Transaction-Types` and `Mw-Message-Functions` as in step 1.2. Check FUNDiMO's own certificate against your trust store, as any HTTPS client does. Never disable that check. A terminal without that check would send its outcomes to any server at the address.

Refusals

What your terminal sees	What it does
A close, a reset or a timeout, with no TLS alert	The normal backoff
`401 certificate_expired` or `certificate_invalid`, or a TLS alert that names the certificate	Wait one hour. After three tries, one hour apart, tell the installer that the terminal needs a new pairing code. Continue to try each hour
`403 certificate_retired`, and it holds a newer certificate	Try again on the newer certificate
`403 certificate_retired` on its newest certificate	Stop. Keep any outcome. Tell a person: something else holds a newer certificate
`403 not_permitted`	Wait one hour. Keep the outcome. A removal or a re-pair gives this refusal
`429` or `503`	The normal backoff

Your terminal never deletes its private key or its certificate after any of these refusals.

2.5 Automate the renewal of every certificate

Your app must renew its terminal's certificate automatically, on every terminal. No person does it. Once a day, your app asks FUNDiMO if it must renew. From day 15, the answer is yes, and your app renews. After its `NotAfter`, a certificate cannot renew. The terminal then needs a new pairing code, which a person brings (step 2.6). Thus put the renewal in your app. Set its daily check on every terminal. `onboarding-device.md`, step 9, sets the check one time each day with Android's WorkManager.

The daily question is `GET /v1/certificates/self`, on the certificate. Its response gives `RenewNow`.

Response: `true` from day 15. Your app then does these steps:

1. It sends `POST /v1/certificates/challenge`, with no body and with `Content-Length: 0`. Over HTTP/1.1, the network edge in front of FUNDiMO refuses the request without that header. The response gives `Challenge`, 64 hex characters. The challenge is valid for 600 seconds.
2. It makes a new private key with a new alias. The challenge is the private key's attestation challenge. It makes a CSR for the private key.

3. It sends `POST /v1/certificates/renew` on the current certificate, within the 10 minutes. The body holds `CertificateSigningRequest` and `AttestationChain`.
4. It uses the new certificate from the next connection. It deletes the old private key only after FUNDiMO responds to a call on the new certificate.

One renewer for each certificate. Each terminal holds its own certificate, and renews its own. Your back office's `admin` server renews as a POS server does. It sends `CertificateSigningRequest` only. It does not ask for a challenge, because FUNDiMO refuses a challenge to a server. Sometimes several copies of your `admin` server share one certificate. Then one copy renews, named in its configuration. The other copies get the new certificate from it (`onboarding-server.md`, step 9).

Test it manually. A terminal's private key never goes out of its keystore. Thus `curl` cannot present it. Test the terminal's check from your app. Your `admin` server asks the same question. You can ask it manually:

```
curl -s "$MW/v1/certificates/self" --cert admin-chain.pem --key admin.key | jq -r .RenewNow
```

Refusals

Status	Code	What to do
409	`renewal_not_due`	Wait until `RenewNow` is `true`.
403	`challenge_not_valid`	Ask for a new challenge, with a new private key.
401	`certificate_expired`	The certificate cannot renew. The terminal needs a new pairing code (step 2.6).

Never delete a private key or a certificate because of a refusal.

2.6 Enrol again with a new pairing code, when a person must

Some terminals enrol again with a new pairing code:

- A terminal that was off for longer than its certificate's life.
- A terminal that was told, after three tries one hour apart, that it needs a new pairing code.
- A terminal that was re-paired.

Your `admin` server gets a new pairing code from `POST /v1/terminals/{POIIdentification}/pairing-code`, with no body and with `Content-Length: 0`, and gives it to the installer. FUNDiMO's operator can also issue it (Get help). FUNDiMO's onboarding team gives only first pairing codes, so it cannot give this one. The installer opens your app's enrolment screen from its menu or settings, behind your app's supervisor sign-in or PIN. The terminal stays as it is, and nothing is installed again. The installer types the pairing code. The app enrols as at the first time, with a new private key. A `201` cancels any wait that a refusal set. Connect immediately on the new certificate. The terminal keeps its `POIIdentification`, and its link to a register stays.

