MaxyLock Documentation

Published by Maxy. Last updated: 1 November 2026.

Contents

1. Introduction
   1.1 Purpose
   1.2 What distinguishes the system
   1.3 The loan assets
   1.4 Scope and audience
2. Design overview
   2.1 Four phases
   2.2 Tranches
   2.3 The two values that connect the chains
   2.4 Properties
   2.5 Comparison with other lending models
3. Key concepts
   3.1 Bitcoin terms
   3.2 Loan and contract terms
   3.3 Roles
4. System components
   4.1 Components
   4.2 Enforcement layers
5. The loan process
   5.1 Steps
   5.2 Worked example
   5.3 The default case
   5.4 The ghost case
   5.5 What the contract never does
6. Flow of funds and custody
   6.1 Stages
   6.2 Encumbrance while in the HTLC
   6.3 Custody on the host chain
   6.4 Custody of the lender's collateral asset
   6.5 Where each party's value ends up

6.6 Fees and costs
7. Tranches
7.1 Why tranches
7.2 Collateral tranches and principal shares
7.3 The interest tranche
7.4 Collateral slices
7.5 Top-up tranches
7.6 Limits
8. Loan and tranche lifecycle
8.1 Loan states
8.2 Tranche states
8.3 Transitions
8.4 The loan state is a summary
8.5 Time parameters
8.6 The timing rule
9. Funding and proof
9.1 One transaction for all tranches
9.2 The proof
9.3 What is checked
9.4 Activation and disbursement
9.5 The verifier fee
9.6 The challenge window and the bounty
9.7 Finalizing funding
9.8 Bitcoin sent that is not proven
10. Repayment and the reveal
10.1 Escrow for secret
10.2 The reveal window
10.3 The early-repayment rebate
10.4 Default
10.5 Reclaiming the Bitcoin
11. Ghost liquidation
11.1 Trigger
11.2 The Mega-Pot
11.3 The order on the settler
11.4 Delivery and settlement
11.5 Expiry and re-posting

11.6 Pot states
11.7 Two-asset loans and redemption routes
11.8 Worked example
12. Protection modules
12.1 Flags
12.2 Module A: health flag and cure window
12.3 Module B: yield-funded hedge
13. Integration guide for wallets and front ends
13.1 Negotiating the terms
13.2 The term sheet
13.3 Signing
13.4 Committing
13.5 Funding the HTLCs
13.6 Submitting the funding proof
13.7 Claiming payouts
13.8 Repaying a tranche
13.9 Revealing
13.10 Spending the Bitcoin
13.11 Permissionless calls
13.12 Monitoring
13.13 Errors
14. Operations for lenders and borrowers
14.1 The lender's keys and secrets
14.2 The borrower's keys
14.3 Lender procedures
14.4 Borrower procedures
14.5 Watchers
14.6 The pauser
14.7 Monitoring
15. Compliance and auditability
15.1 Counterparty selection
15.2 No custody and no pooling
15.3 The audit trail
15.4 Verification by a third party
15.5 Controls
16. Security model

16.1 Guarantees
16.2 Assumptions
16.3 Capabilities of each party
16.4 Failure cases
16.5 Operating rules
17. Contract reference
17.1 Loan contract
17.2 Extensions contract
17.3 Verifier
17.4 Settler
17.5 Oracle adapter
17.6 Redemption adapters
17.7 Module B adapter
17.8 External dependencies at a glance
18. Bitcoin script reference
18.1 What the contract commits to
18.2 The reference witness script
18.3 The address
18.4 Spending a tranche
18.5 The interest tranche
18.6 Checks a wallet performs
18.7 Maxy's template `maxylock-htlc-v1`
19. Deployment and parameters
19.1 Networks
19.2 Live deployments
19.3 Test deployment
19.4 Constructor parameters
19.5 Parameters
19.6 Steps after deployment
19.7 Build settings

1. Introduction

1.1 Purpose

MaxyLock is a lending protocol in which a borrower pledges Bitcoin as collateral and receives a loan in an asset on an EVM chain from a lender. The Bitcoin is never wrapped, bridged or handed to a custodian. It stays on the Bitcoin network, locked in scripts that fix in advance who may take it and under what conditions. The loan contract on the EVM chain holds the lender's principal and the lender's collateral, verifies the Bitcoin lock with proofs drawn from the Bitcoin blockchain, and releases funds only when the conditions the two parties signed have been met.

The protocol is peer to peer. Each loan is an agreement between one borrower and one lender on terms they both signed. There is no lending pool, no shared interest rate and no protocol treasury that takes a position in the loan. The contract pays the principal to the borrower once the Bitcoin deposit has been verified on chain, and it settles each repayment against the secret that returns the borrower's Bitcoin. If the borrower does not repay, the lender takes the Bitcoin. If the lender is repaid but withholds the secret, the lender's own collateral pays for the borrower's compensation.

1.2 What distinguishes the system

In a conventional Bitcoin-backed loan, the borrower transfers Bitcoin to a custodian or wraps it into a token on another chain. The lender relies on the custodian to hold the collateral, on a price oracle to decide when to liquidate it, and on the custodian again to return it after repayment. Each of these is a promise. The borrower cannot verify that the collateral is still there, and the lender cannot verify that a liquidation was fair. Three features separate MaxyLock from both centralized Bitcoin lending and pool-based decentralized lending.

Locks in place of promises. On the Bitcoin side, the collateral sits in hash time-locked contracts that the Bitcoin network enforces. The borrower may spend a lock by presenting a secret. The lender may spend it after a fixed time has passed. No other spending is possible. On the host-chain side, the lender's funds sit in a contract with no administrative path to them. Neither party relies on the other's good conduct or on an intermediary.

Proof in place of reporting. The host chain learns that the collateral has been locked by verifying a proof of inclusion against a copy of the Bitcoin block headers held on chain. No oracle, committee or operator attests to the deposit. The loan contract pays out the principal only after that proof has been checked.

Stakes on both sides. The borrower stakes the Bitcoin. The lender also stakes collateral, posted on the host chain at the moment the loan is committed. The lender can only collect a repayment by

publishing the secret that frees the borrower's Bitcoin. A borrower who does not repay forfeits Bitcoin. A lender who is repaid but withholds the secret forfeits collateral, which is used to buy Bitcoin for the borrower on the open market at a premium over the amount locked. Each party's stake funds the other's remedy.

The result is that settlement does not depend on anyone's cooperation after the loan is committed. Repayment and the return of the collateral are a single mechanical exchange, and every failure to perform has a remedy that the contracts carry out without a decision by any third party. In particular:

- The borrower never gives up control of the Bitcoin to a third party. It is spendable by the borrower with the secret, or by the lender after the tranche's time lock, and in no other way.
- The lender's principal is paid out only against verified Bitcoin. The amount of Bitcoin backing each portion of the loan is a fixed number recorded on chain.
- Every repayment is an atomic exchange. The lender receives the repayment in the same transaction that publishes the secret, and the secret is what the borrower needs to recover the Bitcoin.
- A lender who disappears after being repaid is penalized by its own stake. The penalty requires no court, no oracle and no administrator. Anyone may trigger it.
- There is no price-based liquidation of the borrower's Bitcoin on the main path. A fall in the Bitcoin price does not, by itself, cost the borrower the collateral. An optional protection module, agreed in the terms, adds a health check with a cure window.
- No administrator can touch funds. The only administrative power is to pause the creation of new loans.

1.3 The loan assets

Two assets on the host chain take part in a loan. The **loan asset** is what the lender lends and the borrower repays. The **collateral asset** is what the lender posts as its own stake. Both are chosen in the terms of each loan. Either may be an ERC-20 token or the chain's native currency. The loans Maxy offers use a US dollar stablecoin as the loan asset: USDC on Base and USDG on Robinhood Chain. Section 19.2 gives their addresses.

The two assets may be the same token, or the collateral asset may be a yield-bearing form of the loan asset, such as a lending-pool deposit receipt for the same stablecoin. In the second case a redemption route fixed at deployment converts one into the other when a liquidation requires it. Chapter 11 describes this. Throughout this document the two are referred to as the loan asset and the collateral asset.

The borrower's Bitcoin is not an asset on the host chain at all. It is recorded as a number of satoshis and a set of Bitcoin script hashes, and it never leaves the Bitcoin network.

1.4 Scope and audience

This document describes MaxyLock as deployed on two EVM chains, Base (chain ID 8453) and Robinhood Chain (chain ID 4663), with collateral held on the Bitcoin mainnet. The two deployments share the same contract source. Where the chain does not matter, this document refers to it as the host chain.

It is written for readers who are evaluating or building on the protocol:

- Lenders, borrowers and their risk, treasury and compliance teams: Chapters 2, 6, 11, 12, 14 and 15.
- Borrowers and the wallets that act for them: Chapters 5, 9, 10, 13 and 18.
- Developers integrating wallets, front ends or watchers: Chapters 8, 13, 17 and 18.
- Reviewers of the contracts: Chapters 16, 17 and 19.

Chapter 3 defines the terms used throughout. It does not assume prior familiarity with Bitcoin scripting or with collateralized lending.

2. Design overview

2.1 Four phases

A MaxyLock loan proceeds through four phases.

1. **Commit.** The borrower and the lender agree a term sheet off chain: the principal, the loan asset, the lender's collateral, the amount of Bitcoin to be locked, the deadlines, and the Bitcoin script hash and secret hash of each portion of the collateral. The borrower signs the term sheet. The lender submits it to the loan contract together with the principal and the lender's collateral. The contract records the loan and holds both deposits.
2. **Fund and prove.** The borrower sends Bitcoin to the addresses named in the term sheet. Each address is a hash time-locked contract, or HTLC. Bitcoin held at the address can be spent in exactly two ways: by the borrower with a secret chosen for that portion, or by the lender after a fixed time has passed. A proof that the Bitcoin arrived is submitted to the loan contract, which verifies it against a copy of the Bitcoin block headers maintained by a verifier contract on the host chain. For each portion proven, the contract credits the matching share of the principal to the borrower, who withdraws it. The loan is now active.
3. **Repay and release.** The borrower deposits the repayment for a portion into the loan contract, which holds it in escrow. The lender collects it by submitting the secret for that portion. The contract checks the secret, pays the lender, returns the lender's matching collateral, and publishes the secret in the same transaction. That secret is the key that lets the borrower take the Bitcoin back. The lender cannot be paid without releasing the Bitcoin.
4. **Resolve.** If the borrower does not repay a portion in time, anyone may mark it defaulted. The contract releases the lender's matching collateral back to the lender, and the lender takes the Bitcoin through the time lock in the script. If the borrower repays but the lender does not release the secret within the reveal window, anyone may trigger a liquidation: the lender's matching collateral and the borrower's repayment are pooled and posted as a fixed-quantity order to buy Bitcoin for the borrower. Bonded fillers on the settler fill the order. The borrower receives the portion's Bitcoin plus the premium fixed in the terms, and the lender's collateral pays for it.

The secret is the link between the two chains. The borrower cannot recover the Bitcoin without it. The lender cannot collect the repayment without publishing it. The lender therefore cannot complete the first act without enabling the second. Both endings in the fourth phase are mechanical: no action by the counterparty is required for either, and none is possible.

2.2 Tranches

The collateral is divided into between one and seven portions called tranches. Each tranche is an independent Bitcoin script with its own secret, and each carries a proportional share of the principal. Funding, repayment, default and liquidation all operate tranche by tranche. A borrower can repay part of the loan and recover the corresponding part of the collateral, and a default on one tranche costs only that tranche's Bitcoin. A further small tranche prepays the lender's interest in Bitcoin. Chapter 7 describes tranches in detail.

2.3 The two values that connect the chains

Two 32-byte values do the work of connecting Bitcoin to the host chain. Everything else in the protocol is built around them.

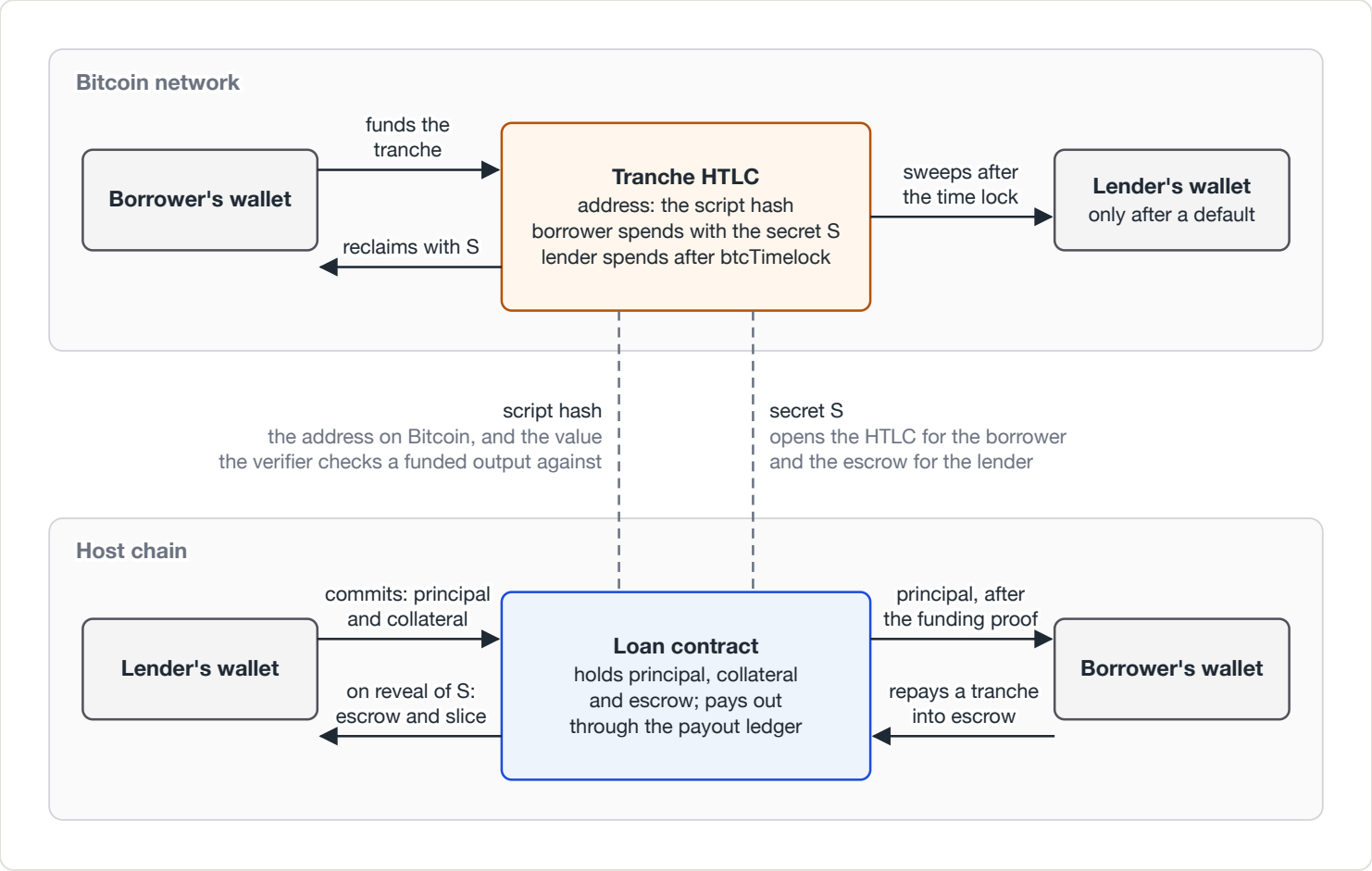


Figure: The two values that connect the chains. The script hash identifies the lock on both sides; the secret opens both sides.

The script hash. Each tranche's Bitcoin script is identified by its hash. On Bitcoin, the hash is the address the borrower pays to. On the host chain, the same hash is what the verifier checks a funded output against. Because both parties inspected the script before it was funded, a match proves every term of the lock: the secret's hash, the lock time and the two parties' keys.

The secret. For each tranche the lender chooses a secret and publishes its hash in the term sheet. The Bitcoin script releases the collateral to the borrower on presentation of the secret. The loan

contract releases the repayment to the lender on presentation of the same secret. Publishing it on the host chain is settlement: it pays the lender and hands the borrower the key in one step.

2.4 Properties

Property	How it is achieved
The borrower's Bitcoin is never held by anyone	It sits in a Bitcoin script spendable only by the borrower with the secret or by the lender after the time lock.
The lender's funds are never held by anyone	They sit in a contract with no administrative function that can move them.
Principal is paid only against verified collateral	The contract credits each tranche's share of the principal only after the verifier has checked a proof that the tranche's address was paid the agreed amount.
Repayment and release are one act	`revealSecret` pays the lender and publishes the secret in one transaction. There is no state in which the lender has the repayment and the borrower lacks the secret.
A lender cannot profit by withholding the secret	A tranche left unrevealed after the reveal window is liquidated from the lender's own collateral and the stuck repayment, at a premium fixed in the terms. Section 12.2 describes the one case, after a lender's health exit, in which no collateral remains to fund this.
A borrower who defaults loses only the defaulted tranches	Each tranche is an independent script with its own secret and lock time. Repaid tranches are unaffected by defaulted ones.
No price oracle on the main path	Funding, repayment, release, default and liquidation use fixed quantities written into the terms. Prices are read only for the optional early-repayment rebate and the optional health module.
No privileged path to funds	The contract has one administrative function, which pauses the creation of new loans. No role can move, freeze or redirect any balance.
Every change of state is public	Every transition emits an event on the host chain, and every Bitcoin movement is a transaction on the Bitcoin network.

2.5 Comparison with other lending models

	Centralized Bitcoin lending	Pool-based lending with wrapped Bitcoin	MaxyLock
Where the Bitcoin is	The lender's or a custodian's wallet	Locked with a bridge or custodian that issues a wrapped token	A Bitcoin script the borrower funded, which neither party controls alone
Counterparty	The lending company	The pool and its governance	One lender, one borrower
Interest rate	Set by the company	Set by pool utilization	Fixed in the term sheet
What the lender relies on	Its own custody and a price feed	The bridge, the pool's oracle and its liquidators	A proof of funding verified on chain, and a fixed time lock
Liquidation	At the company's discretion	Oracle-priced, by any liquidator	Only on non-payment at the tranche's deadline, or by signed health terms if the parties enable them
Effect of a price fall	Forced liquidation	Forced liquidation	None on the main path; an optional cure window if the health module is enabled
What a repaid borrower relies on for the return of collateral	The company's solvency and conduct	The bridge or custodian	The secret the lender must publish to be paid
Remedy if the lender does not perform	Legal process	Not applicable	The lender's collateral buys Bitcoin for the borrower automatically
Record of the loan	The company's ledger	Pool accounting	Public transactions on Bitcoin and the host chain, tied to each other by the script hashes and the secrets

3. Key concepts

The terms below are used throughout this document. Each is defined once here.

3.1 Bitcoin terms

Transactions, outputs and addresses. A Bitcoin transaction moves value from one or more inputs to one or more outputs. Each output holds an amount and a rule that determines who may spend it. An address is a readable encoding of that rule. An unspent output is identified by the transaction identifier (txid) and the output's position within the transaction, called its index or `vout`; the pair is called an outpoint.

Satoshi. The smallest unit of Bitcoin. One bitcoin is 100,000,000 satoshis, abbreviated sats. Every Bitcoin amount in the contracts is a number of satoshis.

Confirmations. A transaction is confirmed when a block includes it, and each later block adds one confirmation. The term sheet fixes how many confirmations a funding transaction needs before the loan contract accepts it: from the verifier's floor (at least 1) to 8.

Block header. An 80-byte summary of a block that commits to every transaction in it. A system that holds the chain of headers can verify that a transaction belongs to a block without holding the block itself.

Hash and preimage. A hash function converts any input into a fixed-size value called a hash. The input is called the preimage. MaxyLock uses SHA-256. Computing the hash from the preimage is immediate. Recovering the preimage from the hash is infeasible. For each tranche the lender chooses a 32-byte secret `S` and gives its hash `H`, called the hashlock, to the borrower before the terms are signed. `H` is written into the term sheet and into the Bitcoin script. `S` stays with the lender until the lender reveals it.

Hash lock and time lock. A hash lock is a spending rule satisfied by presenting the preimage of a stated hash. A time lock is a spending rule satisfied only after a stated time.

HTLC. A hash time-locked contract is a Bitcoin script that combines the two rules. In MaxyLock each tranche is an HTLC with two branches. The borrower may spend it by presenting `S` and signing with the borrower's Bitcoin key. The lender may spend it by signing with the lender's Bitcoin key once the time lock has passed. The Bitcoin network enforces the script. The HTLC is not a contract on the host chain. The loan contract records only the hash of the script and verifies proofs that it was paid. Chapter 18 gives the reference script.

Script hash and P2WSH. The address of an HTLC is the pay-to-witness-script-hash form of the script. Its output script is the two bytes `00 20` followed by the SHA-256 of the witness script. The

term sheet carries the 32-byte script hash of each tranche, and the loan contract builds the output script from it.

Time lock of a tranche. The Unix timestamp written into a tranche's HTLC, called `btcTimelock` in the contracts. Before it, only the borrower's path is available. After it, both paths are available, and the first transaction to confirm takes the funds.

Proof of inclusion. A compact proof that a transaction is contained in a specific Bitcoin block. It consists of the raw transaction, the header of the block, and the Merkle path that connects them. The verifier contract on the host chain maintains a copy of the Bitcoin headers and checks proofs against it.

3.2 Loan and contract terms

Loan contract. The host-chain contract that records loans, holds the lender's deposits, verifies funding proofs, settles repayments against secrets and pays out balances. It is called `MaxyLockLoan` in the source. Part of its code lives in a companion contract, `MaxyLockLoanExtensions`, which holds ghost liquidation and Module B and which the loan contract calls by delegation. All state lives in the loan contract. To users and integrators the two behave as one contract at one address.

Term sheet. The complete set of terms for one loan, signed by the borrower under EIP-712 and submitted by the lender. It names both parties, the loan asset and principal, the collateral asset and amount the lender posts, every tranche's script hash, hashlock, amount and lock time, the deadlines, the interest rebate and protection settings, the Bitcoin script the borrower wants paid in a liquidation, and a nonce. It is called `LoanTerms` in the contracts.

Loan and loan identifier. The record the loan contract creates when the lender commits a term sheet. Its identifier, `loanId`, is a counter starting at 1. A loan passes through the states described in Chapter 8.

Loan asset. The host-chain asset the loan is made in. Any ERC-20 token, or the chain's native asset, may be used. The reference deployments use USDC on Base and USDG on Robinhood Chain.

Principal. The amount of the loan asset the lender provides. It is divided among the collateral tranches in proportion to their Bitcoin amounts.

Principal share. The portion of the principal that one collateral tranche backs. It is fixed when the loan is committed, in proportion to the tranche's satoshis. The borrower receives it when the tranche is proven and repays it when the tranche is repaid.

Lender's collateral. An amount of a host-chain asset the lender posts at commit time. It backs the lender's obligation to release the secret after repayment. It is returned to the lender tranche by tranche as the loan settles, and forfeited tranche by tranche if the lender fails to perform. The

collateral asset may be the loan asset or a yield-bearing form of it for which a redemption route is configured.

Tranche. One portion of the collateral, with its own Bitcoin script hash, secret hash, amount in satoshis and time lock. A loan has `N` collateral tranches, numbered 0 to `N − 1`, and one interest tranche, numbered `N`. Each tranche has its own state.

Interest tranche. The tranche that prepays the lender's interest in Bitcoin. It has no secret and no principal share. The lender collects it on the Bitcoin network shortly after the verification deadline.

Escrow. The repayment for a tranche, held by the loan contract between the borrower's deposit and the lender's reveal. It is released to the lender only when the lender reveals the tranche's secret.

Reveal window. The period, fixed in the term sheet and at least one day, during which the lender may reveal the secret after a repayment. Its end is the point at which the lender is treated as having failed to perform.

Timing rule. Every collateral tranche's Bitcoin lock time must lie after the end of the term plus the reveal window plus a safety margin of two days. The rule guarantees that a borrower who repays always has time to receive the secret, or to start a liquidation, before the lender's time lock can open. Section 8.6 states it precisely.

Collateral slice. The portion of the lender's remaining collateral that one tranche claims when it resolves. It is proportional to the tranche's satoshis among the tranches that have not yet resolved.

Payout ledger. The contract's record of what it owes to whom, in which asset. Every credit the contract makes, whether principal, repayment, collateral or bounty, is written to the ledger. The party owed withdraws it with `claimPayout`. The contract never pushes funds to a party.

