**Lesson 5: Implementing Auction Functions in Smart Contracts**

**Lesson Summary**

* This lesson provides a step-by-step guide to implementing the English auction model in a Tamagotchi auction contract.
* We’ll cover automatic program execution to facilitate the English auction model
* We’ll highlight the contract state and logic required for the auction function
* Expect hands-on development of the auction function solutions

**Lesson Objectives**

By the end of this lesson, you will:

* Comprehend automatic program execution with smart contract automation
* Understand the smart contract state and logic when implementing auction functions
* Understand system signals, gas reservations, and how to add gas reservations to system signals
* Incorporate an auction model to your smart contracts

**Let’s Get Started!**

In this lesson, we'll create a contract where users can put their Tamagotchi up for auction. We'll implement the English auction model.

This type of auction starts with the declaration of the minimum bid, which the Tamagotchi owner sets. After this, the interested bidders start placing their bids in ascending order, i.e., the next bid should be higher than the previous one. This process continues until there is a bid above which any other buyer is not interested in buying Tamagotchi. The highest bid is the selling price of Tamagotchi.

As you might guess, your Tamagotchi contract should be extended with functionality that will make it possible to change ownership (it's exactly your homework from the previous lesson).

During the auction, the auction contract temporarily becomes the owner of Tamagotchi. After the auction ends, the contract appoints the new owner of Tamagotchi - the winner of the auction. If no bids were made, Tamagotchi is returned to the previous owner.

**Automatic program execution with smart contract automation**

Before we start coding the auction smart contract, we'll discuss smart contract automation.

Smart contracts cannot auto-execute. Their code will not run and make state changes on blockchain until triggered by an on-chain transaction. The external transaction serves as a “poke” to wake the smart contract up and initiate its logic. For example, we can start the auction by sending a message to the auction contract.

When the auction time has passed, it's necessary to process the result of the auction. However, the result will not be processed until someone sends an appropriate message to the contract.

In Gear, we solve that problem with delayed messages.

| msg::send\_delayed(program, payload, value, delay) msg::send\_bytes\_delayed(program, payload, value, delay) |
| --- |

The delayed message is executed after the indicated delay. It's quite convenient in our case: we can start the auction by sending a message to the auction contract. After completing all the necessary logic, the auction contract will send a delayed message to itself, which will settle the auction after the indicated time.

So, the ability to send delayed messages allows you to automate the contract execution. The contract can self-execute an unlimited block number provided there’s enough gas for execution. But the execution can be interrupted if the gas runs out.

Gear protocol allows another powerful feature - gas reservation. A developer can reserve gas that can be used to send usual or delayed messages.

To reserve the amount of gas for further usage use the following function:

| let reservation\_id = ReservationId::reserve(RESERVATION\_AMOUNT, TIME)  .expect("reservation across executions"); |
| --- |

That function takes some defined amount of gas from the amount available for this program and reserves it. A reservation gets a unique identifier used by a program to get this reserved gas and use it later.

You also have to indicate the block count within which the reserve must be used. Gas reservation is not free: the reservation for one block costs 100 gas. The reserve function returns *ReservationId*, used for sending a message with that gas. To send a message using the reserved gas:

| msg::send\_from\_reservation(reservation\_id, program, payload, value)  .expect("Failed to send message from reservation"); |
| --- |

If gas is not needed within the reservation period, it can be unreserved and the gas will be returned to the user who made the reservation.

| id.unreserve().expect("unreservation across executions"); |
| --- |

**Understanding contract state and logic**

Let’s start coding the auction contract. First we define the structure of the contract state:

* Tamagotchi\_id - the address of the Tamagotchi contract that is currently on auction;
* status - the status of auction (It can be ReadyToStart or InProcess);
* current\_bid - the current highest bid;
* current\_bidder - the address of the participant who made the highest bid at the moment;
* ended\_at -the time of the end of the auction;
* prev\_tmg\_owner - the previous owner of the Tamagotchi (the auction contract has to store this account in case if no one is participating in the auction and Tamagotchi must be returned to the previous owner).

| static mut AUCTION: Option<Auction> = None;  **#[derive(Default)]** pub struct **Auction** {  Tamagotchi\_id: TamagotchiId,  status: Status,  current\_bid: u128,  current\_bidder: ActorId,  ft\_contract\_id: ActorId,  transaction\_id: TransactionId,  ended\_at: u64,  prev\_tmg\_owner: ActorId, } **#[derive(Debug, PartialEq, Eq)]** pub enum **Status** {  ReadyToStart,  InProcess, }  impl Default for Status {  fn **default**() -> Self {  Self::ReadyToStart  } } |
| --- |

Let's define the messages that the contract will receive:

* *StartAuction {Tamagotchi\_id, minimum\_bid, duration*} - message initiating the start of the auction. The Tamagotchi owner must indicate the address of the Tamagotchi contract, the starting price and the duration of the auction;
* *MakeBid { bid }* - the message from auction participants, to which they must indicate the price (bid) they are ready to pay for Tamagotchi;
* *SettleAuction -* the message that the contract receives after the end of the auction. If there were bids, then the auction contract assigns Tamagotchi to the auction winner. Otherwise, the auction contract assigns Tamagotchi to the previous owner.

