**Integrating Smart Contracts with Websites**

To integrate smart contracts with websites effectively, various programming languages, libraries, front-end frameworks, and wallets can be utilized. Below are some of the key tools and suggestions for integrating smart contracts into a web application.

#### **Programming Languages**

* **Python:** Useful for back-end processing and scripting tasks.
* **JavaScript:** Highly flexible and offers a wide range of libraries for front-end and back-end development, making it a preferred choice for integrating with smart contracts.

#### **Web3 Libraries**

* **Web3.py:** A Python library for interacting with Ethereum.
* **Web3.js:** A JavaScript library for interacting with Ethereum.
* **Ethers.js:** A library that provides a complete and compact library for interacting with the Ethereum Blockchain and its ecosystem.
* **Web3Modal:** A library to help developers onboard their users with ease.
* **Wagmi:** React Hooks library for Ethereum.

#### **Front-end Frameworks**

* **Next.js:** A React framework with server-side rendering capabilities, ideal for building high-performance web applications.
* **React.js:** A JavaScript library for building user interfaces, particularly single-page applications.
* **Basic HTML/CSS/JavaScript:** For simpler applications or static websites.
* **Vue.js:** A progressive JavaScript framework for building user interfaces.

#### **Wallets**

* **MetaMask:** A popular Ethereum wallet available as a browser extension and mobile app.
* **WalletConnect:** An open protocol to connect desktop Dapps to mobile wallets using QR code scanning or deep linking.

### **Recommendations**

1. **JavaScript Flexibility:** JavaScript provides the most flexibility with a wide range of libraries, making it ideal for both front-end and back-end integration with smart contracts.
2. **Wagmi for WalletConnect:** Wagmi, combined with WalletConnect, is recommended for the best user experience, allowing seamless wallet integration and management.
3. **Next.js for Performance:** Next.js is preferred for its server-side rendering and static site generation capabilities, leading to better performance and SEO.
4. **WalletConnect Integration:** Integrating WalletConnect allows users to connect any wallet they prefer, enhancing accessibility and user convenience.

**Example Integration Workflow**

**1. Set Up Next.js Project:**

**npx create-next-app my-dapp**

**cd my-dapp**

**npm install**

**2. Install Necessary Libraries:**

npm install ethers web3modal wagmi @web3-react/core @web3-react/injected-connector

**3. Connect Wallet using Wagmi and WalletConnect:**

| import { Web3Provider } from '@ethersproject/providers';  import { useAccount, useConnect, useDisconnect } from 'wagmi';  function App() {  const { connect, connectors } = useConnect();  const { disconnect } = useDisconnect();  const { data: account } = useAccount();  return (  <div>  {account ? (  <div>  <button onClick={() => disconnect()}>Disconnect</button>  <p>Connected as {account.address}</p>  </div>  ) : (  <button onClick={() => connect(connectors[0])}>Connect Wallet</button>  )}  </div>  );  }  export default App; |
| --- |

**4. Interact with Smart Contract:**

| import { ethers } from 'ethers';  import { useSigner } from 'wagmi';  async function interactWithContract() {  const { data: signer } = useSigner();  const contractAddress = '0xYourContractAddress';  const abi = [/\* Your contract ABI \*/];  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  const response = await contract.someFunction();  console.log('Transaction Response:', response);  } catch (error) {  console.error('Error interacting with contract:', error);  }  } |
| --- |

**All the functions to read from the smart contract**

| **Function Name**: getPaymentToken  **Description**: The getPaymentToken function is a public view function in the TandaPay smart contract that retrieves the address of the current payment token used within the TandaPay community. Specifically, this function will return the address of the LUSD token. It requires no arguments and can be accessed by any user to obtain the token's address. In a front-end application, this function can be used to dynamically display the payment token address to users, ensuring they are aware of the token they need to hold or use for transactions within the TandaPay community.  **Example Code Block using EthersJS**  async function getPaymentToken() {  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the getPaymentToken function  const tokenAddress = await contract.getPaymentToken();  console.log("Payment Token Address:", tokenAddress);  } catch (error) {  console.error("Error fetching payment token address:", error);  }  } else {  console.log("Please install MetaMask");  }  }  **Example Code Block using Wagmi**  const getPaymentToken= async (user) => {  try {  const args = [];  const data = await readContract({  address: contractAddress,  abi: contractABI,  functionName: "getPaymentToken",  args:[],  chainId: expectedChainId,  });  return data;  } catch (error) {  console.log(error);  }  };  **Example Use Case in a Front-End Application**  document.getElementById("fetchTokenBtn").addEventListener("click", async () => {  const tokenAddress = await getPaymentToken();  document.getElementById("tokenAddressDisplay").innerText = `Current Payment Token: ${tokenAddress}`;  }); |
| --- |

| **Function Name:** getCurrentMemberId **Description:**  The getCurrentMemberId function is a public view function in the TandaPay smart contract that retrieves the current member ID. This function does not require any arguments and can be called by any user. The member ID is a unique identifier for each member within the TandaPay community, helping to manage and track individual members. **Use Case** The primary use case of the getCurrentMemberId function is to provide a way to track and manage members within the TandaPay community. This information is crucial for:   * **Membership Management:** Keeping track of the total number of members and their unique identifiers. * **Transaction and Interaction Tracking:** Ensuring that all interactions and transactions involving members are correctly associated with the right member ID. * **Community Organization:** Helping in organizing members within the community, particularly in subgroups or specific roles.  **Why It's Important**  * **Organization:** Helps maintain an organized structure within the TandaPay community by uniquely identifying each member. * **Tracking:** Essential for tracking member participation, transactions, and interactions within the community. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications and other tools that need to reference specific members.   **Example Code Block Using Ethers.js**  async function getCurrentMemberId() {  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the getCurrentMemberId function  const memberId = await contract.getCurrentMemberId();  console.log("Current Member ID:", memberId);  } catch (error) {  console.error("Error fetching current member ID:", error);  }  } else {  console.log("Please install MetaMask");  }  }  **Example Code Block Using Wagmi**  const getCurrentMemberId = async () => {  try {  const data = await readContract({  address: contractAddress,  abi: contractABI,  functionName: "getCurrentMemberId",  args: [],  chainId: expectedChainId,  });  console.log("Current Member ID:", data);  return data;  } catch (error) {  console.error("Error fetching current member ID:", error);  }  };  **Example Use Case in a Front-End Application**  document.getElementById("fetchMemberIdBtn").addEventListener("click", async () => {  const memberId = await getCurrentMemberId();  document.getElementById("memberIdDisplay").innerText = `Current Member ID: ${memberId}`;  }); |
| --- |

| **Function Name:** getCurrentClaimId  **Description:**  The getCurrentClaimId function is a public view function in the TandaPay smart contract that retrieves the current claim ID. This function does not require any arguments and can be called by any user. The claim ID is a unique identifier for each claim within the TandaPay community, helping to manage and track individual claims. This claim ID will be used to fetch all other details regarding that specific claim. **Example Code Block Using Ethers.js** Here is an example of how the getCurrentClaimId function can be called using Ethers.js to interact with the TandaPay smart contract:  async function getCurrentClaimId() {  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the getCurrentClaimId function  const claimId = await contract.getCurrentClaimId();  console.log("Current Claim ID:", claimId);  } catch (error) {  console.error("Error fetching current claim ID:", error);  }  } else {  console.log("Please install MetaMask");  }  } **Example Code Block Using Wagmi** Here is an example of how the getCurrentClaimId function can be called using Wagmi to interact with the TandaPay smart contract:  const getCurrentClaimId = async () => {  try {  const data = await readContract({  address: contractAddress,  abi: contractABI,  functionName: "getCurrentClaimId",  args: [],  chainId: expectedChainId,  });  console.log("Current Claim ID:", data);  return data;  } catch (error) {  console.error("Error fetching current claim ID:", error);  }  };  **Example Use Case in a Front-End Application**  document.getElementById("fetchClaimIdBtn").addEventListener("click", async () => {  const claimId = await getCurrentClaimId();  document.getElementById("claimIdDisplay").innerText = `Current Claim ID: ${claimId}`;  }); |
| --- |

| **Function Name:** getPeriodId  **Description:** The getPeriodId function is a public view function in the TandaPay smart contract that retrieves the current period ID. This function does not require any arguments and can be called by any user. The period ID is a unique identifier for each operational period within the TandaPay community, helping to manage and track the community's progress over time. **Example Code Block Using Ethers.js** Here is an example of how the getPeriodId function can be called using Ethers.js to interact with the TandaPay smart contract:  async function getPeriodId() {  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the getPeriodId function  const periodId = await contract.getPeriodId();  console.log("Current Period ID:", periodId);  } catch (error) {  console.error("Error fetching current period ID:", error);  }  } else {  console.log("Please install MetaMask");  }  } **Example Code Block Using Wagmi** Here is an example of how the getPeriodId function can be called using Wagmi to interact with the TandaPay smart contract:  const getPeriodId = async () => {  try {  const data = await readContract({  address: contractAddress,  abi: contractABI,  functionName: "getPeriodId",  args: [],  chainId: expectedChainId,  });  console.log("Current Period ID:", data);  return data;  } catch (error) {  console.error("Error fetching current period ID:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to dynamically display the current period ID to users, helping them understand the community's current operational phase.  document.getElementById("fetchPeriodIdBtn").addEventListener("click", async () => {  const periodId = await getPeriodId();  document.getElementById("periodIdDisplay").innerText = `Current Period ID: ${periodId}`;  }); |
| --- |

| **Function Name:** getTotalCoverage  **Description:**  The getTotalCoverage function is a public view function in the TandaPay smart contract that retrieves the current total coverage amount. This function does not require any arguments and can be called by any user. The total coverage amount represents the aggregate funds that will be available for claims within the TandaPay community for the upcoming period. **Example Code Block Using Ethers.js** Here is an example of how the getTotalCoverage function can be called using Ethers.js to interact with the TandaPay smart contract:  async function getTotalCoverage() {  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the getTotalCoverage function  const totalCoverage = await contract.getTotalCoverage();  console.log("Current Total Coverage Amount:", totalCoverage);  } catch (error) {  console.error("Error fetching total coverage amount:", error);  }  } else {  console.log("Please install MetaMask");  }  } **Example Code Block Using Wagmi** Here is an example of how the getTotalCoverage function can be called using Wagmi to interact with the TandaPay smart contract:  const getTotalCoverage = async () => {  try {  const data = await readContract({  address: contractAddress,  abi: contractABI,  functionName: "getTotalCoverage",  args: [],  chainId: expectedChainId,  });  console.log("Current Total Coverage Amount:", data);  return data;  } catch (error) {  console.error("Error fetching total coverage amount:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to dynamically display the current total coverage amount to users, helping them understand the financial capacity of the TandaPay community.  document.getElementById("fetchCoverageBtn").addEventListener("click", async () => {  const totalCoverage = await getTotalCoverage();  document.getElementById("coverageDisplay").innerText = `Current Total Coverage Amount: ${totalCoverage}`;  }); |
| --- |