Proof bounty. A reward, paid from the lender's collateral, to a third party who submits a funding proof after the verification deadline. It ensures that a valid funding is recognized even if both parties' systems fail to submit it.

Mega-Pot. The remedy for a lender who is repaid but does not reveal. The lender's collateral slice for the tranche and the borrower's escrowed repayment are combined into a fixed-quantity buy order for Bitcoin, posted to the settler. A bonded filler delivers Bitcoin to the borrower's address and is paid from the pot. The quantity is a premium over the tranche's Bitcoin, fixed in the term sheet.

Verifier. The host-chain contract that stores Bitcoin block headers and verifies that a transaction with specified outputs is contained in a specified block. It is shared with other Maxy systems.

Settler. The host-chain settlement venue to which Mega-Pot orders are posted. Through it, parties exchange host-chain assets for Bitcoin delivered to a named address, with delivery verified by the same verifier. Bonded fillers deliver the Bitcoin. The loan contract uses the settler only for Mega-Pot liquidations. It is shared with other Maxy systems.

Oracle adapter. The host-chain price source used only by two optional features: the early-repayment rebate and the health module. It combines a Pyth feed with a Chainlink feed and rejects prices on which the two disagree. It is not used in funding, repayment, release, default or liquidation.

Redemption route. A pairing, fixed at deployment, of a yield-bearing collateral asset with its underlying asset and an adapter that redeems one into the other. It allows the collateral asset and the loan asset to differ.

3.3 Roles

Borrower. The party that locks Bitcoin and receives the loan. In the contracts the borrower is the host-chain address that signed the term sheet. Only that address may repay, cure a health flag or claim the principal. The borrower also controls a Bitcoin key named in every tranche's script.

Lender. The party that lends the loan asset and posts the collateral asset. In the contracts the lender is the host-chain address named in the term sheet. Only that address may commit the loan, reveal secrets or exit after an uncured health flag. The lender also controls a Bitcoin key named in every tranche's script, and holds the secret for each collateral tranche.

Submitter. Anyone who submits a funding proof. Normally the lender or the borrower. A third party who submits a proof after the verification deadline earns a bounty.

Filler. A bonded third party that fills a Mega-Pot order on the settler by delivering Bitcoin to the borrower's address. Fillers post a bond and are slashed if they commit and fail to deliver.

Pauser. The address that may pause and unpause the creation of new loans. It has no other power.

Anyone. Several functions are open to any caller: submitting proofs, finalizing funding, marking defaults, triggering and settling liquidations, and flagging or clearing health. The contract decides the outcome from its state. The caller only supplies gas.

Maxy's services. In the loans Maxy offers, Maxy is the lender. Its **solver** quotes the loan terms and acts as the lender, and its **keeper** is the automated service that commits loans, proves funding, reveals secrets after repayment, finalizes funding and sweeps defaulted and interest tranches. Maxy also operates a header relayer for the verifier; anyone may run one. These are services built on the protocol, not part of it. Where this document describes Maxy's own template or practice rather than a rule of the contracts, it says so (7.3, 7.6, 9.8, 13.10, 16.4 and 18.7).

4. System components

The protocol spans three environments. The Bitcoin network holds the borrower's collateral in HTLCs. The host chain holds the loan records, the lender's deposits, the verifier, the settler and the oracle adapter. The parties' wallets and any watcher services connect the two: they build Bitcoin transactions and proofs, sign and submit term sheets, and act on events. No protocol operator sits between them.

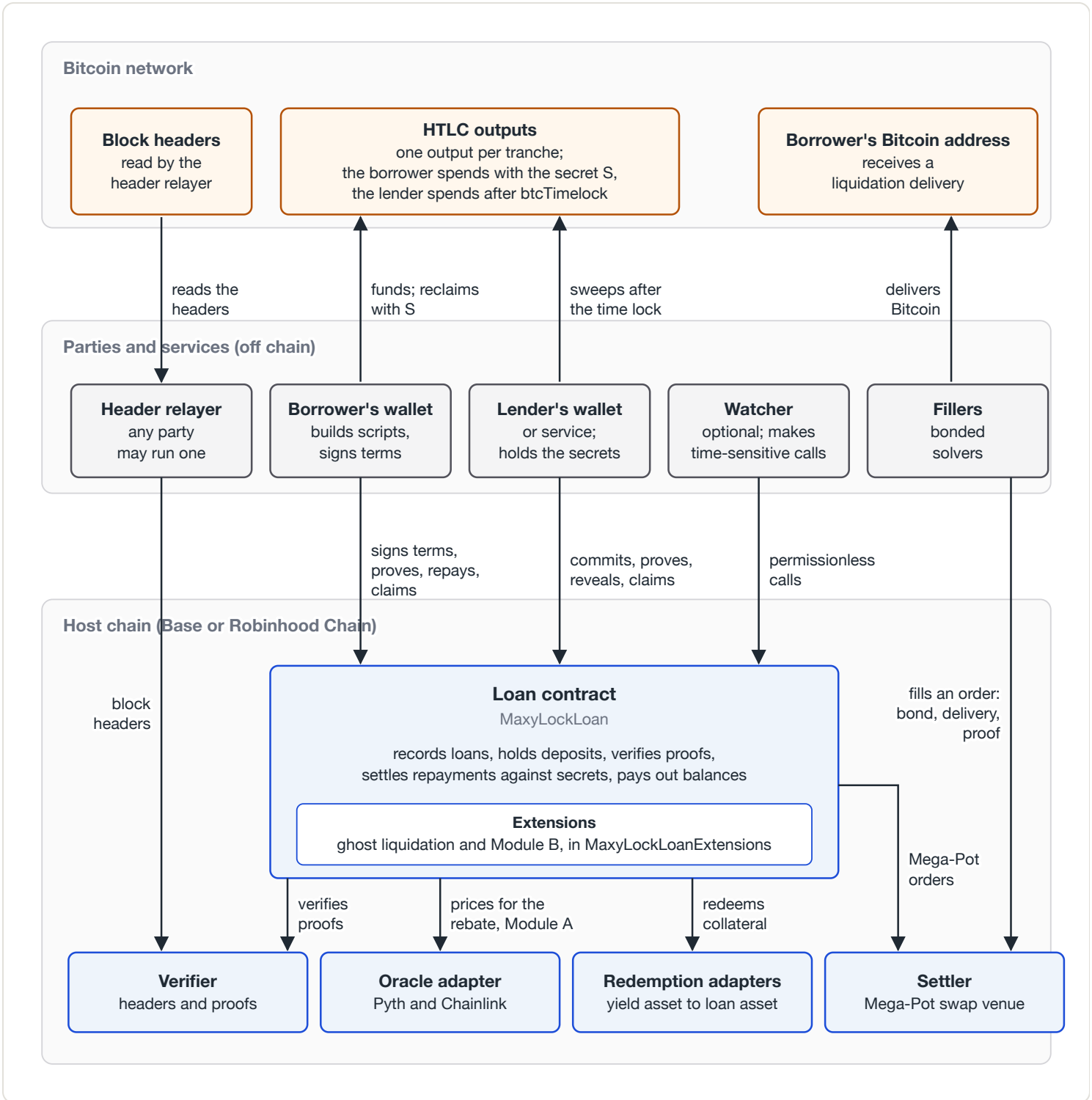


Figure: System components across Bitcoin, the parties' wallets and the host chain.

The borrower funds HTLCs and reclaims Bitcoin on the Bitcoin network, and signs terms, repays and claims on the host chain. The lender commits loans, reveals secrets and claims on the host chain, and sweeps defaulted and interest tranches on the Bitcoin network. Either party, or a watcher, submits proofs. The loan contract verifies funding through the verifier and, if a lender fails to perform, posts a Mega-Pot order to the settler, which fillers deliver Bitcoin against.

4.1 Components

Component	Where it runs	Function
HTLC outputs	Bitcoin	Hold the borrower's collateral, one output per tranche plus the interest tranche. Spendable by the borrower with the tranche's secret, or by the lender after the tranche's time lock.
Borrower's wallet	Borrower's device	Builds the HTLC scripts, signs the term sheet, funds the HTLCs, submits proofs where required, repays tranches, withdraws principal, reads revealed secrets and spends the Bitcoin.
Lender's wallet or service	Lender's systems	Generates secrets and hashlocks, prepares term sheets, commits loans, submits proofs, reveals secrets after repayment, sweeps defaulted tranches and the interest tranche, and withdraws payouts.
Header relayer	Off chain	Supplies Bitcoin block headers to the verifier so that proofs can be checked. Any party may operate one; the verifier validates proof of work itself.
Watcher	Off chain, optional	Monitors loans for reveal windows, cure windows and unproven funding, and submits the permissionless calls. Any party may operate one.
Loan contract (`MaxyLockLoan`)	Host chain	Records loans, holds principal, collateral and escrow, verifies funding proofs, settles repayments against secrets, resolves defaults, credits payouts and runs the protection logic. Part of its code lives in a companion contract (`MaxyLockLoanExtensions`) that it calls by delegation; to users and integrators the two behave as one contract at one address.
Verifier	Host chain	Stores Bitcoin block headers and verifies that a transaction's outputs pay stated amounts to stated scripts in a confirmed block.

Component	Where it runs	Function
Settler	Host chain	An existing intent-based venue for exchanging host-chain assets for Bitcoin delivered on chain. Receives Mega-Pot buy orders, holds the pot, and releases it to a filler that proves delivery of Bitcoin to the borrower.
Fillers	Off chain	Bonded third parties on the settler who deliver Bitcoin to the borrower's address against a Mega-Pot order and are paid from the pot.
Loan asset and collateral asset	Host chain	The token the loan is made in, and the token the lender posts: the loan asset itself, or a yield-bearing form of it for which a redemption route is configured at deployment.
Oracle adapter	Host chain	Prices Bitcoin in the loan asset for the rebate and the health module. Not used in funding, repayment, release, default or liquidation.
Redemption adapters	Host chain	Convert a yield-bearing collateral asset into the loan asset during a liquidation. Fixed at deployment.

4.2 Enforcement layers

Five distinct mechanisms protect the parties. The remainder of this document refers to them separately.

- **Bitcoin consensus** enforces the HTLCs. The borrower cannot spend a tranche without its secret, and the lender cannot spend it before its time lock.
- **The loan contract** enforces the exchange. No principal is released without a verified proof. No repayment reaches the lender without the secret. No collateral is released to the lender except through a reveal, a default, an unfunded tranche or an uncured health flag. No function moves funds to anyone other than the party the terms entitle.
- **The settler** enforces the liquidation. A filler receives the Mega-Pot only after a verified proof that it paid the borrower's Bitcoin address, and forfeits its bond if it commits and does not deliver.
- **The term sheet** governs what neither chain can decide: the amounts, the deadlines, the rate of interest, the premium a non-performing lender pays, and whether the optional protection modules apply. Both parties examine the terms and the Bitcoin scripts before anything is committed.
- **The parties' own procedures** govern what no chain can enforce. The lender keeps its secrets safe and reveals promptly after each repayment. The borrower funds before the deadline and submits the proof if the lender does not. Both parties inspect each Bitcoin script before relying on it. Chapter 16 sets out the consequences of each failure.

5. The loan process

This chapter follows one loan from the term sheet to the return of the collateral. The diagram shows the messages exchanged between the borrower, the lender, the Bitcoin network and the loan contract. Steps 1 to 9 are the normal path. Step 10 is the alternative when the borrower does not repay. Step 11, the alternative when the lender does not reveal, is drawn in full in chapter 11.

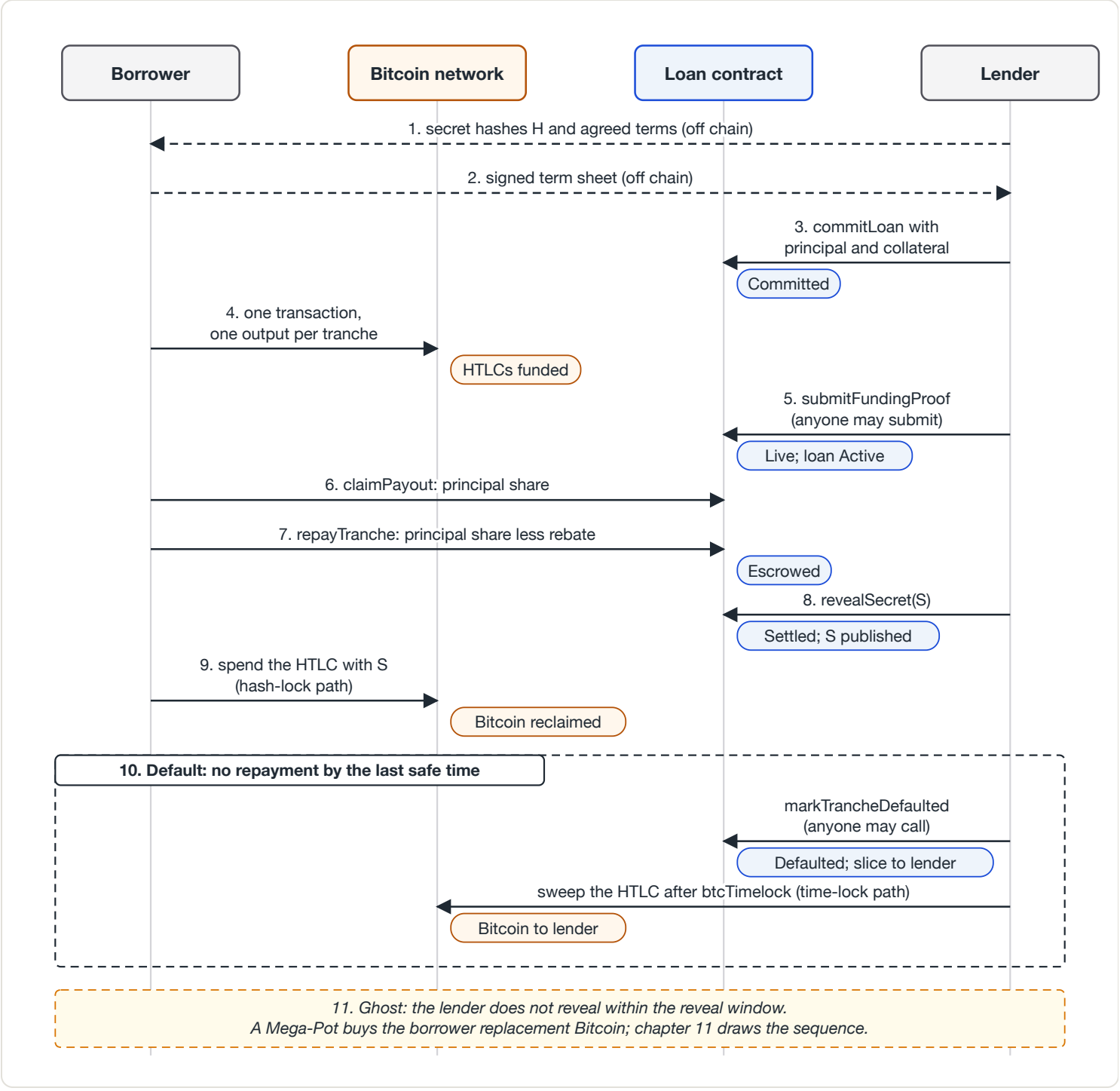


Figure: Loan sequence, with the default alternative.

5.1 Steps

1. **The parties agree the terms.** Off chain, the lender generates one secret per collateral tranche and gives the borrower the hashes. The two parties settle the amounts, the deadlines, the number of tranches and the module settings. Both parties construct the HTLC script for each tranche from its hash, the time lock, the borrower's Bitcoin key and the lender's Bitcoin key, and both verify that the script hashes in the term sheet match what they built. The term sheet carries the SHA-256 of each script.
2. **The borrower signs the term sheet.** The signature is an EIP-712 signature over the `LoanTerms` structure, made with the borrower's host-chain key. The borrower gives it to the lender.
3. **The lender commits the loan.** The lender approves the loan contract to spend the principal and the collateral, then calls `commitLoan` with the term sheet and the signature, transferring both deposits in the same transaction. The contract verifies the signature, checks every term against the rules in Chapters 8 and 13, records the loan as `Committed` with every tranche `Pending`, assigns each tranche its share of the principal, and emits `LoanCommitted` with the script hashes and amounts. The loan now has an identifier.
4. **The borrower funds the HTLCs.** The borrower sends one Bitcoin transaction with one output per tranche, including the interest tranche, each paying the tranche's exact amount to the tranche's address. All tranches must be funded by the same transaction. The borrower waits for the required number of confirmations.
5. **The funding proof is submitted.** The lender, the borrower or anyone else submits a proof of inclusion for the funding transaction, naming which output pays which tranche. The verifier confirms that the transaction is in a confirmed block and that each named output pays the exact amount to the registered script. Each proven collateral tranche becomes `Live`, and its principal share is credited to the borrower's payout ledger. The loan becomes `Active` on the first proven collateral tranche, and the term runs from that moment to the signed `termEnd`.
6. **The borrower withdraws the principal.** The borrower calls `claimPayout` for the loan asset and receives the credited shares.
7. **The borrower repays a tranche.** Before the tranche's last safe repayment time (Section 8.5), the borrower calls `repayTranche`, depositing the tranche's principal share, less any early-repayment rebate, into escrow. The tranche becomes `Escrowed` and the reveal window starts.
8. **The lender reveals the secret and is paid.** Within the reveal window, the lender calls `revealSecret` with the tranche's secret. The contract checks that its SHA-256 equals the hashlock recorded at commit time, credits the escrowed repayment and the tranche's collateral slice to the lender's ledger, publishes the secret in `SecretRevealed`, and marks the tranche `Settled`.
9. **The borrower reclaims the Bitcoin.** The borrower reads the secret from the event and spends the tranche's HTLC through the hash-lock branch, sending the Bitcoin to any address of the borrower's

choosing. Nothing further is required on the host chain. Steps 7 to 9 repeat for each tranche, in any order the borrower chooses.

The lender collects the interest tranche on the Bitcoin network through its time-lock branch, which opens ten minutes after the verification deadline. When every tranche has reached a final state on the host chain, the loan is marked `Settled`. Chapter 8 describes when that occurs; the tranche states, not the loan state, carry the outcome.

10. **Default, in place of steps 7 to 9.** If the borrower does not repay a tranche before its last safe repayment time, anyone may call `markTrancheDefaulted`. The contract credits the tranche's collateral slice back to the lender and marks the tranche `Defaulted`. The lender sweeps the tranche's Bitcoin when its time lock opens.
11. **Ghost liquidation, in place of steps 8 and 9.** If the lender does not reveal within the reveal window, anyone may call `triggerGhostLiquidation`. The contract pools the tranche's collateral slice and the escrowed repayment into a Mega-Pot and posts an order on the settler to buy a fixed quantity of Bitcoin for the borrower. A filler delivers the Bitcoin and takes the pot. Chapter 11 describes the mechanism.

5.2 Worked example

Alice borrows 30,000 USDC from Bob against 0.5 BTC, for a term that ends 180 days after the verification deadline. They agree two collateral tranches of 0.25 BTC (25,000,000 sats each) and an interest tranche of 150,000 sats. Bob posts 30,000 USDC as collateral. The term sheet sets `requiredConfirmations = 3`, a funding deadline one day ahead, a verification deadline one day after that, `termEnd` 180 days after the verification deadline, a reveal window of 7 days and `ghostPayoutBps = 10500`.

Bob generates two secrets and gives Alice their hashes. Alice builds the two collateral scripts with time locks 30 days after `termEnd + 7 days + 2 days`, and the interest script with a time lock exactly 10 minutes after the verification deadline. She signs the term sheet. Bob commits the loan, depositing 60,000 USDC in total. The contract fixes each collateral tranche's principal share at 15,000 USDC.

Alice sends one Bitcoin transaction with three outputs: 25,000,000 sats to each collateral address and 150,000 sats to the interest address. After three confirmations Bob submits the proof for all three outputs. Both collateral tranches become `Live`, the loan becomes `Active`, and Alice's ledger shows 30,000 USDC. She claims it. Ten minutes after the verification deadline Bob sweeps the interest tranche on Bitcoin.

After 90 days Alice repays tranche 0. The rebate is not enabled, so she escrows 15,000 USDC. Bob reveals the secret within the window. The contract credits Bob 15,000 USDC of repayment and 15,000 USDC of collateral, which is tranche 0's slice of his 30,000 USDC. Alice reads the secret and spends

her 0.25 BTC. Tranche 1 proceeds the same way before its deadline. Bob's ledger then holds 60,000 USDC and Alice holds her 0.5 BTC. Both collateral tranches are `Settled`. The loan record itself stays `Active`, because the proven interest tranche has no further transition on the host chain; it was settled on Bitcoin by Bob's sweep. Chapter 8 explains.

If Alice had repaid tranche 0 only, tranche 1 would have been marked `Defaulted` after its last safe repayment time. Bob would have received tranche 1's 15,000 USDC collateral slice back on the host chain and swept 0.25 BTC on Bitcoin when tranche 1's time lock opened. The 15,000 USDC principal share for that tranche would never have been repaid; Alice keeps it, together with the other 0.25 BTC.

Section 10.3 continues with a loan in which the early-repayment rebate is enabled.

5.3 The default case

A default is confined to the tranche. The borrower keeps the principal share for that tranche and loses that tranche's Bitcoin. The lender recovers that tranche's collateral slice on the host chain and takes the Bitcoin on Bitcoin. Other tranches, repaid or still live, are unaffected. The parties set each tranche's principal share below the value of its Bitcoin, so a default is a losing outcome for the borrower and the lender is made whole by the Bitcoin. The contract does not enforce that ratio; the parties set it in the terms. The protocol does not liquidate at a market price to protect the lender; the lender keeps the Bitcoin itself.

The repayment window and the default window are exact complements. For every tranche there is a single moment, the last safe repayment time, before which only repayment is possible and from which only default is possible. There is no moment at which both are possible and none at which neither is. Chapter 8 defines the moment.

5.4 The ghost case

A lender who withholds a secret after being repaid is not trusted to return. Once the reveal window closes, the tranche's collateral slice and the escrowed repayment leave the lender's reach permanently and become a Mega-Pot. The pot is posted on the settler as an order to buy Bitcoin for the borrower on the open market: it offers the whole pot for a fixed quantity of Bitcoin, the tranche's amount plus the premium fixed in the terms (`ghostPayoutBps`, between 100% and 150%; 105% in the loans Maxy offers). Any bonded filler may take the order. Fillers compete at the market price, deliver the Bitcoin to the address the borrower fixed at signing, and keep what remains of the pot, so the order fills whenever the pot is worth more than the Bitcoin it buys. The borrower ends with more Bitcoin than the tranche held, paid for entirely by the lender's forfeited collateral and repayment. The replacement is a term of the loan agreement that the contracts carry out on their own, not a sale by the borrower. Section 12.2 describes the one case, after a lender's health exit, in which the collateral slice has already left. The lender keeps only what the Bitcoin network gives it: the original tranche,

once its time lock opens. The lender has paid for that Bitcoin with the collateral slice and the forfeited repayment, and the borrower has been compensated from the same funds.

The three outcomes for a tranche are mutually exclusive. A reveal publishes the secret and returns the Bitcoin to the borrower. A default never publishes the secret and gives the Bitcoin to the lender. A liquidation never publishes the secret and replaces the Bitcoin from the lender's collateral. There is no sequence of events in which a borrower who repaid ends with neither the Bitcoin nor a standing claim to its replacement.

5.5 What the contract never does

The contract never sends funds to a party on its own initiative. Every credit, including the principal, sits in the payout ledger until its owner withdraws it. The contract never takes custody of Bitcoin; it only verifies proofs about Bitcoin. And it never releases the lender's repayment without publishing the secret, so the borrower's key to the collateral is never withheld from a borrower who has paid.

6. Flow of funds and custody

This chapter states where each asset is at each stage of a loan, who can move it, and what the contract records. Readers responsible for treasury, risk or audit should find here every transfer of value the protocol performs.