SECTION 3

Link

Linking connects a terminal to the register next to it. It pairs the terminal and the register: the register's sales then go to that terminal. Your terminal gets the frames `LinkWord` and `LinkEnded`, and `linking.md` describes them. At the counter, a register asks to link. Each terminal at the store that is not linked shows two words. The cashier types the words from the terminal in their hand into the register. Link on staging first, with real terminals and test cards. Then link on production, at your merchants' stores. The steps are the same, at production's address.

Linking is the POS vendor's responsibility. The POS vendor's register asks to link, and its cashier types the words. Your terminal shows the words. Until a register is linked to your terminal, FUNDiMO refuses that register's sales to it with `403 not_permitted`. Your terminal then gets no instruction from that register.

Before a register can link, make sure of these conditions:

- Your terminal is enrolled at the store (step 2.3), and connected (step 2.4). A terminal gets words only while it is connected.
- Your terminal is not linked to a register yet.
- The merchant has the store's `MerchantIdentification`, which you gave it (step 2.2). The register sends it.

3.1 Show the link words

When a register asks to link, each terminal at the store that is not linked gets a `LinkWord`. Later it gets a `LinkEnded`:

```
{ "MessageType": "LinkWord", "Word": "rabbit.lamp", "ExpiresInSeconds": 120 }
{ "MessageType": "LinkEnded", "Linked": true }
```

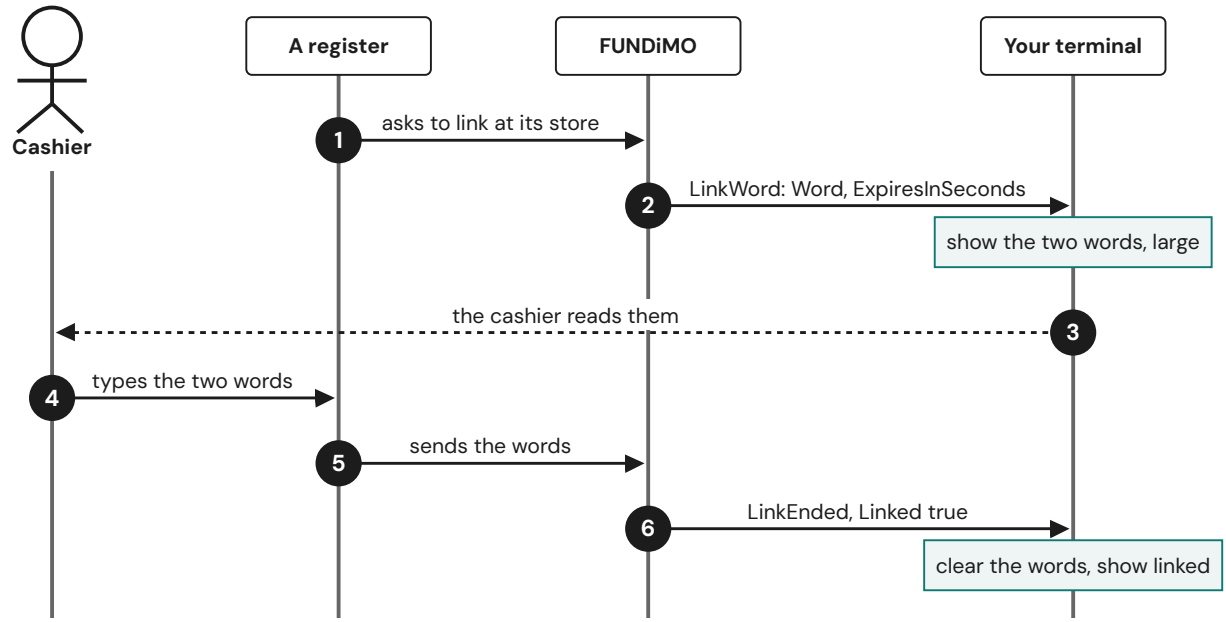
Do these steps:

1. Show the words large enough to read across the counter.
2. Show them until a `LinkEnded` comes, or until `ExpiresInSeconds` pass. Count the time from when your terminal read the frame.
3. On `LinkEnded`, clear the words. If `Linked` is `true`, show that the terminal is linked.