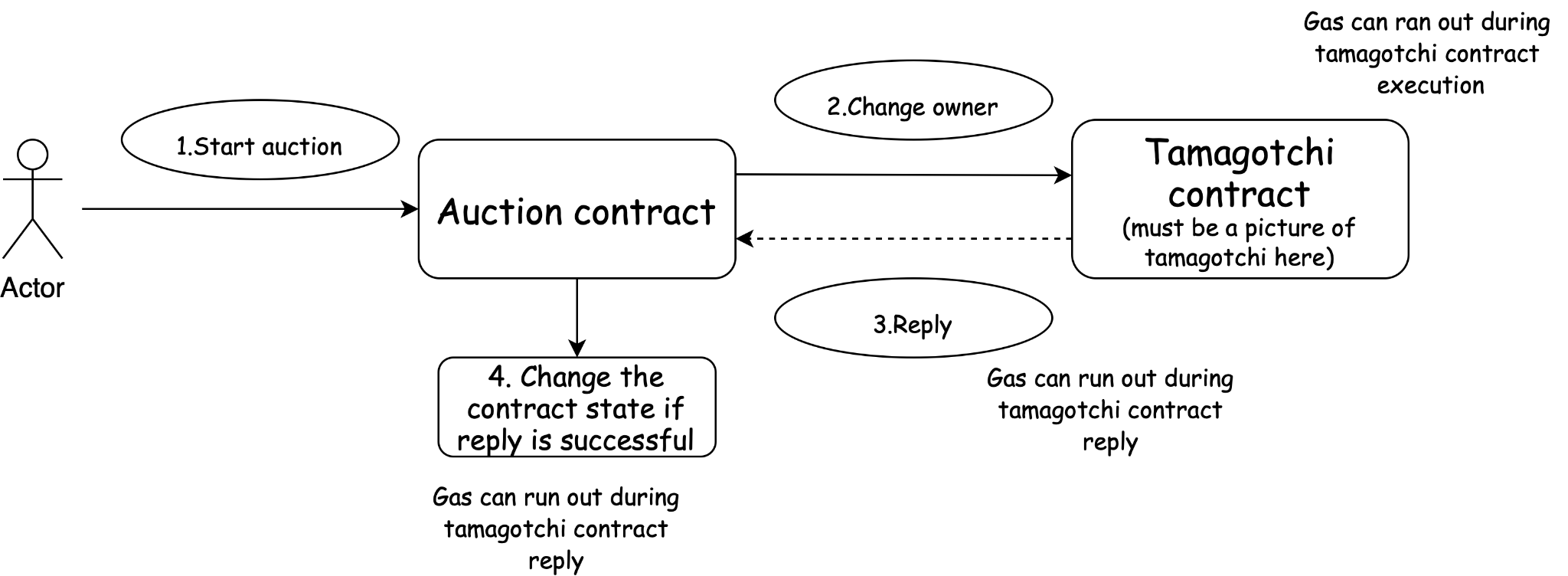
| pub type TamagotchiId = ActorId; pub type Bid = u128;  *#[derive(Encode, Decode)]* pub enum AuctionAction {  StartAuction {  Tamagotchi\_id: TamagotchiId,  minimum\_bid: Bid,  duration: u64,  },  MakeBid {  bid: Bid,  },  SettleAuction, } |
| --- |

As you can guess, the auction contract will interact with the fungible token contract as well as with the Tamagotchi contract. That is, it'll send messages to these contracts and wait for replies. Therefore, the transactions will not be atomic and we’ll have to consider this to maintain the state of 3 contracts consistent.

Let's look at each action of the contract in detail. The action *StartAuction* has to change the owner of Tamagotchi to the auction contract and set the auction parameters.