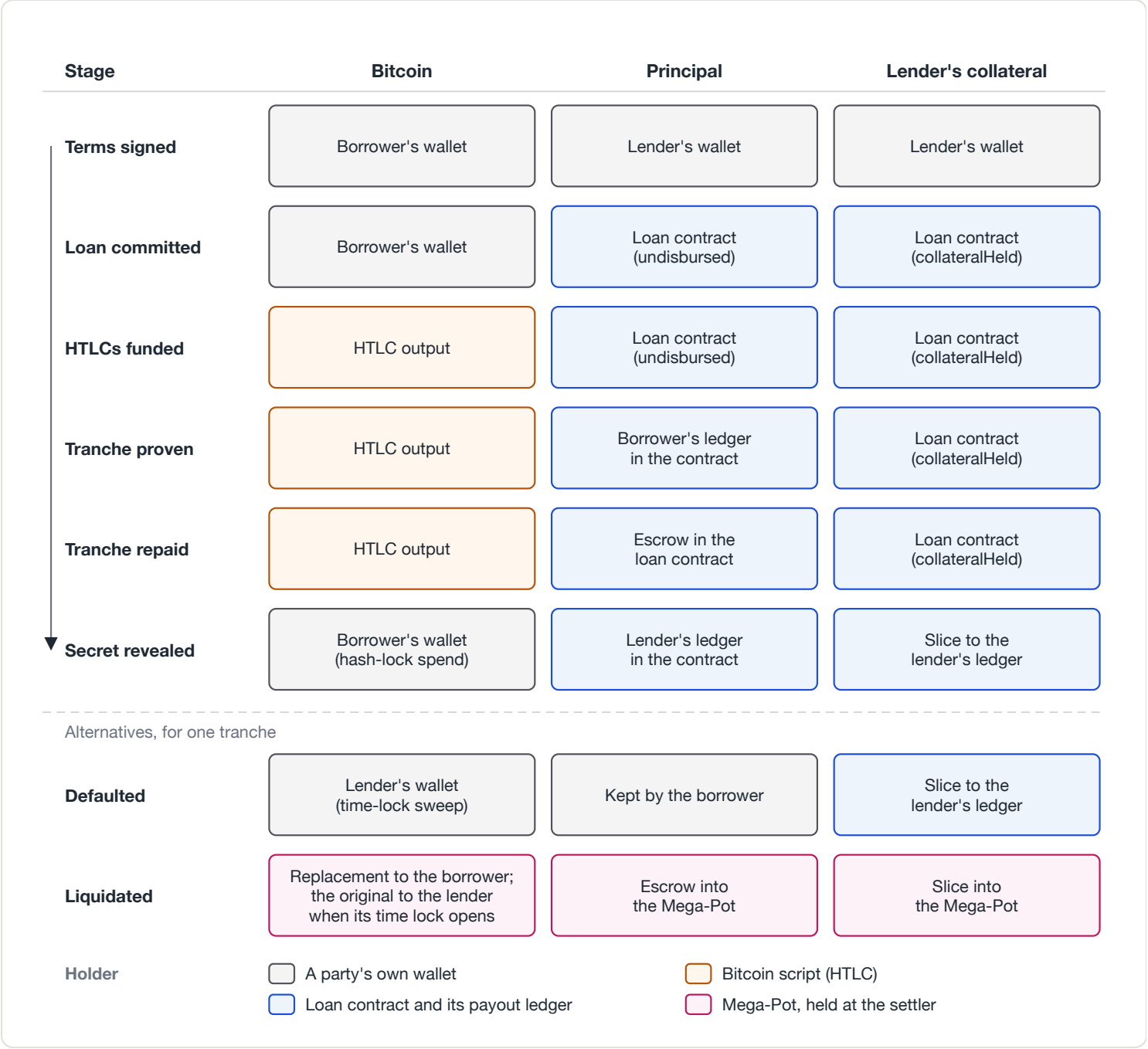


Figure: Custody of the Bitcoin and of the host-chain deposits at each stage.

The lender's principal and collateral move from the lender's wallet into the loan contract at commit. The principal moves to the borrower's ledger tranche by tranche as proofs arrive. Each repayment

moves from the borrower into escrow, then to the lender's ledger at the reveal, together with the tranche's collateral slice. The Bitcoin moves from the borrower's wallet into the HTLCs and back to the borrower at the reveal, or to the lender at a default. The same secret unlocks the repayment on the host chain and the Bitcoin on the Bitcoin network.

6.1 Stages

Stage	Bitcoin	Principal	Lender's collateral	Record on chain
Terms signed	Borrower's wallet	Lender's wallet	Lender's wallet	None; the signed term sheet is held by both parties
Loan committed	Borrower's wallet	Loan contract, undisbursed	Loan contract, `collateralHeld`	`LoanCommitted` on the host chain, with every script hash and sats value
HTLCs funded	HTLC outputs on Bitcoin; the borrower can spend with the secret, the lender after the lock time	As above	As above	The funding transaction on Bitcoin
Tranche proven	HTLC outputs	Tranche's share credited to the borrower's ledger	As above	`TrancheVerified` per tranche and `LoanActivated` on the first
Principal claimed	HTLC outputs	Borrower's wallet	As above	`PayoutClaimed`
Tranche repaid	HTLC outputs	Escrow in the loan contract, as `escrowAmount`	As above	`TrancheRepaid`, with the reveal deadline

Stage	Bitcoin	Principal	Lender's collateral	Record on chain
Secret revealed	Spendable by the borrower with the secret, then the borrower's wallet after the hash-lock spend	Lender's ledger	Tranche's slice credited to the lender's ledger	`SecretRevealed` on the host chain, with the secret; the spend on Bitcoin
Tranche defaulted (alternative)	Lender's wallet, after the time-lock spend	Retained by the borrower	Tranche's slice credited to the lender's ledger	`TrancheDefaulted` on the host chain; the sweep on Bitcoin
Tranche liquidated (alternative)	Replacement Bitcoin at the borrower's address; the original to the lender at the time lock	Escrow moved into the Mega-Pot	Tranche's slice moved into the Mega-Pot	`GhostLiquidationTriggered`, `MegaPotOpened`, `MegaPotDelivered`
Interest tranche	Lender's wallet, after the time-lock spend	None	None	The sweep on Bitcoin, after the verification deadline

6.2 Encumbrance while in the HTLC

While a tranche sits in its HTLC, neither party has ordinary control of it. The borrower can move it only with the secret, which the lender holds until a repayment is made. The lender can move it only after the time lock, which is set past the last point at which the tranche can be repaid, revealed or liquidated. Neither party can direct the funds to any third party except through one of those two paths. The HTLC is not an escrow held by an intermediary. It is a rule enforced by the Bitcoin network with no custodian at all.

Until one of those two paths is taken, the Bitcoin remains the borrower's property. The lender acquires no right to it by its arrival at the address, and none by the borrower's default alone. The lender acquires it only by spending the script after the lock time, which the script permits because the tranche was never repaid.

6.3 Custody on the host chain

The loan contract holds the lender's principal until it is disbursed, the lender's collateral until it is released, each repayment while it is in escrow, and each Mega-Pot until the settler takes it. It never holds anything it does not owe to a named party or to a reclaimed pot awaiting re-posting. Balances owed to a party sit in the payout ledger and leave only when that party calls `claimPayout`. All of these balances are accounted for per loan and per tranche.

Balance	Enters	Leaves
Undisbursed principal	At commit, from the lender	To the borrower's ledger as each tranche is proven; back to the lender's ledger for tranches never funded
Lender's collateral (`collateralHeld`)	At commit, from the lender	To the lender's ledger slice by slice as tranches settle or default; to a Mega-Pot if the lender fails to reveal; to a submitter as a proof bounty; to the lender in full after an uncured health flag
Escrow	At repayment, from the borrower	To the lender's ledger on reveal; to a Mega-Pot if the lender fails to reveal
Reclaimed Mega-Pot	From the settler, when an unfilled order is reclaimed	Back to the settler when the order is re-posted
Payout ledger	Every credit above	To its owner, only through `claimPayout`

The contract has no function that moves any of these balances to an address other than the one the terms entitle. The only administrative function pauses new commitments; it cannot touch an existing loan. The contract's balance in any asset therefore always equals the sum of five quantities in that asset: the ledger balances owed, the escrows, the collateral held, the undisbursed principal, and any reclaimed pot awaiting re-posting. The contract's own test suite checks this identity under random sequences of operations.

6.4 Custody of the lender's collateral asset

The lender's collateral may be the loan asset itself or a yield-bearing form of it, such as an interest-bearing deposit token, provided a redemption route for that token is fixed at deployment. While the loan is active the collateral sits in the loan contract, which returns to the lender exactly the number of units it recorded at commit, slice by slice. For a token whose value per unit grows, the yield returns to the lender with the units. For a token whose balance grows, the growth above the units recorded is not credited to either party by the core logic. The optional yield-hedge module (Chapter 12) forwards a signed fraction of that growth to a hedge account at each skim, and growth that has not been

skimmed stays in the contract. A lender who does not enable Module B therefore posts the loan asset itself or a token whose value per unit grows.

6.5 Where each party's value ends up

For one collateral tranche with principal share `P`, collateral slice `C` and Bitcoin amount `B`, the final position of each party is the following.

Outcome	Borrower	Lender
Repaid and revealed	Received `P`, repaid `P` less any rebate, keeps `B`	Lent `P`, received `P` less any rebate, recovers `C`, keeps the interest tranche
Defaulted	Keeps `P`, loses `B`	Lent `P`, recovers `C`, receives `B`
Liquidated	Received `P`, repaid `P` less any rebate, loses `B`, receives `B × ghostPayoutBps / 10,000` in new Bitcoin	Lent `P`, forfeits the repayment and `C` to the pot, receives `B` at the time lock

In the liquidated case, ignoring any rebate, the lender's net loss is `P + C` less the value of `B`. A lender that revealed would instead have recovered `P` and `C` in full. Ghosting is therefore a losing choice for the lender whenever `C` exceeds the value of `B` less `P`, and the parties should size `C` with that condition in mind. The borrower's net gain is the premium over `B`. The filler's margin is the difference between the pot and the market price of the Bitcoin it delivered.

6.6 Fees and costs

Cost	Paid by
Interest	Borrower, prepaid in Bitcoin through the interest tranche
Bitcoin network fee for the funding transaction	Borrower
Bitcoin network fee for the hash-lock spend after a reveal	Borrower
Bitcoin network fee for the time-lock spend after a default, and for the interest sweep	Lender
Host-chain gas for `commitLoan`	Lender
Host-chain gas and verifier fee for `submitFundingProof`	Whoever submits it
Host-chain gas for `repayTranche` and `revealSecret`	Borrower and lender respectively

Cost	Paid by
Host-chain gas for `claimPayout`	The claimant
Host-chain gas for `markTrancheDefaulted`, `triggerGhostLiquidation` and the other permissionless calls	Whoever makes them
Proof bounty (`bountyBps`)	Deducted from the lender's collateral, only when a third party proves funding after the verification deadline
Mega-Pot premium on a lender's non-performance	Lender, from its collateral slice and the forfeited repayment
Settler bond for a Mega-Pot order (5%)	Filler; forfeited to the pot if the filler commits and does not deliver
Bitcoin network fee for a Mega-Pot delivery	Filler

The verifier may charge a fee per proof, payable in the host chain's native currency or in a token. The submitter supplies the fee with the proof. Any unspent portion of a token fee is credited to the submitter's ledger; a native fee is forwarded in full and should be exactly the quoted amount. A deployment may register the loan contract as a fee-exempt caller, in which case the fee is zero.

7. Tranches

A loan's collateral is not one lump. It is a list of tranches, each an independent HTLC on Bitcoin and an independent record on the host chain. This chapter explains why, how the principal is divided among them, and how the lender's collateral is divided when they resolve.

7.1 Why tranches

If the collateral were a single HTLC with a single secret, the loan could only be repaid in full or defaulted in full. A borrower short of the last 10% would lose all of the Bitcoin. Tranches make every outcome proportional. The borrower repays the tranches it can and defaults on the rest. The lender is compensated tranche by tranche. A liquidation covers one tranche.

Each tranche therefore has its own Bitcoin script, its own secret and hash, its own amount in satoshis, its own time lock, and its own state on the host chain. The original tranches of one loan are funded by a single Bitcoin transaction with one output per tranche.

7.2 Collateral tranches and principal shares

A loan has `N` collateral tranches, where `N` is between 1 and 7. They are numbered 0 to `N − 1`. When the loan is committed, the contract divides the principal among them in proportion to their satoshis:

```
share[i] = floor(principal × sats[i] / totalCollateralSats)     for i = 1 … N−1
share[0] = principal − (share[1] + … + share[N−1])
```

Any rounding remainder goes to tranche 0, so the shares always sum to the principal exactly. A tranche's share is fixed for the life of the loan. It is the amount the borrower receives when the tranche is proven, and the amount the borrower deposits when the tranche is repaid, less any rebate.

Because every tranche is sized the same way, every tranche carries the same ratio of principal to Bitcoin. A borrower cannot repay the cheap tranches and leave the expensive ones; there are none.

7.3 The interest tranche

The tranche numbered `N` is the interest tranche. It prepays the lender's interest for the whole term in Bitcoin, and it differs from the collateral tranches in three ways.

- It has no secret. By convention its hash lock is recorded as zero; the contract does not check the value, and no reveal or repayment applies to the tranche.
- It has no principal share. Proving it releases nothing to the borrower.

- Its time lock is fixed by rule at exactly ten minutes after the verification deadline (`INTEREST_SWEEP_DELAY`). The lender collects it on the Bitcoin network through the time-lock path as soon as that moment passes.

The interest tranche is settled on Bitcoin, not on the host chain. Proving it on the host chain is optional and has no effect on funds. A proven interest tranche is marked `Live` and has no further transition. An interest tranche that is never proven is marked `Unfunded` when funding is finalized. The difference matters only for the loan-level summary state, described in Chapter 8.

The protocol fixes only the interest tranche's time lock. The script behind it is the parties' choice (18.5). In the loans Maxy offers, the interest tranche uses Maxy's script template `maxylock-htlc-v1` (18.7), which has two time-lock branches and no hash lock:

- **Lender branch**, from `verificationDeadline + 10 minutes`. This is the moment the contract fixes, and Maxy sweeps the interest through it.
- **Borrower recovery branch**, from `verificationDeadline + 7 days 10 minutes`. The borrower can take the interest back alone, if the lender has not swept it first.

Maxy sweeps the interest tranche only from a loan that activated. If a loan is canceled or never activates, Maxy leaves the output for the borrower's recovery branch. This is Maxy's policy, not a rule of the contracts: activation is a fact on the host chain that no Bitcoin script can observe, and once both branches are open the Bitcoin network accepts either spend.

7.4 Collateral slices

The lender's collateral backs the loan as a whole, but it is released tranche by tranche. When a collateral tranche resolves, whether by a reveal, a default or a liquidation, it takes a slice of the collateral the contract still holds:

```
slice = collateralHeld × trancheSats / notYetResolvedSats
```

`notYetResolvedSats` is the sum of satoshis over every collateral tranche that is still `Pending`, `Live` or `Escrowed`, counted before the resolving tranche changes state. The interest tranche is never counted. The last tranche to resolve takes whatever remains, so rounding never strands collateral in the contract.

Because the slice is computed against the tranches not yet resolved rather than against the original total, the ratio of collateral to Bitcoin stays constant however the tranches resolve and in whatever order. A tranche that is never funded takes its slice when funding is finalized, so partial funding reduces the collateral at stake in proportion. Two things change the ratio: a proof bounty, which is paid from the collateral and falls on every remaining tranche alike, and a top-up tranche (7.5), which adds Bitcoin without adding collateral.

In the worked example of Chapter 5, Bob's 30,000 USDC backs 50,000,000 sats. Tranche 0 resolves first and takes `30,000 × 25,000,000 / 50,000,000 = 15,000`. Tranche 1 resolves last and takes the remaining 15,000.

7.5 Top-up tranches

If the health module is enabled and a loan is flagged unhealthy, the borrower may cure it by locking more Bitcoin. The new Bitcoin is recorded as an additional tranche, numbered from `N + 1` upward, with a principal share of zero. A top-up tranche is proven on its own, with a proof for whichever Bitcoin transaction paid it; the contract does not tie it to the loan's funding transaction. Once live it is an ordinary collateral tranche: it counts toward the collateral slices and the health check, and it can be repaid, revealed, defaulted or liquidated like any other. Chapter 12 describes the cure.

A top-up differs from an original tranche in who vets it. Its terms are supplied by the borrower alone, with no signature from the lender, and the proof establishes only that an output paid the stated amount to the stated script hash in a confirmed block. The contract cannot tell whether the script gives the lender a time-lock path, or whether the output has since been spent. A lender relying on Module A should treat a top-up as collateral only once it has inspected the script and the output itself.

7.6 Limits

Limit	Value
Collateral tranches per loan	1 to 7 (`MAX_COLLATERAL_TRANCHES`)
Interest tranches per loan	Exactly 1
Tranches per loan including top-ups	At most 255 (`MAX_TOTAL_TRANCHES`)
Satoshis per tranche	Greater than zero
Outputs provable in one `submitFundingProof` call	8, the verifier's limit, which covers every original tranche

These are the contract's limits. Maxy's lending service applies its own minimum of 15,000 sats to every tranche output, collateral and interest, and its own bounds on the other terms. The SDK documentation lists them against the contract bounds in its policy versus protocol table.

8. Loan and tranche lifecycle

A loan record is created by `commitLoan`. Each of its tranches has its own state, and the loan's state is a summary derived from them. Both can be read with `getLoan(loanId)` and `getTranche(loanId, idx)`.

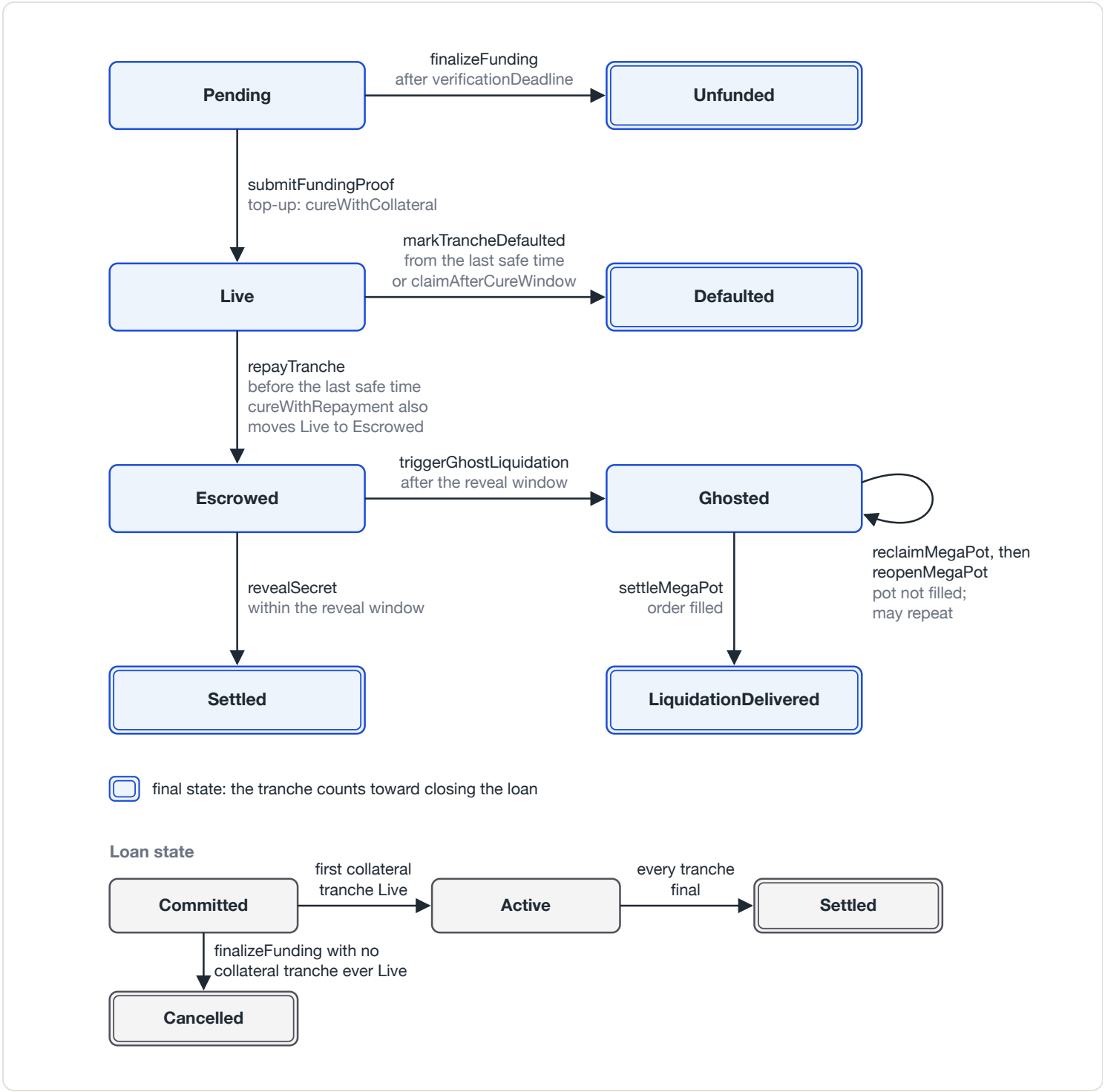


Figure: Tranche states and the calls that move between them, with the loan-level summary below.

Three of the final states are good outcomes for the party that performed: `Settled` for both, `Defaulted` for the lender, `LiquidationDelivered` for the borrower. `Unfunded` returns everyone to where they started for that tranche.

8.1 Loan states

State	Code	Meaning
`None`	0	No record exists for this identifier.
`Committed`	1	The terms are on chain and the lender's deposits are held. No collateral tranche has been proven.
`Active`	2	At least one collateral tranche has been proven and its principal share released. The term has started.
`Settled`	3	Every tranche has reached a final state on the host chain. Final.
`Cancelled`	4	Funding was finalized before any collateral tranche was proven. The lender's deposits have been credited back. Final.

`Settled` means fully resolved on the host chain, not that every tranche went well. A loan with a defaulted tranche and a repaid tranche reaches `Settled` once both are final and the interest tranche is also final. The tranche states carry the outcome, and 8.4 describes when the summary is reached.

8.2 Tranche states

State	Code	Meaning	Final
`Pending`	0	Recorded at commit. No funding proof has been accepted.	No
`Live`	1	A funding proof has been accepted. The principal share, if any, has been released.	No
`Escrowed`	2	The borrower has deposited the repayment. The reveal window is running.	No
`Settled`	3	The lender revealed the secret and was paid. The borrower can spend the Bitcoin.	Yes
`Defaulted`	4	The tranche was not repaid in time. The lender's slice was released and the lender may sweep the Bitcoin.	Yes
`Ghosted`	5	The lender did not reveal. The tranche's value is in a Mega-Pot, open on the settler or awaiting re-posting.	No

State	Code	Meaning	Final
`LiquidationDelivered`	6	A filler delivered the replacement Bitcoin to the borrower and took the pot.	Yes
`Unfunded`	7	No proof arrived before funding was finalized. The lender's share and slice were returned.	Yes

8.3 Transitions

From	To	Call	Who may call	Conditions
`None`	`Committed` (loan)	`commitLoan`	The lender named in the terms	Valid borrower signatu nonce in sequence; ev rule in 8.5 and 13.2; deposits received; commits not paused
`Pending`	`Live`	`submitFundingProof`	Anyone	Loan `Committed` or `Active`; proof verifies the tranche's exact amount and script wit least `requiredConfirmati` same Bitcoin transacti as any earlier proof fo this loan
`Committed`	`Active` (loan)	`submitFundingProof`	Anyone	The first collateral tran proven
`Pending`	`Unfunded`	`finalizeFunding`	Anyone	After `verificationDeadli`
`Committed`	`Cancelled` (loan)	`finalizeFunding`	Anyone	After `verificationDeadli` no collateral tranche v ever proven
`Live`	`Escrowed`	`repayTranche`, `cureWithRepayment`	The borrower	Loan `Active`; before last safe repayment ti not the interest tranch

From	To	Call	Who may call	Conditions
`Live`	`Defaulted`	`markTrancheDefaulted`	Anyone	Loan `Active`; at or af the last safe repaymer time and after `termEr` not the interest tranch
`Live`	`Defaulted`	`claimAfterCureWindow`	The lender	Health flag uncured pa the cure window; appl to every `Live` collate tranche at once
`Escrowed`	`Settled`	`revealSecret`	The lender	Loan `Active`; within t reveal window; `SHA256(preimage)` equals the tranche's h
`Escrowed`	`Ghosted`	`triggerGhostLiquidation`	Anyone	Loan `Active`; reveal window closed; `timeoutBlocks` withi bounds
`Ghosted`	`Ghosted`	`reclaimMegaPot`, then `reopenMegaPot`	Anyone	The order expired, wa never taken, or the fill was slashed; the pot i recovered and then re posted
`Ghosted`	`LiquidationDelivered`	`settleMegaPot`	Anyone	The settler reports the order filled
`Pending`	`Live` (top-up)	`cureWithCollateral`	The borrower	Health flag open and within the cure windo the top-up's own proc verifies
`Active`	`Settled` (loan)	Any call that makes the last tranche final	As above	Every tranche of the lo is final

A top-up tranche created by `cureWithCollateral` enters at `Live` with no principal share and then follows the same transitions as any other collateral tranche.