| **Function Name:** getBasePremium  **Description:**  The getBasePremium function is a public view function in the TandaPay smart contract that retrieves the current base premium amount. This function does not require any arguments and can be called by any user. The base premium amount represents the standard periodic payment that each member needs to make to maintain their coverage within the TandaPay community. **Example Code Block Using Ethers.js** Here is an example of how the getBasePremium function can be called using Ethers.js to interact with the TandaPay smart contract:  async function getBasePremium() {  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the getBasePremium function  const basePremium = await contract.getBasePremium();  console.log("Current Base Premium Amount:", basePremium);  } catch (error) {  console.error("Error fetching base premium amount:", error);  }  } else {  console.log("Please install MetaMask");  }  } **Example Code Block Using Wagmi** Here is an example of how the getBasePremium function can be called using Wagmi to interact with the TandaPay smart contract:  const getBasePremium = async () => {  try {  const data = await readContract({  address: contractAddress,  abi: contractABI,  functionName: "getBasePremium",  args: [],  chainId: expectedChainId,  });  console.log("Current Base Premium Amount:", data);  return data;  } catch (error) {  console.error("Error fetching base premium amount:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to dynamically display the current base premium amount to users, helping them understand their periodic payment requirements.  document.getElementById("fetchBasePremiumBtn").addEventListener("click", async () => {  const basePremium = await getBasePremium();  document.getElementById("basePremiumDisplay").innerText = `Current Base Premium Amount: ${basePremium}`;  }); |
| --- |

| **Function Name:** getManuallyCollapsedPeriod  **Description:**  The getManuallyCollapsedPeriod function is a public view function in the TandaPay smart contract that retrieves the period ID in which a manual collapse occurred. This function does not require any arguments and can be called by any user. A manual collapse is an event where the community is manually transitioned into a collapsed state by the secretary, rather than through the regular process. **Example Code Block Using Ethers.js** Here is an example of how the getManuallyCollapsedPeriod function can be called using Ethers.js to interact with the TandaPay smart contract:  async function getManuallyCollapsedPeriod() {  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the getManuallyCollapsedPeriod function  const collapsedPeriodId = await contract.getManuallyCollapsedPeriod();  console.log("Manually Collapsed Period ID:", collapsedPeriodId);  } catch (error) {  console.error("Error fetching manually collapsed period ID:", error);  }  } else {  console.log("Please install MetaMask");  }  } **Example Code Block Using Wagmi** Here is an example of how the getManuallyCollapsedPeriod function can be called using Wagmi to interact with the TandaPay smart contract:  const getManuallyCollapsedPeriod = async () => {  try {  const data = await readContract({  address: contractAddress,  abi: contractABI,  functionName: "getManuallyCollapsedPeriod",  args: [],  chainId: expectedChainId,  });  console.log("Manually Collapsed Period ID:", data);  return data;  } catch (error) {  console.error("Error fetching manually collapsed period ID:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to dynamically display the manually collapsed period ID to users, helping them understand significant events in the community's operational history.  document.getElementById("fetchCollapsedPeriodBtn").addEventListener("click", async () => {  const collapsedPeriodId = await getManuallyCollapsedPeriod();  document.getElementById("collapsedPeriodDisplay").innerText = `Manually Collapsed Period ID: ${collapsedPeriodId}`;  }); |
| --- |

| **Function Name:** getIsManuallyCollapsed  **Description:**  The getIsManuallyCollapsed function is a public view function in the TandaPay smart contract that indicates whether a manual collapse has occurred. This function returns a boolean value: true if a manual collapse has happened, and false otherwise. It does not require any arguments and can be called by any user. **Use Case** The primary use case of the getIsManuallyCollapsed function is to provide real-time information about the operational state of the TandaPay community. This information is crucial for:   * **State Management:** Understanding whether the community is currently in a manually collapsed state. * **Decision Making:** Helping members and administrators make informed decisions based on the current state of the community. * **Transparency:** Providing clear and accessible information to all members about the community’s status.  **Why It's Important**  * **Operational Clarity:** Ensures that all members are aware of the community's current operational state, particularly in the event of a manual collapse. * **Informed Participation:** Allows members to make informed decisions about their participation and actions within the community. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications and other tools that need to reference the community’s state.  **Example Code Block Using Ethers.js** Here is an example of how the getIsManuallyCollapsed function can be called using Ethers.js to interact with the TandaPay smart contract:  async function getIsManuallyCollapsed() {  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the getIsManuallyCollapsed function  const isCollapsed = await contract.getIsManuallyCollapsed();  console.log("Is Manually Collapsed:", isCollapsed);  } catch (error) {  console.error("Error fetching manual collapse state:", error);  }  } else {  console.log("Please install MetaMask");  }  } **Example Code Block Using Wagmi** Here is an example of how the getIsManuallyCollapsed function can be called using Wagmi to interact with the TandaPay smart contract:  const getIsManuallyCollapsed = async () => {  try {  const data = await readContract({  address: contractAddress,  abi: contractABI,  functionName: "getIsManuallyCollapsed",  args: [],  chainId: expectedChainId,  });  console.log("Is Manually Collapsed:", data);  return data;  } catch (error) {  console.error("Error fetching manual collapse state:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to dynamically display whether a manual collapse has occurred, helping users understand the current operational state of the community.  document.getElementById("fetchIsCollapsedBtn").addEventListener("click", async () => {  const isCollapsed = await getIsManuallyCollapsed();  document.getElementById("isCollapsedDisplay").innerText = `Is Manually Collapsed: ${isCollapsed}`;  }); |
| --- |

| **Function Name:** getCommunityState  **Description:**  The getCommunityState function is a public view function in the TandaPay smart contract that retrieves the current state of the community. This function returns an integer representing the community's state: 0 for initialization, 1 for default, 2 for fractured, and 3 for collapsed. It does not require any arguments and can be called by any user. **Use Case** The primary use case of the getCommunityState function is to provide real-time information about the operational state of the TandaPay community. This information is crucial for:   * **State Management:** Understanding the current phase of the community's lifecycle. * **Decision Making:** Helping members and administrators make informed decisions based on the current state of the community. * **Transparency:** Providing clear and accessible information to all members about the community’s status.  **Why It's Important**  * **Operational Clarity:** Ensures that all members are aware of the community's current operational state. * **Informed Participation:** Allows members to make informed decisions about their participation and actions within the community. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications and other tools that need to reference the community’s state.  **Example Code Block Using Ethers.js** Here is an example of how the getCommunityState function can be called using Ethers.js to interact with the TandaPay smart contract:  async function getCommunityState() {  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the getCommunityState function  const communityState = await contract.getCommunityState();  console.log("Current Community State:", communityState);  } catch (error) {  console.error("Error fetching community state:", error);  }  } else {  console.log("Please install MetaMask");  }  } **Example Code Block Using Wagmi** Here is an example of how the getCommunityState function can be called using Wagmi to interact with the TandaPay smart contract:  const getCommunityState = async () => {  try {  const data = await readContract({  address: contractAddress,  abi: contractABI,  functionName: "getCommunityState",  args: [],  chainId: expectedChainId,  });  console.log("Current Community State:", data);  return data;  } catch (error) {  console.error("Error fetching community state:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to dynamically display the current community state to users, helping them understand the community's operational phase.  document.getElementById("fetchCommunityStateBtn").addEventListener("click", async () => {  const communityState = await getCommunityState();  document.getElementById("communityStateDisplay").innerText = `Current Community State: ${communityState}`;  }); |
| --- |

| **Function Name:** getSubGroupIdToSubGroupInfo  **Description:**  The getSubGroupIdToSubGroupInfo function is a public view function in the TandaPay smart contract that retrieves detailed information about a specific subgroup within the community. This function requires a subGroupId as an argument and returns an object containing the following information about the subgroup:   * **SubGroup ID:** The unique identifier of the subgroup. * **Members:** A list of member addresses in the subgroup. * **Status:** The current status of the subgroup. * **Other relevant details:** Additional information specific to the subgroup, such as creation date, assigned roles, and any custom attributes defined in the smart contract.  **Use Case** The primary use case of the getSubGroupIdToSubGroupInfo function is to provide detailed information about specific subgroups within the TandaPay community. This information is crucial for:   * **Subgroup Management:** Understanding the composition, roles, and status of different subgroups. * **Member Insights:** Providing members with insights into the subgroup they belong to or are interested in. * **Community Organization:** Helping in organizing and managing the community by tracking subgroup details.  **Why It's Important**  * **Detailed Information:** Offers detailed insights into the subgroups, enhancing transparency and trust within the community. * **Operational Clarity:** Ensures that all members have access to information about subgroups, promoting informed participation and decision-making. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications and other tools that need to reference subgroup information.  **Example Code Block Using Ethers.js** Here is an example of how the getSubGroupIdToSubGroupInfo function can be called using Ethers.js to interact with the TandaPay smart contract:  async function getSubGroupIdToSubGroupInfo(subGroupId) {  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the getSubGroupIdToSubGroupInfo function  const subGroupInfo = await contract.getSubGroupIdToSubGroupInfo(subGroupId);  console.log("SubGroup Info:", subGroupInfo);  } catch (error) {  console.error("Error fetching subgroup info:", error);  }  } else {  console.log("Please install MetaMask");  }  } **Example Code Block Using Wagmi** Here is an example of how the getSubGroupIdToSubGroupInfo function can be called using Wagmi to interact with the TandaPay smart contract:  const getSubGroupIdToSubGroupInfo = async (subGroupId) => {  try {  const data = await readContract({  address: contractAddress,  abi: contractABI,  functionName: "getSubGroupIdToSubGroupInfo",  args: [subGroupId],  chainId: expectedChainId,  });  console.log("SubGroup Info:", data);  return data;  } catch (error) {  console.error("Error fetching subgroup info:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to dynamically display detailed information about a specific subgroup to users, helping them understand the composition and status of different subgroups.  document.getElementById("fetchSubGroupInfoBtn").addEventListener("click", async () => {  const subGroupId = document.getElementById("subGroupIdInput").value;  const subGroupInfo = await getSubGroupIdToSubGroupInfo(subGroupId);  document.getElementById("subGroupInfoDisplay").innerText = `SubGroup Info: ${JSON.stringify(subGroupInfo)}`;  }); |
| --- |