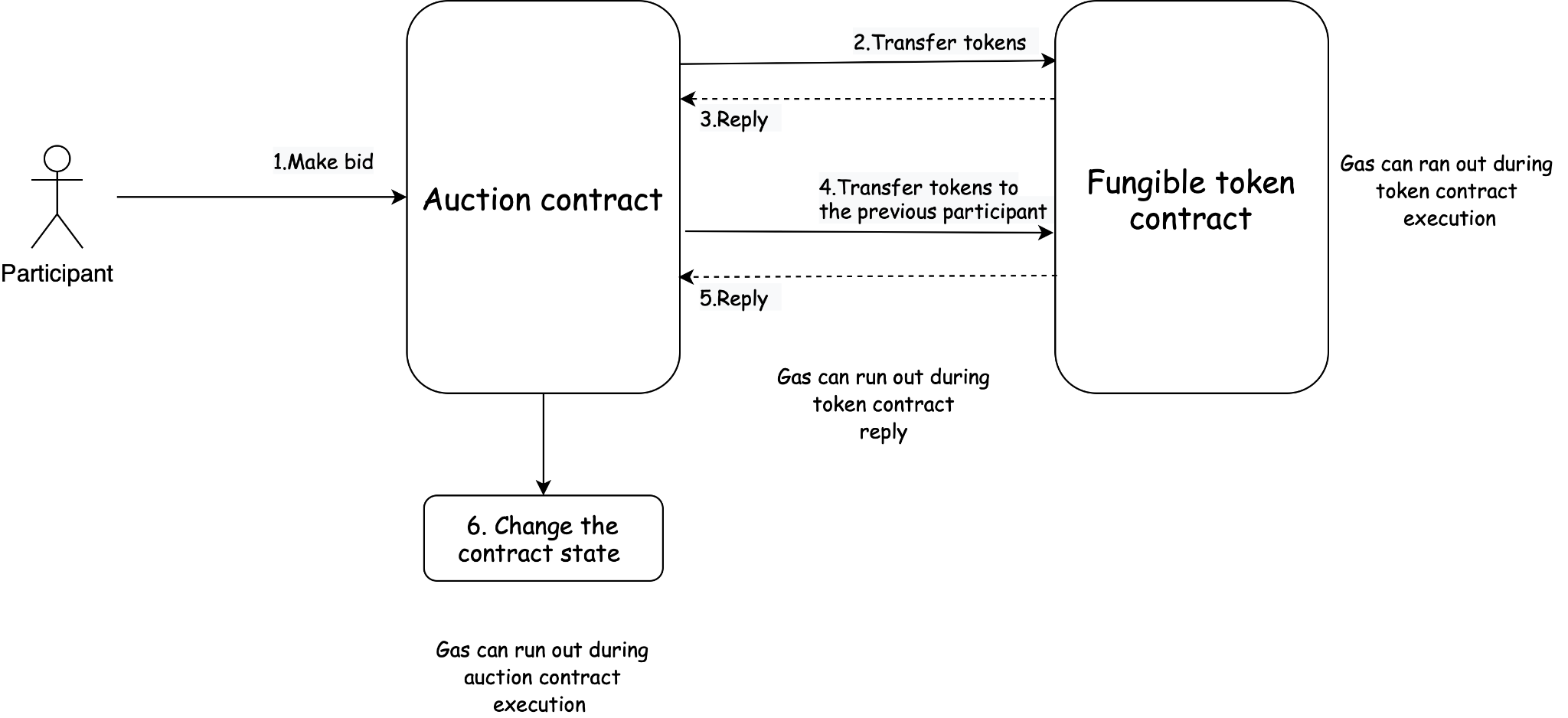
We consider all the possible cases that can lead to state inconsistency. The auction contract sends a message to the Tamagotchi contract and the following cases are possible:

1. The Tamagotchi contract fails during the message execution either from lack of gas or from a logical error. It didn’t save the state and therefore auction and Tamagotchi are in a consistent state, however, the auction contract has no idea what happened in the Tamagotchi contract.
2. The Tamagotchi executes the messages, and saves the state but gas runs out during further operations. Then, the state of the Tamagotchi contract changed, however, this was not reflected in the auction contract.



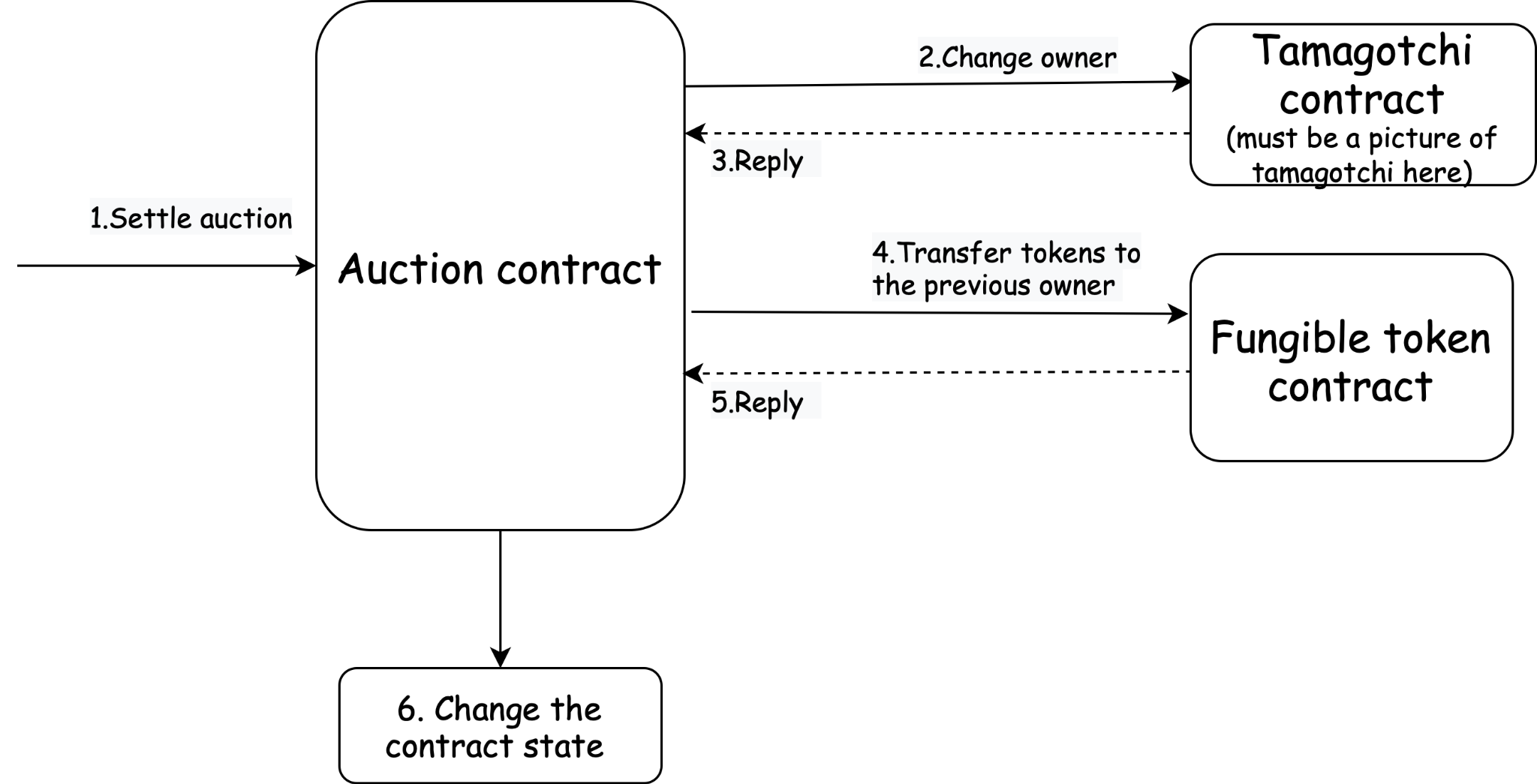
The workflow of *MakeBid* action is as follows:

1. User makes a bid, indicating the number of tokens he would like to pay for Tamagotchi.
2. The contract transfers his tokens to its balance and if that transfer is successful it returns the tokens to the previous bidder. The gas can run out during token contract execution, during a reply to the auction contract or during further execution.



Having received the *SettleAuction* message, the contract performs the following actions:

1. Change the Tamagotchi owner to the auction winner. The gas can run out during the Tamagotchi contract execution, during a reply or during further auction contract execution.
2. Transfer tokens to the previous owner. And again, the gas can run out during the fungible contract execution, during a reply or during further auction contract execution.



So, let’s create the following Enum for tracking transactions:

| *#[derive(Clone)]* enum **Transaction** {  StartAuction {  tamagotchi\_id: ActorId,  bid: Bid,  duration: u64,  },  MakeBid {  transaction\_id: TransactionId,  bidder: ActorId,  bid: u128,  },  SettleAuction {  transaction\_id: TransactionId,  }, } |
| --- |

and add the fields to the contract state:

| **#[derive(Default)]** pub struct Auction {  . . .  transaction: Some<Transaction>,  transaction\_id: TransactionId  . . . } |
| --- |

where the field transaction\_id will be used for tracking the transactions in the fungible token contract.

**Practicing Contract coding**

In this tutorial, instead of using panic!, we'll return Result<AuctionEvent, AuctionError>. Panics use is convenient when the execution of the contract is atomic and there are no asynchronous calls in it.

In our case, there will be quite a few asynchronous messages between which we must carefully track the state of the program. In such a case, using Result enum is the preferred option.

We'll create enums AuctionEvent and AuctionError that we'll expand during the writing of the program.

| *#[derive(Encode, Decode)]* pub enum **AuctionEvent** {  ... }  *#[derive(Encode, Decode)]* pub enum **AuctionError** {  ... } |
| --- |

Accordingly, the main function:

| **#[gstd::async\_main]** async fn **main**() {  let action: AuctionAction = msg::load().expect("Unable to decode `AuctionAction`");  let auction = unsafe { AUCTION.get\_or\_insert(Default::default()) };  let reply = match action {  AuctionAction::StartAuction {  Tamagotchi\_id,  minimum\_bid,  duration,  } => {  auction  .start\_auction(&Tamagotchi\_id, minimum\_bid, duration)  .await  }  AuctionAction::MakeBid { bid } => {  auction.make\_bid(bid).await  }  AuctionAction::SettleAuction => {  auction.settle\_auction().await  }  };  msg::reply(reply, 0).expect("Failed to encode or reply with `Result<MarketEvent, MarketErr>`"); } |
| --- |

Let’s start writing the function start\_auction:

| async fn **start\_auction**(  &mut self,  Tamagotchi\_id: &TamagotchiId,  minimum\_bid: Bid,  duration: u64,  ) -> Result<AuctionEvent, AuctionError> {} |
| --- |

We check that auction is in *ReadyToStart* state:

| if self.status != Status::ReadyToStart {  return Err(AuctionError::WrongState);  } |
| --- |

Then we check if there is a pending transaction. If there is, we:

* Check that it’s the transaction *StartAuction*;
* Check the input arguments for the function. If they differ from those stored in the transaction, the contract replies with the error;
* Get the Tamagotchi owner. If it's already the auction contract, we don’t send the message to the Tamagotchi contract again and just save it in the auction contract. Then we stop the message execution.