8.4 The loan state is a summary

The contract keeps a count of tranches that have reached a final state and marks the loan `Settled` when the count equals the number of tranches. A proven interest tranche is `Live` and never final, since it is settled on Bitcoin rather than on the host chain. A loan whose interest tranche was proven therefore remains `Active` after every collateral tranche has resolved. A loan whose interest tranche was left unproven reaches `Settled` once `finalizeFunding` has marked it `Unfunded` and the collateral tranches are final.

Wallets and watchers should read the tranche states and not wait for `Settled`. A loan is finished, for every practical purpose, when every collateral tranche is final.

8.5 Time parameters

Parameter	Set by	Meaning	Rule
`fundingDeadline`	Terms	The date by which the borrower undertakes to fund the HTLCs. The start of the challenge window (Chapter 9).	Later than the moment of commit
`verificationDeadline`	Terms	End of the challenge window. After this moment funding may be finalized, and a third-party proof earns the bounty. Proofs remain acceptable while a tranche is `Pending`.	Later than `fundingDeadline` by at least `requiredConfirmations × 20 minutes + 1 hour`
`termEnd`	Terms	End of the interest period, against which the rebate and Module A measure. The rebate is zero from this moment. Every tranche's last safe repayment time is later than it.	Later than `verificationDeadline`
`revealWindow`	Terms	Time the lender has to reveal after each repayment	At least 1 day (`MIN_REVEAL_WINDOW`)
`REVEAL_SAFETY_MARGIN`	Contract constant	Buffer between the end of a reveal window and a tranche's time lock	2 days

Parameter	Set by	Meaning	Rule
`btcTimelock` of a collateral tranche	Terms, matching the script	Time after which the lender may sweep the tranche	Later than `termEnd + revealWindow + REVEAL_SAFETY_MARGIN`
`btcTimelock` of the interest tranche	Terms, matching the script	Time after which the lender may sweep the interest	Exactly `verificationDeadline + 10 minutes` (`INTEREST_SWEEP_DELAY`)
Last safe repayment time	Derived	`btcTimelock − revealWindow − REVEAL_SAFETY_MARGIN` for the tranche; the latest moment it may be repaid	Readable with `lastSafeRepayTime(loanId, idx)`
`requiredConfirmations`	Terms	Confirmations a funding proof must show	From the verifier's floor (at least 1) to 8 (`MAX_CONFIRMATIONS`)
`cureWindow`	Terms	Time the borrower has to cure a health flag	Greater than zero if the health module is enabled
`oracleStaleAfter`	Terms	Maximum age of a price the oracle adapter may use	Greater than zero if the rebate or the health module is enabled

The contract does not stop a proof from being submitted after `fundingDeadline`. Proofs are accepted for any tranche still `Pending` until `finalizeFunding` marks it `Unfunded`, which cannot happen before `verificationDeadline`. The interval between the two deadlines is the challenge window, described in Chapter 9.

8.6 The timing rule

Every collateral tranche obeys one inequality, checked when the loan is committed and again at every repayment. The contract's comments call it the timing law.

```
repayment time + revealWindow + REVEAL_SAFETY_MARGIN < btcTimelock
```

A repayment is accepted only while `now < lastSafeRepayTime`. From that moment the tranche may be defaulted instead. The two windows share a boundary and never overlap. The contract's test suite checks at random timestamps around the boundary that exactly one of the two calls is possible.

The reason is the lender's time-lock path. A borrower who repays just before the last safe time leaves the lender a full reveal window, and then two further days in which the borrower can trigger a

liquidation, before the lender's path on Bitcoin opens. A borrower who has paid can therefore always either obtain the secret or obtain the liquidation before the lender could sweep the Bitcoin.

Bitcoin compares a time lock with the median timestamp of the preceding eleven blocks rather than with the current clock. That median trails real time by about an hour, so the lender's sweep becomes possible roughly an hour after `btcTimelock` and never before it. The two-day margin is for the borrower: it is the time after the reveal window closes in which the liquidation can be triggered before the lender's path opens, and it absorbs the hour by which the median trails.

9. Funding and proof

This chapter describes how the borrower's Bitcoin is proven to the loan contract, what happens when it is, and what happens when it is not.

9.1 One transaction for all tranches

The borrower funds every original tranche of a loan, collateral and interest, with a single Bitcoin transaction. Each output pays one tranche's exact amount to that tranche's P2WSH address. The first accepted proof fixes the loan's funding transaction identifier, and a proof for any other transaction is rejected with `FundingTxMismatch`. A loan can never be backed by outputs from two different transactions.

Tranches may be proven in several calls, as long as every call refers to the same transaction. A call may prove any subset of the still-`Pending` tranches, listed in ascending order with one output index per tranche. The verifier accepts up to eight outputs per call, which is enough for every original tranche at once.

Top-up tranches added by the health module are the exception. Each is proven on its own, and the contract does not require its transaction to be the loan's funding transaction.

9.2 The proof

The proof is the `BtcProof` structure.

Field	Type	Source
`blockHeight`	`uint256`	Height of the block containing the funding transaction
`rawTx`	`bytes`	The raw funding transaction; witness serialization is accepted
`txIndex`	`uint256`	Position of the transaction within the block, starting at 0
`merkleProof`	`bytes32[]`	Sibling hashes from the transaction to the Merkle root, in order
`header`	`bytes`	The 80-byte header of the containing block
`prevHeader`	`bytes`	The 80-byte header of the preceding block

Alongside the proof the submitter passes the list of tranche indices and, for each, the index of the output that pays it. The contract builds, for each tranche, a check consisting of the output index, the tranche's satoshis and the output script `00 20 <scriptHash>`, and passes the checks to the verifier.

With Bitcoin Core, `getrawtransaction <txid> true` returns the block hash and the raw transaction; `getblock <hash> 1` returns the height and the ordered transaction identifiers, from which `txIndex` and the Merkle branch are computed with double SHA-256; `getblockheader <hash> false` returns the header, and the same call on `previousblockhash` returns the preceding header.

9.3 What is checked

The verifier confirms that the header is part of its canonical chain, that the transaction is in the block, and that each named output pays exactly the stated satoshis to the stated script. It applies its own global confirmation floor. The loan contract then applies the loan's own requirement: it reads the verifier's latest height and requires `latestHeight − blockHeight + 1 ≥ requiredConfirmations`. A proof for a block the verifier has not yet received reverts; a proof with too few confirmations reverts with `InsufficientConfirmations` and may be retried later.

The proof establishes inclusion in a block, not the current state of the output. It remains valid after the output has been spent.

9.4 Activation and disbursement

For each collateral tranche proven, the contract marks the tranche `Live`, records the output index, credits the tranche's principal share to the borrower's payout ledger in the loan asset, and emits `TrancheVerified`. On the first collateral tranche proven it marks the loan `Active`, records `termStart`, and emits `LoanActivated`. The borrower withdraws the principal with `claimPayout`.

The interest tranche may be included in the same proof. It is marked `Live` and nothing is credited.

9.5 The verifier fee

The verifier may charge a fee per proof. The submitter passes `payAsset` and `maxPayment`. If `payAsset` is the zero address, the fee is paid in the host chain's native currency and the submitter attaches it as the call's value. The whole value is forwarded to the verifier, so it should be exactly the quoted fee. Otherwise the contract pulls `maxPayment` of the token from the submitter, allows the verifier to take what it charges, clears the allowance, and credits any remainder to the submitter's ledger. The verifier exposes `quoteVerificationFee` so that the submitter can size `maxPayment`. A deployment may register the loan contract as fee-exempt.

9.6 The challenge window and the bounty

The interval between `fundingDeadline` and `verificationDeadline` is the challenge window. During it the borrower is expected to have funded, and the lender is expected to prove. If the lender does not, the borrower or anyone else may.

A lender who lets a funded loan go unproven would, without a further rule, be able to reclaim its deposits by finalizing funding and then sweep the borrower's Bitcoin when the time locks open. The bounty closes this. If a proof for one or more collateral tranches is submitted after `verificationDeadline` by an address that is neither the borrower nor the lender, the submitter earns a reward from the lender's collateral:

```
bounty = collateralHeld × bountyBps × provenCollateralSats / (10,000 × totalCollateralSats)
```

The bounty is credited to the submitter's ledger in the collateral asset and deducted from `collateralHeld`. `bountyBps` is a signed term of at most 1,000, that is, 10%. In the worked example, with `bountyBps = 100` and tranche 0 proven by a third party after the deadline, the bounty is `30,000 × 100 × 25,000,000 / (10,000 × 50,000,000) = 150 USDC`. The bounty ensures that a valid funding is recognized even if both parties' systems fail to submit the proof.

The window must be long enough for the required confirmations to arrive. The contract requires `verificationDeadline − fundingDeadline` to be at least `requiredConfirmations × 20 minutes + 1 hour`. The block time assumed is deliberately slow, so that a transaction broadcast by the funding deadline can reach the required confirmations and be proven before the bounty applies, and an honest and prompt proof cannot land after the deadline through ordinary variance in block times alone.

9.7 Finalizing funding

After `verificationDeadline`, anyone may call `finalizeFunding`. Every tranche still `Pending` becomes `Unfunded`. For each unfunded collateral tranche, its never-disbursed principal share and its collateral slice are credited to the lender's ledger. An unfunded interest tranche changes nothing but its state.

If no collateral tranche was ever proven, the loan becomes `Cancelled`. Otherwise it stays `Active`, or becomes `Settled` if every tranche is now final. A partly funded loan continues for the tranches that were funded, with the principal reduced accordingly. `finalizeFunding` is the only way a never-activated loan ends.

After `verificationDeadline` a late proof and `finalizeFunding` are both possible, and whichever confirms first decides the tranche. A borrower who has funded should therefore never rely on the lender to prove. The borrower, or a watcher acting for the borrower, should submit the proof as soon as the confirmations are in.

9.8 Bitcoin sent that is not proven

Bitcoin sent to a tranche's address whose tranche is finalized `Unfunded` is still subject to the script. The borrower can spend it only with the secret, which the lender holds, and the lender can spend it

after the time lock. An output that pays the wrong amount is in the same position: it is never recognized, and its Bitcoin stays in the script. The borrower's protection against both outcomes is to check the amounts before broadcasting and to prove the funding before the deadline. The bounty gives third parties a reason to prove if the borrower does not.

Maxy's disclosure policy. In the loans Maxy offers, Maxy, as lender, discloses the secret of every collateral tranche that was finalized `Unfunded` while Bitcoin sits unspent at its address, whatever the amount. Maxy's keeper watches each such address until 30 days after the tranche's time lock, publishes the secret through the Maxy API, and, if the borrower gave a pre-signed return (13.10), broadcasts the return itself. Maxy never sweeps an unfunded tranche through the time-lock path. This is Maxy's policy as lender. The contract does not require it, and a lender under another policy may keep the secret and sweep the output after the time lock.

10. Repayment and the reveal

10.1 Escrow for secret

A repayment is not a transfer to the lender. It is a deposit into the loan contract that the lender can collect only by publishing the secret. The borrower calls `repayTranche(loanId, idx)` and deposits the tranche's principal share less the rebate. The tranche becomes `Escrowed`, the time of repayment is recorded, and `TrancheRepaid` reports the amount escrowed, the rebate and the deadline by which the lender must reveal.

The lender then calls `revealSecret(loanId, idx, preimage)`. The contract checks that `SHA256(preimage)` equals the tranche's hash lock, credits the escrowed amount and the tranche's collateral slice to the lender's ledger, marks the tranche `Settled`, and emits `SecretRevealed` with the preimage. From that transaction on, the secret is public and the borrower can spend the tranche's Bitcoin.

There is no state in which the lender has been paid and the secret is unpublished. There is no call that pays the lender without a correct preimage.

10.2 The reveal window

The lender must reveal no later than `repaidAt + revealWindow`. A reveal after that moment reverts with `RevealWindowClosed`, and the tranche may be liquidated instead, as Chapter 11 describes. The reveal window is a signed term of at least one day. Seven days is used in the examples.

A lender's service should treat `TrancheRepaid` as a call to action and reveal promptly. This is the lender's one time-critical obligation. The secret has no value to the lender after the window closes, and the delay costs the lender the collateral slice and the repayment.

10.3 The early-repayment rebate

The interest tranche prepays interest for the full term. If the rebate is enabled in the terms, a borrower who repays a tranche before `termEnd` is credited with the unused portion of that tranche's prepaid interest. The credit is valued in the loan asset through the oracle adapter and taken as a reduction of the amount the borrower must escrow:

```
trancheInterestSats = interestSats × trancheSats / totalCollateralSats
unusedSats          = trancheInterestSats × (termEnd − now) / (termEnd − termStart)
rebate              = value of unusedSats in the loan asset, at the oracle adapter's rate
escrowAmount        = principalShare − rebate
```

The rebate is zero if it is not enabled, if `now` is at or past `termEnd`, or for a top-up tranche. It is realized only as the reduction of the escrow; nothing is paid back separately. The lender receives the reduced escrow at the reveal and keeps the full interest tranche on Bitcoin, which is the equivalent of the lender returning the unused interest, and the contract's balances stay exact. `quoteRebate(loanId, idx)` returns the figure `repayTranche` would apply at that moment.

In the worked example of Chapter 5, suppose the rebate is enabled and Alice repays tranche 0 halfway through the term. Tranche 0's interest is `150,000 × 25,000,000 / 50,000,000 = 75,000 sats`, of which half, 37,500 sats, is unused. At a rate of 100,000 USDC per BTC the rebate is 37.50 USDC, and Alice escrows 14,962.50 USDC.

A second example runs on Robinhood Chain. A borrower holds 1 BTC and wants a loan of 60,000 USDG for 90 days. A lender agrees at 5% interest for the term, with the rebate enabled, and asks for the collateral to be split into four tranches of 25,000,000 sats each. The lender posts 60,000 USDG of its own collateral. The interest, 3,000 USDG, is converted at the agreed price of 120,000 USDG per bitcoin to 2,500,000 sats and becomes the interest tranche. The lender generates four secrets, the term sheet lists five script hashes, the borrower signs, and the lender commits 120,000 USDG: 60,000 of principal and 60,000 of collateral. Each collateral tranche carries a principal share of 15,000 USDG. The borrower funds all five scripts in one Bitcoin transaction; after three confirmations the lender submits the proof, the four collateral tranches become `Live`, and 60,000 USDG is credited to the borrower.

On day 60 the borrower repays tranche 2. That tranche's interest is `2,500,000 × 25,000,000 / 100,000,000 = 625,000 sats`, of which a third of the term, 208,333 sats, is unused. If the oracle adapter reports 120,000 USDG per bitcoin at that moment, the rebate is 250 USDG and the borrower escrows 14,750 USDG. The lender reveals the same day and is credited 14,750 USDG from escrow plus 15,000 USDG of its collateral, and the borrower reclaims 25,000,000 sats. The remaining three tranches are repaid on day 89 and settle the same way, each with a rebate of about 8.33 USDG for the one unused day. The lender swept the interest tranche shortly after the verification deadline, so by the end it has received 60,000 USDG of repayments less 275 USDG of rebates, its 60,000 USDG of collateral, and 2,500,000 sats of interest.

The terms may request delivery of the rebate in Bitcoin rather than in the loan asset. That mode is recorded but not available. On a loan with the rebate enabled and Bitcoin delivery requested, every repayment before `termEnd`, including one for a top-up that carries no rebate, reverts with `RebateBitcoinDeliveryNotSupported`. The contract does not pay in the loan asset against a term that promised Bitcoin. A repayment at or after `termEnd` carries no rebate and goes through.

10.4 Default

If the borrower has not repaid a `Live` collateral tranche by its last safe repayment time, anyone may call `markTrancheDefaulted(loanId, idx)` from that time onward. The contract credits the

tranche's collateral slice to the lender's ledger, marks the tranche `Defaulted`, and emits `TrancheDefaulted`. The lender spends the tranche's HTLC through the time-lock path once `btcTimelock` has passed. No action on the host chain is needed for the sweep; the script permits it on its own. The borrower keeps the principal share for that tranche. Nothing else changes.

A defaulted tranche's secret is never published. If the lender were to publish it by mistake, the borrower could spend the Bitcoin before the lender; the lender's secrets should be handled accordingly.

10.5 Reclaiming the Bitcoin

After `SecretRevealed`, the borrower builds a Bitcoin transaction spending the tranche's output through the hash-lock path, with the preimage and the borrower's signature in the witness. Chapter 18 gives the witness layout. Nothing is required on the host chain. The borrower should do this promptly: the lender's time-lock path opens at `btcTimelock` regardless of the reveal, and from that moment both parties can spend the output.

11. Ghost liquidation

A lender that has been repaid and does not reveal is said to have ghosted. The dual-collateral design exists for this case. The borrower's Bitcoin would otherwise remain locked until the lender's time lock opened, and then be lost. Instead, anyone may trigger a liquidation of the lender's collateral in favor of the borrower. This chapter describes how the borrower is compensated.

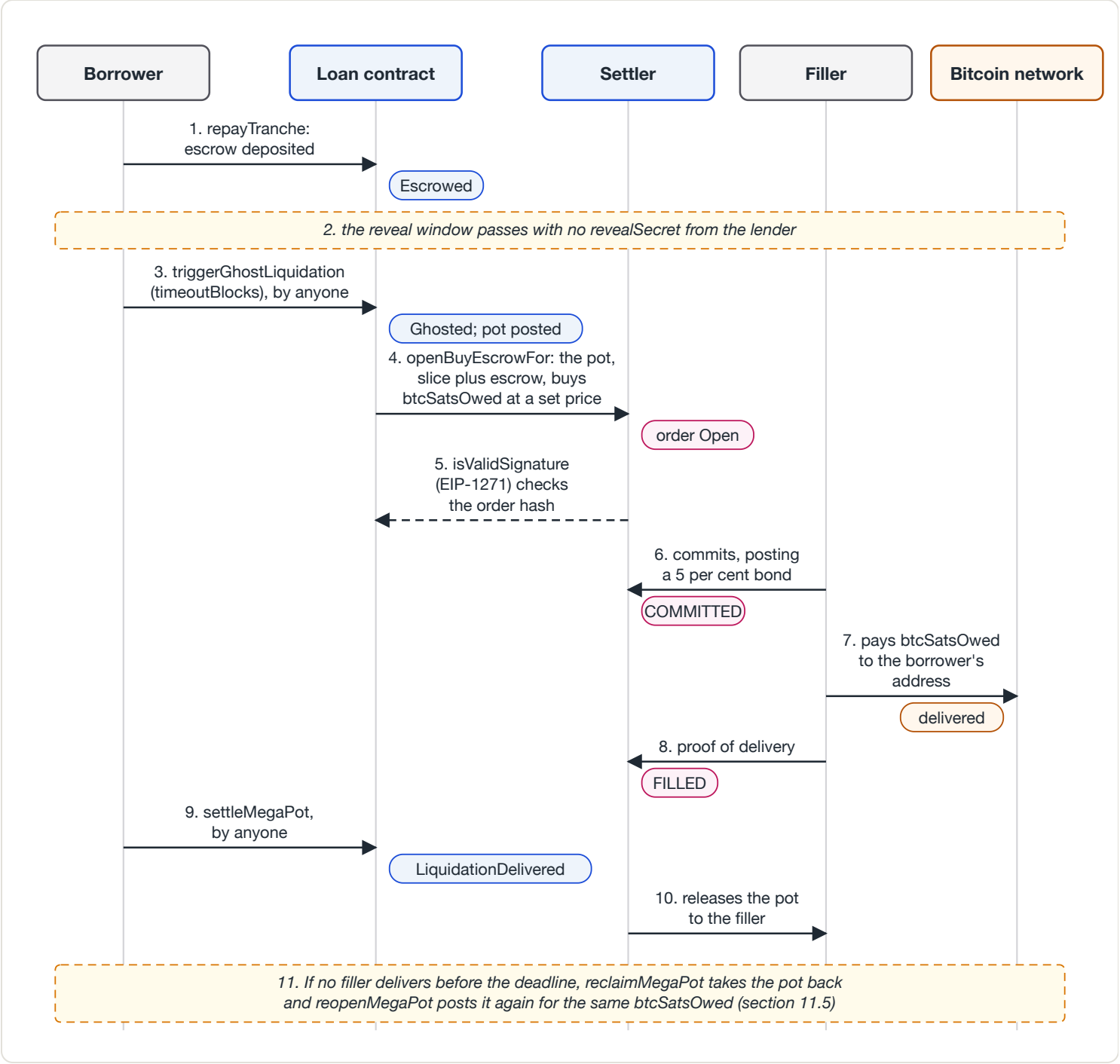


Figure: Ghost liquidation sequence: the pot is posted to the settler and a filler delivers Bitcoin.

11.1 Trigger

Once `now > repaidAt + revealWindow` for an `Escrowed` tranche, anyone may call `triggerGhostLiquidation(loanId, idx, timeoutBlocks)`. The lender's reveal is no longer accepted. The tranche becomes `Ghosted`.

`timeoutBlocks` is the number of Bitcoin blocks a filler has to deliver after committing to the order. It must lie between `requiredConfirmations + 10` and 4,320, which is about thirty days of blocks.

11.2 The Mega-Pot

The contract takes the tranche's collateral slice from `collateralHeld` and the escrowed repayment from the tranche, and combines them into a pot. If the collateral is a yield-bearing token, the slice is first redeemed to the loan asset through the route fixed at deployment (11.7). The pot owes the borrower a fixed quantity of Bitcoin:

```
btcSatsOwed = trancheSats × ghostPayoutBps / 10,000
```

`ghostPayoutBps` is a signed term between 10,000 and 15,000, so the borrower receives between 100% and 150% of the tranche's Bitcoin. The quantity is fixed when the pot opens and never changes. No price is read anywhere in this path. The contract names a quantity of Bitcoin and offers the whole pot for it, so a filler takes the order when the pot is worth more than the Bitcoin it must deliver. The lender's collateral is sized at signing to keep the order attractive across the price range the parties anticipate.

The pot is identified by `potKey = (loanId << 8) | idx` and can be read with `getMegaPot(potKey)`.

11.3 The order on the settler

The contract posts the pot to the settler as a buy order: it offers the whole pot in exchange for `btcSatsOwed` delivered to the borrower's Bitcoin address, which is the `borrowerBtcScript` fixed in the terms. The order uses the settler's fixed-price mode, with the price derived from the pot and the quantity. Its other terms are the following.

- Fillers must post a bond of 5% of the order.
- The delivery proof must show the loan's `requiredConfirmations`.
- A filler that commits must deliver within `timeoutBlocks`.
- The order expires thirty days after posting if no filler commits.
- The loan contract is the recipient of any refund.

The settler holds the pot from this point.

The loan contract is a contract, not a key, so it cannot sign the order. It instead pre-authorizes the order's hash and answers the settler's EIP-1271 signature check for that hash alone.

The amount the settler takes is computed with the settler's own price formula, rounding down. Any residue of a few base units is credited to the borrower's ledger before the order is posted, so that the contract's balances remain exact.

11.4 Delivery and settlement

A filler commits to the order on the settler, pays `btcSatsOwed` to the borrower's Bitcoin address, and submits a proof of that payment to the settler. The settler verifies the delivery against the Bitcoin headers and marks the order filled. Anyone then calls `settleMegaPot(potKey)`. The loan contract reads the settler's state, marks the pot `Delivered` and the tranche `LiquidationDelivered`, and closes the loan if every tranche is now final. The settler releases the pot to the filler.

The borrower ends with more Bitcoin than was locked, bought with the lender's own collateral and the repayment the lender forfeited. The original tranche remains in its HTLC. The borrower cannot spend it without the secret, and the lender may sweep it once its time lock opens. The lender has paid for it, at a premium, with the collateral slice and the repayment.

