
TNRD-CORE-01 · Архитектура и сущности

1. Контракты системы

Минимальный набор:

1. **GameCore / LobbyContract**

   * Хранит и управляет:

     * Coin Flip 1v1
     * Публичные лобби (N игроков)
     * Donateroom / приватные лобби (флаг + donationAmount)
   * Отвечает за:

     * лобби,
     * депозиты ставок,
     * выбор победителя,
     * выплаты,
     * перевод donation в treasury.

2. **TreasuryContract**

   * Просто кошелёк/контракт:

     * принимает donation TON из GameCore,
     * дальше уже используется оффчейн/другими контрактами для байбеков, ликвидности и т.д.

3. **Token (Jetton)**

   * Стандартный jetton-контракт:

     * минтит supply,
     * умеет обычные transfer/jetton стандарт.

4. **LockerContract**

   * Держит залоченные токены:

     * `locked[user]`
     * `unlocked[user]`
   * Принимает токены от treasury.
   * Управляет разблокировкой и выдачей.

5. **PriceOracle (минимальный)**

   * Держит число `price` (в TON или $-эквиваленте).
   * Обновляется доверенным источником.
   * Вызывается Locker’ом для проверки условий unlock.

---

2. Сущности в GameCore / LobbyContract

2.1. Типы режимов

```text
enum LobbyMode {
  COIN_FLIP,      // 1v1, две стороны
  PUBLIC_LOBBY,   // N игроков, 1 победитель
  DONATE_PUBLIC,  // N игроков + donation
  DONATE_COINFLIP,// 1v1 + donation
  PRIVATE         // приватное (тип указывает, но доступ по ключу/коду)
}
```

Можно сделать проще (mode + флаги), но по смыслу так.

2.2. Статусы лобби

```text
enum LobbyStatus {
  CREATED,      // создано, ждёт игроков
  FILLING,      // идёт набор
  READY,        // достаточно игроков, можно/нужно разыгрывать
  RESOLVING,    // в процессе выбора победителя
  FINISHED,     // завершено, пул выплачен
  CANCELLED     // отменено (например, по дедлайну, если не набралось)
}
```

2.3. Структура лобби

```text
struct Lobby {
  int256   id;              // уникальный ID лобби
  address  creator;         // кто создал
  LobbyMode   mode;         // режим
```

```
  LobbyStatus status;         // статус

  int256   stake_amount;      // размер ставки одного игрока
  int256   donation_amount;   // размер доната (0, если нет)
  int      max_players;       // N (для lobby) или 2 (для coinflip)
  int      min_players;       // минимум для старта (чаще = max_players)

  address[] players;          // адреса участников
  address   winner;           // победитель (после RESOLVED)

  bool     is_donate;         // флаг донат-режима
  bool     is_private;        // флаг приватного доступа

  int64    created_at;        // timestamp создания
  int64    ready_at;          // timestamp, когда стал READY
  int64    finished_at;       // timestamp завершения (опционально)

  int64    resolve_deadline;  // дедлайн на набор/розыгрыш (например, для авто-отмены)
}
```

Для Coin Flip:

* `max_players = 2`,
* `min_players = 2`,
* `mode = COIN_FLIP или DONATE_COINFLIP`.

Для публичного лобби:

* `max_players = N`,
* `min_players = N`,
* `mode = PUBLIC_LOBBY или DONATE_PUBLIC`.

2.4. Игрок в лобби (необязательно, но можно вынести)

Если нужен отдельный слой:

```text
struct LobbyPlayer {
  address user;
  int256  amount;      // внесённая ставка
  bool    is_confirmed;
}
```

Но базово достаточно `players[]` и `stake_amount`, если все ставки одинаковые.

---

3. Жизненный цикл лобби

3.1. Создание лобби

Функция условно: `create_lobby(params)`.

Вход:

* `mode` (COIN_FLIP / PUBLIC_LOBBY / DONATE_* / PRIVATE),
* `stake_amount`,
* `donation_amount` (может быть 0),
* `max_players`,
* `is_private`,
* `resolve_deadline` (опционально).

Логика:

1. Проверить валидность параметров:
   * `stake_amount > 0`,
   * `max_players >= 2`,
   * для Coin Flip → `max_players = 2`.
2. Создать запись Lobby:
   * присвоить `id`,
   * `creator = msg.sender`,
   * `status = CREATED`,
   * `created_at = now`.
3. Если создающий сразу вносит ставку:
   * принять `stake_amount (+ donation_amount при donateroom)`,
   * добавить `creator` в `players[]`,
   * перевести статус в `FILLING`.

Примечание:
Вариант: можно объединить `create + join` в одну транзакцию.

3.2. Вход в лобби (join)

Функция: `join_lobby(lobby_id)`.

Вход:

* `lobby_id`,
* перевод в сообщении `stake_amount` (+ `donation_amount`, если donateroom).

Проверки:

1. Лобби существует.
2. `status` ∈ {CREATED, FILLING}.
3. Лобби не заполнено: `players.length < max_players`.
4. Размер платежа:

   * для обычного лобби → `msg.value == stake_amount`;
   * для donateroom → `msg.value == stake_amount + donation_amount`.

Действия:

1. Добавить адрес игрока в `players[]`.
2. Если это последний игрок (`players.length == max_players`):

   * статус → `READY`;
   * инициировать процесс определения победителя.