| **Function Name:** getPeriodIdToClaimIdToClaimInfo  **Description:**  The getPeriodIdToClaimIdToClaimInfo function is a public view function in the TandaPay smart contract that retrieves detailed information about a specific claim within a given period. This function requires two arguments: periodId and claimId. It returns an object containing the following information about the claim:   * **Claim ID:** The unique identifier of the claim. * **Claimant:** The address of the member who submitted the claim. * **Amount:** The amount being claimed. * **Status:** The current status of the claim (e.g., pending, approved, rejected). * **Submission Date:** The date the claim was submitted. * **Other relevant details:** Additional information specific to the claim, such as supporting documents, reasons for the claim, and any custom attributes defined in the smart contract.  **Use Case** The primary use case of the getPeriodIdToClaimIdToClaimInfo function is to provide detailed information about specific claims within the TandaPay community. This information is crucial for:   * **Claim Management:** Understanding the details of claims submitted during specific periods. * **Member Insights:** Providing members and administrators with insights into the claims process and individual claims. * **Community Organization:** Helping in organizing and managing the community by tracking claim details and their statuses.  **Why It's Important**  * **Detailed Information:** Offers detailed insights into claims, enhancing transparency and trust within the community. * **Operational Clarity:** Ensures that all members have access to information about claims, promoting informed participation and decision-making. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications and other tools that need to reference claim information.  **Example Code Block Using Ethers.js** Here is an example of how the getPeriodIdToClaimIdToClaimInfo function can be called using Ethers.js to interact with the TandaPay smart contract:  async function getPeriodIdToClaimIdToClaimInfo(periodId, claimId) {  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the getPeriodIdToClaimIdToClaimInfo function  const claimInfo = await contract.getPeriodIdToClaimIdToClaimInfo(periodId, claimId);  console.log("Claim Info:", claimInfo);  } catch (error) {  console.error("Error fetching claim info:", error);  }  } else {  console.log("Please install MetaMask");  }  } **Example Code Block Using Wagmi** Here is an example of how the getPeriodIdToClaimIdToClaimInfo function can be called using Wagmi to interact with the TandaPay smart contract:  const getPeriodIdToClaimIdToClaimInfo = async (periodId, claimId) => {  try {  const data = await readContract({  address: contractAddress,  abi: contractABI,  functionName: "getPeriodIdToClaimIdToClaimInfo",  args: [periodId, claimId],  chainId: expectedChainId,  });  console.log("Claim Info:", data);  return data;  } catch (error) {  console.error("Error fetching claim info:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to dynamically display detailed information about a specific claim to users, helping them understand the status and details of their claims.  document.getElementById("fetchClaimInfoBtn").addEventListener("click", async () => {  const periodId = document.getElementById("periodIdInput").value;  const claimId = document.getElementById("claimIdInput").value;  const claimInfo = await getPeriodIdToClaimIdToClaimInfo(periodId, claimId);  document.getElementById("claimInfoDisplay").innerText = `Claim Info: ${JSON.stringify(claimInfo)}`;  }); |
| --- |

| **Function Name:** getPeriodIdToClaimIds  **Description:**  The getPeriodIdToClaimIds function is a public view function in the TandaPay smart contract that retrieves a list of claim IDs for a specific period. This function requires one argument: periodId. It returns an array of claim IDs that were submitted during the specified period. **Use Case** The primary use case of the getPeriodIdToClaimIds function is to provide a way to track and manage all claims submitted within a specific period in the TandaPay community. This information is crucial for:   * **Claim Management:** Keeping track of all claims submitted during a specific period. * **Administrative Insights:** Providing administrators with an overview of the claims activity within each period. * **Community Organization:** Helping in organizing and resolving claims efficiently by period.  **Why It's Important**  * **Detailed Information:** Offers detailed insights into the claims activity within specific periods, enhancing transparency and trust within the community. * **Operational Clarity:** Ensures that all members and administrators have access to information about the volume and timing of claims. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications and other tools that need to reference claim information by period.  **Example Code Block Using Ethers.js** Here is an example of how the getPeriodIdToClaimIds function can be called using Ethers.js to interact with the TandaPay smart contract:  async function getPeriodIdToClaimIds(periodId) {  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the getPeriodIdToClaimIds function  const claimIds = await contract.getPeriodIdToClaimIds(periodId);  console.log("Claim IDs for Period", periodId, ":", claimIds);  } catch (error) {  console.error("Error fetching claim IDs:", error);  }  } else {  console.log("Please install MetaMask");  }  } **Example Code Block Using Wagmi** Here is an example of how the getPeriodIdToClaimIds function can be called using Wagmi to interact with the TandaPay smart contract:  const getPeriodIdToClaimIds = async (periodId) => {  try {  const data = await readContract({  address: contractAddress,  abi: contractABI,  functionName: "getPeriodIdToClaimIds",  args: [periodId],  chainId: expectedChainId,  });  console.log("Claim IDs for Period", periodId, ":", data);  return data;  } catch (error) {  console.error("Error fetching claim IDs:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to dynamically display the list of claim IDs for a specific period to users, helping them understand the claims activity within each period.  document.getElementById("fetchClaimIdsBtn").addEventListener("click", async () => {  const periodId = document.getElementById("periodIdInput").value;  const claimIds = await getPeriodIdToClaimIds(periodId);  document.getElementById("claimIdsDisplay").innerText = `Claim IDs for Period ${periodId}: ${claimIds.join(", ")}`;  }); |
| --- |

| **Function Name:** getPeriodIdToDefectorsId  **Description:**  The getPeriodIdToDefectorsId function is a public view function in the TandaPay smart contract that retrieves a list of defectors' IDs for a specific period. This function requires one argument: periodId. It returns an array of member IDs who defected during the specified period. **Use Case** The primary use case of the getPeriodIdToDefectorsId function is to provide a way to track and manage members who have defected from the TandaPay community during a specific period. This information is crucial for:   * **Member Management:** Understanding the number and identity of members who have chosen to leave the community in a given period. * **Operational Analysis:** Analyzing trends and reasons behind defections to improve community retention. * **Community Organization:** Helping in organizing and managing the community by tracking membership changes.  **Why It's Important**  * **Detailed Information:** Offers a comprehensive list of defectors for each period, enhancing transparency and trust within the community. * **Operational Clarity:** Ensures that all members have access to information about defections, promoting informed participation and decision-making. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications and other tools that need to reference defectors by period.  **Example Code Block Using Ethers.js** Here is an example of how the getPeriodIdToDefectorsId function can be called using Ethers.js to interact with the TandaPay smart contract:  async function getPeriodIdToDefectorsId(periodId) {  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the getPeriodIdToDefectorsId function  const defectorsIds = await contract.getPeriodIdToDefectorsId(periodId);  console.log("Defectors IDs for Period", periodId, ":", defectorsIds);  } catch (error) {  console.error("Error fetching defectors IDs:", error);  }  } else {  console.log("Please install MetaMask");  }  } **Example Code Block Using Wagmi** Here is an example of how the getPeriodIdToDefectorsId function can be called using Wagmi to interact with the TandaPay smart contract:  const getPeriodIdToDefectorsId = async (periodId) => {  try {  const data = await readContract({  address: contractAddress,  abi: contractABI,  functionName: "getPeriodIdToDefectorsId",  args: [periodId],  chainId: expectedChainId,  });  console.log("Defectors IDs for Period", periodId, ":", data);  return data;  } catch (error) {  console.error("Error fetching defectors IDs:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to dynamically display the list of defectors' IDs for a specific period, helping users understand the membership changes during that period.  document.getElementById("fetchDefectorsIdsBtn").addEventListener("click", async () => {  const periodId = document.getElementById("periodIdInput").value;  const defectorsIds = await getPeriodIdToDefectorsId(periodId);  document.getElementById("defectorsIdsDisplay").innerText = `Defectors IDs for Period ${periodId}: ${defectorsIds.join(", ")}`;  }); |
| --- |

| **Function Name:** getPeriodIdToManualCollapse  **Description:**  The getPeriodIdToManualCollapse function is a public view function in the TandaPay smart contract that retrieves information about a manual collapse for a specific period. This function requires one argument: periodId. It returns an object containing details about the manual collapse that occurred during the specified period.  The information returned includes:   * **Period ID:** The unique identifier of the period. * **Manual Collapse Details:** Information about the manual collapse, such as the reason for collapse, the date of collapse, and any actions taken by the community or administrators.  **Use Case** The primary use case of the getPeriodIdToManualCollapse function is to provide detailed information about manual collapses that have occurred within specific periods in the TandaPay community. This information is crucial for:   * **Operational Analysis:** Understanding the reasons and consequences of manual collapses to improve future operations. * **Community Transparency:** Providing members with clear information about significant events affecting the community’s state. * **Historical Tracking:** Keeping a record of manual collapses for future reference and analysis.  **Why It's Important**  * **Detailed Information:** Offers comprehensive details about manual collapses, enhancing transparency and trust within the community. * **Operational Clarity:** Ensures that all members have access to information about significant events, promoting informed participation and decision-making. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications and other tools that need to reference manual collapse information.  **Example Code Block Using Ethers.js** Here is an example of how the getPeriodIdToManualCollapse function can be called using Ethers.js to interact with the TandaPay smart contract:  async function getPeriodIdToManualCollapse(periodId) {  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the getPeriodIdToManualCollapse function  const collapseInfo = await contract.getPeriodIdToManualCollapse(periodId);  console.log("Manual Collapse Info for Period", periodId, ":", collapseInfo);  } catch (error) {  console.error("Error fetching manual collapse info:", error);  }  } else {  console.log("Please install MetaMask");  }  } **Example Code Block Using Wagmi** Here is an example of how the getPeriodIdToManualCollapse function can be called using Wagmi to interact with the TandaPay smart contract:  const getPeriodIdToManualCollapse = async (periodId) => {  try {  const data = await readContract({  address: contractAddress,  abi: contractABI,  functionName: "getPeriodIdToManualCollapse",  args: [periodId],  chainId: expectedChainId,  });  console.log("Manual Collapse Info for Period", periodId, ":", data);  return data;  } catch (error) {  console.error("Error fetching manual collapse info:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to dynamically display information about manual collapse events for a specific period, helping users understand the community's operational history.  document.getElementById("fetchCollapseInfoBtn").addEventListener("click", async () => {  const periodId = document.getElementById("periodIdInput").value;  const collapseInfo = await getPeriodIdToManualCollapse(periodId);  document.getElementById("collapseInfoDisplay").innerText = `Manual Collapse Info for Period ${periodId}: ${JSON.stringify(collapseInfo)}`;  }); |
| --- |