11.5 Expiry and re-posting

If no filler fills the order, the pot is recovered and posted again. Anyone calls `reclaimMegaPot(potKey)`. If a filler had committed and the delivery deadline passed, the contract calls the settler's refund, which returns the pot together with whatever share of the filler's forfeited bond the settler pays out. If no filler ever committed and the order expired, which happens thirty days after posting, the contract cancels the open order, which returns the pot. In either case the contract measures what came back in the pot's asset and records it as the pot's new amount. A bond share paid in the pot's asset therefore grows the pot; one paid in another token is not captured. If the order had already been refunded or slashed by a call made directly on the settler, the contract keeps the pot's last recorded amount, and any bond that came back that way is not attributed to the pot. The pot becomes `Reclaimed` and the tranche stays `Ghosted`.

Anyone then calls `reopenMegaPot(potKey, timeoutBlocks)`, which posts the same `btcSatsOwed` with the current pot and a fresh deadline. This may be repeated without limit. The pot never shrinks except by rounding dust, which is credited to the borrower, so the borrower's claim does not expire.

If the pot was already filled, `reclaimMegaPot` reverts and `settleMegaPot` is the correct call.

11.6 Pot states

State	Code	Meaning
`None`	0	No pot exists for this key.
`Open`	1	An order is posted on the settler.
`Delivered`	2	The order was filled and the tranche is `LiquidationDelivered`. Final.
`Reclaimed`	3	The order expired or its filler was slashed, and the pot has been recovered. May be reopened.

11.7 Two-asset loans and redemption routes

The settler accepts one asset per order. If the collateral asset and the loan asset are the same, the pot is their sum. If they differ, the collateral slice must first be converted into the loan asset.

MaxyLock allows the two to differ only when a redemption route for the collateral asset exists. A route pairs a yield-bearing asset with its underlying asset and an adapter that redeems one into the other through the issuing protocol. An Aave deposit receipt redeemed through the Aave pool is one example; an ERC-4626 vault share redeemed through its vault is another. Routes are written once, in the loan contract's constructor, and can never be added or changed afterwards. There is no governance surface for them.

At commit, a term sheet whose collateral asset differs from the loan asset is accepted only if the collateral asset's route leads to the loan asset. Otherwise it is rejected with `IncoherentAssetPair`. At liquidation, the contract redeems the collateral slice through the route's adapter, measures the loan asset received, and forms the pot from that amount plus the escrow. A redemption that returns less than expected, because of rounding or a stressed venue, shrinks the pot rather than blocking the liquidation.

The settler itself must allow the pot's asset. Allowing an asset is a one-time step by the settler's governance, described in Chapter 19.

A pot in the host chain's native currency is posted through the same settler call as a token pot, with the amount attached as value. The contract's own tests exercise token pots only; a native-currency pot has not been exercised. A lender choosing native collateral or a native loan asset should confirm that path against the settler before relying on liquidation.

11.8 Worked example

Alice repaid tranche 0 with 15,000 USDC in escrow. Bob does not reveal within seven days. A watcher calls `triggerGhostLiquidation` with `timeoutBlocks = 144`. The pot is tranche 0's collateral slice, 15,000 USDC, plus the escrow, 15,000 USDC: 30,000 USDC in total. With `ghostPayoutBps = 10500`,

the pot owes `25,000,000 × 10,500 / 10,000 = 26,250,000 sats`, or 0.2625 BTC. The order offers 30,000 USDC for 0.2625 BTC, which is 114,285.71 USDC per BTC.

If the market price is 100,000 USDC per BTC, a filler can buy 0.2625 BTC for 26,250 USDC, deliver it to Alice's address, and take 30,000 USDC. Alice receives 0.2625 BTC in place of the 0.25 BTC still locked in the HTLC. Bob, once the time lock opens, sweeps that 0.25 BTC, worth 25,000 USDC. To get it, Bob has given up the 15,000 USDC he lent, which was never returned to him, and the 15,000 USDC of collateral that went into the pot. Bob's net position on the tranche is a loss of 5,000 USDC. Alice's is a gain of 0.0125 BTC. The filler's is 3,750 USDC. Had Bob revealed, he would have recovered 30,000 USDC and swept nothing.

If the price were above 114,285 USDC per BTC, no filler would take the order. The pot would be reclaimed and reopened until the price allowed a fill or a slashed bond had grown the pot enough. The borrower's claim waits; it does not lapse.

12. Protection modules

Two optional modules can be enabled in the terms. Both are chosen when the term sheet is signed and cannot be changed afterwards. Module A and the early-repayment rebate are the only places in the protocol where a price source influences an outcome, and each applies only to loans whose parties chose it.

12.1 Flags

The `flagsBits` term is a bit field.

Bit	Name	Meaning
0	Module A	Health flag with a cure window
1	Module B	Yield-funded hedge
2	Module C	Reserved; no behavior in this version
3	Rebate enabled	The early-repayment rebate of Chapter 10 applies
4	Rebate in Bitcoin	Delivery of the rebate in Bitcoin; recorded but not available, so with the rebate enabled every repayment before `termEnd` reverts

12.2 Module A: health flag and cure window

On the main path a fall in the Bitcoin price has no effect on a loan. A lender who wants protection against a fall can require Module A. It adds a health check that any party may invoke, a cure window during which the borrower can restore health, and an exit for the lender if the borrower does not.

The health check. A loan is unhealthy when the Bitcoin in its `Live` collateral tranches, valued in the loan asset through the oracle adapter, is worth less than `healthThresholdBps` of the principal still outstanding:

```
value(liveSats) × 10,000 < healthThresholdBps × outstandingPrincipal
```

`liveSats` is the sum of satoshis over `Live` collateral tranches, including top-ups, and `outstandingPrincipal` is the sum of their principal shares. Repaid, defaulted and liquidated tranches count on neither side. `healthThresholdBps` is a signed term of at most 10,000. `healthStatus(loanId)` returns the current answer.

In the worked example, with both tranches live and `healthThresholdBps = 9000`, the loan is unhealthy when 0.5 BTC is worth less than `0.9 × 30,000 = 27,000 USDC`, that is, when the price falls below 54,000 USDC per BTC.

Flagging. Anyone may call `flagUnhealthy(loanId)` on an `Active` loan that reads unhealthy. The contract records the time and emits `HealthFlagged` with the cure deadline, `now + cureWindow`, and the collateral value it read. A loan can carry one flag at a time.

Clearing. If a later reading shows the loan healthy again, anyone may call `clearFlag(loanId)`. A recovered market cures the loan by itself.

Curing by repayment. Within the cure window the borrower may call `cureWithRepayment(loanId, idx)`, which repays a tranche exactly as `repayTranche` does and then re-reads health. Because every original tranche carries the same ratio of principal to Bitcoin, repaying one of them reduces both sides of the inequality in proportion and leaves the ratio unchanged. On a loan with no top-up, repayment alone clears the flag only when the price has recovered or when no collateral tranche remains `Live`. Once a top-up is live, each original tranche repaid improves the ratio, because the top-up adds Bitcoin and no principal. A partial cure is not an error; the borrower may continue.

Curing with collateral. Within the cure window the borrower may call `cureWithCollateral` with the terms of a new tranche, a proof that it is funded, and the output index. The new tranche is appended with a principal share of zero, verified through the same path as an original tranche, and marked `Live`. Its time lock must obey the same rule as an original collateral tranche. It adds Bitcoin to the numerator and nothing to the denominator, so it moves the ratio. Health is re-read and the flag cleared if the loan is now healthy. In the example above, a top-up of 0.1 BTC at a price of 50,000 USDC per BTC brings the live collateral to 0.6 BTC, worth 30,000 USDC, which clears the flag. The lender does not sign the top-up's terms, and 7.5 describes what that means for the lender.

The lender's exit. If the cure window ends with the flag still set, the lender may call `claimAfterCureWindow(loanId)`. The contract re-reads health first; a flag that the market has since cured pays nothing, and the call reverts. If the loan is still unhealthy, all remaining `collateralHeld` is credited to the lender in one sum, every `Live` collateral tranche is marked `Defaulted`, and the lender sweeps each one's Bitcoin as its time lock opens.

Repayments in flight. A health event never touches a repayment in flight. An `Escrowed` tranche keeps its state and its escrow. The lender can still reveal and collect the repayment, and the borrower can still trigger a liquidation if the lender does not. But the exit takes the whole of `collateralHeld`, including the slice that would have backed that tranche's liquidation, so a pot opened afterwards holds the escrow alone against an unchanged `btcSatsOwed` and is unlikely to fill. A borrower with a repayment in escrow at the moment of the lender's exit therefore depends on the lender revealing, and a lender in that position has no collateral at stake in the choice. A borrower on a Module A loan should be cautious about repaying while a flag is open and the cure deadline is near or past, because the backstop for that repayment can leave with the lender's exit.

Oracle availability. Every Module A call reads the oracle adapter, and reverts if the adapter cannot produce a trustworthy price. So does a rebated repayment before `termEnd`. The cure window does not pause while the adapter is unavailable. During an outage, both cures revert. The one action a borrower can still take is a plain `repayTranche` on a loan without the rebate, which needs no price. A loan whose every collateral tranche has been repaid reads healthy once the adapter returns. The lender's exit also waits until the adapter recovers, and the fresh reading it then takes honors anything the borrower did in the meantime. The contract prefers a delayed exit to one taken on a price it cannot trust.

12.3 Module B: yield-funded hedge

If the collateral asset earns yield, for example a lending-pool deposit receipt whose balance grows, Module B routes a share of that yield to a hedging contract chosen by the parties. The intent is to buy price protection from the collateral's own earnings without touching the collateral.

Anyone may call `skimYieldToHedge(loanId)` at most once per day per loan. The contract measures the yield as the contract's balance of the collateral asset, less everything it owes in that asset through the payout ledger, less `collateralHeld`. It forwards `moduleBYieldBps` of that figure to `moduleBAdapter`, a contract named in the terms, by approving the amount and calling `buyProtection(amount)`. The adapter routes the funds to an account that buys price protection at a venue outside the protocol. The approval is cleared afterwards. If there is no yield the call forwards nothing, but it still counts as that day's skim. `YieldSkimmed` reports the amount.

Module B requires the collateral asset to be a token with a growing balance. It cannot be enabled on a loan with native collateral, and it requires `moduleBAdapter` and `moduleBYieldBps` to be set.

The measurement is exact when the collateral asset differs from the loan asset and backs one active loan at a time. It has two limits that the contract does not enforce. First, it assumes that the collateral asset backs one active loan at a time in this contract. If two active loans share the same collateral asset, the yield of each cannot be separated from the balance of the other, and a skim on one could draw on the other's collateral. Second, it assumes that the collateral asset is not also the loan asset. If it is, escrowed repayments and undisbursed principal sit in the same balance and are measured as yield, and a skim could forward part of them to the hedge adapter. A lender enabling Module B should use a yield-bearing collateral asset that differs from the loan asset and that no other active loan on the same contract uses. Chapter 16 lists both limits among the assumptions.

13. Integration guide for wallets and front ends

This chapter describes what a wallet, a lender's service or a front end must do to take a borrower and a lender through a loan. It assumes access to a host-chain RPC endpoint and to a Bitcoin node or indexer that can return raw transactions, block headers and Merkle branches. Chapter 17 gives every signature in full.

13.1 Negotiating the terms

The two parties exchange the following before anything is signed. How they exchange it (a lender's order service, a marketplace, or direct negotiation) is outside the protocol.

1. **Keys.** Each party supplies a Bitcoin public key for the tranche scripts and the host-chain address it will act from. The borrower also supplies the Bitcoin address, as an output script, to which a liquidation should deliver Bitcoin. This becomes `borrowerBtcScript`.
2. **Secrets.** The lender generates one 32-byte secret per collateral tranche from a cryptographically secure source, computes each SHA-256 hash, and sends the borrower the hashes. The secrets themselves never leave the lender. The interest tranche has a zero hashlock.
3. **Amounts and settings.** The parties agree the assets and amounts: the loan asset, the collateral asset, the principal and the lender's collateral. They agree the Bitcoin side: the number of collateral tranches, each tranche's satoshis and the interest tranche's satoshis. They agree the timing and the risk terms: the deadlines, the reveal window, the confirmation depth, the liquidation premium, the bounty and the module settings. The contract allocates the principal across the collateral tranches in proportion to their sats.
4. **Scripts.** Both parties construct the witness script for each tranche from the tranche's hash, the tranche's time lock, the borrower's Bitcoin public key and the lender's Bitcoin public key, following Chapter 18. Both parties check every script: the hash, the time lock and the two keys. The SHA-256 of each script becomes the tranche's `scriptHash`, and each party verifies that the hashes it computed match the ones in the term sheet before proceeding.

The contract will later check every amount, deadline and hash against the rules below, but it cannot check that a script hash corresponds to a script with the agreed branches. That check belongs to the wallets, before signing and before funding.

13.2 The term sheet

The `LoanTerms` structure carries the following fields.

Field	Meaning	Rule enforced at commit
`borrower`	Host-chain address that signs the terms, repays and receives the principal	Must be the signer
`lender`	Host-chain address that commits the loan and reveals secrets	Must be the caller of `commitLoan`
`collateralAsset`	The lender's collateral token; zero address for native	Equal to `loanAsset`, or with a redemption route to it
`collateralAmount`	The lender's collateral	Greater than zero
`loanAsset`	The token lent; zero address for native	
`principal`	The amount lent	Greater than zero
`fundingDeadline`	See 8.5	Later than now
`verificationDeadline`	See 8.5	Later than `fundingDeadline` by at least `requiredConfirmations × 20 min + 1 h`
`termEnd`	See 8.5	Later than `verificationDeadline`
`revealWindow`	Seconds the lender has to reveal after a repayment	At least 86,400
`requiredConfirmations`	Confirmations a proof must show	From the verifier's `minConfirmations()` to 8
`ghostPayoutBps`	Bitcoin owed by a Mega-Pot, in basis points of the tranche	10,000 to 15,000
`bountyBps`	Third-party proof bounty, in basis points of the collateral	At most 1,000
`rebateSlippageBps`	Recorded with the loan; not used by the token-delivered rebate	At most 10,000
`flagsBits`	Module and rebate toggles; see 12.1	Bit 4 is accepted at commit but makes every rebated repayment revert, so it is left clear
`loanAssetFeedId`	Pyth feed identifier for the loan asset against USD	Non-zero if Module A or the rebate is enabled

Field	Meaning	Rule enforced at commit
`oracleStaleAfter`	Maximum price age in seconds	Non-zero if Module A or the rebate is enabled
`healthThresholdBps`	Module A threshold; see 12.2	At most 10,000; non-zero if Module A is enabled
`cureWindow`	Module A cure window in seconds	Non-zero if Module A is enabled
`moduleBAdapter`	Module B hedge contract	Non-zero if Module B is enabled
`moduleBYieldBps`	Share of yield skimmed, in basis points	Non-zero if Module B is enabled
`borrowerBtcScript`	Bitcoin output script paid by a liquidation	Non-empty
`nonce`	The borrower's next nonce, read from `borrowerNonce(borrower)`	Equal to `borrowerNonce(borrower)`
`tranches`	`N + 1` tranche terms; index `N` is the interest tranche	2 to 8 entries

Each `TrancheTerms` entry carries `scriptHash`, `hashlock`, `satsValue` and `btcTimelock`. Every `satsValue` must be greater than zero. The interest tranche's `hashlock` is zero by convention and its `btcTimelock` must equal `verificationDeadline + 600`. Every collateral tranche's `btcTimelock` must exceed `termEnd + revealWindow + 172,800`.

If Module B is enabled, `collateralAsset` must not be the zero address.

13.3 Signing

The borrower signs the EIP-712 digest of the term sheet. The domain is name `MaxyLock`, version `1`, the host chain's ID, and the loan contract's address. The type string is given in Chapter 17. The `borrowerBtcScript` field enters the structure hash as its `keccak256`, and the tranche array enters as the `keccak256` of the concatenated tranche structure hashes.

Before signing, the wallet calls the view `hashTerms(terms)` on the loan contract and confirms that it equals the digest the wallet computed. A mismatch means the wallet's encoding is wrong, and a signature over it would be rejected. The wallet also checks that `terms.nonce` equals `borrowerNonce(borrower)`; nonces are sequential per borrower and each term sheet consumes one.

13.4 Committing

From the lender's account:

1. Approve the loan contract to spend `collateralAmount` of `collateralAsset` and `principal` of `loanAsset`, for whichever of the two are tokens.
2. Call `commitLoan(terms, signature)`, attaching as value the sum of whichever of the two amounts are native. Any other value reverts with `WrongNativeValue`.
3. Read `loanId` from the return value or from `LoanCommitted`. Every later call uses it.

The contract pulls both deposits and checks what arrived by balance difference. The principal must arrive exactly. The collateral may arrive up to ten base units short, to accommodate tokens whose transfers round down, and the amount actually received is what the contract records as `collateralHeld`. A token that takes a fee on transfer is rejected with `TransferAmountMismatch`. The call reverts if any term is out of bounds; 13.13 lists the reasons.

13.5 Funding the HTLCs

The borrower constructs one Bitcoin transaction with one output per tranche. The following rules apply.

- Every original tranche, collateral and interest, is paid in the same transaction. A second transaction cannot fund any of them.
- Each output pays the tranche's exact `satsValue`. The verifier requires exact equality; an output for any other amount cannot be proven, and its Bitcoin stays locked in the script (9.8). The amounts are checked before the transaction is broadcast.
- Each output pays the tranche's P2WSH address. The output script is `00 20 <scriptHash>`, which the view `trancheScriptPubKeys(loanId, idx)` returns byte for byte. Encode it as a bech32 address beginning `bc1q` for display, and compare it with the address derived from the script the wallet built.
- Commit before paying. Bitcoin sent before `LoanCommitted` has no record on the host chain.
- Pay before `fundingDeadline`, and allow for the confirmations and the proof to land before `verificationDeadline`.

Wait for `requiredConfirmations` before building the proof.

13.6 Submitting the funding proof

Build the `BtcProof` of Chapter 9 and call:

```
submitFundingProof(uint256 loanId, uint8[] trancheIdx, uint256[] outputIndexes, BtcProof proof,
address payAsset, uint256 maxPayment)
```

`trancheIdx` lists the tranches being proven in strictly ascending order, and `outputIndexes[k]` is the index of the output that pays `trancheIdx[k]`. Call `quoteVerificationFee` on the verifier to size the fee. For a native fee, pass the zero address as `payAsset` and attach exactly the quoted fee as value; the whole value is forwarded to the verifier and nothing comes back through the loan contract. For a token fee, approve the loan contract for `maxPayment` first; the unspent part comes back to the submitter's ledger.

The call may be made more than once for different tranches of the same transaction, and it is accepted at any time while a tranche is `Pending`. If the call reverts with `InsufficientConfirmations`, the verifier has not yet seen enough blocks. Retry later. The borrower's wallet should be prepared to submit the proof itself: the lender normally does, but nothing requires it to, and a tranche that is not proven by `verificationDeadline` can be finalized unfunded.

13.7 Claiming payouts

Every amount the contract owes an address is in `payoutLedger(address, asset)`. That address withdraws it with `claimPayout(asset)`, which transfers the whole balance for that asset. The borrower claims the principal this way after the proof. The lender claims repayments, collateral and returned deposits the same way. A bounty hunter claims its bounty the same way. A claim of a zero balance reverts with `NothingToClaim`.

13.8 Repaying a tranche

From the borrower's account, before `lastSafeRepayTime(loanId, idx)`:

1. Read `escrowAmount = getTranche(loanId, idx).principalShare − quoteRebate(loanId, idx)`.
2. For a token loan asset, approve the loan contract for `escrowAmount`. For a native loan asset, attach exactly `escrowAmount` as value.
3. Call `repayTranche(loanId, idx)`. The `TrancheRepaid` event gives the reveal deadline.

The rebate quote changes with time and, if enabled, with the oracle's price. The rebate shrinks as the term runs, so the amount pulled at the moment of the call can exceed a figure quoted earlier, and an approval for exactly that figure can fail. For a token loan asset, approving the full `principalShare` is the simplest course; the contract pulls only what it needs. For a native loan asset the attached value must match the escrow amount exactly, and a rebated amount changes from one second to the next, so a rebated repayment in native currency is impractical before `termEnd`. If the rebate is enabled and the oracle adapter cannot supply a price, the repayment reverts until it can, so a rebated repayment should not be left to the last moment. After `termEnd` the rebate is zero and no price is read.

13.9 Revealing

From the lender's account, no later than `repaidAt + revealWindow`, call `revealSecret(loanId, idx, preimage)`. The preimage is the 32-byte secret whose SHA-256 is the tranche's `hashlock`. The repayment and the collateral slice are credited to the lender's ledger in the same transaction, and the lender then claims them with `claimPayout` for the loan asset and for the collateral asset.

13.10 Spending the Bitcoin

After `SecretRevealed`, the borrower spends the tranche's output through the hash-lock path. The witness is `[signature, S, 0x01, witnessScript]` for the reference script of Chapter 18, signed with the borrower's key over the BIP-143 digest. Under Maxy's template (18.7) the witness also carries the borrower's public key: `[signature, pubkey, S, 0x01, witnessScript]`. The wallet should broadcast promptly and confirm before `btcTimelock`.

A lender spends a defaulted or interest tranche through the time-lock path, with `nLockTime` at least `btcTimelock` and the witness `[signature, <empty>, witnessScript]`. Maxy's template uses different witnesses for its sweeps, listed in 18.7.

If the reveal deadline passes with no `SecretRevealed`, the wallet calls `triggerGhostLiquidation(loanId, idx, timeoutBlocks)` with a timeout between `requiredConfirmations + 10` and 4,320 blocks, then monitors `MegaPotDelivered` and the Bitcoin delivery to `borrowerBtcScript`. `settleMegaPot(potKey)` closes the tranche once the settler reports the fill; `reclaimMegaPot` and `reopenMegaPot` re-post an unfilled order.

Pre-signed returns in loans Maxy offers. The protocol leaves the spend to the borrower. Maxy's service also accepts, optionally, a borrower's pre-signed return, so that the Bitcoin comes back even if the borrower's wallet is offline when the secret is published. The borrower signs it once the loan is committed and the funding transaction is known, and Maxy's service stores it only after rebuilding the transaction from its own records and the chain and checking that the signature commits to exactly that transaction.

	Collateral return	**Interest recovery**
What it spends	A collateral tranche, through the hash-lock branch	The interest tranche, through the borrower recovery branch of `maxylock-htlc-v1` (18.7)
Valid when	Only once the tranche's secret is public, from `SecretRevealed` or a disclosure (9.8). The signature alone cannot spend the output; the witness also needs the secret.	Only once the chain's median time past exceeds `verificationDeadline + 7 days 10 minutes`, the transaction's `nLockTime`
When Maxy broadcasts it	As soon as the secret is public and the output is unspent	Only if the loan never activated and the output is unspent

	Collateral return	Interest recovery
Accepted while the loan is	`Committed` or `Active`	`Committed`, `Active`, `Settled` or `Cancelled`

The return pays the borrower's own Bitcoin address, the one whose key is in the tranche scripts, and nothing else. It comes in two forms. Signed with `SIGHASH_ALL`, it pays the tranche less a network fee the borrower fixes when signing; that fee cannot be raised later. Signed with `SIGHASH_SINGLE|ANYONECANPAY`, it pays the borrower the tranche's full amount at the same output index, and Maxy's keeper adds its own input to pay the fee and can raise it. Changing the amount or the destination of the borrower's output invalidates the signature.

By signing a return the borrower gives up nothing it would otherwise keep: the return can only send the tranche's Bitcoin to the borrower, and the borrower can still spend the output itself at any time the script allows. Two practical points follow. A return is tied to the funding transaction's txid, so the borrower funds without replace-by-fee and raises a low funding fee only by child-pays-for-parent; a replaced funding transaction makes its returns useless. And a `SIGHASH_ALL` return fixes its fee rate at signing, so in a period of high fees the borrower may prefer to spend the output itself.

13.11 Permissionless calls

A wallet acting for either party may make the following calls on the party's behalf, and so may any watcher.