| *// Check if there is already a pending transaction*  if let Some(tx) = self.transaction.clone() {  match tx {  Transaction::StartAuction {   tamagotchi: prev\_tmg\_id,  bid,  duration: prev\_duration,  } => {  if \*tamagotchi\_id != prevamagotchi\_id  || bid != minimum\_bid  || prev\_duration != duration  {  return Err(AuctionError::WrongParams);  }   *// get the Tamagotchi owner*  let tmg\_owner =  if let Ok(tmg\_owner) = get\_owner(&self.Tamagotchi\_id).await {  tmg\_owner  } else {  self.transaction = None;  return Err(AuctionError::WrongReceivedMessage);  };   *// if Tamagotchi owner is already the current contract*  *// we just change its state and start the auction*  if tmg\_owner == exec::program\_id() {  self.tamagotchi\_id = \*tamagotchi\_id;  self.status = Status::InProcess;  self.current\_bid = bid;  self.ended\_at = exec::block\_timestamp() + duration;  self.transaction = None;  return Ok(AuctionEvent::AuctionStarted)  };    *// check that owner starts the auction*  if tmg\_owner != msg::source() {  return Err(AuctionError::NotOwner);  }    if change\_owner(&self.Tamagotchi\_id, &exec::program\_id())  .await  .is\_err()  {  self.transaction = None;  return Err(AuctionError::UnableToChangeOwner);  } else {  self.tamagotchi\_id = \*tamagotchi\_id;  self.status = Status::InProcess;  self.current\_bid = bid;  self.prev\_tmg\_owner = tmg\_owner;  self.ended\_at = exec::block\_timestamp() + duration;  self.transaction = None;  msg::send\_delayed(  exec::program\_id(),  AuctionAction::SettleAuction,  0,  duration as u32,  )  .expect("Error in sending a delayed message `AuctionAction::SettleAuction`");  return Ok(AuctionEvent::AuctionStarted);  }  }  \_ => {  return Err(AuctionError::WrongTx);  }  }  } |
| --- |

Where the function for getting owner:

| async fn **get\_owner**(Tamagotchi\_id: &TamagotchiId) -> Result<ActorId, AuctionError> {  let reply = msg::send\_for\_reply\_as(\*Tamagotchi\_id, TmgAction::Owner, 0)  .expect("Error in sending a message `TmgAction::Owner` to Tamagotchi contract")  .await;  match reply {  Ok(TmgEvent::Owner(tmg\_owner)) => {  Ok(tmg\_owner)  },  \_ => Err(AuctionError::WrongReceivedMessage)  } } |
| --- |

And the function for changing owner:

| async fn change\_owner(  Tamagotchi\_id: &TamagotchiId,  new\_owner: &ActorId, ) -> Result<TmgEvent, ContractError> {  msg::send\_for\_reply\_as::<\_, TmgEvent>(  \*Tamagotchi\_id,  TmgAction::ChangeOwner {  new\_owner: \*new\_owner,  },  0,  )  .expect("Error in sending a message `TmgAction::ChangeOwner` to Tamagotchi contract")  .await } |
| --- |

If there is no a pending transaction, the following logic is simple:

| if duration < MIN\_DURATION {  return Err(AuctionError::WrongDuration);  }    self.transaction = Some(Transaction::StartAuction {  tamagotchi\_id: \*tamagotchi\_id,  bid: minimum\_bid,  duration,  });   let tmg\_owner = if let Ok(tmg\_owner) = get\_owner(&self.Tamagotchi\_id).await {  tmg\_owner  } else {  self.transaction = None;  return Err(AuctionError::WrongReceivedMessage);  };    *// check that owner starts the auction*  if tmg\_owner != msg::source() {  self.transaction = None;  return Err(AuctionError::NotOwner);  }   if change\_owner(&self.Tamagotchi\_id, &exec::program\_id())  .await  .is\_err()  {  self.transaction = None;  Err(AuctionError::UnableToChangeOwner)  } else {  self.tamagotchi\_id = \*tamagotchi\_id;  self.status = Status::InProcess;  self.current\_bid = minimum\_bid;  self.prev\_tmg\_owner = tmg\_owner;  self.ended\_at = exec::block\_timestamp() + duration;  self.transaction = None;  msg::send\_delayed(  exec::program\_id(),  AuctionAction::SettleAuction,  0,  duration as u32,  )  .expect("Error in sending a delayed message `AuctionAction::SettleAuction`");  Ok(AuctionEvent::AuctionStarted)  } |
| --- |

As you can see, the code is repeated when we continue the previous transaction or execute the current one.

Let’s write the function complete\_tx:

| async fn **complete\_tx**(&mut self, tx: Transaction) -> Result<AuctionEvent, AuctionError> {  match tx {  Transaction::StartAuction { bid, duration } => {  let tmg\_owner = if let Ok(tmg\_owner) = get\_owner(&self.Tamagotchi\_id).await {  tmg\_owner  } else {  self.transaction = None;  return Err(AuctionError::WrongReceivedMessage);  };  *// if Tamagotchi owner is already the current contract*  *// we just change its state and start the auction*  if tmg\_owner == exec::program\_id()   self.status = Status::InProcess;  self.current\_bid = bid;  self.ended\_at = exec::block\_timestamp() + duration;  self.transaction = None;  return Ok(AuctionEvent::AuctionStarted);  };   *// check that owner starts the auction*  if tmg\_owner != msg::source() {  return Err(AuctionError::NotOwner);  }   if change\_owner(&self.Tamagotchi\_id, &exec::program\_id())  .await  .is\_err()  {  self.transaction = None;  Err(AuctionError::UnableToChangeOwner)  } else {  self.status = Status::InProcess;  self.current\_bid = bid;  self.prev\_tmg\_owner = tmg\_owner;  self.ended\_at = exec::block\_timestamp() + duration;  self.transaction = None;  msg::send\_delayed(  exec::program\_id(),  AuctionAction::SettleAuction,  0,  duration as u32,  )  .expect("Error in sending a delayed message `AuctionAction::SettleAuction`");  Ok(AuctionEvent::AuctionStarted)  }  }  Transaction::MakeBid {  transaction\_id,  bidder,  bid,  } => Ok(AuctionEvent::BidMade { bid }),  Transaction::SettleAuction { transaction\_id } => Ok(AuctionEvent::AuctionSettled),  }  } |
| --- |