| **Function Name:** getMemberToMemberId  **Description:**  The getMemberToMemberId function is a public view function in the TandaPay smart contract that retrieves the unique member ID associated with a given wallet address. This function requires one argument: memberAddress. It returns the member ID corresponding to the specified wallet address. **Information Returned**  * **Member ID:** The unique identifier associated with the provided wallet address.  **Use Case** The primary use case of the getMemberToMemberId function is to provide a way to look up the unique member ID for a given wallet address within the TandaPay community. This information is crucial for:   * **Member Identification:** Allowing members and administrators to identify and reference members using their unique IDs. * **Transaction and Interaction Tracking:** Ensuring that all interactions and transactions involving members are correctly associated with the right member ID. * **Community Organization:** Helping in organizing and managing the community by tracking member details.  **Why It's Important**  * **Member Identification:** Ensures that each member can be uniquely identified and referenced within the community. * **Operational Clarity:** Helps in maintaining an organized structure within the TandaPay community by uniquely identifying each member. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications and other tools that need to reference specific members.  **Example Code Block Using Ethers.js** Here is an example of how the getMemberToMemberId function can be called using Ethers.js to interact with the TandaPay smart contract:  async function getMemberToMemberId(memberAddress) {  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the getMemberToMemberId function  const memberId = await contract.getMemberToMemberId(memberAddress);  console.log("Member ID for Address", memberAddress, ":", memberId);  } catch (error) {  console.error("Error fetching member ID:", error);  }  } else {  console.log("Please install MetaMask");  }  } **Example Code Block Using Wagmi** Here is an example of how the getMemberToMemberId function can be called using Wagmi to interact with the TandaPay smart contract:  const getMemberToMemberId = async (memberAddress) => {  try {  const data = await readContract({  address: contractAddress,  abi: contractABI,  functionName: "getMemberToMemberId",  args: [memberAddress],  chainId: expectedChainId,  });  console.log("Member ID for Address", memberAddress, ":", data);  return data;  } catch (error) {  console.error("Error fetching member ID:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to dynamically display the member ID for a given wallet address, helping users understand their unique identifier within the TandaPay community.  document.getElementById("fetchMemberIdBtn").addEventListener("click", async () => {  const memberAddress = document.getElementById("memberAddressInput").value;  const memberId = await getMemberToMemberId(memberAddress);  document.getElementById("memberIdDisplay").innerText = `Member ID for Address ${memberAddress}: ${memberId}`;  }); |
| --- |

| **Function Name:** getPeriodIdWhiteListedClaims  **Description:**  The getPeriodIdWhiteListedClaims function is a public view function in the TandaPay smart contract that retrieves a list of whitelisted claim IDs for a specific period. This function requires one argument: periodId. It returns an array of claim IDs that have been whitelisted during the specified period. **Information Returned**  * **Whitelisted Claim IDs:** An array of claim IDs that have been approved (whitelisted) for the given period.  **Use Case** The primary use case of the getPeriodIdWhiteListedClaims function is to provide a way to track and manage all whitelisted claims within a specific period in the TandaPay community. This information is crucial for:   * **Claim Management:** Understanding the volume and specifics of claims that have been approved in each period. * **Member Insights:** Providing members and administrators with an overview of whitelisted claims for review and tracking. * **Community Organization:** Helping in organizing and resolving claims efficiently by period.  **Why It's Important**  * **Detailed Information:** Offers a comprehensive list of whitelisted claims for each period, enhancing transparency and trust within the community. * **Operational Clarity:** Ensures that all members have access to information about whitelisted claims, promoting informed participation and decision-making. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications and other tools that need to reference whitelisted claims by period.  **Example Code Block Using Ethers.js** Here is an example of how the getPeriodIdWhiteListedClaims function can be called using Ethers.js to interact with the TandaPay smart contract:  async function getPeriodIdWhiteListedClaims(periodId) {  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the getPeriodIdWhiteListedClaims function  const claimIds = await contract.getPeriodIdWhiteListedClaims(periodId);  console.log("Whitelisted Claim IDs for Period", periodId, ":", claimIds);  } catch (error) {  console.error("Error fetching whitelisted claim IDs:", error);  }  } else {  console.log("Please install MetaMask");  }  } **Example Code Block Using Wagmi** Here is an example of how the getPeriodIdWhiteListedClaims function can be called using Wagmi to interact with the TandaPay smart contract:  const getPeriodIdWhiteListedClaims = async (periodId) => {  try {  const data = await readContract({  address: contractAddress,  abi: contractABI,  functionName: "getPeriodIdWhiteListedClaims",  args: [periodId],  chainId: expectedChainId,  });  console.log("Whitelisted Claim IDs for Period", periodId, ":", data);  return data;  } catch (error) {  console.error("Error fetching whitelisted claim IDs:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to dynamically display the list of whitelisted claim IDs for a specific period, helping users understand the volume and details of claims that have been approved during that period.  document.getElementById("fetchWhitelistedClaimsBtn").addEventListener("click", async () => {  const periodId = document.getElementById("periodIdInput").value;  const claimIds = await getPeriodIdWhiteListedClaims(periodId);  document.getElementById("whitelistedClaimsDisplay").innerText = `Whitelisted Claim IDs for Period ${periodId}: ${claimIds.join(", ")}`;  }); |
| --- |

| **Function Name:** getMemberToMemberInfo  **Description:**  The getMemberToMemberInfo function is a public view function in the TandaPay smart contract that retrieves detailed information about a specific member. This function requires two arguments: memberAddress and expectedPeriodId. It returns an object containing the following information about the member:   * **Member ID:** The unique identifier of the member. * **Wallet Address:** The wallet address of the member. * **Status:** The current status of the member (e.g., active, inactive). * **Period Joined:** The period in which the member joined. * **SubGroup ID:** The ID of the subgroup to which the member belongs. * **Other relevant details:** Additional information specific to the member, such as their role, contributions, and any custom attributes defined in the smart contract.  **Use Case** The primary use case of the getMemberToMemberInfo function is to provide detailed information about specific members within the TandaPay community. This information is crucial for:   * **Member Management:** Understanding the roles, status, and contributions of different members. * **Member Insights:** Providing members with insights into their status and details within the community. * **Community Organization:** Helping in organizing and managing the community by tracking member details.  **Why It's Important**  * **Detailed Information:** Offers detailed insights into members, enhancing transparency and trust within the community. * **Operational Clarity:** Ensures that all members have access to information about their status and contributions, promoting informed participation and decision-making. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications and other tools that need to reference member information.  **Example Code Block Using Ethers.js** Here is an example of how the getMemberToMemberInfo function can be called using Ethers.js to interact with the TandaPay smart contract:  async function getMemberToMemberInfo(memberAddress, expectedPeriodId) {  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the getMemberToMemberInfo function  const memberInfo = await contract.getMemberToMemberInfo(memberAddress, expectedPeriodId);  console.log("Member Info for Address", memberAddress, ":", memberInfo);  } catch (error) {  console.error("Error fetching member info:", error);  }  } else {  console.log("Please install MetaMask");  }  } **Example Code Block Using Wagmi** Here is an example of how the getMemberToMemberInfo function can be called using Wagmi to interact with the TandaPay smart contract:  const getMemberToMemberInfo = async (memberAddress, expectedPeriodId) => {  try {  const data = await readContract({  address: contractAddress,  abi: contractABI,  functionName: "getMemberToMemberInfo",  args: [memberAddress, expectedPeriodId],  chainId: expectedChainId,  });  console.log("Member Info for Address", memberAddress, ":", data);  return data;  } catch (error) {  console.error("Error fetching member info:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to dynamically display detailed information about a specific member to users, helping them understand their status and contributions within the TandaPay community.  document.getElementById("fetchMemberInfoBtn").addEventListener("click", async () => {  const memberAddress = document.getElementById("memberAddressInput").value;  const expectedPeriodId = document.getElementById("expectedPeriodIdInput").value;  const memberInfo = await getMemberToMemberInfo(memberAddress, expectedPeriodId);  document.getElementById("memberInfoDisplay").innerText = `Member Info for Address ${memberAddress}: ${JSON.stringify(memberInfo)}`;  }); |
| --- |

| **Function Name:** getIsAMemberDefectedInPeriod  **Description:**  The getIsAMemberDefectedInPeriod function is a public view function in the TandaPay smart contract that checks if a specific member has defected during a specific period. This function requires two arguments: memberAddress and periodId. It returns a boolean value: true if the member defected during the specified period, and false otherwise. **Information Returned**  * **Defection Status:** A boolean value indicating whether the member has defected during the specified period.  **Use Case** The primary use case of the getIsAMemberDefectedInPeriod function is to provide a way to track and manage member defections within specific periods in the TandaPay community. This information is crucial for:   * **Member Management:** Understanding the defection status of members during specific periods. * **Operational Analysis:** Analyzing trends and reasons behind defections to improve community retention. * **Community Organization:** Helping in organizing and managing the community by tracking membership changes.  **Why It's Important**  * **Detailed Information:** Offers specific insights into member defections for each period, enhancing transparency and trust within the community. * **Operational Clarity:** Ensures that all members have access to information about defections, promoting informed participation and decision-making. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications and other tools that need to reference member defection status by period.  **Example Code Block Using Ethers.js** Here is an example of how the getIsAMemberDefectedInPeriod function can be called using Ethers.js to interact with the TandaPay smart contract:  async function getIsAMemberDefectedInPeriod(memberAddress, periodId) {  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the getIsAMemberDefectedInPeriod function  const isDefected = await contract.getIsAMemberDefectedInPeriod(memberAddress, periodId);  console.log(`Member ${memberAddress} defection status in period ${periodId}:`, isDefected);  } catch (error) {  console.error("Error fetching defection status:", error);  }  } else {  console.log("Please install MetaMask");  }  } **Example Code Block Using Wagmi** Here is an example of how the getIsAMemberDefectedInPeriod function can be called using Wagmi to interact with the TandaPay smart contract:  const getIsAMemberDefectedInPeriod = async (memberAddress, periodId) => {  try {  const data = await readContract({  address: contractAddress,  abi: contractABI,  functionName: "getIsAMemberDefectedInPeriod",  args: [memberAddress, periodId],  chainId: expectedChainId,  });  console.log(`Member ${memberAddress} defection status in period ${periodId}:`, data);  return data;  } catch (error) {  console.error("Error fetching defection status:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to dynamically display the defection status of a specific member for a specific period, helping users understand membership changes.  document.getElementById("fetchDefectionStatusBtn").addEventListener("click", async () => {  const memberAddress = document.getElementById("memberAddressInput").value;  const periodId = document.getElementById("periodIdInput").value;  const isDefected = await getIsAMemberDefectedInPeriod(memberAddress, periodId);  document.getElementById("defectionStatusDisplay").innerText = `Member ${memberAddress} defection status in period ${periodId}: ${isDefected}`;  }); |
| --- |