Situation	Call
A tranche is `Pending` after `verificationDeadline`	`finalizeFunding(loanId)`
A tranche is `Live` after its last safe repayment time	`markTrancheDefaulted(loanId, idx)`
A tranche is `Escrowed` after its reveal window	`triggerGhostLiquidation(loanId, idx, timeoutBlocks)`
A pot's order was filled	`settleMegaPot(potKey)`
A pot's order expired or its filler was slashed	`reclaimMegaPot(potKey)`, then `reopenMegaPot(potKey, timeoutBlocks)`
A Module A loan reads unhealthy	`flagUnhealthy(loanId)`
A flagged loan reads healthy	`clearFlag(loanId)`

Situation	Call
A Module B loan has earned yield	`skimYieldToHedge(loanId)`

13.12 Monitoring

Event	Who acts	Action
`LoanCommitted`	Borrower	Fund the HTLCs
`TrancheVerified`, `LoanActivated`	Borrower	Claim the principal
`TrancheRepaid`	Lender	Reveal before the deadline in the event
`SecretRevealed`	Borrower	Spend the Bitcoin with the preimage in the event
`TrancheDefaulted`	Lender	Sweep the Bitcoin at `btcTimelock`
`GhostLiquidationTriggered`, `MegaPotOpened`	Borrower, fillers	Watch the settler for a fill
`MegaPotDelivered`	Borrower	The replacement Bitcoin has arrived
`HealthFlagged`	Borrower	Cure before the deadline in the event
`CollateralClaimedAfterCure`	Lender	Sweep each defaulted tranche at its time lock
`PayoutQueued`, `ProofBountyPaid`	The named address	Claim when convenient; a bounty is announced by `ProofBountyPaid` alone

The borrower should also watch each tranche's HTLC output on Bitcoin, and the lender should watch each tranche's time lock.

13.13 Errors

Error	Meaning	Response
`CommitsArePaused`	New loans are paused	Wait
`NotLender`, `NotBorrower`	The caller is not the party required	Use the right account

Error	Meaning	Response
`NotBorrowerSignature`	The signature does not recover to `borrower`	Check the EIP-712 encoding against `hashTerms` and re-sign the exact term sheet
`NonceUsed`	`nonce` is not the borrower's next nonce	Read `borrowerNonce` and re-sign
`BadTrancheCount`	Fewer than 2 or more than 8 tranches	Fix the term sheet
`ConfirmationsOutOfRange`	Outside the verifier's floor (at least 1) to 8	Fix the term sheet
`DeadlinesUnordered`	The deadlines are not in order, or the reveal window is under one day	Fix the term sheet
`ChallengeWindowTooTight`	The verification deadline is too close to the funding deadline for the confirmations required	Move the verification deadline later
`InterestTimelockWrong`	The interest tranche's time lock is not `verificationDeadline + 600`	Fix the script and the term sheet
`TimelockTooTight`	A collateral tranche's time lock is not past `termEnd + revealWindow + 2 days`	Fix the script and the term sheet
`IncoherentAssetPair`	The collateral asset has no route to the loan asset	Use the same asset, or an asset with a route
`BpsOutOfRange`, `ZeroAmount`, `ZeroSats`, `ZeroAddress`, `BtcScriptEmpty`	A term is out of range or missing	Fix the term sheet
`WrongNativeValue`	The attached value is not what the call needs	Attach the exact amount
`TransferAmountMismatch`	A token delivered a different amount than requested	Check approvals and amounts; do not use fee-on-transfer tokens

Error	Meaning	Response
`WrongLoanState`, `WrongTrancheState`	The call is not valid in the current state	Read `getLoan` and `getTranche` and consult Chapter 8
`DuplicateTranche`, `ArrayLengthMismatch`, `NoTranchesListed`	The proof's tranche list is malformed	List tranches once, ascending, with one output index each
`TrancheIndexOutOfRange`	The index is out of range, or names the interest tranche where a collateral tranche is required	Check the index against `getLoan`
`VoutOutOfRange`	An output index exceeds 32 bits	Check the output index
`VerifierRejected`	The proof does not match the registered scripts and amounts	Check the output indices, the amounts, the scripts and the Merkle branch
`InsufficientConfirmations`	The verifier has not yet seen enough blocks	Retry later
`FundingTxMismatch`	A different transaction already funded this loan	Only the first transaction can be proven
`FundingNotOver`	`verificationDeadline` has not passed	Wait
`RepayTooLate`	The last safe repayment time has passed	The tranche will default
`RebateBitcoinDeliveryNotSupported`	The terms asked for the rebate in Bitcoin and the repayment is before `termEnd`	Not available; sign terms with token delivery, or repay after `termEnd`
`WrongPreimage`	The value does not hash to the tranche's hash lock	Check the secret for this tranche
`RevealWindowClosed`	The reveal window has passed	The tranche will be liquidated

Error	Meaning	Response
`DeadlineNotReached`, `RevealWindowStillOpen`, `CureWindowOpen`, `CureWindowOver`, `SkimTooSoon`	Too early or too late for the call	Wait, or the opportunity has passed
`TimeoutOutOfRange`	`timeoutBlocks` is outside `[requiredConfirmations + 10, 4320]`	Pass a value in range
`TwoAssetPotNotSupported`	No route normalizes this collateral into the loan asset	Should not occur for a committed loan
`PotNotOpen`, `PotNotExpired`, `PotNotDelivered`, `PotNotReclaimed`	The pot is not in the state the call requires	Read `getMegaPot` and the settler's state
`ModuleNotEnabled`	The terms did not enable the module the call belongs to	None
`NotFlagged`, `AlreadyFlagged`, `NotUnhealthy`, `StillUnhealthy`	A Module A precondition is not met	Read `healthStatus`
`TooManyTranches`	The loan already has 255 tranches	No further top-ups
`NothingToClaim`	The ledger balance is zero	None
`NativeSendFailed`	The claimant rejected the native transfer	Claim from an address that accepts value
`NotSettler`	Native value was sent to the contract outside a function	None; only the settler may do this
`NotPauser`	The caller is not the pauser	None

14. Operations for lenders and borrowers

This chapter is written for the teams that act as lender or borrower, in particular for a lender operating at scale, and for the wallets that act for borrowers. It covers the keys each side holds, the duties each must perform on time, and the conditions that should be monitored.

14.1 The lender's keys and secrets

The lender holds three distinct credentials. They should be held on separate systems with separate access controls.

Credential	Network	Purpose	Consequence of compromise
Host-chain key	Host chain	Commits loans, reveals secrets, exits after an uncured flag, claims payouts	An attacker can claim the lender's ledger balances and commit loans with any funds the key has approved. Revoke approvals and claim outstanding balances to a new key where possible.
Bitcoin sweep key	Bitcoin	Spends defaulted tranches and interest tranches through the time-lock path	An attacker can sweep every tranche whose time lock opens. It cannot take a tranche before its lock time, and nothing on the host chain is affected.
Secret store	Off chain	Holds `S` for each collateral tranche, revealed after repayment	A borrower who obtains a secret can spend the tranche's Bitcoin without repaying. The tranche then defaults with no Bitcoin left to sweep, and the lender loses that tranche's principal share.

Loss of a secret is as costly as theft. A lender that cannot reveal within the window loses the tranche's collateral slice and the repayment to a liquidation, so the secret store is treated with the same care as the Bitcoin key. Each secret should be 32 bytes from a cryptographically secure source, never reused, and recoverable. Deriving each secret from an offline master secret and the loan and tranche identifiers, for example with HMAC-SHA256, allows a secret to be recomputed after a database failure. Secrets are made available only to the reveal process, and never to the borrower before a repayment.

A lender uses a fresh Bitcoin key for each loan. A tranche's script, and the key in it, become public when the tranche is spent, so a reused key links the lender's loans on the Bitcoin blockchain and a compromised key exposes every loan it was used in.

14.2 The borrower's keys

Credential	Network	Purpose	Consequence of loss or compromise
Host-chain key	Host chain	Signs the term sheet, repays, cures, claims the principal	An attacker can sign term sheets in the borrower's name and claim the borrower's ledger balance. Loss before the principal is claimed strands it; loss afterwards prevents repayment, and every tranche defaults.
Bitcoin key	Bitcoin	Spends tranches through the hash-lock path after a reveal	Loss means the Bitcoin cannot be reclaimed even with the secret, and the lender takes it at the time lock. Compromise lets an attacker race the borrower for the output once the secret is public.

14.3 Lender procedures

Before committing. The lender completes any customer identification its obligations require before committing a loan (Chapter 15), and chooses terms within the bounds in Chapter 13.

Sizing the terms. The lender decides the principal, the collateral it posts, the premium a liquidation pays and the deadlines. Three figures deserve attention. The collateral should be sized so that each tranche's slice exceeds the value of that tranche's Bitcoin less its principal share, so that ghosting is always a loss; Chapter 6 gives the condition. The reveal window should be long enough for the lender's service to act with margin, since a lapse is unrecoverable. The challenge window between the two funding deadlines should allow for the required confirmations and the time to build a proof, beyond the contract's minimum.

Committing. The lender generates and stores the secrets, constructs and verifies the scripts, verifies the borrower's signature locally against `hashTerms`, confirms every term is as agreed, approves the two deposits, and calls `commitLoan`. The contract rejects anything these checks would reject, but a rejected commit costs the lender gas.

Proving. For each committed loan the lender watches the tranche addresses. When the funding transaction reaches `requiredConfirmations`, the lender submits the proof for every output at once. The lender is normally the first to submit, and should prove before `verificationDeadline`; after it, a third party may prove and claim the bounty from the lender's collateral.

Revealing. On every `TrancheRepaid`, the lender's service loads the tranche's secret and calls `revealSecret` without delay. This is the lender's one time-critical obligation. The event carries the deadline. A reveal after it is refused, and the tranche is liquidated from the lender's collateral.

Sweeping. For each `TrancheDefaulted` and for each interest tranche, the lender builds the time-lock spend once `btcTimelock` has passed, allowing about an hour for Bitcoin's median-time rule, and broadcasts it. The interest tranche's lock opens shortly after the verification deadline. A defaulted tranche's secret is never used.

Top-ups. On a `TrancheVerified` for an index above `N`, the lender inspects the top-up's script and confirms that the output is unspent and that the script gives the lender a time-lock path, before treating the cleared flag as a cure. The contract checks neither.

Claiming. The lender calls `claimPayout` for each asset when convenient. Balances accumulate in the ledger and do not expire.

Finalizing. After `verificationDeadline`, if any tranche remains `Pending`, the lender calls `finalizeFunding` to recover the undisbursed principal and the matching collateral.

14.4 Borrower procedures

Before signing. The borrower's wallet checks each script against the agreed hash, time lock and keys: in particular, that the borrower's own Bitcoin key appears in each collateral script's hash-lock branch, that every collateral tranche's lock time satisfies the timing rule, and that the interest tranche's lock time is the verification deadline plus ten minutes. It computes the script hashes itself, verifies `hashTerms`, and keeps a copy of every script. The scripts are needed to spend the Bitcoin later and are not stored on either chain.

Funding. The borrower waits for `LoanCommitted`, confirms the script hashes in the event match the scripts it built, and pays every tranche in one transaction with exact amounts, well before `fundingDeadline`.

Pre-signing. In a loan Maxy offers, the borrower may also give Maxy's service pre-signed returns for its collateral tranches and its interest recovery, so that the Bitcoin comes back without the borrower's wallet being online (13.10). This is optional, and the borrower keeps every spend it could otherwise make.

Proving. If the lender has not proven the funding within a reasonable time after the confirmations are in, the borrower submits the proof itself, so that the principal is credited. The borrower should not allow `verificationDeadline` to pass with a tranche unproven.

Withdrawing. The borrower claims the principal with `claimPayout`.

Repaying. The borrower repays each tranche before its last safe repayment time, allowing for the oracle if the rebate is enabled, and records the reveal deadline for each. The wallet reads `lastSafeRepayTime` and `quoteRebate` immediately before the call.

Spending. On `SecretRevealed`, the borrower spends the tranche's output at once. The lender's time lock on that tranche will eventually open regardless of the reveal.

Liquidation. If a reveal window lapses, the borrower or its watcher triggers the liquidation, watches the settler, and settles, reclaims and reopens the pot as needed until the Bitcoin is delivered.

Curing. On `HealthFlagged`, the borrower decides within the cure window whether to add Bitcoin, repay, or allow the lender to exit.

14.5 Watchers

Many calls are permissionless, and a party that is offline at the wrong moment loses by it. A watcher is a service that reads the loan contract's events and the Bitcoin chain and makes the time-sensitive calls on a party's behalf: submitting proofs, finalizing funding, marking defaults, triggering and managing liquidations, flagging and clearing health. A watcher needs only a funded host-chain key for gas. It holds no secrets and cannot redirect any payout, since every credit goes to the ledger of the address the contract determines.

14.6 The pauser

`PAUSER` is the one administrative address. It may call `setCommitsPaused(true)` to stop new loans from being committed, and `setCommitsPaused(false)` to resume. The pause has no effect on funding, repayment, reveal, default, liquidation, cure or payout for any existing loan. The pauser cannot move funds, change terms or close a loan.

14.7 Monitoring

The following conditions warrant an alert.

- A `Committed` loan with tranches still `Pending` as `verificationDeadline` approaches. The borrower should fund or prove; the lender should prepare to finalize.
- A funding transaction confirmed on Bitcoin with no `TrancheVerified` after `requiredConfirmations`. Someone should prove.
- An `Escrowed` tranche with no `SecretRevealed` within a fraction of the reveal window. The lender's service may be failing.
- A `Live` tranche approaching its last safe repayment time. The borrower must decide.
- A `Settled` tranche whose output is still unspent as `btcTimelock` approaches. The borrower must spend.
- A `Defaulted` tranche, or the interest tranche, whose time lock has opened and has not been swept.
- A `Ghosted` tranche whose pot has been `Open` past its delivery deadline, or `Reclaimed` without being reopened.
- A `HealthFlagged` loan approaching its cure deadline.
- The verifier's `getLatestHeight()` falling more than a few blocks behind the Bitcoin chain tip.
- Any `commitLoan` call that reverts, which indicates that the wallets and the contract disagree about validity.

15. Compliance and auditability

MaxyLock is a bilateral agreement carried out by contracts and scripts rather than by an intermediary. The compliance obligations that apply are therefore the parties' own, and the protocol's contribution is to make every element of the agreement and its performance verifiable by either party, by an auditor and by a regulator without reliance on anyone's records. This chapter describes where the decision points are and what the record contains.

15.1 Counterparty selection

No loan exists until the borrower has signed the term sheet and the lender has submitted it. Each party chooses the other before any value moves. A lender that is subject to customer identification requirements completes them before committing a loan, and the borrower's host-chain address in the term sheet ties the identified customer to every subsequent action on the loan: the signature, the repayments and the withdrawals all come from that address. The borrower's Bitcoin payout script is likewise fixed in the signed terms, so a Mega-Pot delivery can only go to a destination the borrower committed to at signing.

The lender's decision to commit is the last point at which the lender can decline without cost. After commit, the protocol runs on the signed terms. A lender who wishes to screen the Bitcoin that will fund the loan does so before committing, by agreeing with the borrower which outputs will be spent into the tranches.

15.2 No custody and no pooling

The protocol never takes custody of either party's assets in the sense of holding them under a third party's control. The Bitcoin remains in scripts spendable only by the two parties. The loan asset and the lender's collateral are held by a contract that has no administrator able to move them and that accounts for them per loan and per tranche. Nothing is pooled across loans or across parties, and no party's funds are exposed to another loan's outcome. The single administrative function, a pause on new commitments, cannot affect any existing loan.

15.3 The audit trail

Every step of a loan leaves a record that can be read without either party's cooperation.

Record	Where	What it establishes
Signed term sheet	Held by both parties; its hash is in `LoanCommitted`	The complete terms both parties agreed, and the borrower's signature
`LoanCommitted`	Host chain	Both parties' addresses, the hash of the term sheet, every tranche's script hash and sats, and the funding and verification deadlines
Funding transaction	Bitcoin	The amounts locked, the scripts they were locked to, and the inputs they came from
`TrancheVerified`, `LoanActivated`	Host chain	The funding transaction was verified against the Bitcoin headers; the principal credited per tranche; the term start
`TrancheRepaid`	Host chain	The repayment escrowed, any rebate applied, and the reveal deadline
`SecretRevealed`	Host chain	The secret, the amount paid to the lender and the collateral returned
Tranche spend	Bitcoin	Whether the borrower reclaimed the tranche with the secret or the lender swept it after the lock time
`TrancheDefaulted`, `GhostLiquidationTriggered`, `MegaPotOpened`, `MegaPotDelivered`	Host chain	Which remedy ran, when, and with what amounts
`PayoutQueued`, `PayoutClaimed`	Host chain	Every credit to a party and every withdrawal

15.4 Verification by a third party

An auditor can reconstruct a loan in full from public data.

1. Read `LoanCommitted` for the loan. It gives both addresses, the script hashes and each tranche's sats; the principal, the collateral and the remaining terms are in the signed term sheet and in the loan record returned by `getLoan`.

2. Obtain the signed term sheet from either party and confirm that its hash matches the `termsHash` in the event and that its signature recovers to the borrower's address.
3. Read `TrancheVerified` for each tranche. It names the Bitcoin transaction and output. Confirm on the Bitcoin blockchain that the output paid the stated amount to the stated script hash.
4. Follow each tranche's outpoint on Bitcoin to see how it was spent: by the borrower with a secret matching the hashlock, or by the lender after the lock time.
5. Read the repayment, reveal, default and liquidation events on the host chain and reconcile them against the Bitcoin spends. A `SecretRevealed` event should correspond to a borrower reclaim; a `TrancheDefaulted` event to a lender sweep; a `MegaPotDelivered` event to a filler's payment to the borrower's script.
6. Sum the payout events to confirm that each party received exactly what the terms entitled.

None of these steps requires access to either party's systems.

15.5 Controls

Control	Provided by
Counterparty identification before any value moves	The term sheet; nothing is committed without both parties' agreement
Segregation of each loan's funds	Per-loan and per-tranche accounting in the contract; per-tranche scripts on Bitcoin
No custodial risk	Bitcoin in two-party scripts; loan assets in a contract with no administrative access
Principal paid only against verified collateral	Proof verification tranche by tranche
Settlement without discretion	The escrow-for-secret exchange and the time-locked default path
Remedy for each party's non-performance, without recourse to a third party	The time-lock sweep and the Mega-Pot
Separation of duties	Distinct keys for signing, for Bitcoin spending and for secrets, on each side (Chapter 14)
Complete and tamper-evident records	Events on the host chain and transactions on Bitcoin
Bounded administrative power	A pause on new commitments only

These controls give each party, and anyone reviewing a loan, verifiable evidence of who agreed to what, what was delivered, and how every obligation was discharged. Legal requirements vary by

jurisdiction and by the nature of each party; the protocol supplies the technical record on which a compliance program is built.

16. Security model

16.1 Guarantees

Property	Enforced by	Mechanism
The lender cannot take the Bitcoin before the time lock	Bitcoin consensus	The lender's path is guarded by `OP_CHECKLOCKTIMEVERIFY`
The borrower cannot take the Bitcoin without the secret	Bitcoin consensus	The borrower's path requires a preimage whose SHA-256 equals the hash lock
No principal is released without a funded tranche	Loan contract and verifier	A tranche becomes `Live` only through a proof that an output pays its exact amount to its script
The amount proven equals the amount agreed	Verifier	The output value is checked for exact equality
Every proof shows the agreed confirmations	Loan contract	`latestHeight − blockHeight + 1 ≥ requiredConfirmations` is checked on every proof
One Bitcoin transaction backs a loan	Loan contract	A second transaction identifier is rejected
The lender is paid only by publishing the secret	Loan contract	`revealSecret` requires `SHA256(preimage) == hashlock` and emits the preimage
A paid borrower always has time to act before the lender's time lock	Loan contract and the timing rule	Repayment is refused after `btcTimelock − revealWindow − 2 days`, so the reveal window plus the safety margin ends before any lender time lock, and the Mega-Pot is available after the window
A withheld secret costs the lender	Loan contract and settler	After the reveal window, the slice and the escrow fund a Bitcoin purchase for the borrower at a premium, for as long as the lender's collateral remains in the contract (see 12.2)
A default is confined to its tranche	Loan contract	Each tranche has its own state and slice

Property	Enforced by	Mechanism
Collateral leaves only through a resolution	Loan contract	`collateralHeld` decreases only in a reveal, a default, an unfunded return, a liquidation, a bounty or an uncured flag
No one but the owed party receives a payout	Loan contract	Every credit goes to the ledger of the address the contract determines; `claimPayout` pays only the caller's own balance
No price can liquidate the main path	Loan contract	Funding, repayment without the rebate, default and liquidation read no oracle
No administrator can reach funds	Loan contract	The only privileged function pauses new commitments; `setCommitsPaused` gates `commitLoan` only, so a pause cannot trap anyone
Fee-on-transfer tokens cannot under-fund a loan	Loan contract	Every transfer in is checked by balance difference; the principal must arrive exactly and the collateral within ten base units
The borrower's liquidation claim never expires	Loan contract	A pot can be reclaimed and reopened without limit

16.2 Assumptions

- Bitcoin and the host chain are operating and are not reorganized beyond the confirmation depths in use.
- The verifier holds the canonical Bitcoin header chain. A relayer that supplies an invalid header is rejected by the verifier's proof-of-work checks; a relayer that supplies nothing only delays proofs.
- Each tranche's script hash corresponds to a script with the agreed branches, keys, hash and time lock. The contract records the hash and cannot inspect the script. Both wallets check it before signing and before funding.
- Every tranche of every loan has a new script. The contract does not prevent the same script hash from being registered on two loans, so a reused script could be counted twice. A fresh secret from the lender makes a fresh script, and the parties' wallets enforce the rule.
- A top-up tranche is vetted by the lender outside the contract. The contract accepts the borrower's own terms for it and checks only that an output paid the stated amount to the stated script hash. It cannot tell whether the lender has a path in that script, or whether the output is still unspent.
- The lender keeps every secret from the borrower until the repayment and retains it until the reveal.

- Each party acts within its deadlines: the borrower funds before `fundingDeadline`, ensures that a proof is submitted before `verificationDeadline`, repays before the last safe repayment time and spends each tranche promptly after the secret is published; the lender reveals within the reveal window.
- The interest tranche's script is agreed between the parties. The contract pins its time lock and does not check how the borrower would recover the interest if the loan never activated. The parties settle that in the script.
- The verifier, the settler and the oracle adapter are shared infrastructure governed outside this contract. The settler's governance must allow each asset a pot may hold. The oracle adapter's governance affects only the rebate and Module A, and where either is enabled the adapter is assumed to return accurate prices.
- Under Module B, the collateral asset differs from the loan asset and backs one active loan on this contract at a time.
- The settler operates and has fillers who will take a pot at its posted price. If no filler delivers, the pot can be re-posted indefinitely, but the borrower's compensation waits until one does.