A later `LinkWord` replaces the words on the screen. The two frames need no response. Never log the words, and never send them back. A terminal gets no words during a sale. It gets no words after it is linked.

Make and try this in the sandbox. There, your world's register is linked to every terminal from the start. To see the words, unlink `-yours` and connect it. Then ask to link at your world's store. Use your sandbox key for the two calls:

```
curl -s -X DELETE "$SANDBOX/v1/links/sbx-3f9a1c2e7b10-yours" \
  -H 'Authorization: Bearer sbx_...'
curl -s -X POST "$SANDBOX/v1/links" \
  -H 'Authorization: Bearer sbx_...' \
  -H 'Content-Type: application/json' \
  -d '{"MerchantIdentification":"sbx-3f9a1c2e7b10-M1"}'
```



Linking at the counter, from your terminal's side. The register never sees the words (2). The cashier reads them on your terminal (3), and types them into the register (4).

3.2 Monitor the links at your stores

Your `admin` server sees each terminal of your store, with the register and the POS vendor linked to it. Use `GET /v1/merchant-stores/{StoreIdentification}/terminals`. Thus you can see a link that you did not expect. In that list, a terminal's `State` reads `paired` after it enrols. This is true whether or not a register is linked to it. It is the pages' word for enrolled. Your `admin` server can do two things that a POS cannot do.

End a link of any register, with `DELETE /v1/terminals/{POIIdentification}/binding`. The response is `204`, also if the terminal had no link:

```
curl -s -X DELETE "$MW/v1/terminals/poi_.../binding" \
  --cert admin-chain.pem --key admin.key
```

Limit the POS vendors whose registers the store accepts, with `PUT /v1/merchant-stores/{StoreIdentification}/accepted-vendors`. The response is `204`:

```
curl -s -X PUT "$MW/v1/merchant-stores/mst_.../accepted-vendors" \
  --cert admin-chain.pem --key admin.key \
  -H 'Content-Type: application/json' \
  -d '{"AcceptsAnyVendor":false,"Vendors":["ven_..."]}'
```

Every store starts with registers of all onboarded POS vendors accepted. `{"AcceptsAnyVendor":true}` accepts all of them again. A link that exists stays until somebody ends it. The words prove only that a person can see the terminal. They do not prove that the person is the store's cashier. Thus end a link that you do not expect. Limit the store to the POS vendors whose registers it uses.

What green conformance means for your terminal software

A green run means one thing: your terminal software did each step one time, in the sandbox. It is not a certification. On every other day, your terminal software must obey these rules.

- **One payment at a time.** Do not start or queue an `Instruction` that comes during a payment. Store `RJCT` with `AB11` as its outcome, and send it. A queue can make a customer pay two times.
- **Store first, and keep it.** Store the `TransactionReference` before the card, and the `Outcome` before you send it. A crash or a reboot must not lose them. Keep every reference with its outcome for 24 hours or more. Know how far back your store goes.
- **Never charge two times.** Never charge again for a `TransactionReference` that your terminal has already started, also after a restart. Send its stored outcome.
- **The truth, or nothing yet.** `ACSP` means the full `TotalAmount`, given in `AuthorisedAmount` with no change. There is no partial approval, and no added tip, surcharge or fee. Send `RJCT`, always with its `Reason`, only when no payment was taken. Send `DS02` only after an `Abort`. If your terminal does not know, it sends nothing yet. It gets the result from its host by an enquiry or a confirmed reversal, and sends it, however late.
- **Send again safely.** After every new connection, send your most recent outcome again, with no change. Send a reference's outcome a maximum of one time on a connection, and never on a timer. A different result for the same sale is a contradiction: the sale then reads `UNKN`.
- **Respond to what you declare.** Declare `SSMQ` and `SSAB` only when your terminal responds to each. Stop only a payment that is not with its host yet.
- **One connection for each terminal id.** Two connections for one id replace each other.
- **Renew automatically, and never delete your private key after a refusal.** Your app renews each terminal's certificate from day 15 (step 2.5). Make a new private key only to renew, or when a person brings a new pairing code.
- **An enrolment screen that each installer can open.** An item of your app's own menu or settings opens it, with only the device. It stays there after the terminal is enrolled (step 1.10).