| **Function Name:** getPeriodIdToPeriodInfo  **Description:**  The getPeriodIdToPeriodInfo function is a public view function in the TandaPay smart contract that retrieves detailed information about a specific period. This function requires one argument: periodId. It returns an object containing the following information about the period:   * **Period ID:** The unique identifier of the period. * **Start Date:** The date when the period began. * **End Date:** The date when the period ended or is expected to end. * **Total Claims:** The total number of claims submitted during the period. * **Total Defectors:** The total number of members who defected during the period. * **Status:** The current status of the period (e.g., active, completed, collapsed). * **Other relevant details:** Additional information specific to the period, such as the total premiums collected, payouts made, and any custom attributes defined in the smart contract.  **Use Case** The primary use case of the getPeriodIdToPeriodInfo function is to provide detailed information about specific periods within the TandaPay community. This information is crucial for:   * **Period Management:** Understanding the start and end dates, status, and activities within each period. * **Operational Analysis:** Analyzing trends and events within periods to improve future community operations. * **Community Organization:** Helping in organizing and managing the community by tracking period details.  **Why It's Important**  * **Detailed Information:** Offers detailed insights into periods, enhancing transparency and trust within the community. * **Operational Clarity:** Ensures that all members have access to information about the community’s periods, promoting informed participation and decision-making. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications and other tools that need to reference period information.  **Example Code Block Using Ethers.js** Here is an example of how the getPeriodIdToPeriodInfo function can be called using Ethers.js to interact with the TandaPay smart contract:  async function getPeriodIdToPeriodInfo(periodId) {  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the getPeriodIdToPeriodInfo function  const periodInfo = await contract.getPeriodIdToPeriodInfo(periodId);  console.log("Period Info for Period ID", periodId, ":", periodInfo);  } catch (error) {  console.error("Error fetching period info:", error);  }  } else {  console.log("Please install MetaMask");  }  } **Example Code Block Using Wagmi** Here is an example of how the getPeriodIdToPeriodInfo function can be called using Wagmi to interact with the TandaPay smart contract:  const getPeriodIdToPeriodInfo = async (periodId) => {  try {  const data = await readContract({  address: contractAddress,  abi: contractABI,  functionName: "getPeriodIdToPeriodInfo",  args: [periodId],  chainId: expectedChainId,  });  console.log("Period Info for Period ID", periodId, ":", data);  return data;  } catch (error) {  console.error("Error fetching period info:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to dynamically display detailed information about a specific period to users, helping them understand the community's operational history and current status.  document.getElementById("fetchPeriodInfoBtn").addEventListener("click", async () => {  const periodId = document.getElementById("periodIdInput").value;  const periodInfo = await getPeriodIdToPeriodInfo(periodId);  document.getElementById("periodInfoDisplay").innerText = `Period Info for Period ID ${periodId}: ${JSON.stringify(periodInfo)}`;  }); |
| --- |

| **Function Name:** getIsAllMemberNotPaidInPeriod  **Description:**  The getIsAllMemberNotPaidInPeriod function is a public view function in the TandaPay smart contract that checks if all members have not paid their premiums in a specific period. This function requires one argument: periodId. It returns a boolean value: true if all members have not paid their premiums during the specified period, and false otherwise. **Information Returned**  * **Payment Status:** A boolean value indicating whether all members have not paid their premiums in the specified period.  **Use Case** The primary use case of the getIsAllMemberNotPaidInPeriod function is to provide a way to track and manage premium payments within specific periods in the TandaPay community. This information is crucial for:   * **Financial Management:** Understanding the payment compliance of members in each period. * **Operational Analysis:** Analyzing trends and issues in payment compliance to improve future community operations. * **Community Organization:** Helping in organizing and managing the community by tracking payment statuses.  **Why It's Important**  * **Payment Compliance:** Ensures that the community is aware of the payment compliance status of its members. * **Operational Clarity:** Provides clear information about premium payment statuses, promoting informed decision-making and participation. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications and other tools that need to reference payment statuses by period.  **Example Code Block Using Ethers.js** Here is an example of how the getIsAllMemberNotPaidInPeriod function can be called using Ethers.js to interact with the TandaPay smart contract:  async function getIsAllMemberNotPaidInPeriod(periodId) {  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the getIsAllMemberNotPaidInPeriod function  const isNotPaid = await contract.getIsAllMemberNotPaidInPeriod(periodId);  console.log(`All members not paid status in period ${periodId}:`, isNotPaid);  } catch (error) {  console.error("Error fetching payment status:", error);  }  } else {  console.log("Please install MetaMask");  }  } **Example Code Block Using Wagmi** Here is an example of how the getIsAllMemberNotPaidInPeriod function can be called using Wagmi to interact with the TandaPay smart contract:  const getIsAllMemberNotPaidInPeriod = async (periodId) => {  try {  const data = await readContract({  address: contractAddress,  abi: contractABI,  functionName: "getIsAllMemberNotPaidInPeriod",  args: [periodId],  chainId: expectedChainId,  });  console.log(`All members not paid status in period ${periodId}:`, data);  return data;  } catch (error) {  console.error("Error fetching payment status:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to dynamically display the payment compliance status of members for a specific period, helping users understand the community's financial health.  document.getElementById("fetchPaymentStatusBtn").addEventListener("click", async () => {  const periodId = document.getElementById("periodIdInput").value;  const isNotPaid = await getIsAllMemberNotPaidInPeriod(periodId);  document.getElementById("paymentStatusDisplay").innerText = `All members not paid status in period ${periodId}: ${isNotPaid}`;  }); |
| --- |

**All the functions to send transaction to the smart contract**

**By secretary**

| **Function Name:** addToCommunity  **Description:**  The addToCommunity function is a write function in the TandaPay smart contract that allows the secretary to add a new member to the community. This function requires one argument: the wallet address of the new member. By invoking this function, the secretary can expand the community by including new participants. **Use Case** The primary use case of the addToCommunity function is to enable the growth of the TandaPay community by adding new members. This action is crucial for:   * **Community Expansion:** Allowing the community to grow by including new members. * **Member Management:** Keeping track of new members joining the community. * **Operational Scalability:** Ensuring the community can scale and accommodate more participants.  **Why It's Important**  * **Community Growth:** Helps in expanding the community, making it more robust and diverse. * **Member Tracking:** Ensures that new members are formally recognized and integrated into the community structure. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications that manage member additions.  **Example Code Block Using Ethers.js** Here is an example of how the addToCommunity function can be called using Ethers.js to interact with the TandaPay smart contract:  async function addToCommunity(memberAddress) {  console.log(`Adding ${memberAddress} to the community...`);  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Request accounts from the user  await provider.send("eth\_requestAccounts", []);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the addToCommunity function  const transactionResponse = await contract.addToCommunity(memberAddress);  await listenForTransactionMine(transactionResponse, provider);  console.log("Member added successfully.");  } catch (error) {  console.error("Error adding member to the community:", error);  }  } else {  console.log("Please install MetaMask");  }  }  async function listenForTransactionMine(transactionResponse, provider) {  console.log(`Mining ${transactionResponse.hash}...`);  return new Promise((resolve, reject) => {  provider.once(transactionResponse.hash, (transactionReceipt) => {  console.log(`Completed with ${transactionReceipt.confirmations} confirmations.`);  resolve();  });  });  } **Example Code Block Using Wagmi** Here is an example of how the addToCommunity function can be called using Wagmi to interact with the TandaPay smart contract:  import { getAccount, prepareWriteContract, writeContract, waitForTransaction } from '@wagmi/core';  const addToCommunity = async (memberAddress) => {  try {  const account = getAccount();  const config = await prepareWriteContract({  address: contractAddress,  abi: contractABI,  functionName: "addToCommunity",  args: [memberAddress],  chainId: expectedChainId,  account: account,  });  const { hash } = await writeContract(config);  const data = await waitForTransaction({  confirmations: 1,  chainId: expectedChainId,  hash: hash,  });  console.log("Transaction successful:", data);  } catch (error) {  console.error("Error adding member to the community:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to add new members to the TandaPay community through a user interface, ensuring the community grows and operates smoothly.  <button id="addMemberBtn">Add Member</button>  <input type="text" id="memberAddressInput" placeholder="Enter member address" />  <script>  document.getElementById("addMemberBtn").addEventListener("click", async () => {  const memberAddress = document.getElementById("memberAddressInput").value;  await addToCommunity(memberAddress);  });  </script> |
| --- |

| **Function Name:** createSubGroup  **Description:**  The createSubGroup function is a write function in the TandaPay smart contract that allows the secretary to create a new subgroup within the community. This function does not require any arguments. By invoking this function, the secretary can organize members into smaller, manageable groups for better governance and accountability. **Use Case** The primary use case of the createSubGroup function is to enable the organization of the TandaPay community into smaller subgroups. This action is crucial for:   * **Community Organization:** Structuring the community into subgroups for better management and accountability. * **Operational Efficiency:** Ensuring that each subgroup can function autonomously while still being part of the larger community. * **Member Management:** Helping in the allocation and distribution of members into subgroups for specific purposes or roles.  **Why It's Important**  * **Structured Growth:** Helps in maintaining an organized community structure as the number of members grows. * **Improved Governance:** Ensures that the community can govern itself more effectively by managing smaller groups. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications that manage subgroup creation.  **Example Code Block Using Ethers.js** Here is an example of how the createSubGroup function can be called using Ethers.js to interact with the TandaPay smart contract:  async function createSubGroup() {  console.log(`Creating a new subgroup...`);  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Request accounts from the user  await provider.send("eth\_requestAccounts", []);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the createSubGroup function  const transactionResponse = await contract.createSubGroup();  await listenForTransactionMine(transactionResponse, provider);  console.log("Subgroup created successfully.");  } catch (error) {  console.error("Error creating subgroup:", error);  }  } else {  console.log("Please install MetaMask");  }  }  async function listenForTransactionMine(transactionResponse, provider) {  console.log(`Mining ${transactionResponse.hash}...`);  return new Promise((resolve, reject) => {  provider.once(transactionResponse.hash, (transactionReceipt) => {  console.log(`Completed with ${transactionReceipt.confirmations} confirmations.`);  resolve();  });  });  } **Example Code Block Using Wagmi** Here is an example of how the createSubGroup function can be called using Wagmi to interact with the TandaPay smart contract:  import { getAccount, prepareWriteContract, writeContract, waitForTransaction } from '@wagmi/core';  const createSubGroup = async () => {  try {  const account = getAccount();  const config = await prepareWriteContract({  address: contractAddress,  abi: contractABI,  functionName: "createSubGroup",  chainId: expectedChainId,  account: account,  });  const { hash } = await writeContract(config);  const data = await waitForTransaction({  confirmations: 1,  chainId: expectedChainId,  hash: hash,  });  console.log("Subgroup created successfully:", data);  } catch (error) {  console.error("Error creating subgroup:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to create new subgroups within the TandaPay community through a user interface, ensuring the community is organized and manageable.  <button id="createSubGroupBtn">Create Subgroup</button>  <script>  document.getElementById("createSubGroupBtn").addEventListener("click", async () => {  await createSubGroup();  });  </script> |
| --- |

