

ChatGPT

What is the transaction cost in gas for this Solidity function? Please provide a detailed analysis.

```
function createBatch(uint256 _quantity) external  
onlyRole(msg.sender, Registration.Role.Manufacturer) {  
    batchCounter++;  
    batches[batchCounter] = Batch(batchCounter,  
msg.sender, _quantity, MedicationState.Created);  
    emit BatchCreated(batchCounter, msg.sender,  
_quantity, block.timestamp);  
}
```

To estimate the gas cost of the given `createBatch` function in Solidity, we need to analyze each operation and factor in the storage and event emission costs. Here's a detailed breakdown of the operations and their associated gas costs:

1. Function Modifier: `onlyRole(msg.sender, Registration.Role.Manufacturer)`

- **Gas cost:** Depends on the implementation of `onlyRole`.
 - If `onlyRole` checks the sender's role by reading from a mapping or similar structure, there will be a **read operation from storage**.
 - Gas cost for a `SLOAD` (storage read): **~2100 gas** per access.
 - Total depends on the complexity of the modifier, which is typically constant for simple role checks.
-

2. Incrementing `batchCounter`

- **Operation:** `batchCounter++`
- **Gas cost:** Incrementing a state variable involves a **storage write**.

- Gas cost for a `SSTORE` (storage write):
 - If the variable is set to a new non-zero value: **~20,000 gas**.
 - If it changes from one non-zero value to another: **~5,000 gas** (since it's just an update).
-

3. Creating and Storing the `Batch` Struct

- **Operation:** `batches[batchCounter] = Batch(batchCounter, msg.sender, _quantity, MedicationState.Created)`
 - This involves:
 - Calculating the storage location for `batches[batchCounter]` (hashed index).
 - Writing multiple fields of the struct to storage.
 - Cost depends on the size of the struct:
 - `batchId` : **256-bit integer** → 1 storage slot.
 - `msg.sender` : **address** → 1 storage slot.
 - `_quantity` : **256-bit integer** → 1 storage slot.
 - `MedicationState.Created` : Enum typically stored as a **uint8** → packed with `_quantity` or occupies 1 slot.
 - Total: **3-4 storage writes** → ~20,000 gas per `SSTORE` × 3-4 = **60,000-80,000 gas**.
-

4. Event Emission: `BatchCreated`

- **Operation:** Emits `BatchCreated` with arguments: `batchCounter` , `msg.sender` , `_quantity` , `block.timestamp` .
 - Each indexed parameter: **8 gas per byte** for 32 bytes = **256 gas** per indexed parameter.
 - Non-indexed parameters: Stored in the logs as data:
 - **~68 gas per byte** for the length of the data.

- Estimated size of data:
 - `batchCounter` (32 bytes), `msg.sender` (20 bytes), `_quantity` (32 bytes), `block.timestamp` (32 bytes).
 - Total gas for logs and data: ~1,000–2,000 gas (depends on implementation).
-

5. Other Overheads

- **Function call and return:** Includes the base transaction cost and the intrinsic gas cost.
 - Base gas: 21,000 gas.
 - Additional overhead for function logic: ~500–1,000 gas.
-

Estimated Total Gas Cost

Adding up the components:

1. `onlyRole` : ~2,100–4,000 gas (depends on complexity).
2. Incrementing `batchCounter` : ~5,000–20,000 gas.
3. Storing `Batch` : ~60,000–80,000 gas.
4. Event emission: ~1,000–2,000 gas.
5. Overhead: ~21,500–22,000 gas.

Total Gas Estimate: ~90,600–128,000 gas.

Key Considerations

- **Optimizations:**
 - If `MedicationState` can be packed into a single slot with `_quantity`, storage costs might be slightly lower.

- Using `memory` instead of directly writing to `storage` before final storage updates can reduce gas costs.
- **Exact Cost Measurement:**
 - To measure the precise gas cost, deploy the contract on a test network and use tools like **Truffle**, **Hardhat**, or Ethereum development frameworks to analyze transaction receipts.

Let me know if you need further assistance with testing or optimizing this function!

Do the same for this function:

```
function createPrescription(bytes32 _patientHash, uint256
_dosage, address _facility) external onlyRole(msg.sender,
Registration.Role.HealthcareProvider) {
    require(facilityInventory[_facility] >= _dosage,
    "Insufficient inventory");
    prescriptionCounter++;
    prescriptions[prescriptionCounter] = Prescription(
        prescriptionCounter,
        _patientHash,
        _dosage,
        _facility,
        msg.sender,
        PrescriptionState.Created
    );
    emit PrescriptionCreated(prescriptionCounter,
    _patientHash, msg.sender, block.timestamp);
}
```

Here's a detailed gas cost analysis for the `createPrescription` function:

1. Function Modifier: `onlyRole(msg.sender, Registration.Role.HealthcareProvider)`

- **Operation:** Validates the sender's role.
- **Gas cost:** Depends on the implementation of `onlyRole`.
 - Typical implementation involves a **storage read**.