What your terminal can send

Send only the outcome's own fields: `MessageType`, `TransactionReference`, `TransactionStatus`, `StatusReasonInformation`, `TransactionDetails`, `TransactionDateTime` and `PaymentReference`. `StatusReasonInformation` holds `Reason`, `ResponseCode` and `AdditionalInformation`. `TransactionDetails` holds only `AuthorisedAmount`. `PaymentReference` holds the group of the kind of payment, with only the keys that its issuer gave:

Group	Keys
`Card`	`RetrievalReferenceNumber`, `SystemTraceAuditNumber`, `AuthorisationCode`, `AcquirerPOIIdentification`, `AcquirerMerchantIdentification`, `RefundReference`
`Bank`	`AccountServicerReference`
`Wallet`	`ProviderIdentification`, `RecipientTransactionIdentification`
`NPP`	`EndToEndIdentification`, `TransactionIdentification`, `MandateIdentification`

Never send a card number, an expiry, track data, a cardholder name or a bank account. FUNDiMO forbids them in every field, but its patterns cannot stop all of them. For example, a 12 digit card number matches a retrieval reference. FUNDiMO would log it, send it to the register and keep it with the sale. FUNDiMO removes any other field before it logs anything, and logs only the field's name. A printed card line stays on your terminal's own receipt.

The sandbox's MCP server

What it is. The sandbox runs an MCP (Model Context Protocol) server at `POST /mcp`. With it, an AI coding assistant can read FUNDiMO's pages and check an outcome before your terminal sends it. The assistant can also send test sales, operate a terminal on `-manual` and read your world's log.

Who it is for. Your developer's AI coding assistant, in the sandbox only. Your terminal's own code holds the WebSocket of section 1. A conformance run counts only what has your certificate.

Connect your assistant. You can use any MCP client on revision `2026-07-28` or `2025-11-25`. It connects over Streamable HTTP at `$SANDBOX/mcp`, with your sandbox key on every request. There is no session and no OAuth. The pages give this command for the command line of one assistant:

```
claude mcp add --transport http mw-sandbox "$SANDBOX/mcp" --header "Authorization: Bearer sbx_..."
```

Then start a new session, because a client reads its servers when a session starts. You can also call the server over plain HTTP, with no client. Keep the line with your sandbox key in a file that only you can read. Send one request at a time:

```
touch ~/.mw-sandbox-auth && chmod 600 ~/.mw-sandbox-auth
(read -rs key && printf 'Authorization: Bearer %s\n' "$key" > ~/.mw-sandbox-auth)
curl -s "$SANDBOX/mcp" -H @"$HOME/.mw-sandbox-auth" \
  -H 'Content-Type: application/json' \
  -H 'Accept: application/json, text/event-stream' \
  -H 'MCP-Protocol-Version: 2025-11-25' \
  -d '{"jsonrpc":"2.0","id":1,"method":"resources/read","params":{"uri":"mw://world"}}'
```

Without a sandbox key, you can read every resource except `mw://world`. Also, `describe_field`, `validate_message` and `explain_refusal` respond. Every other tool responds `key_required`. A sandbox key that is unknown or expired gets `key_not_recognised`. A staging or production credential gets `401 credential_not_for_sandbox`. If you sent one, rotate it.