| **Function Name:** assignToSubGroup  **Description:**  The assignToSubGroup function is a write function in the TandaPay smart contract that allows the secretary to assign a member to a specific subgroup. This function requires three arguments:   * **memberAddress:** The wallet address of the member to be assigned. * **subGroupId:** The ID of the subgroup to which the member will be assigned. * **isReorg:** A boolean value indicating whether the assignment is part of a reorganization.   By invoking this function, the secretary can effectively distribute members into different subgroups for better management and governance. **Use Case** The primary use case of the assignToSubGroup function is to enable the organization and management of the TandaPay community by assigning members to specific subgroups. This action is crucial for:   * **Community Organization:** Structuring the community into manageable subgroups. * **Operational Efficiency:** Ensuring each subgroup has the necessary members to function effectively. * **Member Management:** Allocating and redistributing members as needed for various roles or purposes within the community.  **Why It's Important**  * **Structured Growth:** Helps in maintaining an organized community structure as the number of members grows. * **Improved Governance:** Ensures that the community can govern itself more effectively by managing smaller groups. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications that manage subgroup assignments.  **Example Code Block Using Ethers.js** Here is an example of how the assignToSubGroup function can be called using Ethers.js to interact with the TandaPay smart contract:  async function assignToSubGroup(memberAddress, subGroupId, isReorg) {  console.log(`Assigning ${memberAddress} to subgroup ${subGroupId}...`);  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Request accounts from the user  await provider.send("eth\_requestAccounts", []);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the assignToSubGroup function  const transactionResponse = await contract.assignToSubGroup(memberAddress, subGroupId, isReorg);  await listenForTransactionMine(transactionResponse, provider);  console.log("Member assigned to subgroup successfully.");  } catch (error) {  console.error("Error assigning member to subgroup:", error);  }  } else {  console.log("Please install MetaMask");  }  }  async function listenForTransactionMine(transactionResponse, provider) {  console.log(`Mining ${transactionResponse.hash}...`);  return new Promise((resolve, reject) => {  provider.once(transactionResponse.hash, (transactionReceipt) => {  console.log(`Completed with ${transactionReceipt.confirmations} confirmations.`);  resolve();  });  });  } **Example Code Block Using Wagmi** Here is an example of how the assignToSubGroup function can be called using Wagmi to interact with the TandaPay smart contract:  import { getAccount, prepareWriteContract, writeContract, waitForTransaction } from '@wagmi/core';  const assignToSubGroup = async (memberAddress, subGroupId, isReorg) => {  try {  const account = getAccount();  const config = await prepareWriteContract({  address: contractAddress,  abi: contractABI,  functionName: "assignToSubGroup",  args: [memberAddress, subGroupId, isReorg],  chainId: expectedChainId,  account: account,  });  const { hash } = await writeContract(config);  const data = await waitForTransaction({  confirmations: 1,  chainId: expectedChainId,  hash: hash,  });  console.log("Member assigned to subgroup successfully:", data);  } catch (error) {  console.error("Error assigning member to subgroup:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to assign members to subgroups within the TandaPay community through a user interface, ensuring the community is organized and manageable.  <button id="assignToSubGroupBtn">Assign to Subgroup</button>  <input type="text" id="memberAddressInput" placeholder="Enter member address" />  <input type="text" id="subGroupIdInput" placeholder="Enter subgroup ID" />  <input type="checkbox" id="isReorgInput" /> Is Reorg  <script>  document.getElementById("assignToSubGroupBtn").addEventListener("click", async () => {  const memberAddress = document.getElementById("memberAddressInput").value;  const subGroupId = document.getElementById("subGroupIdInput").value;  const isReorg = document.getElementById("isReorgInput").checked;  await assignToSubGroup(memberAddress, subGroupId, isReorg);  });  </script> |
| --- |

| **Function Name:** initiatDefaultStateAndSetCoverage  **Description:**  The initiatDefaultStateAndSetCoverage function is a write function in the TandaPay smart contract that initializes the default state of the community and sets the coverage amount. This function requires one argument:   * **coverageAmount:** The amount of coverage to be set for the community.   By invoking this function, the secretary can initialize the community's default state and establish the coverage amount that members will be eligible for. **Use Case** The primary use case of the initiatDefaultStateAndSetCoverage function is to initialize the community's default state and define the coverage amount. This action is crucial for:   * **Community Initialization:** Setting up the community in its default state at the start. * **Coverage Definition:** Defining the coverage amount that members will be eligible for in case of claims. * **Operational Readiness:** Ensuring that the community is ready to function with defined coverage.  **Why It's Important**  * **Community Setup:** Helps in establishing the initial setup of the community, making it operational. * **Defined Coverage:** Ensures that members are aware of the coverage amount they are eligible for. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications that manage community initialization and coverage settings.  **Example Code Block Using Ethers.js** Here is an example of how the initiatDefaultStateAndSetCoverage function can be called using Ethers.js to interact with the TandaPay smart contract:  async function initiatDefaultStateAndSetCoverage(coverageAmount) {  console.log(`Initializing default state and setting coverage to ${coverageAmount}...`);  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Request accounts from the user  await provider.send("eth\_requestAccounts", []);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the initiatDefaultStateAndSetCoverage function  const transactionResponse = await contract.initiatDefaultStateAndSetCoverage(coverageAmount);  await listenForTransactionMine(transactionResponse, provider);  console.log("Default state initialized and coverage set successfully.");  } catch (error) {  console.error("Error initializing default state and setting coverage:", error);  }  } else {  console.log("Please install MetaMask");  }  }  async function listenForTransactionMine(transactionResponse, provider) {  console.log(`Mining ${transactionResponse.hash}...`);  return new Promise((resolve, reject) => {  provider.once(transactionResponse.hash, (transactionReceipt) => {  console.log(`Completed with ${transactionReceipt.confirmations} confirmations.`);  resolve();  });  });  } **Example Code Block Using Wagmi** Here is an example of how the initiatDefaultStateAndSetCoverage function can be called using Wagmi to interact with the TandaPay smart contract:  import { getAccount, prepareWriteContract, writeContract, waitForTransaction } from '@wagmi/core';  const initiatDefaultStateAndSetCoverage = async (coverageAmount) => {  try {  const account = getAccount();  const config = await prepareWriteContract({  address: contractAddress,  abi: contractABI,  functionName: "initiatDefaultStateAndSetCoverage",  args: [coverageAmount],  chainId: expectedChainId,  account: account,  });  const { hash } = await writeContract(config);  const data = await waitForTransaction({  confirmations: 1,  chainId: expectedChainId,  hash: hash,  });  console.log("Default state initialized and coverage set successfully:", data);  } catch (error) {  console.error("Error initializing default state and setting coverage:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to initialize the community's default state and set the coverage amount through a user interface, ensuring the community is set up and ready to operate.  <button id="initializeCommunityBtn">Initialize Community</button>  <input type="number" id="coverageAmountInput" placeholder="Enter coverage amount" />  <script>  document.getElementById("initializeCommunityBtn").addEventListener("click", async () => {  const coverageAmount = document.getElementById("coverageAmountInput").value;  await initiatDefaultStateAndSetCoverage(coverageAmount);  });  </script> |
| --- |

| **Function Name:**whitelistClaim  **Description:**  The whitelistClaim function is a write function in the TandaPay smart contract that allows the secretary to approve or whitelist a claim submitted by a member. This function requires two arguments:   * **claimId:** The ID of the claim to be whitelisted. * **isApproved:** A boolean value indicating whether the claim is approved (true) or not (false).   By invoking this function, the secretary can officially approve claims, making them eligible for payout or further processing. **Use Case** The primary use case of the whitelistClaim function is to enable the approval process for claims within the TandaPay community. This action is crucial for:   * **Claim Management:** Approving or rejecting claims submitted by members. * **Operational Governance:** Ensuring that only valid claims are approved and processed for payouts. * **Member Trust:** Providing transparency and trust in the claim approval process.  **Why It's Important**  * **Claim Approval:** Helps in managing the claim approval process, ensuring that valid claims are processed. * **Operational Clarity:** Ensures that members are aware of the status of their claims, promoting transparency and trust. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications that manage claim approvals.  **Example Code Block Using Ethers.js** Here is an example of how the whitelistClaim function can be called using Ethers.js to interact with the TandaPay smart contract:  async function whitelistClaim(claimId, isApproved) {  console.log(`Whitelisting claim ${claimId}...`);  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Request accounts from the user  await provider.send("eth\_requestAccounts", []);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the whitelistClaim function  const transactionResponse = await contract.whitelistClaim(claimId, isApproved);  await listenForTransactionMine(transactionResponse, provider);  console.log("Claim whitelisted successfully.");  } catch (error) {  console.error("Error whitelisting claim:", error);  }  } else {  console.log("Please install MetaMask");  }  }  async function listenForTransactionMine(transactionResponse, provider) {  console.log(`Mining ${transactionResponse.hash}...`);  return new Promise((resolve, reject) => {  provider.once(transactionResponse.hash, (transactionReceipt) => {  console.log(`Completed with ${transactionReceipt.confirmations} confirmations.`);  resolve();  });  });  } **Example Code Block Using Wagmi** Here is an example of how the whitelistClaim function can be called using Wagmi to interact with the TandaPay smart contract:  import { getAccount, prepareWriteContract, writeContract, waitForTransaction } from '@wagmi/core';  const whitelistClaim = async (claimId, isApproved) => {  try {  const account = getAccount();  const config = await prepareWriteContract({  address: contractAddress,  abi: contractABI,  functionName: "whitelistClaim",  args: [claimId, isApproved],  chainId: expectedChainId,  account: account,  });  const { hash } = await writeContract(config);  const data = await waitForTransaction({  confirmations: 1,  chainId: expectedChainId,  hash: hash,  });  console.log("Claim whitelisted successfully:", data);  } catch (error) {  console.error("Error whitelisting claim:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to approve or reject claims within the TandaPay community through a user interface, ensuring that the claim approval process is transparent and trustworthy.  <button id="whitelistClaimBtn">Whitelist Claim</button>  <input type="text" id="claimIdInput" placeholder="Enter claim ID" />  <label for="isApprovedInput">Approve:</label>  <input type="checkbox" id="isApprovedInput" />  <script>  document.getElementById("whitelistClaimBtn").addEventListener("click", async () => {  const claimId = document.getElementById("claimIdInput").value;  const isApproved = document.getElementById("isApprovedInput").checked;  await whitelistClaim(claimId, isApproved);  });  </script> |
| --- |