- Storage read (`SLOAD`): ~2100 gas per access.
 - Total gas for role verification: ~2100–4000 gas.
-

2. Inventory Check: `require(facilityInventory[_facility] >= _dosage, "Insufficient inventory")`

- **Operation:** Reads the `facilityInventory` mapping for the given `_facility` address and compares it to `_dosage`.
 - Gas cost for a `SLOAD` (storage read): ~2100 gas.
 - Comparison operation: Negligible gas (~3–5 gas).
 - Total: ~2100 gas.
-

3. Incrementing `prescriptionCounter`

- **Operation:** `prescriptionCounter++`.
 - **Gas cost:** Incrementing a state variable involves a **storage write**.
 - Storage write (`SSTORE`):
 - If it changes from a non-zero value to another non-zero value: ~5000 gas.
 - If it changes from 0 to a non-zero value: ~20,000 gas.
 - Assuming the counter starts at a non-zero value: ~5000 gas.
-

4. Creating and Storing the `Prescription` Struct

- **Operation:**

solidity

```
prescriptions[prescriptionCounter] = Prescription(  
    prescriptionCounter,
```

```
_patientHash,  
_dosage,  
_facility,  
msg.sender,  
PrescriptionState.Created  
);
```

- This involves:
 - Calculating the storage location for `prescriptions[prescriptionCounter]` (hashed index).
 - Writing multiple fields of the struct to storage.
 - Size of the struct:
 - `prescriptionCounter` : 256-bit integer → 1 slot.
 - `_patientHash` : bytes32 → 1 slot.
 - `_dosage` : 256-bit integer → 1 slot.
 - `_facility` : address → 1 slot.
 - `msg.sender` : address → 1 slot.
 - `PrescriptionState.Created` : Enum (typically stored as `uint8`) → packed with another value or takes 1 slot.
 - Total: 5–6 storage writes → ~20,000 gas per `SSTORE` × 5–6 = ~100,000–120,000 gas.
-

5. Event Emission: PrescriptionCreated

- **Operation:** Emits `PrescriptionCreated` with arguments: `prescriptionCounter`, `_patientHash`, `msg.sender`, `block.timestamp`.
 - Each indexed parameter:
 - `prescriptionCounter` (indexed): 32 bytes.
 - `_patientHash` (indexed): 32 bytes.
 - Total for indexed parameters: 256 gas × 2 = 512 gas.
 - Non-indexed parameters:
 - `msg.sender` (20 bytes), `block.timestamp` (32 bytes).

- Gas cost: $\sim 68 \text{ gas per byte} \times (20 + 32) = \sim 3536 \text{ gas}$.
 - Total event emission gas: $\sim 4000\text{--}5000 \text{ gas}$.
-

6. Other Overheads

- **Function call and return:** Includes the base transaction cost and intrinsic gas costs.
 - Base gas: **21,000 gas**.
 - Additional overhead for function logic: $\sim 500\text{--}1,000 \text{ gas}$.
-

Estimated Total Gas Cost

Adding up the components:

1. `onlyRole` : $\sim 2100\text{--}4000 \text{ gas}$.
2. Inventory check: $\sim 2100 \text{ gas}$.
3. Incrementing `prescriptionCounter` : $\sim 5000\text{--}20,000 \text{ gas}$.
4. Storing `Prescription` : $\sim 100,000\text{--}120,000 \text{ gas}$.
5. Event emission: $\sim 4000\text{--}5000 \text{ gas}$.
6. Overhead: $\sim 21,500\text{--}22,000 \text{ gas}$.

Total Gas Estimate: $\sim 135,000\text{--}173,000 \text{ gas}$.

Key Considerations

- **Optimizations:**
 - Packing `PrescriptionState` into an existing storage slot with `_dosage` or another small field can save storage costs.

- Reading `facilityInventory[_facility]` only once and storing it in memory could slightly optimize gas usage.
- **Exact Gas Measurement:**
 - Use tools like **Truffle** or **Hardhat** to measure the actual gas cost on a test network.
 - Consider simulating the function call using a development environment.

Let me know if you'd like further help optimizing or testing this function!

Do the same for this function:

```
function approveBatch(uint256 _batchId) external
onlyRegulatoryAuthority {
    Batch storage batch = batches[_batchId];
    require(batch.state == MedicationState.Created, "Batch
not in Created state");
    batch.state = MedicationState.Approved;
    emit BatchApproved(_batchId, block.timestamp);
}
```

Here is a detailed gas cost analysis for the `approveBatch` function:

1. Function Modifier: `onlyRegulatoryAuthority`

- **Operation:** Ensures that the caller has the appropriate role.
 - **Gas cost:** Depends on the implementation of the modifier.
 - Typical role checks involve reading a value from storage.
 - Storage read (`SLOAD`): ~2100 gas per access.
 - Total cost for the role check: ~2100–4000 gas.
-

2. Accessing `batches[_batchId]`

- **Operation:** Retrieves the storage pointer for `batches[_batchId]` .