What it offers

Resource	Key	What it is for
`mw://start`	No	The sandbox's start page, `sandbox.md`
`mw://environments`	No	The two environments that the pages name, the sandbox and staging, and the address of each
`mw://world`	Yes	Your world's ids, its expiry, and the timers and TLS mode in force. How each terminal does a refund and responds to a `StatusRequest` and an `Abort`
`mw://fields`	No	The field catalogue: one row for each field of each message
`mw://fields/{Message}/{Path}`	No	One row, for example `mw://fields/Outcome/TransactionStatus`
`mw://refusals`	No	Every code that FUNDiMO and the sandbox give, what it means, and what to do next
`mw://conformance`	No	The conformance scenarios, and the assertions that judge each

`mw://docs/terminal.md`
`mw://docs/openapi.yaml`
`mw://docs/llms.txt`
	No	The terminal contract, the OpenAPI document and the index

`mw://onboarding/device`
`mw://onboarding/server`
`mw://onboarding/linking`
	No	Enrolment of an app on a device, enrolment of a server, and linking at the counter

`mw://report-key`	No	The public key that verifies a conformance report's signature

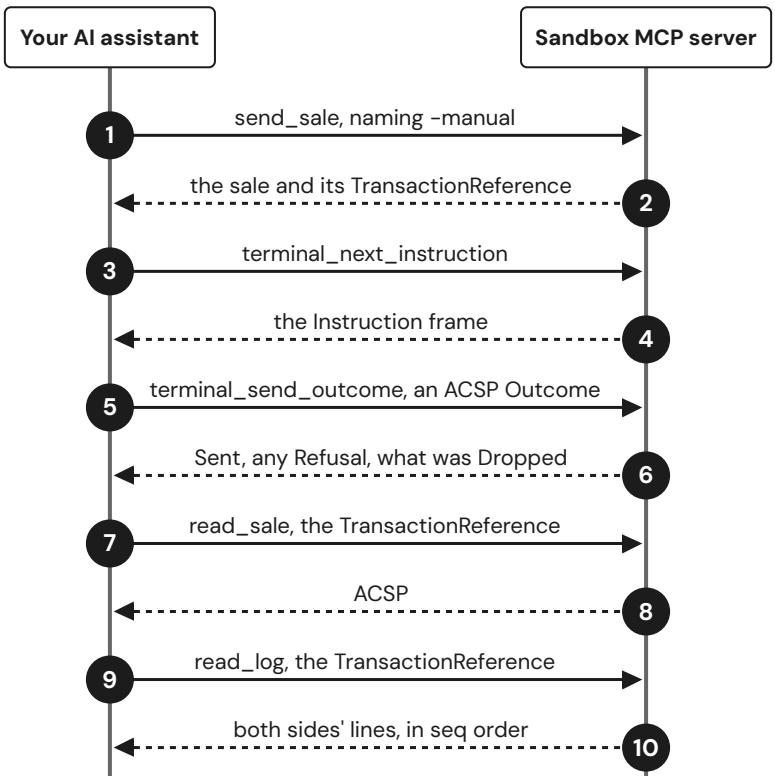
Tool	Key	What it is for
`describe_field`	No	The catalogue's rows for a field, by its `Path`
`validate_message`	No	Checks a `Sale` or an `Outcome` as FUNDiMO does. It gives `Accepted`, or the refusal with its fields, what FUNDiMO would remove (`Dropped`), and `Next`. It keeps nothing
`explain_refusal`	No	What a code means, what to do next, and which side gives it
`send_sale`	Yes	Sends a sale as your world's POS, for example one to `-yours`. It can wait up to 25 seconds for the result
`read_sale`	Yes	Reads one of your sales. It can wait for the sale to change
`terminal_next_instruction`	Yes	The next instruction, `StatusRequest` or `Abort` that FUNDiMO sent to your `-manual` terminal. A maximum of eight wait, of the three kinds together
`terminal_send_outcome`	Yes	Sends an outcome as your `-manual` terminal. It tells if the outcome was `Sent`, any `Refusal`, and what FUNDiMO removed (`Dropped`)
`read_log`	Yes	The log lines of the sales that a `TransactionReference` or a `SaleReferenceIdentification` names: the newest five, each in `seq` order