Then the function start\_auctionwill be rewritten as follows

| async fn **start\_auction**(  &mut self,  Tamagotchi\_id: &TamagotchiId,  minimum\_bid: Bid,  duration: u64,  ) -> Result<AuctionEvent, AuctionError> {  if self.status != Status::ReadyToStart {  return Err(AuctionError::WrongState);  }   *// Check if there is already a pending transaction*  if let Some(tx) = self.transaction.clone() {  match tx {  Transaction::StartAuction {   tamagotchi: prev\_tmg\_id,  bid,  duration: prev\_duration,  } => {  if \*tamagotchi\_id != prevamagotchi\_id  || bid != minimum\_bid  || prev\_duration != duration  {  return Err(AuctionError::WrongParams);  }  return self  .complete\_tx(Transaction::StartAuction { bid, duration })  . await;  }  \_ => {  return Err(AuctionError::WrongTx);  }  }  }   if duration < MIN\_DURATION {  return Err(AuctionError::WrongDuration);  }  let tx = Transaction::StartAuction {  tamagotchi\_id: \*tamagotchi\_id,  bid: minimum\_bid,  duration,  };  self.transaction = Some(tx.clone());   self.complete\_tx(tx).await  } |
| --- |

Great, we're done with this function and now we'll start writing the function for making bids:

1. First, we check if there is no pending transaction *MakeBid*;
2. Next, we check the input arguments. If they differ from those saved in transactions, we complete the previous transaction and execute the current one. If they are the same, we complete the pending transaction and stop the function execution.
3. If there is no pending transaction, we execute the current transaction.

| async fn **make\_bid**(&mut self, bid: u128) -> Result<AuctionEvent, AuctionError> {  if self.status != Status::InProcess {  return Err(AuctionError::WrongState);  }   *// Check if there is already a pending transaction*  if let Some(tx) = self.transaction.clone() {  match tx {  Transaction::MakeBid {  transaction\_id,  bidder,  bid: prev\_bid,  } => {  let result = self  .complete\_tx(tx).await;  if bidder == msg::source() && bid == prev\_bid {  return result;  }  }  \_ => {  return Err(AuctionError::WrongTx);  }  }  }   if bid <= self.current\_bid {  return Err(AuctionError::WrongBid);  }   let transaction\_id = self.transaction\_id;  let bidder = msg::source();  *// We reserve 2 transaction ids since there will be 2 messages to the token contract*  self.transaction\_id = self.transaction\_id.wrapping\_add(2);  let tx = Transaction::MakeBid {  transaction\_id,  bidder,  bid,  };  self.transaction = Some(tx.clone());  self.complete\_tx(tx).await  } |
| --- |

Let’s expand the function *complete\_tx*:

| async fn **complete\_tx**(&mut self, tx: Transaction) -> Result<AuctionEvent, AuctionError> {  match tx {  ...  Transaction::MakeBid {  transaction\_id,  bidder,  bid,  } => {  if transfer\_tokens(  transaction\_id,  &self.ft\_contract\_id,  &bidder,  &exec::program\_id(),  bid,  )  .await  .is\_err()  {  self.transaction = None;  return Err(AuctionError::UnableToTransferTokens);  }   *// if it's not the first bet*  *// we have to return the tokens to the previous bidder*  *// since the tokens are on the auction contract*  *// the transaction can fail only due to lack of gas*  *// it's necessary to rerun the transaction*  if !self.current\_bidder.is\_zero()  && transfer\_tokens(  transaction\_id + 1,  &self.ft\_contract\_id,  &exec::program\_id(),  &self.current\_bidder,  self.current\_bid,  )  .await  .is\_err()  {  return Err(AuctionError::RerunTransaction);  }   self.current\_bid = bid;  self.current\_bidder = bidder;  Ok(AuctionEvent::BidMade { bid })  }  ...  }  } |
| --- |

So, the next step is writing the function *settle\_auction.* And again, here, we check the presence of a pending transaction.

But it's possible that there is a transaction *MakeBid* left from the state when users were making bids.