- Accessing a mapping involves hashing the `_batchId` and using the hash to calculate the storage slot.
 - Storage read (`SLOAD`): ~2100 gas.
-

3. Condition Check: `require(batch.state == MedicationState.Created)`

- **Operation:** Reads the `state` field of the `Batch` struct and compares it to `MedicationState.Created`.
 - Storage read for `batch.state` : ~2100 gas.
 - Comparison operation: Negligible gas (~3–5 gas).
 - Total gas for this step: ~2100 gas.
-

4. Updating `batch.state`

- **Operation:** Updates the `state` field of the `Batch` struct to `MedicationState.Approved`.
 - Storage write (`SSTORE`):
 - If `batch.state` changes from one non-zero value to another: ~5000 gas.
 - If it changes from zero to non-zero: ~20,000 gas.
 - Assuming `state` starts at `MedicationState.Created` (non-zero): ~5000 gas.
-

5. Event Emission: `emit BatchApproved(_batchId, block.timestamp)`

- **Operation:** Emits the `BatchApproved` event with parameters `_batchId` and `block.timestamp`.
 - `_batchId` (indexed): 32 bytes.
 - `block.timestamp` (non-indexed): 32 bytes.

- Indexed parameter: $256 \text{ gas} \times 1 = 256 \text{ gas}$.
 - Non-indexed parameter:
 - $32 \text{ bytes} \times 68 \text{ gas per byte} = 2176 \text{ gas}$.
 - Total event emission gas: **~2432 gas**.
-

6. Other Overheads

- **Function call and return:** Includes base transaction cost and intrinsic gas costs.
 - Base gas: **21,000 gas**.
 - Additional overhead for function logic: **~500–1000 gas**.
-

Estimated Total Gas Cost

Adding up the components:

1. `onlyRegulatoryAuthority` : **~2100–4000 gas**.
2. Accessing `batches[_batchId]` : **~2100 gas**.
3. Condition check: **~2100 gas**.
4. Updating `batch.state` : **~5000–20,000 gas**.
5. Event emission: **~2432 gas**.
6. Overhead: **~21,500–22,000 gas**.

Total Gas Estimate: ~35,232–52,632 gas.

Key Considerations

- Optimizations:

- Avoid redundant storage reads by assigning `batch.state` to a local variable for the condition check.
- If `MedicationState` is stored as a smaller type (e.g., `uint8`), it can save storage and retrieval costs.
- **Exact Gas Measurement:**
 - Use tools like **Truffle**, **Hardhat**, or **Remix** to deploy and simulate the function to get precise gas estimates.

Let me know if you want assistance testing or optimizing this function!

Do the same for this function:

```
function shipBatch(uint256 _batchId, address _to) external
onlyRole(msg.sender, Registration.Role.Manufacturer) {
    Batch storage batch = batches[_batchId];
    require(batch.state == MedicationState.Approved,
"Batch not approved");
    batch.state = MedicationState.Shipped;
    emit BatchShipped(_batchId, _to, batch.quantity,
block.timestamp);
}
```

Here's a detailed gas cost analysis for the `shipBatch` function:

1. Function Modifier: `onlyRole(msg.sender, Registration.Role.Manufacturer)`

- **Operation:** Ensures the caller has the appropriate role.
- **Gas cost:**
 - Role validation typically involves reading a value from storage.
 - Storage read (`SLOAD`): ~2100 gas per access.
 - Total cost for role check: ~2100–4000 gas.

2. Accessing `batches[_batchId]`

- **Operation:** Retrieves the storage pointer for the batch with ID `_batchId`.
 - Accessing a mapping requires hashing `_batchId` to compute the storage slot.
 - Storage read (`SLOAD`): ~2100 gas.
-

3. Condition Check: `require(batch.state == MedicationState.Approved)`

- **Operation:** Reads the `state` field of the `Batch` struct and compares it to `MedicationState.Approved`.
 - Storage read for `batch.state` : ~2100 gas.
 - Comparison operation: Negligible gas (~3–5 gas).
 - Total gas for this step: ~2100 gas.
-

4. Updating `batch.state`

- **Operation:** Updates the `state` field of the `Batch` struct to `MedicationState.Shipped`.
 - Storage write (`SSTORE`):
 - If `batch.state` changes from one non-zero value to another: ~5000 gas.
 - If it changes from zero to non-zero: ~20,000 gas.
 - Assuming `state` starts at `MedicationState.Approved` (non-zero): ~5000 gas.
-

5. Event Emission: `emit BatchShipped(_batchId, _to, batch.quantity, block.timestamp)`

- **Operation:** Emits the `BatchShipped` event with parameters `_batchId`, `_to`, `batch.quantity`, and `block.timestamp`.

- `_batchId` and `_to` (indexed parameters):
 - Each 32 bytes, $256 \text{ gas} \times 2 = 512 \text{ gas}$.
 - `batch.quantity` and `block.timestamp` (non-indexed parameters):
 - Combined size: $64 \text{ bytes} \times 68 \text{ gas per byte} = 4352 \text{ gas}$.
 - Total event emission gas: **~4864 gas**.
-

6. Other Overheads

- **Function call and return:** Includes base transaction cost and intrinsic gas costs.
 - Base gas: **21,000 gas**.
 - Additional overhead for function logic: **~500–1000 gas**.
-

Estimated Total Gas Cost

Adding up the components:

1. `onlyRole` : **~2100–4000 gas**.
2. Accessing `batches[_batchId]` : **~2100 gas**.
3. Condition check: **~2100 gas**.
4. Updating `batch.state` : **~5000–20,000 gas**.
5. Event emission: **~4864 gas**.
6. Overhead: **~21,500–22,000 gas**.