`conformance_start`
`conformance_status`
`conformance_report`
	Yes	Start a conformance run, judge and read it, and get its signed report (step 1.11)

An example

Ask your assistant: "Do a sale on my manual terminal, and show me its log." The assistant does these steps:

1. It sends a sale with `send_sale`, to your `<tester>-manual` terminal. The instruction waits there.
2. It reads the instruction with `terminal_next_instruction`. It gets the `Instruction` frame, as your terminal would get it.
3. It responds with `terminal_send_outcome` and an `ACSP` outcome. The tool sends the outcome as written. Thus the assistant gives `AuthorisedAmount` in it. The tool tells if the outcome was `Sent`, any `Refusal`, and what FUNDiMO removed (`Dropped`).
4. It reads the sale with `read_sale`, as the register does. Then it reads the log with `read_log`. The log gives every message of the sale, in the two directions, in `seq` order.



The assistant sends a sale to `-manual` (1, 2). It reads the instruction as your terminal would (3, 4), and responds to it (5, 6). It reads the sale as the register does (7, 8). Then it reads the sale's log (9, 10).

If a tool fails, its result has `isError` `true`, an `Error`, and a `Next` that says what to do. FUNDiMO never logs a tool's arguments. No tool changes a credential: sandbox keys and pairing codes come only from the routes under `/sandbox/v1/`.

Get help

In the sandbox, before you sign with FUNDiMO. The sandbox is self-help. Use this guide and the MCP server first (The sandbox's MCP server). If you need more help, send an email to `support@fundimo.com.au`.

When you sign with FUNDiMO. FUNDiMO invites your team to a Slack Connect channel. The channel is private: only your organisation and FUNDiMO are in it. Use it for enrolment and linking, and for your questions on staging and production.

If your organisation does not use Slack. Send an email to `support@fundimo.com.au` instead.

> **Never send a secret in the channel or in an email.** Send no pairing code, no sandbox key and no private key. FUNDiMO never asks for one there. A pairing code goes only by a path that nobody else reads (step 2.2). It never goes in the channel.

The pages

Every page is public at `mw.sandbox.fundimo.com.au`. Each page gives the details that this guide does not give.

Page	What it is for
`/llms.txt`	The index of every page below
`/docs/api/sandbox.md`	Start here. Your sandbox key and test world, the terminal journey, the MCP server, conformance, and every code that the sandbox adds
`/docs/api/environments.md`	The sandbox and staging: the address of each, the differences, and how to move from the sandbox to staging
`/docs/api/terminal.md`	The terminal's contract: the connection, every frame, the deadline, refunds, and what to send again
`/docs/api/onboarding-device.md`	Your app's enrolment on the device, with Kotlin. The pairing code, the private key and its attestation, renewal, refusals and the enrolment screen
`/docs/api/openapi.yaml`	The schemas of the frames. Your back office's routes, tagged `operator`: stores, terminals and their pairing codes
`/docs/api/linking.md`	Linking at the counter. What a store's fleet vendor can do
`/docs/api/onboarding-server.md`	Your back office's own server and its certificate
`/docs/api/fields.json`	Every field: who sends it, its ISO 20022 element if ISO has one, its limits, and an example
`/docs/api/conformance.json`	The conformance scenarios, and how FUNDiMO judges each
`POST /mcp`	The sandbox's MCP server, for your AI coding assistant

Addendum: the ISO 20022 terms in this guide

Each ISO 20022 term in this guide, with ISO's definition as `fields.json` and `terminal.md` quote it, without ISO's usage notes. Then what the term is in this guide.