In this case, we must first complete this transaction and then execute the transaction *SettleAuction*:

| async fn **settle\_auction**(&mut self) -> Result<AuctionEvent, AuctionError> {  if self.ended\_at < exec::block\_timestamp() {  return Err(AuctionError::WrongState);  }   *// it's possible that there is a pending transaction `MakeBid`*  if let Some(tx) = self.transaction.clone() {  match tx {  Transaction::MakeBid { .. } => {  self.complete\_tx(tx).await;  }  Transaction::SettleAuction { .. } => {  return self.complete\_tx(tx).await;  }  \_ => {  return Err(AuctionError::WrongTx);  }  }  }   let transaction\_id = self.transaction\_id;  self.transaction\_id = self.transaction\_id.wrapping\_add(1);  let tx = Transaction::SettleAuction { transaction\_id };  self.transaction = Some(tx.clone());  return self.complete\_tx(tx).await;  } |
| --- |

And accordingly, the *complete\_tx* function:

| async fn **complete\_tx**(&mut self, tx: Transaction) -> Result<AuctionEvent, AuctionError> {  match tx {  ...  Transaction::SettleAuction { transaction\_id } => {  let tmg\_owner = if let Ok(tmg\_owner) = get\_owner(&self.Tamagotchi\_id).await  {  tmg\_owner  } else {  return Err(AuctionError::WrongReceivedMessage);  };  if tmg\_owner == exec::program\_id() {  if self.current\_bidder.is\_zero() {  if change\_owner(&self.Tamagotchi\_id, &self.prev\_tmg\_owner)  .await  .is\_err()  {  return Err(AuctionError::RerunTransaction);  };  } else {  if transfer\_tokens(  transaction\_id,  &self.ft\_contract\_id,  &exec::program\_id(),  &self.prev\_tmg\_owner,  self.current\_bid,  )  .await  .is\_err()  {  return Err(AuctionError::RerunTransaction);  };   if change\_owner(&self.Tamagotchi\_id, &self.current\_bidder)  .await  .is\_err()  {  return Err(AuctionError::RerunTransaction);  };  }  }  self.transaction = None;  self.prev\_tmg\_owner = ActorId::zero();  self.current\_bidder = ActorId::zero();  self.status = Status::ReadyToStart;  self.ended\_at = 0;  self.tamagotchi\_id = ActorId::zero();   Ok(AuctionEvent::AuctionSettled)  } |
| --- |

**System signals**

In Gear programs, there are three common entry points: **init**, **handle**, and **handle\_reply**. The Gear Protocol also introduces the **handle\_signa**l entry point, which enables the system to communicate with programs and notify them (signal) of events related to the program's messages. Only the system (Gear node runtime) can send signal messages to a program.

The system sends messages to a program if some errors during program execution occurred. For example, a program can panic or run out of gas.

The gstd library provides a separate function for reserving gas specifically for system signal messages.

| exec::system\_reserve\_gas(1\_000\_000\_000).expect("Error during system gas reservation"); |
| --- |

This cannot be used for sending other regular cross-actor messages. While signals can be used for inter-actor communication, they are not suitable for sending regular cross-actor messages.

Signal messages use gas that is specifically reserved for them. If no gas has been reserved for system messages, they will be skipped, and the program will not receive them.

If gas has been reserved but no system messages occur during the current execution, then this gas returns back from where it was taken.

If your program uses asynchronous messages and the #[gstd::async\_main] macro is used to expand the handle\_signal entrypoint, it will free up resources occupied by the program.

In Gear, using custom async logic involves storing Futures in the program's memory. The execution context of these Futures can occupy a significant amount of memory, especially when dealing with many Futures.

It's important to note that if a program sends a message and waits for a reply, but the reply is unable to be received, it could be due to a lack of gas. For example, if the initial message in the waitlist runs out of gas or the gas amount is insufficient, the reply cannot be received.

To handle signals in your program, you can define your own my\_handle\_signal entrypoint and write custom logic for it. In the next section, we'll see an example of how to write this function in the auction contract.

**Adding gas reservation and system signals**

Let’s add the entrypoint my\_handle\_signal to our auction contract. That function will confirm whether there is a pending transaction. If there is, the function takes the gas reserved in advance and sends a message CompleteTx using that gas.

First, we have to add an action for gas reservation by expanding the enums AuctionAction and AuctionEvent:

| *#[derive(Encode, Decode)]* pub enum **AuctionAction** {  ...   MakeReservation, }  *#[derive(Encode, Decode)]* pub enum **AuctionEvent** {  ...  ReservationMade, } |
| --- |

We also need to add a field reservations to the Auction struct to save the ids of gas reservations:

| **#[derive(Default)]** pub struct **Auction** {  ...   reservations: Vec<ReservationId>, } |
| --- |

Next, we'll define the method make\_reservation for the Auction struct:

| impl Auction {  ...  fn **make\_reservation**(&mut self) -> Result<AuctionEvent, AuctionError> {  let reservation\_id = ReservationId::reserve(RESERVATION\_AMOUNT, RESERVATION\_DURATION)  .expect("reservation across executions");  self.reservations.push(reservation\_id);  Ok(AuctionEvent::ReservationMade)  } } |
| --- |

Here, RESERVATION\_AMOUNT and RESERVATION\_DURATION are constants defined as follows:

| const RESERVATION\_AMOUNT: u64 = 50\_000\_000\_000; const RESERVATION\_DURATION: u32 = 86400; |
| --- |

Then we have to add the action for completing the transaction.