Total Gas Estimate: ~37,664–54,064 gas.

Key Considerations

- Optimizations:

- Cache `batch.state` in memory to avoid redundant storage reads.
- Ensure `MedicationState` is efficiently packed into storage to reduce costs.
- **Testing for Exact Gas Usage:**
 - Use frameworks like **Truffle** or **Hardhat** to deploy and simulate the function to get precise gas estimates.

Let me know if you'd like to explore testing, optimizations, or detailed comparisons!

Do the same for this function:

```
function receiveBatch(uint256 _batchId) external
onlyRole(msg.sender, Registration.Role.HealthcareFacility) {
    Batch storage batch = batches[_batchId];
    require(batch.state == MedicationState.Shipped, "Batch
not shipped");
    batch.state = MedicationState.Received;
    facilityInventory[msg.sender] += batch.quantity;
    emit BatchReceived(_batchId, msg.sender,
batch.quantity, block.timestamp);
}
```

Here's a detailed gas cost analysis for the `receiveBatch` function:

1. Function Modifier: `onlyRole(msg.sender, Registration.Role.HealthcareFacility)`

- **Operation:** Ensures the caller has the appropriate role.
- **Gas cost:**
 - Role validation typically involves reading a value from storage.
 - Storage read (`SLOAD`): ~2100 gas per access.
 - Total cost for role check: ~2100–4000 gas.

2. Accessing `batches[_batchId]`

- **Operation:** Retrieves the storage pointer for the batch with ID `_batchId`.
 - Accessing a mapping requires hashing `_batchId` to compute the storage slot.
 - Storage read (`SLOAD`): ~2100 gas.
-

3. Condition Check: `require(batch.state == MedicationState.Shipped)`

- **Operation:** Reads the `state` field of the `Batch` struct and compares it to `MedicationState.Shipped`.
 - Storage read for `batch.state` : ~2100 gas.
 - Comparison operation: Negligible gas (~3–5 gas).
 - Total gas for this step: ~2100 gas.
-

4. Updating `batch.state`

- **Operation:** Updates the `state` field of the `Batch` struct to `MedicationState.Received`.
 - Storage write (`SSTORE`):
 - If `batch.state` changes from one non-zero value to another: ~5000 gas.
 - If it changes from zero to non-zero: ~20,000 gas.
 - Assuming `state` starts at `MedicationState.Shipped` (non-zero): ~5000 gas.
-

5. Updating `facilityInventory[msg.sender]`

- **Operation:** Increments the inventory of the healthcare facility by `batch.quantity`.
 - Reading `facilityInventory[msg.sender]` : ~2100 gas.

- Updating `facilityInventory[msg.sender]` : ~5000 gas if updated from non-zero to non-zero, ~20,000 gas if updated from zero to non-zero.
 - Total gas for this step: ~7100–22,100 gas.
-

6. Event Emission: `emit BatchReceived(_batchId, msg.sender, batch.quantity, block.timestamp)`

- **Operation:** Emits the `BatchReceived` event with parameters `_batchId`, `msg.sender`, `batch.quantity`, and `block.timestamp`.
 - `_batchId` and `msg.sender` (indexed parameters):
 - Each 32 bytes, $256 \text{ gas} \times 2 = 512 \text{ gas}$.
 - `batch.quantity` and `block.timestamp` (non-indexed parameters):
 - Combined size: $64 \text{ bytes} \times 68 \text{ gas per byte} = 4352 \text{ gas}$.
 - Total event emission gas: ~4864 gas.
-

7. Other Overheads

- **Function call and return:** Includes base transaction cost and intrinsic gas costs.
 - Base gas: 21,000 gas.
 - Additional overhead for function logic: ~500–1000 gas.
-

Estimated Total Gas Cost

Adding up the components:

1. `onlyRole` : ~2100–4000 gas.
2. Accessing `batches[_batchId]` : ~2100 gas.
3. Condition check: ~2100 gas.

4. Updating `batch.state` : ~5000–20,000 gas.
5. Updating `facilityInventory[msg.sender]` : ~7100–22,100 gas.
6. Event emission: ~4864 gas.
7. Overhead: ~21,500–22,000 gas.

Total Gas Estimate: ~44,764–76,164 gas.

Key Considerations

- **Optimizations:**
 - Cache `batch.state` in a local variable to avoid redundant storage reads.
 - If `facilityInventory[msg.sender]` updates frequently, ensuring that values are initialized to non-zero can reduce storage write costs.
 - Efficient packing of data types in the `Batch` struct can also reduce storage costs.
- **Testing for Exact Gas Usage:**
 - Use frameworks like **Truffle** or **Hardhat** to deploy and simulate the function to measure exact gas costs.

Let me know if you'd like help testing, optimizing, or refining this function!