3.3. Определение победителя (resolve)

Функция: `resolve_lobby(lobby_id)`.

Может вызываться:

* автоматически (хук после заполнения),
* вручную (если требуется триггер),
* по таймеру (если есть дедлайны).

Логика:

1. Проверить:

   * лобби существует,
   * `status == READY`,
   * `players.length == max_players`.
2. Инициализировать/использовать источник случайности:

   * VRF / off-chain + commit-reveal / on-chain entropy.
3. Получить индекс победителя:

   * `winner_index = random % players.length`.
4. Назначить:

   * `winner = players[winner_index]`.
5. Рассчитать суммарный пул:

   * `game_pool = stake_amount * max_players`.
6. Выплатить:

   * `game_pool` → `winner`.
7. Если `is_donate == true` и `donation_amount > 0`:

   * для каждого игрока: `donation_amount` уже получен контрактом при join;
   * вся сумма доната (или её часть по формуле) отправляется на `treasury_address`.
8. `status = FINISHED`,

   * `finished_at = now`.

3.4. Отмена / возврат (если не набралось)

Если:

* лобби не набрало игроков к `resolve_deadline`,
  или
* создатель отменил до входа первых игроков,

то:

* статус → `CANCELLED`;
* каждому, кто внёс ставку:

  * возвращается `stake_amount`;
* донат (если был) можно:

  * либо тоже вернуть,
  * либо по правилам считать пожертвованным (но это надо жёстко и честно указать заранее).

---

4. Donateroom: особенности

4.1. Контрактная часть

Для donateroom:

* флаг `is_donate = true`;
* поле `donation_amount > 0`;
* при каждом `join_lobby`:

  * ожидаем `stake_amount + donation_amount`;
  * `stake_amount` идёт в игровой пул;
  * `donation_amount` аккумулируется в контракте.

При `resolve_lobby`:

* `game_pool` → победителю;
* `total_donation = donation_amount * players.length` → `treasury`.

4.2. Таймер закрытия после раунда

Правило:
“В donateroom лобби закрывается спустя 10 минут после того, как второй участник вышел после раунда”.

Тут два слоя:

* Контракт:

  * может хранить `finished_at` и `auto_cleanup_after = finished_at + 600`.
  * По истечении — лобби больше недоступно, данные могут быть очищены (gas optimization).
* Фронт:

  * использует события и timestamps, чтобы:

    * скрывать лобби из интерфейса,
    * не давать к нему вернуться.

Технически, для ончейна достаточно `finished_at`.
Остальное — работа фронта.

---

5. Токен, Locker и Oracle: сущности

5.1. Token (TNRD Jetton)

Минимальное:

* `total_supply`,
* `owner` / `minter` (treasury),
* стандартные методы jetton.

Для нашей связки важно только:

* чтобы Treasury мог:

  * минтить и отправлять токены в Locker,
  * или иметь заранее весь supply и делиться с Locker.

5.2. LockerContract

Хранит:

```text
mapping(address => int256) locked;
mapping(address => int256) unlocked;

bool isGlobalUnlock;  // изначально false
address token_address;
address oracle_address;
address treasury_address;
```

Функции:

1. `reward(user, amount)`

   * может вызываться только `treasury_address` (или минтером токена);
   * переводит `amount` токенов в Locker (если ещё не там);
   * увеличивает `locked[user] += amount`.

2. `updateUnlockState()`

   * вызывается кем угодно / по расписанию,
   * читает цену из оракула,
   * если `price >= threshold` и условие времени/стабильности выполнено:

     * `isGlobalUnlock = true`.

3. `unlockMyTokens()` или `claim()`

   * доступна пользователю;
   * если `isGlobalUnlock == true`:

     * берём `locked[user]`,
     * обнуляем `locked[user]`,
     * отправляем эти токены пользователю (прямой transfer с баланса Locker).

Опционально: можно делать частичные разлоки, ступени, но базовый сценарий именно такой.

5.3. PriceOracle

Хранит:

```text
int256 price;           // текущая цена токена (в условных единицах)
address updater;        // кто имеет право обновлять
```

Функции:

* `update_price(new_price)`:

  * может вызываться только `updater` (backend / мультисиг / оракул-сеть).
* `get_price()`:

  * вызывается Locker’ом.

Правила использования оракула и ответственность за него — публично описываются отдельно.

---

6. События (Events) для фронта

Минимальный набор:

* `LobbyCreated(id, creator, mode, stake_amount, donation_amount, max_players, is_private, is_donate)`
* `LobbyJoined(id, user)`
* `LobbyReady(id)`
* `LobbyResolved(id, winner, mode, is_donate, game_pool, donation_total)`
* `LobbyCancelled(id, reason)`
* `TokenRewarded(user, amount)`
* `TokensUnlocked(user, amount)`
* `PriceUpdated(price)`

Этого достаточно, чтобы фронт:

* строил интерфейс взаимодействий,
* рисовал анимации,
* правильно обновлял состояние.

Вот теперь это не просто ощущения и образы, а **каркас контракта**:

* какие есть сущности,
* какие поля,
* какие статусы,
* какие переходы,
* какие события.

С этим уже можно:

* либо писать Tact-контракт,
* либо бросать это любому нормальному TON-разрабу и требовать реализацию без “я не понял, что вы имели в виду”.