We’ll add the action to enum AuctionAction using:

| *#[derive(Encode, Decode)]* pub enum **AuctionAction** {  ...  CompleteTx(Transaction), } |
| --- |

Where the transaction is an enum that we have defined before:

| *#[derive(Clone, Encode, Decode, PartialEq, Eq)]* pub enum **Transaction** {  StartAuction {  tamagotchi: ActorId,  bid: Bid,  duration: u64,  },  MakeBid {  transaction\_id: TransactionId,  bidder: ActorId,  bid: u128,  },  SettleAuction {  transaction\_id: TransactionId,  }, } |
| --- |

We’ll also extend the entrypoint main():

| **#[gstd::async\_main]** async fn **main**() {  ...  AuctionAction::MakeReservation => auction.make\_reservation(),  AuctionAction::CompleteTx(tx) => {  let result = if let Some(\_tx) = &auction.transaction {  if tx == \_tx.clone() {  auction.complete\_tx(tx).await  } else {  Err(AuctionError::WrongTx)  }  } else {  Err(AuctionError::NoTx)  };  result   },  }; } |
| --- |

Let’s write the function my\_handle\_signal. This function is responsible for checking if there is a pending transaction and if there is a gas reservation available. If so, it sends a message CompleteTx using that gas.

| **#[no\_mangle]** extern "C" fn **my\_handle\_signal**() {  let auction = unsafe { AUCTION.get\_or\_insert(Default::default()) };  if let Some(tx) = &auction.transaction {  let reservation\_id = if !auction.reservations.is\_empty() {  auction.reservations.remove(0)  } else {  return;  };  msg::send\_from\_reservation(  reservation\_id,  exec::program\_id(),  AuctionAction::CompleteTx(tx.clone()),  0,  )  .expect("Failed to send message");  } } |
| --- |

It's also necessary to reserve gas for system messages before every transaction. Here’s how we’ll implement it:

| **#[gstd::async\_main]** async fn **main**() {  ...   let reply = match action {  AuctionAction::StartAuction {  Tamagotchi\_id,  minimum\_bid,  duration,  } => {  system\_reserve\_gas();  auction  .start\_auction(&Tamagotchi\_id, minimum\_bid, duration)  .await  }  AuctionAction::MakeBid { bid } => {  system\_reserve\_gas();  auction.make\_bid(bid).await  }  AuctionAction::SettleAuction => {  system\_reserve\_gas();  auction.settle\_auction().await  }  ...  msg::reply(reply, 0).expect("Failed to encode or reply with `Result<MarketEvent, MarketErr>`"); }  fn **system\_reserve\_gas**() {  exec::system\_reserve\_gas(SYSTEM\_GAS).expect("Error during system gas reservation"); } |
| --- |

**Assignment:**

In this assignment, you will add a new feature to your Tamagotchi contract, which will allow it to send a delayed message to itself at a specified interval to check its state. If the Tamagotchi is tired, hungry, or not entertained, it should send a message to the user asking for feeding or playing.

To implement this feature, you need to extend the Tamagotchi state with the *reservations* field, as shown below:

| **#[derive(Default, Encode, Decode, TypeInfo)]** pub struct **Tamagotchi** {  pub name: String,  pub date\_of\_birth: u64,  pub owner: ActorId,  pub fed: u64,  pub fed\_block: u64,  pub entertained: u64,  pub entertained\_block: u64,  pub rested: u64,  pub rested\_block: u64,  pub allowed\_account: Option<ActorId>,  pub ft\_contract\_id: ActorId,  pub ft\_transaction\_id: TransactionId,  pub approve\_transaction: Option<(TransactionId, ActorId, u128)>,  pub reservations: Vec<ReservationId>, } |
| --- |

Next, you need to add two new incoming messages to the *TmgAction* enum: *CheckState* and *ReserveGas*, as shown below:

| *#[derive(Encode, Decode, TypeInfo)]* pub enum **TmgAction** {  Name,  Age,  Feed,  Play,  Sleep,  Transfer(ActorId),  Approve(ActorId),  RevokeApproval,  ApproveTokens {  account: ActorId,  amount: u128,  },  SetFTokenContract(ActorId),  BuyAttribute {  store\_id: ActorId,  attribute\_id: AttributeId,  },  CheckState,  ReserveGas {  reservation\_amount: u64,  duration: u32,  }, } |
| --- |

You also need to add three new outcoming messages to the *TmgEvent* enum: *FeedMe*, *PlayWithMe*, and *WantToSleep*. If the Tamagotchi runs out of gas, it should send the message *MakeReservation* asking the owner to reserve gas to continue checking the state.

You should also add the outcoming message *GasReserved* to indicate a successful gas reservation, as shown below:

| #[derive(Encode, Decode, TypeInfo)] pub enum TmgEvent {  Name(String),  Age(u64),  Fed,  Entertained,  Slept,  Transfer(ActorId),  Approve(ActorId),  RevokeApproval,  ApproveTokens { account: ActorId, amount: u128 },  FTokenContractSet,  ApprovalError,  AttributeBought(AttributeId),  CompletePrevPurchase(AttributeId),  ErrorDuringPurchase,  FeedMe,  PlayWithMe,  WantToSleep,  MakeReservation,  GasReserved, } |
| --- |

So, the Tamagotchi must send a message to itself once in a certain time interval. Define this interval and determine at what levels of *fed*, *slept* or *entertained* the Tamagotchi will start sending messages.

Connect your Tamagotchi to the application and see how it communicates with you!