Do the same for this function:

```
function administerMedication(uint256 _prescriptionId)
external onlyRole(msg.sender,
Registration.Role.HealthcareProvider) {
    Prescription storage prescription =
prescriptions[_prescriptionId];
    require(prescription.state ==
PrescriptionState.Approved, "Prescription not approved");

    require(facilityInventory[prescription.healthcareFacility] >=
prescription.dosage, "Insufficient inventory");

    facilityInventory[prescription.healthcareFacility] -=
prescription.dosage;
```

```
emit  
MedicationAdministered(prescription.prescriptionId,  
prescription.dosage, prescription.healthcareFacility,  
block.timestamp);  
}
```

Here is a detailed gas cost analysis for the `administerMedication` function:

1. Function Modifier: `onlyRole(msg.sender, Registration.Role.HealthcareProvider)`

- **Operation:** Ensures the caller has the appropriate role.
 - **Gas cost:**
 - Typical implementation involves reading a value from storage.
 - Storage read (`SLOAD`): ~2100 gas per access.
 - Total cost for role validation: ~2100–4000 gas.
-

2. Accessing `prescriptions[_prescriptionId]`

- **Operation:** Retrieves the storage pointer for the prescription with ID `_prescriptionId`.
 - Accessing a mapping requires hashing `_prescriptionId` to compute the storage slot.
 - Storage read (`SLOAD`): ~2100 gas.
-

3. Checking Prescription State

- **Operation:** `require(prescription.state == PrescriptionState.Approved)`
 - Reads the `state` field of the `Prescription` struct and compares it to `PrescriptionState.Approved`.

- Storage read (`SLOAD`) for `prescription.state` : ~2100 gas.
 - Comparison operation: Negligible gas (~3–5 gas).
 - Total gas for this step: ~2100 gas.
-

4. Checking Facility Inventory

- Operation: `require(facilityInventory[prescription.healthcareFacility] >= prescription.dosage)`
 - Reads the `facilityInventory` mapping for the healthcare facility's address.
 - Storage read (`SLOAD`) for `facilityInventory[prescription.healthcareFacility]` : ~2100 gas.
 - Comparison operation: Negligible gas (~3–5 gas).
 - Total gas for this step: ~2100 gas.
-

5. Updating Facility Inventory

- Operation: `facilityInventory[prescription.healthcareFacility] -= prescription.dosage`
 - Reads the current value of `facilityInventory[prescription.healthcareFacility]` : ~2100 gas.
 - Updates the value by decrementing the dosage:
 - Storage write (`SSTORE`):
 - If updating from one non-zero value to another: ~5000 gas.
 - If updating from zero to a non-zero value: ~20,000 gas.
 - Total gas for this step: ~7100–22,100 gas.
-

6. Event Emission: `emit MedicationAdministered(...)`

- **Operation:** Emits the `MedicationAdministered` event with parameters: `prescription.prescriptionId`, `prescription.dosage`, `prescription.healthcareFacility`, and `block.timestamp`.
 - Indexed parameters: `prescriptionId` (32 bytes), `healthcareFacility` (20 bytes).
 - Indexed parameters: $256 \text{ gas} \times 2 = 512 \text{ gas}$.
 - Non-indexed parameters: `dosage` (32 bytes), `block.timestamp` (32 bytes).
 - Combined size: $64 \text{ bytes} \times 68 \text{ gas per byte} = 4352 \text{ gas}$.
 - Total event emission gas: **~4864 gas**.
-

7. Other Overheads

- **Function call and return:** Includes base transaction cost and intrinsic gas costs.
 - Base gas: **21,000 gas**.
 - Additional overhead for function logic: **~500–1000 gas**.
-

Estimated Total Gas Cost

Adding up the components:

1. `onlyRole` : **~2100–4000 gas**.
2. Accessing `prescriptions[_prescriptionId]` : **~2100 gas**.
3. Checking prescription state: **~2100 gas**.
4. Checking facility inventory: **~2100 gas**.
5. Updating facility inventory: **~7100–22,100 gas**.
6. Event emission: **~4864 gas**.
7. Overhead: **~21,500–22,000 gas**.

Total Gas Estimate: ~42,864–59,264 gas.

Key Considerations

- **Optimizations:**
 - Cache `prescription.state` in memory to avoid redundant storage reads.
 - Ensure `facilityInventory[prescription.healthcareFacility]` values are initialized to non-zero to avoid higher gas costs for updates.
 - Efficient packing of the `Prescription` struct can slightly reduce gas usage.
- **Testing for Exact Gas Usage:**
 - Use frameworks like **Truffle** or **Hardhat** to simulate this function on a test network for precise measurements.

Let me know if you'd like help with testing or optimizing this function further!

Do the same for this function:

```
function disposeMedication(uint256 _quantity) external
onlyRole(msg.sender, Registration.Role.HealthcareFacility) {
    require(facilityInventory[msg.sender] >= _quantity,
        "Insufficient inventory");
    facilityInventory[msg.sender] -= _quantity;
    emit MedicationDisposed(msg.sender, _quantity,
        block.timestamp);
}
```

Here's a detailed gas cost analysis for the `disposeMedication` function:

1. Function Modifier: `onlyRole(msg.sender, Registration.Role.HealthcareFacility)`

- **Operation:** Ensures the caller has the appropriate role.
- **Gas cost:**
 - Role validation typically involves reading a value from storage.