16.3 Capabilities of each party

Party	Can	Cannot
Lender	Commit on signed terms; prove funding; reveal; sweep defaulted and interest tranches after their time locks; exit after an uncured health flag; recover unfunded shares; decline to reveal	Take a repayment without the secret; take the collateral back before a resolution; take a tranche's Bitcoin before its time lock; reveal after the window; withhold the secret without forfeiting collateral; move the borrower's escrow anywhere but to the Mega-Pot; change the terms; stop a liquidation
Borrower	Sign terms; fund or not fund; prove; claim the principal; repay in any order before the last safe time; cure a flag; spend with the secret; trigger a liquidation	Spend a tranche without the secret; repay after the last safe time; receive principal for an unproven tranche; change the terms; recover Bitcoin sent to an unproven tranche without the lender
Filler	Commit to a pot, deliver Bitcoin and take the pot	Take the pot without a verified delivery; keep the bond after a missed delivery

Party	Can	Cannot
Anyone	Prove funding; finalize; mark defaults; trigger, settle, reclaim and reopen pots; flag and clear health; skim yield to the signed hedge adapter	Direct any payout to itself, except a bounty for a late proof; move any balance to an address the terms do not name; close a loan early
Pauser	Pause and unpause commits	Anything else; touch any existing loan or any balance
Holder of a stolen secret	Give the borrower the means to spend one tranche without repaying	Take any Bitcoin or any host-chain balance
Settler governance	Allow or disallow pot assets	Touch any balance in the loan contract
Oracle adapter governance	Change the sources behind the rebate and Module A prices	Affect funding, repayment, default or liquidation

16.4 Failure cases

Event	Outcome	Cost borne by
The borrower never funds	After `verificationDeadline`, funding is finalized and the lender recovers both deposits; the loan is `Cancelled`	Gas only
The borrower funds some tranches	The others are finalized unfunded; the loan proceeds with the rest	No one
The borrower pays a wrong amount or uses a second transaction	The output cannot be proven. The tranche is finalized unfunded. The Bitcoin is spendable by the borrower only with the secret and by the lender at the time lock	The borrower, unless the lender gives up the secret or returns the funds. Maxy, as lender, discloses the secret as policy (9.8)
The borrower funds but the proof lands after the lender has finalized	As above	The borrower, unless the lender gives up the secret. Maxy, as lender, discloses it as policy (9.8)

Event	Outcome	Cost borne by
The lender does not prove	The borrower or a third party proves; after `verificationDeadline` the third party is paid from the lender's collateral	The lender, up to `bountyBps`
The lender loses a secret, reveals late, or does not reveal	The tranche is liquidated from the lender's slice and the escrow. If Maxy, as lender, misses a reveal window, it discloses the secret through the Maxy API so that the borrower can reclaim the Bitcoin, and triggers the liquidation itself only if it cannot recover the secret	The lender
The lender discloses a secret before repayment	The borrower can reclaim that tranche without paying; it then defaults with no Bitcoin to sweep	The lender
The borrower does not repay	The tranche defaults; the lender recovers the slice and sweeps the Bitcoin	The borrower
The borrower repays and the lender reveals, but the borrower does not spend before the time lock	Both parties can spend; the lender may sweep	The borrower
The borrower loses the Bitcoin key	The borrower cannot reclaim even with the secret; the lender sweeps after the lock time	The borrower
No filler takes a pot	The pot is reclaimed and reopened until one does	The borrower, in time
A filler commits and does not deliver	The filler is slashed; the share of the bond the settler pays in the pot's asset joins the pot	The filler
The oracle adapter is unavailable	Rebated repayments before `termEnd` and every Module A call revert until it recovers; the cure window keeps running	Delay, and a borrower's cure window can run out during the outage
A price fall under Module A	The borrower cures or the lender exits with its collateral; tranches in escrow keep their escrow	The borrower, on uncured tranches
The lender exits after an uncured flag while a tranche is `Escrowed`	The whole collateral leaves with the lender; the escrowed tranche's liquidation pot would hold the escrow alone and is unlikely to fill; the borrower depends on the lender revealing	The borrower, against a lender that then withholds the secret

Event	Outcome	Cost borne by
A borrower cures a flag with a top-up the lender cannot sweep or that is already spent	The flag clears and the lender's exit is blocked; the loan continues on the original tranches	The lender, by not inspecting the top-up
The verifier lags the Bitcoin chain	Proofs revert and are retried	Delay only
The settler does not allow the pot's asset	The liquidation cannot be posted and the tranche stays `Escrowed` until governance allows it; the lender can no longer reveal	Delay for the borrower
The same script hash is registered on two loans	Both loans could count one Bitcoin output	The lender whose process allowed it, against a dishonest borrower
The pauser key is stolen	New loans can be paused; nothing else	Contained
Two active loans share a collateral asset under Module B, or the collateral asset is the loan asset	A skim can draw on another loan's collateral, or on escrowed repayments and undisbursed principal	The lenders, by misconfiguration

16.5 Operating rules

1. Both parties verify every tranche script before signing and before funding.
2. The lender reveals promptly after every repayment and never lets a reveal window lapse.
3. The borrower proves funding itself if the lender has not, before `verificationDeadline`.
4. The borrower repays before the last safe repayment time or accepts the default. Every collateral tranche obeys the timing rule; the contract checks it at commit and at every repayment.
5. The borrower spends each tranche as soon as its secret is public.
6. Collateral leaves the contract only through a reveal, a default, an unfunded return, a liquidation, a bounty or an uncured health flag. There is no other exit. Every balance leaves the contract only through `claimPayout`, by its owner.
7. Every tranche of every loan has a fresh script. A script hash is never reused, and the lender uses a fresh Bitcoin key for each loan.

8. Secrets are backed up, withheld from the borrower until the repayment, used exactly once, and stored as carefully as keys.

17. Contract reference

A deployment consists of the loan contract, the extensions contract, and three shared contracts that the loan contract calls: the verifier, the settler and the oracle adapter. Redemption adapters are deployed as needed, one per yield-bearing collateral asset. All are written in Solidity 0.8 on OpenZeppelin Contracts 5. This chapter documents their interfaces.

17.1 Loan contract

The loan contract is `MaxyLockLoan`. Its storage, constants, errors and events are declared in `MaxyLockLoanStorage`, which both it and the extensions contract inherit, so that the two share one layout.

Structures

```
struct TrancheTerms {
    bytes32 scriptHash;        // SHA-256 of the witness script
    bytes32 hashlock;          // H = SHA256(S); zero for the interest tranche
    uint64  satsValue;
    uint32  btcTimelock;       // Unix time in the script
}

struct LoanTerms {
    address borrower;
    address lender;
    address collateralAsset;   // zero address for native
    uint256 collateralAmount;
    address loanAsset;         // zero address for native
    uint256 principal;
    uint64  fundingDeadline;
    uint64  verificationDeadline;
    uint64  termEnd;
    uint32  revealWindow;
    uint16  requiredConfirmations;
    uint16  ghostPayoutBps;
    uint16  bountyBps;
    uint16  rebateSlippageBps;
    uint8   flagsBits;
    bytes32 loanAssetFeedId;   // Pyth feed ID of the loan asset
    uint32  oracleStaleAfter;
    uint16  healthThresholdBps;
    uint32  cureWindow;
    address moduleBAdapter;
    uint16  moduleBYieldBps;
```

```
    bytes   borrowerBtcScript; // output script paid by a liquidation
    uint256 nonce;
    TrancheTerms[] tranches;   // N collateral tranches, then the interest tranche
}

struct BtcProof {
    uint256   blockHeight;
    bytes     rawTx;
    uint256   txIndex;
    bytes32[] merkleProof;
    bytes     header;
    bytes     prevHeader;
}

struct Tranche {
    bytes32 scriptHash;
    bytes32 hashlock;
    uint64  satsValue;
    uint32  btcTimelock;
    uint32  paymentVout;       // set by the funding proof
    uint8   state;             // TrancheState
    uint64  repaidAt;          // 0 until Escrowed
    uint256 principalShare;    // 0 for the interest tranche and for top-ups
    uint256 escrowAmount;      // 0 except while Escrowed
}

struct Loan {
    address borrower;
    uint8   state;             // LoanState
    uint8   collateralTrancheCount;   // N
    uint8   liveOrTerminalCount;      // tranches in a final state
    uint16  requiredConfirmations;
    uint16  ghostPayoutBps;
    uint16  bountyBps;
    uint16  rebateSlippageBps;
    address lender;
    uint32  revealWindow;
    uint64  fundingDeadline;
    address collateralAsset;
    uint64  verificationDeadline;
    uint32  cureWindow;
    address loanAsset;
    uint64  termStart;         // set at activation
    uint16  healthThresholdBps;
    uint8   flagsBits;
    uint8   _gap0;
    bytes32 loanAssetFeedId;
    uint64  termEnd;
    uint32  oracleStaleAfter;
```

```
    uint256 collateralAmount;  // as received at commit
    uint256 collateralHeld;    // collateral not yet released
    uint256 principal;
    uint256 principalDisbursed;
    bytes32 termsHash;
    bytes32 fundingTxId;       // display byte order; zero until the first proof
    Tranche[] tranches;
    bytes   borrowerBtcScript;
    address moduleBAdapter;
    uint64  healthFlaggedAt;   // 0 when no flag is open
    uint16  moduleBYieldBps;
}

struct MegaPot {
    bytes32   intentHash;      // the settler's order
    uint64    btcSatsOwed;
    uint64    openedAtHeight;
    uint8     state;           // PotState
    address[] assets;          // one entry
    uint256[] amounts;         // one entry
}

struct AssetRouteInput {       // constructor only
    address asset;
    address underlying;
    address adapter;
}
```

EIP-712. Domain name `MaxyLock`, version `1`, the host chain's ID, the contract address. Type string:

```
LoanTerms(address borrower,address lender,address collateralAsset,uint256
collateralAmount,address loanAsset,uint256 principal,uint64 fundingDeadline,uint64
verificationDeadline,uint64 termEnd,uint32 revealWindow,uint16 requiredConfirmations,uint16
ghostPayoutBps,uint16 bountyBps,uint16 rebateSlippageBps,uint8 flagsBits,bytes32
loanAssetFeedId,uint32 oracleStaleAfter,uint16 healthThresholdBps,uint32 cureWindow,address
moduleBAdapter,uint16 moduleBYieldBps,bytes borrowerBtcScript,uint256 nonce,TrancheTerms[]
tranches)TrancheTerms(bytes32 scriptHash,bytes32 hashlock,uint64 satsValue,uint32 btcTimelock)
```

`borrowerBtcScript` enters the structure hash as its `keccak256`. `tranches` enters as the `keccak256` of the concatenation of each tranche's structure hash, in order. `hashTerms(LoanTerms)` returns the final digest and `hashTrancheTerms(TrancheTerms)` returns one tranche's structure hash.

Constants

Constant	Value	Meaning
`MAX_COLLATERAL_TRANCHES`	7	Largest `N`

Constant	Value	Meaning
`MAX_CONFIRMATIONS`	8	Largest `requiredConfirmations`
`MIN_REVEAL_WINDOW`	1 day	Smallest `revealWindow`
`REVEAL_SAFETY_MARGIN`	2 days	Buffer between a reveal window and a time lock
`INTEREST_SWEEP_DELAY`	10 minutes	Interest time lock relative to `verificationDeadline`
`MAX_BOUNTY_BPS`	1,000	Largest `bountyBps`
`MIN_GHOST_PAYOUT_BPS`, `MAX_GHOST_PAYOUT_BPS`	10,000, 15,000	Range of `ghostPayoutBps`
`BPS`	10,000	Basis-point denominator
`ASSUMED_BTC_BLOCK_TIME`	20 minutes	Block time assumed when checking the challenge window
`CHALLENGE_WINDOW_BUFFER`	1 hour	Added to the challenge-window minimum
`MAX_TOTAL_TRANCHES`	255	Tranches per loan including top-ups
`MAX_PULL_ROUNDING_DUST`	10	Base units of shortfall tolerated when pulling collateral
`GHOST_TIMEOUT_MIN_EXTRA_BLOCKS`, `GHOST_TIMEOUT_MAX_BLOCKS`	10, 4,320	`timeoutBlocks` must lie in `[requiredConfirmations + 10, 4320]`
`GHOST_INTENT_BOND_BPS`	500	Filler bond on a Mega-Pot order
`GHOST_INTENT_DEADLINE_SLACK`	30 days	Life of a Mega-Pot order on the settler
`SKIM_INTERVAL`	1 day	Minimum interval between Module B skims
`MODULE_A_BIT`, `MODULE_B_BIT`, `MODULE_C_BIT`, `REBATE_ENABLED_BIT`, `REBATE_BITCOIN_DELIVERY_BIT`	bits 0 to 4 of `flagsBits`	Toggles; see 12.1

Enumerations. `LoanState`: `None` 0, `Committed` 1, `Active` 2, `Settled` 3, `Cancelled` 4. `TrancheState`: `Pending` 0, `Live` 1, `Escrowed` 2, `Settled` 3, `Defaulted` 4, `Ghosted` 5,

`LiquidationDelivered` 6, `Unfunded` 7. `PotState`: `None` 0, `Open` 1, `Delivered` 2, `Reclaimed` 3.

Functions

Function	Caller	Preconditions	Effect
`commitLoan(LoanTerms t, bytes sig) payable returns (uint256 loanId)`	`t.lender`	Not paused; `sig` recovers to `t.borrower`; `t.nonce == borrowerNonce[t.borrower]`; every rule in 13.2; value equals the native legs	Pulls the deposits, recor loan `Committed` with ev tranche `Pending`, fixes principal shares, emits `LoanCommitted`
`submitFundingProof(uint256 loanId, uint8[] trancheIdx, uint256[] outputIndexes, BtcProof proof, address payAsset, uint256 maxPayment) payable`	Anyone	Loan `Committed` or `Active`; indices ascending, each `Pending`; proof verifies; confirmations sufficient; same funding transaction	Marks each tranche `Liv` credits principal shares, activates the loan on the collateral tranche, pays bounty if due, emits `TrancheVerified` and `LoanActivated` and `ProofBountyPaid`
`finalizeFunding(uint256 loanId)`	Anyone	Loan `Committed` or `Active`; after `verificationDeadline`	Marks `Pending` tranche `Unfunded`, credits the l sets `Cancelled` or `Set` where applicable, emits `LoanFinalized`
`repayTranche(uint256 loanId, uint8 idx) payable`	The borrower	Loan `Active`; tranche `Live`; not the interest tranche; before the last safe time	Pulls `principalShare` + `rebate`, marks `Escrowe` emits `TrancheRepaid`
`revealSecret(uint256 loanId, uint8 idx, bytes32 preimage)`	The lender	Loan `Active`; tranche `Escrowed`; within the reveal window; `sha256(preimage) == hashlock`	Credits the escrow and to the lender, marks `Set` emits `SecretRevealed` the loan if every tranche
`markTrancheDefaulted(uint256 loanId, uint8 idx)`	Anyone	Loan `Active`; tranche `Live`; not the interest tranche; at or after the last safe time and after `termEnd`	Credits the slice to the l marks `Defaulted`, emit `TrancheDefaulted`, clo loan if every tranche is f
`flagUnhealthy(uint256 loanId)`	Anyone	Loan `Active`; Module A; no flag; unhealthy	Records the flag, emits `HealthFlagged`
`clearFlag(uint256 loanId)`	Anyone	Flag open; healthy	Clears the flag, emits `FlagCleared`

Function	Caller	Preconditions	Effect
`cureWithRepayment(uint256 loanId, uint8 idx) payable`	The borrower	Loan `Active`; Module A; flag open; within the cure window; repayment preconditions	Repays as `repayTranch` clears the flag if healthy, emitting `HealthCured`
`cureWithCollateral(uint256 loanId, TrancheTerms topUp, uint256 outputIndex, BtcProof proof, address payAsset, uint256 maxPayment) payable returns (uint8 idx)`	The borrower	Loan `Active`; Module A; flag open; within the cure window; `topUp.satsValue > 0`; fewer than 255 tranches; `topUp.btcTimelock > termEnd + revealWindow + 2 days`; proof verifies	Appends a `Live` tranch no principal share, emits `TrancheVerified`; clea flag if healthy
`claimAfterCureWindow(uint256 loanId)`	The lender	Flag open; cure window passed; still unhealthy	Credits all remaining col to the lender, marks eve collateral tranche `Defau` emits `CollateralClaimedAft` closes the loan if every t is final
`claimPayout(address asset)`	Anyone	`payoutLedger[caller][asset] > 0`	Transfers the caller's wh balance in `asset`, emits `PayoutClaimed`
`setCommitsPaused(bool paused)`	`PAUSER`		Gates `commitLoan` only `CommitsPauseSet`
`receive()`	`SETTLER` only		Accepts native refunds f settler
`fallback()`			Forwards any other call `EXTENSIONS` by `delegatecall`; not pay

Views. `hashTerms(LoanTerms) returns (bytes32)`; `hashTrancheTerms(TrancheTerms) returns (bytes32)` (pure); `getLoan(uint256) returns (Loan)`; `getTranche(uint256, uint8) returns (Tranche)`; `getMegaPot(uint256 potKey) returns (MegaPot)`; `principalOutstanding(uint256) returns (uint256)`, the sum of principal shares over `Live` collateral tranches; `lastSafeRepayTime(uint256, uint8) returns (uint256)`; `healthStatus(uint256) returns (bool unhealthy, int256 answer)`; `quoteRebate(uint256, uint8) returns (uint256)`; `trancheScriptPubKeys(uint256, uint8) returns (bytes paymentScriptPubKey)`; the public state `loanCount`, `commitsPaused`, `borrowerNonce(address)`,

`payoutLedger(address who, address asset)` and `totalLedgered(address asset)`; and the immutable addresses `VERIFIER`, `SETTLER`, `PAUSER`, `ORACLE_ADAPTER`, `EXTENSIONS` and the immutable `BTC_FEED_ID`.

Every view that reports a figure a mutator will act on calls the same internal function as the mutator. `lastSafeRepayTime` is the value `RepayTooLate` reports, `quoteRebate` is the amount `repayTranche` deducts, and `healthStatus` is the reading `flagUnhealthy` acts on.

Events

Event	Fields
`LoanCommitted`	`loanId`, `borrower`, `lender`, `termsHash`, `scriptHashes[]`, `sats[]`, `fundingDeadline`, `verificationDeadline`
`TrancheVerified`	`loanId`, `idx`, `btcTxId`, `paymentVout`, `principalReleased`
`LoanActivated`	`loanId`, `termStart`, `termEnd`
`ProofBountyPaid`	`loanId`, `submitter`, `amount`
`LoanFinalized`	`loanId`, `fundedCount` (original collateral tranches `Live` at the time of the call), `principalBack`, `collateralBack`
`LoanCancelled`	`loanId`
`LoanSettled`	`loanId`
`TrancheRepaid`	`loanId`, `idx`, `escrowed`, `rebate`, `revealDeadline`
`SecretRevealed`	`loanId`, `idx`, `preimage`, `paidToLender`, `collateralReturned`
`TrancheDefaulted`	`loanId`, `idx`, `collateralReleased` (0 when released in one sum by `claimAfterCureWindow`)
`GhostLiquidationTriggered`	`loanId`, `idx`, `potKey`
`MegaPotOpened`	`potKey`, `intentHash`, `btcSats`, `assets[]`, `amounts[]`
`MegaPotDelivered`	`potKey`
`MegaPotReclaimed`	`potKey`
`HealthFlagged`	`loanId`, `cureDeadline`, `answer`
`HealthCured`	`loanId`
`FlagCleared`	`loanId`
`CollateralClaimedAfterCure`	`loanId`, `amount`

Event	Fields
`YieldSkimmed`	`loanId`, `amount`
`PayoutQueued`	`who`, `asset`, `amount`
`PayoutClaimed`	`who`, `asset`, `amount`
`CommitsPauseSet`	`paused`

Errors. `ZeroAddress`, `ExtensionsMismatch(extensionsVerifier, extensionsSettler)`, `DuplicateRoute(asset)`, `NotLender(caller, expectedLender)`, `NotBorrower(caller, expectedBorrower)`, `TransferAmountMismatch(expected, actual)`, `NotBorrowerSignature(recovered)`, `BadTrancheCount(n)`, `ConfirmationsOutOfRange(c)`, `DeadlinesUnordered`, `TimelockTooTight(tranche)`, `ChallengeWindowTooTight(requiredConfirmations, window, minWindow)`, `InterestTimelockWrong`, `ZeroAmount`, `ZeroSats(t)`, `BpsOutOfRange`, `BtcScriptEmpty`, `NonceUsed(nonce)`, `CommitsArePaused`, `WrongNativeValue(want, got)`, `IncoherentAssetPair`, `WrongLoanState(have, want)`, `WrongTrancheState(idx, have)`, `NoTranchesListed`, `DuplicateTranche(idx)`, `TrancheIndexOutOfRange(idx, max)`, `ArrayLengthMismatch`, `VerifierRejected`, `FundingTxMismatch(have, expected)`, `VoutOutOfRange(index)`, `FundingNotOver`, `InsufficientConfirmations(have, required)`, `RepayTooLate(idx, lastSafe)`, `WrongPreimage(idx)`, `RevealWindowClosed(idx)`, `RevealWindowStillOpen(idx)`, `DeadlineNotReached(idx)`, `RebateBitcoinDeliveryNotSupported`, `PotNotOpen(key)`, `PotNotExpired(key)`, `PotNotDelivered(key)`, `PotNotReclaimed(key)`, `TimeoutOutOfRange(have, min, max)`, `TwoAssetPotNotSupported`, `ModuleNotEnabled`, `NotUnhealthy`, `StillUnhealthy`, `AlreadyFlagged`, `NotFlagged`, `CureWindowOpen`, `CureWindowOver`, `TooManyTranches`, `SkimTooSoon(nextSkimAt)`, `NothingToClaim`, `NativeSendFailed`, `NotSettler(caller)`, `NotPauser`. `UnknownLoan`, `PastFundingDeadline` and `Reentrancy` are declared and unused; a reentrant call reverts with OpenZeppelin's own error.

17.2 Extensions contract

`MaxyLockLoanExtensions` holds the ghost-liquidation and Module B functions. It exists because the loan contract's code would otherwise exceed the host chain's contract-size limit. It is constructed with the same `verifier` and `settler` as the loan contract, and the loan contract's constructor checks that they match. Callers never address it directly: a call to any of its functions is made to the loan contract's address, which forwards it by `delegatecall`, so the function runs against the loan contract's storage and balances. A direct call to the extensions contract would act on empty storage and is of no use.

Function	Caller	Preconditions	Effect
`triggerGhostLiquidation(uint256 loanId, uint8 idx, uint32 timeoutBlocks) returns (uint256 potKey)`	Anyone	Loan `Active`; tranche `Escrowed`; reveal window closed; `timeoutBlocks` in range; a route if the assets differ	Marks the tranche `Ghosted`, forms the pot, posts the order, emits `GhostLiquidationTriggered` and `MegaPotOpened`
`settleMegaPot(uint256 potKey)`	Anyone	Pot `Open`; the settler reports `FILLED`	Marks the pot `Delivered` and the tranche `LiquidationDelivered`, emits `MegaPotDelivered`, closes the loan if every tranche is final
`reclaimMegaPot(uint256 potKey)`	Anyone	Pot `Open`; the settler reports `COMMITTED` past its deadline, `NONE` past the order's deadline, `REFUNDED` or `SLASHED`	Triggers the settler's refund or cancellation where one is due and measures what returns; otherwise keeps the last recorded amount. Marks the pot `Reclaimed` and the tranche `Ghosted`, emits `MegaPotReclaimed`
`reopenMegaPot(uint256 potKey, uint32 timeoutBlocks)`	Anyone	Pot `Reclaimed`; `timeoutBlocks` in range	Posts a new order for the same `btcSatsOwed`, marks the pot `Open`, emits `MegaPotOpened`
`skimYieldToHedge(uint256 loanId)`	Anyone	Loan `Active`; Module B; at least one day since the last skim	Forwards `moduleBYieldBps` of the measured yield to `moduleBAdapter`, emits `YieldSkimmed`; forwards nothing when there is no yield, but still consumes the day's skim
`isValidSignature(bytes32 hash, bytes) view returns (bytes4)`	The settler		Returns the EIP-1271 magic value for a pot order hash the contract itself posted, and a rejection for anything else

17.3 Verifier

The verifier maintains the Bitcoin header chain and verifies proofs against it. The loan contract uses the following functions.

Function	Purpose
`verifyTransactionWithTranches(blockHeight, transaction, txIndex, merkleProof, header, prevHeader, TrancheCheck[] tranches, payAsset, maxPayment) returns (bool)`	Confirms that the header is canonical and sufficiently confirmed by the verifier's own floor, that the transaction is in the block, and that each `TrancheCheck` names an output paying exactly `satsValue` to `scriptPubKey`. Up to 8 checks per call. May charge a fee.
`getLatestHeight()`	The verifier's view of the chain tip; used for the loan's own confirmation check and for pot opening heights
`minConfirmations()`	The verifier's global confirmation floor; the lower bound of `requiredConfirmations`
`calculateTxId(transaction)`	Transaction identifier of a raw transaction in display byte order, with witness data excluded; stored as `fundingTxId`
`quoteVerificationFee(outputValueSats, payAsset)`, `isFeeExemptCaller(caller)`	Fee discovery

`TrancheCheck` is `{uint256 outputIndex; uint64 satsValue; bytes scriptPubKey}`. Any party may extend the header chain through the verifier's own header-submission functions, and the verifier validates proof of work and difficulty. The Maxy Mint documentation describes the verifier in full; the loan contract uses only the functions above.

17.4 Settler

The settler is the swap venue for Mega-Pot orders. The loan contract acts as a buy-side maker and uses the following functions.

Function	Purpose
`hashIntent(Intent)`	The order hash the contract pre-authorizes

Function	Purpose
`openBuyEscrowFor(Intent, bytes scriptPubKey, OracleConfig, bytes signature, address refundRecipient)`	Posts the order and deposits the pot; the loan contract passes its own address as the refund recipient and an empty signature, relying on EIP-1271
`commitments(bytes32 intentHash)`	The order's state; `FILLED` releases the pot to the filler
`refundExpired(bytes32 intentHash)`	Returns the pot, with any share of a slashed filler's bond the settler pays out, after a committed filler's deadline passes
`cancelBuyEscrow(bytes32 intentHash)`	Returns the pot of an order no filler committed to, after the order's own deadline

Order states are `NONE`, `COMMITTED`, `FILLED`, `SLASHED` and `REFUNDED`. The order the loan contract posts has the following fields.