| **Function Name:** updateCoverageAmount  **Description:**  The updateCoverageAmount function is a write function in the TandaPay smart contract that allows the secretary to update the coverage amount for the community. This function requires one argument:   * **newCoverageAmount:** The new amount of coverage to be set for the community.   By invoking this function, the secretary can adjust the coverage amount that members will be eligible for, ensuring that the coverage remains adequate and relevant. **Use Case** The primary use case of the updateCoverageAmount function is to enable the adjustment of the community's coverage amount. This action is crucial for:   * **Coverage Adjustment:** Updating the coverage amount to reflect changes in the community's needs or financial status. * **Operational Flexibility:** Ensuring that the coverage amount can be adjusted as required to maintain the community's financial health. * **Member Trust:** Providing transparency and confidence to members regarding the coverage they are entitled to.  **Why It's Important**  * **Adequate Coverage:** Helps in maintaining appropriate coverage levels for the community, ensuring members are adequately protected. * **Operational Flexibility:** Allows for adjustments in coverage based on the community's financial health and needs. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications that manage coverage settings.  **Example Code Block Using Ethers.js** Here is an example of how the updateCoverageAmount function can be called using Ethers.js to interact with the TandaPay smart contract:  async function updateCoverageAmount(newCoverageAmount) {  console.log(`Updating coverage amount to ${newCoverageAmount}...`);  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Request accounts from the user  await provider.send("eth\_requestAccounts", []);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the updateCoverageAmount function  const transactionResponse = await contract.updateCoverageAmount(newCoverageAmount);  await listenForTransactionMine(transactionResponse, provider);  console.log("Coverage amount updated successfully.");  } catch (error) {  console.error("Error updating coverage amount:", error);  }  } else {  console.log("Please install MetaMask");  }  }  async function listenForTransactionMine(transactionResponse, provider) {  console.log(`Mining ${transactionResponse.hash}...`);  return new Promise((resolve, reject) => {  provider.once(transactionResponse.hash, (transactionReceipt) => {  console.log(`Completed with ${transactionReceipt.confirmations} confirmations.`);  resolve();  });  });  } **Example Code Block Using Wagmi** Here is an example of how the updateCoverageAmount function can be called using Wagmi to interact with the TandaPay smart contract:  import { getAccount, prepareWriteContract, writeContract, waitForTransaction } from '@wagmi/core';  const updateCoverageAmount = async (newCoverageAmount) => {  try {  const account = getAccount();  const config = await prepareWriteContract({  address: contractAddress,  abi: contractABI,  functionName: "updateCoverageAmount",  args: [newCoverageAmount],  chainId: expectedChainId,  account: account,  });  const { hash } = await writeContract(config);  const data = await waitForTransaction({  confirmations: 1,  chainId: expectedChainId,  hash: hash,  });  console.log("Coverage amount updated successfully:", data);  } catch (error) {  console.error("Error updating coverage amount:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to update the community's coverage amount through a user interface, ensuring that the coverage remains adequate and relevant.  <button id="updateCoverageBtn">Update Coverage Amount</button>  <input type="number" id="newCoverageAmountInput" placeholder="Enter new coverage amount" />  <script>  document.getElementById("updateCoverageBtn").addEventListener("click", async () => {  const newCoverageAmount = document.getElementById("newCoverageAmountInput").value;  await updateCoverageAmount(newCoverageAmount);  });  </script> |
| --- |

| **Function Name:** defineSecretarySuccessor  **Description:**  The defineSecretarySuccessor function is a write function in the TandaPay smart contract that allows the current secretary to appoint a successor. This function requires one argument:   * **successorAddress:** The wallet address of the successor to be appointed.   By invoking this function, the current secretary can designate a new secretary to take over responsibilities, ensuring a smooth transition of leadership within the community. **Use Case** The primary use case of the defineSecretarySuccessor function is to enable the orderly transition of the secretary role within the TandaPay community. This action is crucial for:   * **Leadership Transition:** Ensuring that there is a clear successor for the secretary role to maintain continuity in community governance. * **Operational Stability:** Preventing disruptions in community operations by having a designated successor ready to assume the secretary's responsibilities. * **Member Confidence:** Providing transparency and trust in the leadership transition process.  **Why It's Important**  * **Smooth Transition:** Helps in maintaining stability and continuity in community leadership. * **Operational Readiness:** Ensures that the community is always prepared with a designated leader. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications that manage leadership transitions.  **Example Code Block Using Ethers.js** Here is an example of how the defineSecretarySuccessor function can be called using Ethers.js to interact with the TandaPay smart contract:  async function defineSecretarySuccessor(successorAddress) {  console.log(`Defining successor with address ${successorAddress}...`);  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Request accounts from the user  await provider.send("eth\_requestAccounts", []);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the defineSecretarySuccessor function  const transactionResponse = await contract.defineSecretarySuccessor(successorAddress);  await listenForTransactionMine(transactionResponse, provider);  console.log("Secretary successor defined successfully.");  } catch (error) {  console.error("Error defining secretary successor:", error);  }  } else {  console.log("Please install MetaMask");  }  }  async function listenForTransactionMine(transactionResponse, provider) {  console.log(`Mining ${transactionResponse.hash}...`);  return new Promise((resolve, reject) => {  provider.once(transactionResponse.hash, (transactionReceipt) => {  console.log(`Completed with ${transactionReceipt.confirmations} confirmations.`);  resolve();  });  });  } **Example Code Block Using Wagmi** Here is an example of how the defineSecretarySuccessor function can be called using Wagmi to interact with the TandaPay smart contract:  import { getAccount, prepareWriteContract, writeContract, waitForTransaction } from '@wagmi/core';  const defineSecretarySuccessor = async (successorAddress) => {  try {  const account = getAccount();  const config = await prepareWriteContract({  address: contractAddress,  abi: contractABI,  functionName: "defineSecretarySuccessor",  args: [successorAddress],  chainId: expectedChainId,  account: account,  });  const { hash } = await writeContract(config);  const data = await waitForTransaction({  confirmations: 1,  chainId: expectedChainId,  hash: hash,  });  console.log("Secretary successor defined successfully:", data);  } catch (error) {  console.error("Error defining secretary successor:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to define a successor for the secretary role through a user interface, ensuring a smooth transition of leadership within the community.  <button id="defineSuccessorBtn">Define Secretary Successor</button>  <input type="text" id="successorAddressInput" placeholder="Enter successor address" />  <script>  document.getElementById("defineSuccessorBtn").addEventListener("click", async () => {  const successorAddress = document.getElementById("successorAddressInput").value;  await defineSecretarySuccessor(successorAddress);  });  </script> |
| --- |

| **Function Name:** handoverSecretary  **Description:**  The handoverSecretary function is a write function in the TandaPay smart contract that allows the current secretary to hand over their responsibilities to the appointed successor. This function does not require any arguments. By invoking this function, the current secretary can officially transfer the role and responsibilities to the designated successor, ensuring a seamless transition of leadership within the community. **Use Case** The primary use case of the handoverSecretary function is to enable the smooth transition of the secretary role within the TandaPay community. This action is crucial for:   * **Leadership Transition:** Officially transferring the secretary role to the appointed successor. * **Operational Stability:** Preventing disruptions in community operations by ensuring a clear and formal handover process. * **Member Confidence:** Providing transparency and trust in the leadership transition process.  **Why It's Important**  * **Seamless Transition:** Helps in maintaining stability and continuity in community leadership. * **Operational Readiness:** Ensures that the community is always prepared with a designated leader. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications that manage leadership transitions.  **Example Code Block Using Ethers.js** Here is an example of how the handoverSecretary function can be called using Ethers.js to interact with the TandaPay smart contract:  async function handoverSecretary() {  console.log(`Handing over secretary role...`);  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Request accounts from the user  await provider.send("eth\_requestAccounts", []);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the handoverSecretary function  const transactionResponse = await contract.handoverSecretary();  await listenForTransactionMine(transactionResponse, provider);  console.log("Secretary role handed over successfully.");  } catch (error) {  console.error("Error handing over secretary role:", error);  }  } else {  console.log("Please install MetaMask");  }  }  async function listenForTransactionMine(transactionResponse, provider) {  console.log(`Mining ${transactionResponse.hash}...`);  return new Promise((resolve, reject) => {  provider.once(transactionResponse.hash, (transactionReceipt) => {  console.log(`Completed with ${transactionReceipt.confirmations} confirmations.`);  resolve();  });  });  } **Example Code Block Using Wagmi** Here is an example of how the handoverSecretary function can be called using Wagmi to interact with the TandaPay smart contract:  import { getAccount, prepareWriteContract, writeContract, waitForTransaction } from '@wagmi/core';  const handoverSecretary = async () => {  try {  const account = getAccount();  const config = await prepareWriteContract({  address: contractAddress,  abi: contractABI,  functionName: "handoverSecretary",  chainId: expectedChainId,  account: account,  });  const { hash } = await writeContract(config);  const data = await waitForTransaction({  confirmations: 1,  chainId: expectedChainId,  hash: hash,  });  console.log("Secretary role handed over successfully:", data);  } catch (error) {  console.error("Error handing over secretary role:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to hand over the secretary role through a user interface, ensuring a smooth transition of leadership within the community.  <button id="handoverSecretaryBtn">Handover Secretary Role</button>  <script>  document.getElementById("handoverSecretaryBtn").addEventListener("click", async () => {  await handoverSecretary();  });  </script> |
| --- |