- Storage read (`SLOAD`): ~2100 gas per access.
 - Total cost for role validation: ~2100–4000 gas.
-

2. Checking Facility Inventory

- **Operation:** `require(facilityInventory[msg.sender] >= _quantity, "Insufficient inventory")`
 - Reads the `facilityInventory` mapping for the caller's (`msg.sender`) address.
 - Storage read (`SLOAD`) for `facilityInventory[msg.sender]` : ~2100 gas.
 - Comparison operation: Negligible gas (~3–5 gas).
 - Total gas for this step: ~2100 gas.
-

3. Updating Facility Inventory

- **Operation:** `facilityInventory[msg.sender] -= _quantity`
 - Reads the current value of `facilityInventory[msg.sender]` : ~2100 gas.
 - Updates the value by decrementing `_quantity` :
 - Storage write (`SSTORE`):
 - If updating from one non-zero value to another: ~5000 gas.
 - If updating from zero to a non-zero value: ~20,000 gas.
 - Total gas for this step: ~7100–22,100 gas.
-

4. Event Emission: `emit MedicationDisposed(...)`

- **Operation:** Emits the `MedicationDisposed` event with parameters: `msg.sender` , `_quantity` , and `block.timestamp` .

- Indexed parameter: `msg.sender` (20 bytes).
 - Indexed parameter: **256 gas**.
 - Non-indexed parameters: `_quantity` (32 bytes), `block.timestamp` (32 bytes).
 - Combined size: 64 bytes × 68 gas per byte = **4352 gas**.
 - Total event emission gas: **~4608 gas**.
-

5. Other Overheads

- **Function call and return:** Includes base transaction cost and intrinsic gas costs.
 - Base gas: **21,000 gas**.
 - Additional overhead for function logic: **~500–1000 gas**.
-

Estimated Total Gas Cost

Adding up the components:

1. `onlyRole` : **~2100–4000 gas**.
2. Checking facility inventory: **~2100 gas**.
3. Updating facility inventory: **~7100–22,100 gas**.
4. Event emission: **~4608 gas**.
5. Overhead: **~21,500–22,000 gas**.

Total Gas Estimate: ~37,408–54,808 gas.

Key Considerations

- **Optimizations:**

- Cache the `facilityInventory[msg.sender]` value in a memory variable to avoid redundant reads.
- Ensure `facilityInventory` values are initialized to non-zero to reduce higher gas costs for updates.
- **Testing for Exact Gas Usage:**
 - Use tools like **Truffle**, **Hardhat**, or **Remix** to deploy and simulate this function to measure precise gas costs.

Let me know if you need further assistance testing or optimizing this function!

Do the same for this function:

```
function approvePrescription(uint256 _prescriptionId)
external onlyRegulatoryAuthority {
    Prescription storage prescription =
prescriptions[_prescriptionId];
    require(prescription.state == PrescriptionState.Created,
"Prescription not in Created state");
    prescription.state = PrescriptionState.Approved;
    emit PrescriptionApproved(_prescriptionId,
block.timestamp);
}
```

Here is a detailed gas cost analysis for the `approvePrescription` function:

1. Function Modifier: `onlyRegulatoryAuthority`

- **Operation:** Ensures the caller has the authority to approve prescriptions.
- **Gas cost:**
 - This typically involves reading a value from storage to check roles.
 - Storage read (`SLOAD`): ~2100 gas per access.
 - Total cost for role validation: ~2100–4000 gas.

2. Accessing `prescriptions[_prescriptionId]`

- **Operation:** Retrieves the storage pointer for the prescription with ID `_prescriptionId`.
 - Accessing a mapping requires hashing `_prescriptionId` to compute the storage slot.
 - Storage read (`SLOAD`): ~2100 gas.
-

3. Checking Prescription State

- **Operation:** `require(prescription.state == PrescriptionState.Created)`
 - Reads the `state` field of the `Prescription` struct and compares it to `PrescriptionState.Created`.
 - Storage read for `prescription.state`: ~2100 gas.
 - Comparison operation: Negligible gas (~3–5 gas).
 - Total gas for this step: ~2100 gas.
-

4. Updating Prescription State

- **Operation:** `prescription.state = PrescriptionState.Approved`
 - Updates the `state` field of the `Prescription` struct to `PrescriptionState.Approved`.
 - Storage write (`SSTORE`):
 - If updating from one non-zero value to another: ~5000 gas.
 - If updating from zero to a non-zero value: ~20,000 gas.
 - Assuming `state` starts at `PrescriptionState.Created` (non-zero): ~5000 gas.
-

5. Event Emission: `emit PrescriptionApproved(...)`

- **Operation:** Emits the `PrescriptionApproved` event with parameters: `_prescriptionId` and `block.timestamp`.
 - `_prescriptionId` (indexed parameter): 32 bytes.
 - Indexed parameter: **256 gas**.
 - `block.timestamp` (non-indexed parameter): 32 bytes.
 - Non-indexed parameter: **32 bytes × 68 gas per byte = 2176 gas**.
 - Total event emission gas: **~2432 gas**.
-

6. Other Overheads

- **Function call and return:** Includes base transaction cost and intrinsic gas costs.
 - Base gas: **21,000 gas**.
 - Additional overhead for function logic: **~500–1000 gas**.
-

Estimated Total Gas Cost

Adding up the components:

1. `onlyRegulatoryAuthority` : **~2100–4000 gas**.
2. Accessing `prescriptions[_prescriptionId]` : **~2100 gas**.
3. Checking prescription state: **~2100 gas**.
4. Updating prescription state: **~5000–20,000 gas**.
5. Event emission: **~2432 gas**.
6. Overhead: **~21,500–22,000 gas**.