Fields

Field	ISO 20022's definition, and what it is here
`MerchantIdentification` `Merchant/Identification, Mrchnt/Id`	"Identification of the merchant." The store's id. Every sale names it.
`POIIdentification` `POIId`	"Identification of the POI (Point Of Interaction) performing the transaction." Your terminal's id.
`TransactionReference` `TxRef`	"Identification of the transaction that has to be unique for a time period." FUNDiMO's id for a sale or a refund. Your outcome sends it back with no change.
`SaleReferenceIdentification` `SaleRefId`	"Global reference of the sale transaction for the sale system." The register's own reference, for example an invoice number.
`CashierIdentification` `CshrId`	"Identification of the cashier who carried out the transaction." The register's id for the user who made the sale or the refund, if the register sent one.
`TransactionType` `TxTp`	"Type of transaction being undertaken for the main service." `RFND` on a refund's instruction.
`OriginalTransaction` `OrgnlTx`	"Identification of the original transaction." On a refund, the sale that it refunds.
`TransactionDetails` `TxDtls`	"Details of the transaction." In an instruction: the amount, its currency and the items. In an outcome: `AuthorisedAmount` only.
`Currency` `Ccy`	"Currency associated with the transaction." `AUD`.
`TotalAmount` `TtlAmt`	"Total amount of the transaction." The amount to charge, or to refund.
`SaleItem` `SaleItm`	"Item purchased with the transaction." One line of the basket. Show it if your terminal can.
`ProductDescription` `PdctDesc`	"Description of the product or item." The line's description.
`ProductQuantity` `PdctQty`	"Product quantity." How many.
`UnitPrice` `UnitPric`	"Price per unit of product." The price of one.
`ProductAmount` `PdctAmt`	"Monetary value of purchased product." The line's amount.
`AuthorisedAmount` `AuthrsdAmt`	"Amount authorised for the payment transaction." The amount that the host approved. Give it on every `ACSP`.
`TransactionDateTime` `TxDtTm`	"Local date and time of the transaction assigned by the POI (Point Of Interaction)." Your terminal's time of the payment.
`TransactionStatus` `TxSts`	"Specifies the status of a transaction, as published in an external payment transaction status code list." What your outcome says happened.

Field	ISO 20022's definition, and what it is here
`StatusReasonInformation` `StsRsnInf`	"Provides detailed information on the status reason." It holds `Reason`, `ResponseCode` and `AdditionalInformation`.
`Reason` `Rsn`	"Specifies the reason for the status report." With `RJCT`, why no payment was taken.
`ResponseCode` `RspnCd`	"Detailed results of the processing, conforming to ISO 8583 Response codes list. This code list is maintained by the ISO 8583/MA (maintenance agency)." The host's own code, for example `00`.
`AdditionalInformation` `AddtlInf`	"Further details on the status reason." The host's own words, for example `APPROVED`.
`RetrievalReferenceNumber` `RtrvlRefNb`	"Reference supplied by the system retaining the original source information and used to assist in locating that information or a copy thereof." In `Card`, for the receipt and settlement.
`SystemTraceAuditNumber` `SysTracAudtNb`	"Number assigned by a transaction originator to assist in identifying a transaction uniquely. The trace number remains unchanged for all messages within a two-message exchange (for example, request/repeat and response)." In `Card`, for the receipt and settlement.
`AuthorisationCode` `AuthstnCd`	"Value assigned by the authorising party." In `Card`, for the receipt and settlement.
`AccountServicerReference` `AcctSvcrRef`	"Unique reference, as assigned by the account servicing institution, to unambiguously identify the instruction." In `Bank`.
`ProviderIdentification` `PrvdrId`	"Identification of the wallet provider." In `Wallet`.
`RecipientTransactionIdentification` `RcptTxId`	"Identification of the transaction assigned by the recipient party for the initiating party." In `Wallet`.
`EndToEndIdentification` `EndToEndId`	"Unique identification, as assigned by the initiating party, to unambiguously identify the transaction. This identification is passed on, unchanged, throughout the entire end-to-end chain." In `NPP`.
`TransactionIdentification` `TxId`	"Unique identification, as assigned by the first instructing agent, to unambiguously identify the transaction that is passed on, unchanged, throughout the entire interbank chain." In `NPP`.
`MandateIdentification` `MndtRltdInf/MndtId`	"Unique identification, as assigned by the creditor, to unambiguously identify the mandate." In `NPP`.