Field	Value
`side`	`BUY`
`maker`	The loan contract
`asset`	The pot's asset
`scriptHash`	`keccak256(borrowerBtcScript)`
`satAmount`	`btcSatsOwed`
`designatedFiller`	None
`pricingMode`, `limitPrice`	`LIMIT`; `pot × 10^18 / (btcSatsOwed × 10^assetDecimals)`
`bondBps`	500
`requiredConfirmations`	The loan's
`settlementTimeoutBlocks`	`timeoutBlocks`
`deadline`	Thirty days after posting
`nonce`	Derived from the opening height, time and block

The settler takes `btcSatsOwed × 10^assetDecimals × limitPrice / 10^18` from the pot, and any rounding residue is credited to the borrower's ledger.

17.5 Oracle adapter

The oracle adapter implements `resolveRate(OracleFeedsConfig feeds, bytes32 expectedFeedsHash) returns (uint256 rateWad)`. The loan contract calls it with `counterFeedId = BTC_FEED_ID`, `originFeedId = loanAssetFeedId`, `maxAge = oracleStaleAfter`, `counterTokenDecimals = 8` and `originTokenDecimals` equal to the loan asset's decimals, and converts satoshis to loan-asset units as `sats × rateWad / 10^18`. The same conversion serves the rebate and the health check.

The reference adapter reads each feed from Pyth and, where a Chainlink aggregator is registered for the feed, from Chainlink. When both are fresh it rejects the reading if they diverge beyond a configured tolerance and otherwise uses their midpoint. When only one is fresh it uses that one. When neither is fresh, or a price is not positive, it reverts. Its configuration is governed by its owner, outside the loan contract.

17.6 Redemption adapters

A redemption adapter implements `redeem(address asset, uint256 amount, address to) returns (uint256 received)`. It pulls `amount` of `asset` from the caller, which has approved it in the same transaction, redeems through its one fixed venue, sends the underlying asset to `to`, and returns the amount delivered. Adapters are ownerless and stateless, and hold nothing between calls. Three are provided.

Adapter	Collateral asset	Venue
`AaveV3RedemptionAdapter`	Aave v3 deposit receipts, such as aUSDC	The Aave pool's `withdraw`
`ERC4626RedemptionAdapter`	Any ERC-4626 vault share	The vault's `redeem`
`Rebasing4626RedemptionAdapter`	Rebasing wrappers of ERC-4626 shares	The wrapper's `withdraw`

17.7 Module B adapter

A Module B adapter implements `buyProtection(uint256 amount)`. It receives an exact approval for `amount` of the collateral asset and is expected to pull it within the call. The approval is cleared afterwards. The adapter is a per-loan contract named in the terms.

The reference implementation is `HedgeAccountAdapter`. It is constructed with the collateral asset and a hedge account, both fixed. `buyProtection` pulls the approved amount, measures what actually arrived, forwards that amount to the hedge account, and emits `HedgeFunded`. The purchase of

protection itself happens off chain, at a venue the hedge account operates. The adapter holds nothing between calls and has no owner.

17.8 External dependencies at a glance

Dependency	Interface used	Role
Verifier	`verifyTransactionWithTranches` (up to 8 outputs of one transaction in one call), `calculateTxId`, `getLatestHeight`, `minConfirmations`, `quoteVerificationFee`	Verifies funding proofs and top-up proofs
Settler	`hashIntent`, `openBuyEscrowFor`, `commitments`, `refundExpired`, `cancelBuyEscrow`; the loan contract answers the settler's EIP-1271 signature check for its own pot orders	Fills Mega-Pot orders with Bitcoin delivered to the borrower
Oracle adapter	`resolveRate`	Values Bitcoin in the loan asset for the rebate and for Module A
Redemption adapters	`redeem`	Convert a yield-bearing collateral asset to the loan asset before a pot is posted; fixed at deployment
Hedge adapter	`buyProtection`	Receives Module B yield skims; named per loan in the term sheet

18. Bitcoin script reference

18.1 What the contract commits to

The loan contract records, for each tranche, the SHA-256 of a witness script, a hash lock and a time lock. It builds the P2WSH output script from the script hash when it verifies a proof. It never constructs, parses or validates the script itself. The script is built by the borrower's wallet and inspected by both parties before the terms are signed. The reference form below is the construction the system is designed around, and the spending rules of the rest of this document assume it.

18.2 The reference witness script

In the reference form, every collateral tranche uses the same script with four values filled in: the hash lock `H`, the borrower's compressed public key, the time lock, and the lender's compressed public key. All data pushes are of fixed size, so the script is 119 bytes and each value sits at a fixed offset.

Offset	Bytes	Opcode or data	Meaning
0	`63`	`OP_IF`	Branch on the last witness element
1	`82`	`OP_SIZE`	Push the length of the preimage
2 to 3	`01 20`	push 1 byte: `0x20`	32
4	`88`	`OP_EQUALVERIFY`	The preimage must be exactly 32 bytes
5	`a8`	`OP_SHA256`	Hash the preimage
6	`20`	push 32 bytes	
7 to 38	`<hashlock>`		`H`
39	`88`	`OP_EQUALVERIFY`	The hash must equal `H`
40	`21`	push 33 bytes	
41 to 73	`<borrowerPubKey>`		The borrower's key
74	`ac`	`OP_CHECKSIG`	The borrower's signature
75	`67`	`OP_ELSE`	The lender's branch
76	`04`	push 4 bytes	

Offset	Bytes	Opcode or data	Meaning
77 to 80	`<btcTimelock>`	little-endian	Time lock as a script number
81	`b1`	`OP_CHECKLOCKTIMEVERIFY`	The spending transaction must be locked to at least this time
82	`75`	`OP_DROP`	
83	`21`	push 33 bytes	
84 to 116	`<lenderPubKey>`		The lender's key
117	`ac`	`OP_CHECKSIG`	The lender's signature
118	`68`	`OP_ENDIF`	

The first branch pays the borrower on presentation of the secret. The second pays the lender after the time lock. The roles are the reverse of a Maxy Mint deposit. There, the secret holder is the party that takes the funds. Here, the secret holder is the lender, and presenting the secret is what returns the funds to the borrower.

Lock time encoding. `btcTimelock` is a Unix timestamp. It is at least 500,000,000, so Bitcoin interprets it as a time rather than a block height. Until January 2038 it fits in four bytes with the top bit clear, so the minimal encoding is a four-byte little-endian push. The value 1,760,000,000 encodes as `04 00 78 e7 68`.

18.3 The address

The tranche's address is the pay-to-witness-script-hash form of the script:

```
scriptPubKey = 00 20 <SHA256(witnessScript)>     34 bytes
```

For display, witness version 0 and the 32-byte program are encoded as a bech32 address with the `bc` prefix. The result is 62 characters beginning `bc1q`. The term sheet's `scriptHash` is the 32-byte program, the view `trancheScriptPubKeys` returns the 34-byte output script, and `LoanCommitted` carries every tranche's `scriptHash`.

18.4 Spending a tranche

Both spends are SegWit version 0 inputs. The signature covers the BIP-143 digest with the witness script as the script code and uses `SIGHASH_ALL`.

	Reclaim (borrower)	Sweep (lender)
Witness, bottom to top	`<signature>`, `<S>` (32 bytes), `01`, `<witnessScript>`	`<signature>`, `<empty>`, `<witnessScript>`
Signing key	`borrowerPubKey`	`lenderPubKey`
`nLockTime`	Any	At least `btcTimelock`
Input `nSequence`	Any	Below `0xffffffff` (`0xfffffffe` is conventional)
Valid when	After `S` is public, while the output is unspent	Once the chain's median time past exceeds `nLockTime`
Used after	`SecretRevealed`	`TrancheDefaulted`, or for the interest tranche

The branch selector must be minimal: a single `0x01` byte for the reclaim and an empty byte string for the sweep. The network's MINIMALIF rule rejects any other encoding, including a `0x00` byte.

Size and fees. A reclaim or sweep with one input and one output is approximately 150 virtual bytes: about 41 non-witness bytes for the input, 31 to 43 for the output, 11 of overhead, and a witness of 195 to 230 weight units. The borrower pays the funding and reclaim fees; the lender pays the sweep fees.

18.5 The interest tranche

The interest tranche's time lock is fixed at `verificationDeadline + 10 minutes`, and the lender sweeps it through the time-lock branch once that moment has passed. The term sheet records its hash lock as zero, and the contract applies no hash-lock logic to it. The script the parties use for it is their choice. The reference form with the hash-lock branch omitted gives the lender the interest after the time lock and nothing else. A form that lets the borrower recover the interest if the loan never activates needs an additional branch, which the parties agree between themselves. Whatever the form, both parties inspect it, and its SHA-256 is the tranche's `scriptHash`.

In the loans Maxy offers, the interest tranche uses the interest script of Maxy's template `maxylock-htlc-v1`, given in full in 18.7. It has the additional branch: the lender may spend from `verificationDeadline + 10 minutes`, and the borrower may spend alone from `verificationDeadline + 7 days 10 minutes`. Its first time lock is the tranche's `btcTimelock`, so the contract's rule is met.

18.6 Checks a wallet performs

Before signing and again before funding, a wallet verifies each collateral script in four steps.

1. Compute `SHA256(witnessScript)` and compare it with the tranche's `scriptHash`.
2. Compare bytes 7 to 38 with the tranche's `hashlock`.
3. Decode bytes 77 to 80 as a little-endian number and compare it with `btcTimelock`.
4. Compare bytes 41 to 73 with the borrower's key and bytes 84 to 116 with the lender's key.

The lender's wallet performs the same checks before committing. The borrower's wallet also compares the script hashes in `LoanCommitted` with its own before paying. Any mismatch means the address must not be paid and the terms must not be signed.

The offsets above are those of the reference form. A loan built on Maxy's template is checked against the offsets in 18.7.

18.7 Maxy's template `maxylock-htlc-v1`

This section describes Maxy's script template, used in every loan Maxy offers. It is built on top of the protocol, not required by it: the contract commits only to each script's hash, hash lock and time lock (18.1), and a loan between other parties may use the reference form or any other script both of them accept. The template differs from the reference form in two ways. Both scripts pay to key hashes rather than to bare public keys, so the spending witness carries the public key. And the interest tranche has a borrower recovery branch.

Encoding rules.

Item	Rule
Key hash	`RIPEMD160(SHA256(pubkey))` of a 33-byte compressed secp256k1 key. Uncompressed and x-only keys are rejected. The borrower's and the lender's key hashes must differ.
Time locks	Unix times only, in `[500,000,000, 0x7fffffff]`, always pushed as `04` followed by four little-endian bytes. Block heights are never used.
Branch selector	True is the single byte `01`; false is the empty item.
Address	P2WSH: `scriptPubKey = 00 20 <SHA256(witnessScript)>`, byte-equal to what the loan contract builds from `scriptHash`.
Canonical form	Both scripts are miniscript-canonical, so a descriptor wallet rebuilds the same address and can satisfy both spends.
Amounts	Every tranche output, collateral and interest, is at least 15,000 sats (Maxy's lending policy, 7.6).

Collateral tranche `i` (99 bytes). Miniscript `or_i(and_v(v:sha256(H_i),pkh(B)),and_v(v:after(T_i),pkh(L)))`.

```
63 82 01 20 88 a8 20 <H_i> 88 76 a9 14 <PKH_B> 88 ac 67 04 <LE32(T_i)> b1 69 76 a9 14 <PKH_L> 88 ac 68
```

Offset	Bytes	Opcode or data	Meaning
0	`63`	`OP_IF`	Hash-lock branch; pays the borrower
1 to 4	`82 01 20 88`	`OP_SIZE` 32 `OP_EQUALVERIFY`	The preimage is exactly 32 bytes
5 to 6	`a8 20`	`OP_SHA256`, push 32 bytes	
7 to 38	`<H_i>`		The tranche's `hashlock`
39	`88`	`OP_EQUALVERIFY`	The hash must equal `H_i`
40 to 42	`76 a9 14`	`OP_DUP OP_HASH160`, push 20 bytes	
43 to 62	`<PKH_B>`		The borrower's key hash
63 to 64	`88 ac`	`OP_EQUALVERIFY OP_CHECKSIG`	The borrower's signature
65	`67`	`OP_ELSE`	Time-lock branch; pays the lender
66	`04`	push 4 bytes	
67 to 70	`<LE32(T_i)>`	little-endian	The tranche's `btcTimelock`
71 to 72	`b1 69`	`OP_CHECKLOCKTIMEVERIFY OP_VERIFY`	
73 to 75	`76 a9 14`	`OP_DUP OP_HASH160`, push 20 bytes	
76 to 95	`<PKH_L>`		The lender's key hash for this loan
96 to 97	`88 ac`	`OP_EQUALVERIFY OP_CHECKSIG`	The lender's signature
98	`68`	`OP_ENDIF`	

Interest tranche `N` (67 bytes). Miniscript `or_i(and_v(v:after(V+600),pkh(L)),and_v(v:after(V+600+604800),pkh(B)))`, where `V` is `verificationDeadline`.

```
63 04 <LE32(V+600)> b1 69 76 a9 14 <PKH_L> 88 ac 67 04 <LE32(V+600+604800)> b1 69 76 a9 14
<PKH_B> 88 ac 68
```

Offset	Bytes	Opcode or data	Meaning
0	`63`	`OP_IF`	Lender branch
1	`04`	push 4 bytes	
2 to 5	`<LE32(V+600)>`	little-endian	`verificationDeadline + 10 minutes`; must equal the tranche's `btcTimelock`
6 to 7	`b1 69`	`OP_CHECKLOCKTIMEVERIFY OP_VERIFY`	
8 to 10	`76 a9 14`	`OP_DUP OP_HASH160`, push 20 bytes	
11 to 30	`<PKH_L>`		The lender's key hash
31 to 32	`88 ac`	`OP_EQUALVERIFY OP_CHECKSIG`	The lender's signature
33	`67`	`OP_ELSE`	Borrower recovery branch
34	`04`	push 4 bytes	
35 to 38	`<LE32(V+600+604800)>`	little-endian	`verificationDeadline + 7 days 10 minutes`
39 to 40	`b1 69`	`OP_CHECKLOCKTIMEVERIFY OP_VERIFY`	
41 to 43	`76 a9 14`	`OP_DUP OP_HASH160`, push 20 bytes	
44 to 63	`<PKH_B>`		The borrower's key hash
64 to 65	`88 ac`	`OP_EQUALVERIFY OP_CHECKSIG`	The borrower's signature
66	`68`	`OP_ENDIF`	

The recovery delay of 604,800 seconds (seven days) is part of the template's identity: a script with any other delay is not `maxylock-htlc-v1`. Note that in the interest script the lender is the `OP_IF` branch, selected with `01`, and the borrower is the `OP_ELSE` branch, selected with the empty item; in the collateral script it is the other way round.

Spends. Every spend is a version 2 transaction, and every HTLC input uses `nSequence` `0xfffffffd`, which signals replace-by-fee and keeps the time lock enforced. Witness stacks are listed bottom to top.

Spend	Signer	Witness	`nLockTime`	Valid when
Collateral reclaim	Borrower	`<signature> <pubkey_B> <S_i> 01 <witnessScript>`	0	Once `S_i` is known; confirm before `T_i`
Collateral sweep	Lender	`<signature> <pubkey_L> <empty> <witnessScript>`	`T_i`	Median time past exceeds `T_i`
Interest sweep	Lender	`<signature> <pubkey_L> 01 <witnessScript>`	`V + 600`	Median time past exceeds `V + 600`
Interest recovery	Borrower	`<signature> <pubkey_B> <empty> <witnessScript>`	`V + 600 + 604800`	Median time past exceeds `V + 600 + 604800`

After `V + 600 + 604800`, both interest branches are open and the Bitcoin network accepts whichever spend confirms first. Which party takes the interest of a loan that never activated is therefore decided by Maxy's policy of not sweeping it (7.3), not by the script.

Checks a wallet performs. For a collateral script: the length is 99 bytes; `SHA256(witnessScript)` equals the tranche's `scriptHash`; bytes 7 to 38 equal `hashlock`; bytes 67 to 70, read as a little-endian number, equal `btcTimelock`; bytes 43 to 62 are the hash of the borrower's own key, and bytes 76 to 95 are the lender's key hash for this loan. For the interest script: the length is 67 bytes; its SHA-256 equals the interest tranche's `scriptHash`; bytes 2 to 5 equal `verificationDeadline + 600`; bytes 35 to 38 equal that value plus 604,800; bytes 44 to 63 are the borrower's key hash and bytes 11 to 30 the lender's. A wallet may instead rebuild each script from these values and compare the bytes, which covers every check at once.

Test vectors. With the borrower's key `sha256("borrower")`, the lender's key `sha256("lender")`, `H_0 = SHA256(SHA256("secret-0"))`, `T_0 = 1,800,000,000` and `V = 1,789,639,404`:

	Value
`PKH_B`	`d6662c380b81c5583ef7e9811b09f29451e343c1`

	Value
`PKH_L`	`e51268abee9468db6c34cc97d765b1fedb74bb51`
`H_0`	`45da2f299ea5c009a54395815c8099d904a700178c3f828d93d0b92d7a24d25b`
Collateral script	`6382012088a82045da2f299ea5c009a54395815c8099d904a700178c3f828d93d0b92d7a24d25b8876a914d6662c380b81c5583ef7e9811b09f29451e343c188ac670400d2496bb16976a914e51268abee9468db6c34cc97d765b1fedb74bb5188ac68`
Collateral address	`bc1qn2eu98z9ycra6feyefqjqg0u7pxryqq282zgfqt4f7rq35y3j3ns8nk8a6`
Interest script	`630444bdab6ab16976a914e51268abee9468db6c34cc97d765b1fedb74bb5188ac6704c4f7b46ab16976a914d6662c380b81c5583ef7e9811b09f29451e343c188ac68`
Interest address	`bc1q6fw8xuuwzt7rydjv2mf6x3uvmqhkjgq0my0n76cacnfwfqzf7kgqlrh2pj`

The keys in the vectors are public test values and must never hold funds.

Keys. Maxy derives a fresh lender key for every loan (14.1), so `PKH_L` differs from loan to loan. The borrower's key is the key of the borrower's Bitcoin payment address, which must be a native SegWit (P2WPKH) or nested SegWit (P2SH-P2WPKH) address.

19. Deployment and parameters

19.1 Networks

Network	Chain ID	Role
Bitcoin mainnet		Collateral HTLCs, interest sweeps, liquidation deliveries
Base	8453	Loan contract, extensions, verifier, settler, oracle adapter
Robinhood Chain	4663	Loan contract, extensions, verifier, settler, oracle adapter

19.2 Live deployments

Chain	Component	Address
Base (8453)	Loan contract	0x74E7064c386Cb4c852c9B72d54182a60872ea93A
Base (8453)	Extensions	0xFe8992261FC141B49f9F679Acdc94b54168e1EaE
Base (8453)	Verifier	0x8dFED41f68b3749a06a0A9451E6d1701A9C75F9f
Base (8453)	Loan asset in the loans Maxy offers	USDC (6 decimals), 0x833589fCD6eDb6E08f4c7C32D4f71b54bdA02913
Robinhood Chain (4663)	Loan contract	0xCEf3C36A332043D8840277197D7f6B0651FBD0B8
Robinhood Chain (4663)	Extensions	0x519FeF5f9c6b88249340055A71fBdc09ce2EDdfb
Robinhood Chain (4663)	Verifier	0xDbA3FaD120aE7bBBD42a39eAB65D2A41E030e29A
Robinhood Chain (4663)	Loan asset in the loans Maxy offers	USDG (Global Dollar, 6 decimals), 0x5fc5360D0400a0Fd4f2af552ADD042D716F1d168

The verifier on Robinhood Chain is the same contract that Maxy Mint uses. Each deployment's settler, oracle adapter, pauser and Bitcoin feed identifier are read from the loan contract's `SETTLER()`, `ORACLE_ADAPTER()`, `PAUSER()` and `BTC_FEED_ID()`. The redemption routes are fixed in the constructor, are not readable from the contract, and are published with the deployment record. The

extensions contract is an implementation detail of the loan contract and is never called directly; all calls go to the loan contract's address. The deployed contracts come from the source tree covered by Maxy's internal engineering review of 16 September 2026 (607 tests, including fork tests against live verifier and settler deployments). No independent third-party audit has been published yet. A reviewer should compare against the verified on-chain source before relying on a line-by-line match with any copy of the source.

19.3 Test deployment

The reference test environment is Base Sepolia (chain ID 84532), wired to the live shared verifier, settler and oracle adapter on that network. Its verifier tracks the Bitcoin mainnet header chain.

Component	Address
Verifier	0x16707BF8C9411fc7b8111969A8FD7FE0a109fd1F
Settler	0xb8E451b23B80e46B9236Dc288558d42E2Bb91bbf
Oracle adapter	0x169d64B531816d66e653EcAb7B8AEd778AcaB3d4
Pyth BTC/USD feed	0xe62df6c8b4a85fe1a67db44dc12de5db330f7ac66b72dc658afedf0f4a415b43
Settler governance	0x29DD80da5870c1052d59BC436e4215B571721Ad6

The loan contract and extensions are deployed to this environment with `script/DeployMaxyLockRealBaseSepolia.s.sol`. A separate script, `DeployMaxyLockBaseSepolia.s.sol`, deploys the stack against mock infrastructure for lifecycle demonstrations, since the real verifier accepts only real Bitcoin proofs.

19.4 Constructor parameters

The extensions contract is deployed first, with the verifier and the settler. The loan contract is deployed second, with the following parameters, and its constructor reverts with `ExtensionsMismatch` if the extensions contract was built with a different verifier or settler.

Parameter	Meaning
`verifier_`	The verifier
`settler_`	The settler
`pauser_`	The address that may pause commits
`oracleAdapter_`	The oracle adapter

Parameter	Meaning
`btcFeedId_`	The Pyth feed identifier for BTC/USD
`extensions_`	The extensions contract
`routes_`	The redemption routes, as `{asset, underlying, adapter}` triples; may be empty

Every address must be non-zero and every route must name a distinct asset. The routes are written once and cannot be changed. A deployment that needs a new route is a new deployment. The contracts are not upgradeable.

19.5 Parameters

Parameter	Value
Collateral tranches	1 to 7; at most 255 tranches in total with top-ups
`requiredConfirmations`	From the verifier's floor (at least 1) to 8
`revealWindow`	At least 1 day; 7 days in the examples
`REVEAL_SAFETY_MARGIN`	2 days
`INTEREST_SWEEP_DELAY`	10 minutes
Challenge window minimum	`requiredConfirmations × 20 minutes + 1 hour`
`ghostPayoutBps`	10,000 to 15,000
`bountyBps`	0 to 1,000
`healthThresholdBps`	0 to 10,000
Mega-Pot `timeoutBlocks`	`requiredConfirmations + 10` to 4,320
Mega-Pot filler bond	5%
Mega-Pot order life	30 days
Module B skim interval	1 day
Collateral pull tolerance	10 base units
EIP-712 domain	`MaxyLock` version `1`

19.6 Steps after deployment

1. Confirm the verifier's header chain is current, and that a header relayer will keep it so.
2. For every asset a Mega-Pot may hold, the settler's governance calls `setSettlementAssetAllowed(asset, true)` on the settler. Until this is done, a liquidation in that asset cannot be posted.
3. Optionally, register the loan contract as a fee-exempt caller of the verifier.
4. Run one canary loan end to end: commit, fund, prove, repay, reveal, claim. The contract accepts commits from the moment it is deployed, so the canary should run before the addresses are published.

19.7 Build settings

Solidity `0.8.33` with `via_ir = true` and the optimizer at 200 runs, per the repository's `foundry.toml`; the sources declare `pragma solidity ^0.8.13`. OpenZeppelin Contracts 5 provides `ReentrancyGuard`, `SafeERC20`, `ECDSA`, `MessageHashUtils` and `Math`. Both contracts inherit the reentrancy guard through the shared storage base, so the guard occupies the same slot in both, and a guarded function in the extensions contract reads the loan contract's guard when it runs under `delegatecall`. The loan contract's compiled size is within a few hundred bytes of the EVM's 24,576-byte contract size limit, which is the reason for the split. Neither contract is upgradeable.