Total Gas Estimate: ~35,232–52,632 gas.

Key Considerations

- **Optimizations:**
 - Cache the `prescription.state` value in memory to avoid redundant storage reads.
 - Efficient packing of the `Prescription` struct can reduce storage costs slightly.
- **Testing for Exact Gas Usage:**
 - Use tools like **Truffle**, **Hardhat**, or **Remix** to deploy and simulate the function for precise measurements.

Let me know if you'd like help testing, optimizing, or deploying this function!

Do the same for this function:

```
function registerEntity(address _entityAddress, Role _role)
external {
    require(entities[_entityAddress].entityAddress ==
address(0), "Entity already registered");
    entities[_entityAddress] = Entity(_entityAddress, _role,
false);
    emit EntityRegistered(_entityAddress, _role,
block.timestamp);
}
```

Here's a detailed gas cost analysis for the `registerEntity` function:

1. Checking if Entity is Already Registered

- **Operation:**

solidity

```
require(entities[_entityAddress].entityAddress == address(0), "Entity already
registered");
```

- Reads the `entityAddress` field from the `entities` mapping for `_entityAddress`.
- Storage read (`SLOAD`): ~2100 gas.
- Comparison operation: Negligible gas (~3–5 gas).

- Total gas for this step: ~2100 gas.
-

2. Registering the Entity

- Operation:

solidity

```
entities[_entityAddress] = Entity(_entityAddress, _role, false);
```

- Writes a new `Entity` struct to the `entities` mapping for `_entityAddress`.
 - Fields in the struct:
 - `entityAddress` : `address` → 1 storage slot.
 - `_role` : Enum (typically stored as `uint8`) → packed with another field or 1 storage slot.
 - `false` (boolean): Typically part of the same storage slot as `_role`.
 - Total storage writes: 1-2 `SSTORE` operations:
 - If the slot is being initialized from zero: ~20,000 gas per slot.
 - If the slot is being updated from a non-zero value: ~5000 gas per slot.
 - Assuming initialization: ~20,000–40,000 gas.
-

3. Emitting the `EntityRegistered` Event

- Operation:

solidity

```
emit EntityRegistered(_entityAddress, _role, block.timestamp);
```

- Indexed parameter: `_entityAddress` (20 bytes).
 - Indexed parameter: 256 gas.

- Non-indexed parameters:
 - `_role` (typically `uint8`, padded to 32 bytes) and `block.timestamp` (32 bytes).
 - Total size: 64 bytes × 68 gas per byte = **4352 gas**.
 - Total event emission gas: **~4608 gas**.
-

4. Other Overheads

- **Function call and return:** Includes base transaction cost and intrinsic gas costs.
 - Base gas: **21,000 gas**.
 - Additional overhead for function logic: **~500–1000 gas**.
-

Estimated Total Gas Cost

Adding up the components:

1. Checking if entity is already registered: **~2100 gas**.
2. Registering the entity: **~20,000–40,000 gas**.
3. Emitting the event: **~4608 gas**.
4. Overhead: **~21,500–22,000 gas**.

Total Gas Estimate: ~48,208–68,708 gas.

Key Considerations

- **Optimizations:**
 - Minimize storage writes by ensuring efficient packing of data in the `Entity` struct.
 - Cache `entities[_entityAddress]` in memory for reuse if accessed multiple times.
- **Testing for Exact Gas Usage:**

- Deploy and simulate the function using tools like **Truffle**, **Hardhat**, or **Remix** for precise gas cost measurements.

Let me know if you need help testing, optimizing, or deploying this function!

Do the same for this function:

```
function approveEntity(address _entityAddress) external  
onlyRegulatoryAuthority {  
    require(entities[_entityAddress].entityAddress !=  
address(0), "Entity not registered");  
    require(!entities[_entityAddress].isApproved, "Entity  
already approved");  
  
    entities[_entityAddress].isApproved = true;  
    emit EntityApproved(_entityAddress,  
entities[_entityAddress].role, block.timestamp);  
}
```

Here is a detailed gas cost analysis for the `approveEntity` function:

1. Checking if Entity is Registered

- **Operation:**

solidity

```
require(entities[_entityAddress].entityAddress != address(0), "Entity not  
registered");
```

- Reads the `entityAddress` field from the `entities` mapping for `_entityAddress`.
- Storage read (`SLOAD`): ~2100 gas.
- Comparison operation: Negligible gas (~3–5 gas).
- Total gas for this step: ~2100 gas.