Codes

Code	ISO 20022's name and definition, and what it is here
`ACSP` Status	AcceptedSettlementInProcess: "All preceding checks such as technical validation and customer profile were successful and therefore the payment instruction has been accepted for execution." The host approved the full `TotalAmount`.
`RJCT` Status	Rejected: "Payment instruction has been rejected." No payment was taken. `Reason` says why.
`PDNG` Status	Pending: "Payment instruction is pending. Further checks and status update will be performed." What a register reads while the sale waits for your outcome. Your terminal never sends it.
`AG07` Reason	UnsuccesfulDirectDebit: "Debtor account cannot be debited for a generic reason." The card or the host declined.

Code	ISO 20022's name and definition, and what it is here
`AB11` Reason	TimeoutDebtorAgent: "Transaction stopped due to timeout at the Debtor Agent." No payment was taken for a different cause. A cancel at the terminal is one cause.
`DS02` Reason	OrderCancelled: "An authorized user has cancelled the order" Your terminal stopped the payment before it got to its host, after an `Abort`.
`CRDP` Transaction type	"Card payment." A sale.
`RFND` Transaction type	"Refund transaction." A refund.
`SSMQ` Message function	"Request the status of a previous message for which the Sale system has no response." In `Mw-Message-Functions`: your terminal responds to a `StatusRequest`.
`SSAB` Message function	"Abort the current process or the last request." In `Mw-Message-Functions`: your terminal responds to an `Abort`.

FUNDiMO's own names

These names are FUNDiMO's, not ISO 20022's. The other names of FUNDiMO's API in this guide are FUNDiMO's too. `openapi.yaml`, `sandbox.md` and `conformance.json` define them.

Name	What it is
`IdempotencyKey` FUNDiMO's: field	A register's idempotency key for a sale. A retry after a lost response cannot make a second sale.
`PaymentReference`, `Card`, `Bank`, `Wallet`, `NPP` FUNDiMO's: fields	Your terminal's references for the payment, in the group of the kind of payment.
`AcquirerPOIIdentification` FUNDiMO's: field	In `Card`: the acquirer's terminal id.
`AcquirerMerchantIdentification` FUNDiMO's: field	In `Card`: the acquirer's merchant id.
`RefundReference` FUNDiMO's: field	In `Card`: the host's reference for a later refund.
`SecondsSinceInstruction` FUNDiMO's: field	In a `StatusRequest`: the whole seconds from the creation of the sale, by FUNDiMO's clock.
`ResponseReason` FUNDiMO's: field	In a `Refusal`: why FUNDiMO refused your last frame.
`Word`, `ExpiresInSeconds`, `Linked` FUNDiMO's: fields	In `LinkWord` and `LinkEnded`: the words to show, for how long, and if this terminal was linked.
`UNKN` FUNDiMO's: status	What a register reads when FUNDiMO does not know the outcome. Your terminal never sends it.
`Mw-Transaction-Types`, `Mw-Message-Functions` FUNDiMO's: headers	On every upgrade: the transaction types that your terminal does, and the requests that it responds to.
`MessageType` FUNDiMO's: field	The name of the frame.
`Sale`, `Error`, `Instruction`, `Outcome`, `Refusal`, `StatusRequest`, `LinkWord`, `LinkEnded`, `Abort` FUNDiMO's: messages	A register's `Sale`, FUNDiMO's `Error`, and the frames between FUNDiMO and your terminal. `Abort` is FUNDiMO's cancel to a terminal.