2. Checking if Entity is Already Approved

- Operation:

```
solidity
```

```
require(!entities[_entityAddress].isApproved, "Entity already approved");
```

- Reads the `isApproved` field from the `entities` mapping for `_entityAddress`.
 - Storage read (`SLOAD`): ~2100 gas.
 - Comparison operation: Negligible gas (~3–5 gas).
 - Total gas for this step: ~2100 gas.
-

3. Updating Approval Status

- Operation:

```
solidity
```

```
entities[_entityAddress].isApproved = true;
```

- Writes the new value (`true`) to the `isApproved` field of the `entities` mapping for `_entityAddress`.
 - Storage write (`SSTORE`):
 - If the slot is being updated from `false` to `true`: ~5000 gas (non-zero to non-zero change).
 - If the slot is being initialized from zero: ~20,000 gas (zero to non-zero change).
 - Assuming the field was initialized but previously set to `false`: ~5000 gas.
-

4. Emitting the `EntityApproved` Event

- Operation:

solidity

```
emit EntityApproved(_entityAddress, entities[_entityAddress].role,  
block.timestamp);
```

- Indexed parameter: `_entityAddress` (20 bytes).
 - Indexed parameter: **256 gas**.
 - Non-indexed parameters:
 - `role` (typically `uint8`, padded to 32 bytes) and `block.timestamp` (32 bytes).
 - Total size: 64 bytes × 68 gas per byte = **4352 gas**.
 - Total event emission gas: **~4608 gas**.
-

5. Function Modifier: `onlyRegulatoryAuthority`

- **Operation:** Ensures the caller has the authority to approve entities.
 - **Gas cost:**
 - Typical implementation involves reading a value from storage to check roles.
 - Storage read (`SLOAD`): **~2100 gas** per access.
 - Total cost for role validation: **~2100–4000 gas**.
-

6. Other Overheads

- **Function call and return:** Includes base transaction cost and intrinsic gas costs.
 - Base gas: **21,000 gas**.
 - Additional overhead for function logic: **~500–1000 gas**.
-

Estimated Total Gas Cost

Adding up the components:

1. Checking if entity is registered: ~2100 gas.
2. Checking if entity is already approved: ~2100 gas.
3. Updating approval status: ~5000 gas.
4. Emitting the event: ~4608 gas.
5. Role validation: ~2100–4000 gas.
6. Overhead: ~21,500–22,000 gas.

Total Gas Estimate: ~37,408–39,808 gas.

Key Considerations

- **Optimizations:**
 - Cache the `entities[_entityAddress]` struct in memory for reuse, minimizing redundant storage reads.
 - If possible, pack fields like `isApproved` and `role` into a single storage slot for efficiency.
- **Testing for Exact Gas Usage:**
 - Use tools like **Truffle**, **Hardhat**, or **Remix** to deploy and simulate this function on a test network to get precise gas cost measurements.

Let me know if you need further assistance with testing, optimizing, or deploying this function!

Display all the calculated transaction costs in a table.
Return them in the following order:

- 1- `registerEntity()`
- 2- `approveEntity()`
- 3- `createBatch()`
- 4- `approveBatch()`
- 5- `shipBatch()`
- 6- `receiveBatch()`
- 7- `createPrescription()`
- 8- `approvePrescription()`

9- administerMedication()
10- disposeMedication()

Here is a summary of the transaction cost estimates for each function, displayed in a table:

Function	Estimated Gas Cost (Min)	Estimated Gas Cost (Max)
registerEntity()	48,208	68,708
approveEntity()	37,408	39,808
createBatch()	90,600	128,000
approveBatch()	35,232	52,632
shipBatch()	37,664	54,064
receiveBatch()	44,764	76,164
createPrescription()	135,000	173,000
approvePrescription()	35,232	52,632
administerMedication()	42,864	59,264
disposeMedication()	37,408	54,808

This table organizes all the functions and their estimated gas costs for easy comparison. Let me know if you'd like further refinements or assistance!

Now calculate the costs in Ether and USD for each function. Assume a gas price of 4.336 gwei and an Ether price of 3,276.46 USD. Display the results in a table with three columns: one for the cost in gas, one for the cost in Ether, and one for the cost in USD, including the range for each.

Transaction Cost Estimates

I have displayed the transaction cost estimates in Ether and USD for the given functions in the table. Let me know if you need further assistance or additional details!

Display the results in a standard table format.

Here is the transaction cost analysis for each function, presented in a standard table format:

Function	Gas Cost	Ether Cost	USD Cost
registerEntity()	48,208–68,708	0.000209–0.000298 ETH	\$0.68–\$0.98
approveEntity()	37,408–39,808	0.000162–0.000173 ETH	\$0.53–\$0.57
createBatch()	90,600–128,000	0.000393–0.000555 ETH	\$1.29–\$1.82
approveBatch()	35,232–52,632	0.000153–0.000228 ETH	\$0.50–\$0.75
shipBatch()	37,664–54,064	0.000163–0.000234 ETH	\$0.54–\$0.77
receiveBatch()	44,764–76,164	0.000194–0.000330 ETH	\$0.64–\$1.08
createPrescription()	135,000–173,000	0.000586–0.000750 ETH	\$1.92–\$2.46
approvePrescription()	35,232–52,632	0.000153–0.000228 ETH	\$0.50–\$0.75
administerMedication()	42,864–59,264	0.000186–0.000257 ETH	\$0.61–\$0.84
disposeMedication()	37,408–54,808	0.000162–0.000238 ETH	\$0.53–\$0.78

Let me know if you need further details or refinements!