| **Function Name:** injectFunds  **Description:**  The injectFunds function is a write function in the TandaPay smart contract that allows the secretary or an authorized party to inject additional funds into the community pool. This function requires one argument:   * **amount:** The amount of funds to be injected.   By invoking this function, the authorized party can increase the pool of funds available for coverage, ensuring the community has sufficient resources to handle claims and other financial obligations. **Use Case** The primary use case of the injectFunds function is to enable the addition of funds to the community pool. This action is crucial for:   * **Financial Management:** Ensuring that the community pool has sufficient funds to cover claims and other financial needs. * **Operational Stability:** Maintaining the financial health of the community by injecting additional funds as needed. * **Member Confidence:** Providing assurance to members that there are adequate funds to meet coverage obligations.  **Why It's Important**  * **Adequate Funding:** Helps in maintaining a well-funded community pool, ensuring members are protected. * **Operational Readiness:** Ensures that the community is financially prepared to handle claims and other expenses. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications that manage fund injections.  **Example Code Block Using Ethers.js** Here is an example of how the injectFunds function can be called using Ethers.js to interact with the TandaPay smart contract:  async function injectFunds(amount) {  console.log(`Injecting ${amount} funds into the community pool...`);  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Request accounts from the user  await provider.send("eth\_requestAccounts", []);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the injectFunds function  const transactionResponse = await contract.injectFunds({ value: ethers.utils.parseEther(amount.toString()) });  await listenForTransactionMine(transactionResponse, provider);  console.log("Funds injected successfully.");  } catch (error) {  console.error("Error injecting funds:", error);  }  } else {  console.log("Please install MetaMask");  }  }  async function listenForTransactionMine(transactionResponse, provider) {  console.log(`Mining ${transactionResponse.hash}...`);  return new Promise((resolve, reject) => {  provider.once(transactionResponse.hash, (transactionReceipt) => {  console.log(`Completed with ${transactionReceipt.confirmations} confirmations.`);  resolve();  });  });  } **Example Code Block Using Wagmi** Here is an example of how the injectFunds function can be called using Wagmi to interact with the TandaPay smart contract:  import { getAccount, prepareWriteContract, writeContract, waitForTransaction } from '@wagmi/core';  const injectFunds = async (amount) => {  try {  const account = getAccount();  const config = await prepareWriteContract({  address: contractAddress,  abi: contractABI,  functionName: "injectFunds",  args: [],  overrides: {  value: ethers.utils.parseEther(amount.toString()),  },  chainId: expectedChainId,  account: account,  });  const { hash } = await writeContract(config);  const data = await waitForTransaction({  confirmations: 1,  chainId: expectedChainId,  hash: hash,  });  console.log("Funds injected successfully:", data);  } catch (error) {  console.error("Error injecting funds:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to inject funds into the community pool through a user interface, ensuring that the community remains financially stable.  <button id="injectFundsBtn">Inject Funds</button>  <input type="number" id="amountInput" placeholder="Enter amount" />  <script>  document.getElementById("injectFundsBtn").addEventListener("click", async () => {  const amount = document.getElementById("amountInput").value;  await injectFunds(amount);  });  </script> |
| --- |

| **Function Name:** divideShortFall  **Description:**  The divideShortFall function is a write function in the TandaPay smart contract that allows the secretary or an authorized party to manage shortfalls in the community pool. This function typically involves redistributing or managing the shortfall amounts among members to ensure the community remains financially balanced. The function may not require specific arguments, but this will depend on the exact implementation in the smart contract. **Use Case** The primary use case of the divideShortFall function is to handle situations where there is a financial shortfall in the community pool. This action is crucial for:   * **Financial Management:** Ensuring that any shortfalls in the community pool are addressed and managed appropriately. * **Operational Stability:** Maintaining the financial stability of the community by addressing shortfalls promptly. * **Member Confidence:** Providing assurance to members that financial shortfalls are managed transparently and efficiently.  **Why It's Important**  * **Financial Health:** Helps in maintaining a financially healthy community pool by managing shortfalls effectively. * **Operational Readiness:** Ensures that the community can handle financial challenges and remain stable. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications that manage financial redistributions.  **Example Code Block Using Ethers.js** Here is an example of how the divideShortFall function can be called using Ethers.js to interact with the TandaPay smart contract:  async function divideShortFall() {  console.log(`Dividing shortfall among members...`);  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Request accounts from the user  await provider.send("eth\_requestAccounts", []);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the divideShortFall function  const transactionResponse = await contract.divideShortFall();  await listenForTransactionMine(transactionResponse, provider);  console.log("Shortfall divided successfully.");  } catch (error) {  console.error("Error dividing shortfall:", error);  }  } else {  console.log("Please install MetaMask");  }  }  async function listenForTransactionMine(transactionResponse, provider) {  console.log(`Mining ${transactionResponse.hash}...`);  return new Promise((resolve, reject) => {  provider.once(transactionResponse.hash, (transactionReceipt) => {  console.log(`Completed with ${transactionReceipt.confirmations} confirmations.`);  resolve();  });  });  } **Example Code Block Using Wagmi** Here is an example of how the divideShortFall function can be called using Wagmi to interact with the TandaPay smart contract:  import { getAccount, prepareWriteContract, writeContract, waitForTransaction } from '@wagmi/core';  const divideShortFall = async () => {  try {  const account = getAccount();  const config = await prepareWriteContract({  address: contractAddress,  abi: contractABI,  functionName: "divideShortFall",  chainId: expectedChainId,  account: account,  });  const { hash } = await writeContract(config);  const data = await waitForTransaction({  confirmations: 1,  chainId: expectedChainId,  hash: hash,  });  console.log("Shortfall divided successfully:", data);  } catch (error) {  console.error("Error dividing shortfall:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to manage financial shortfalls through a user interface, ensuring that the community remains financially stable.  <button id="divideShortFallBtn">Divide Shortfall</button>  <script> document.getElementById("divideShortFallBtn").addEventListener("click", async () => {  await divideShortFall();  });  </script> |
| --- |

| **Function Name:** addAdditionalDay  **Description:**  The addAdditionalDay function is a write function in the TandaPay smart contract that allows the secretary to extend the current operational period by one additional day. This function does not require any arguments. The function updates the willEndAt property of the current period by adding one day (measured in seconds) to it. This ensures that the period lasts one day longer than initially planned. **Use Case** The primary use case of the addAdditionalDay function is to provide flexibility in extending the operational timeline of the TandaPay community. This action is crucial for:   * **Operational Flexibility:** Allowing the community to extend the period to accommodate additional activities or delays. * **Event Management:** Ensuring that important events or processes have enough time to be completed within the period. * **Member Communication:** Providing members with an updated timeline to align their activities accordingly.  **Why It's Important**  * **Timeline Management:** Helps in managing the operational timeline effectively, ensuring all necessary activities are completed. * **Operational Readiness:** Ensures that the community can adapt to changes and extend the period as needed. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications that manage period extensions.  **Example Code Block Using Ethers.js** Here is an example of how the addAdditionalDay function can be called using Ethers.js to interact with the TandaPay smart contract:  async function addAdditionalDay() {  console.log(`Adding an additional day to the period...`);  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Request accounts from the user  await provider.send("eth\_requestAccounts", []);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the addAdditionalDay function  const transactionResponse = await contract.addAdditionalDay();  await listenForTransactionMine(transactionResponse, provider);  console.log("Additional day added successfully.");  } catch (error) {  console.error("Error adding additional day:", error);  }  } else {  console.log("Please install MetaMask");  }  }  async function listenForTransactionMine(transactionResponse, provider) {  console.log(`Mining ${transactionResponse.hash}...`);  return new Promise((resolve, reject) => {  provider.once(transactionResponse.hash, (transactionReceipt) => {  console.log(`Completed with ${transactionReceipt.confirmations} confirmations.`);  resolve();  });  });  } **Example Code Block Using Wagmi** Here is an example of how the addAdditionalDay function can be called using Wagmi to interact with the TandaPay smart contract:  import { getAccount, prepareWriteContract, writeContract, waitForTransaction } from '@wagmi/core';  const addAdditionalDay = async () => {  try {  const account = getAccount();  const config = await prepareWriteContract({  address: contractAddress,  abi: contractABI,  functionName: "addAdditionalDay",  chainId: expectedChainId,  account: account,  });  const { hash } = await writeContract(config);  const data = await waitForTransaction({  confirmations: 1,  chainId: expectedChainId,  hash: hash,  });  console.log("Additional day added successfully:", data);  } catch (error) {  console.error("Error adding additional day:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to extend the operational period through a user interface, ensuring that the community can adapt to changes in the timeline.  <button id="addAdditionalDayBtn">Add Additional Day</button>  <script>  document.getElementById("addAdditionalDayBtn").addEventListener("click", async () => {  await addAdditionalDay();  });  </script> |
| --- |

| **Function Name:** manualCollapsBySecretary  **Description:**  The manualCollapsBySecretary function is a write function in the TandaPay smart contract that allows the secretary to manually collapse the community. This function does not require any arguments. By invoking this function, the secretary can transition the community into a collapsed state, which might be necessary in emergency situations or when the community can no longer operate effectively. **Use Case** The primary use case of the manualCollapsBySecretary function is to handle emergency situations or severe operational issues within the TandaPay community. This action is crucial for:   * **Emergency Management:** Allowing the community to be collapsed manually in case of emergencies. * **Operational Governance:** Ensuring that the community can be put into a collapsed state when it is no longer viable to operate. * **Member Protection:** Protecting members by transitioning the community to a safe state during crises.  **Why It's Important**  * **Crisis Management:** Helps in managing crises effectively by allowing the community to be collapsed manually. * **Operational Control:** Ensures that the community has a controlled mechanism for transitioning to a collapsed state. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications that manage emergency collapses.  **Example Code Block Using Ethers.js** Here is an example of how the manualCollapsBySecretary function can be called using Ethers.js to interact with the TandaPay smart contract:  async function manualCollapsBySecretary() {  console.log(`Manually collapsing the community...`);  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Request accounts from the user  await provider.send("eth\_requestAccounts", []);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the manualCollapsBySecretary function  const transactionResponse = await contract.manualCollapsBySecretary();  await listenForTransactionMine(transactionResponse, provider);  console.log("Community collapsed manually.");  } catch (error) {  console.error("Error collapsing the community:", error);  }  } else {  console.log("Please install MetaMask");  }  }  async function listenForTransactionMine(transactionResponse, provider) {  console.log(`Mining ${transactionResponse.hash}...`);  return new Promise((resolve, reject) => {  provider.once(transactionResponse.hash, (transactionReceipt) => {  console.log(`Completed with ${transactionReceipt.confirmations} confirmations.`);  resolve();  });  });  } **Example Code Block Using Wagmi** Here is an example of how the manualCollapsBySecretary function can be called using Wagmi to interact with the TandaPay smart contract:  import { getAccount, prepareWriteContract, writeContract, waitForTransaction } from '@wagmi/core';  const manualCollapsBySecretary = async () => {  try {  const account = getAccount();  const config = await prepareWriteContract({  address: contractAddress,  abi: contractABI,  functionName: "manualCollapsBySecretary",  chainId: expectedChainId,  account: account,  });  const { hash } = await writeContract(config);  const data = await waitForTransaction({  confirmations: 1,  chainId: expectedChainId,  hash: hash,  });  console.log("Community collapsed manually:", data);  } catch (error) {  console.error("Error collapsing the community:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to manually collapse the community through a user interface, ensuring that the community can be transitioned to a safe state during emergencies.  <button id="manualCollapsBySecretaryBtn">Manually Collapse Community</button>  <script>  document.getElementById("manualCollapsBySecretaryBtn").addEventListener("click", async () => {  await manualCollapsBySecretary();  });  </script> |
| --- |

| **Function Name:** cancelManualCollapsBySecretary  **Description:**  The cancelManualCollapsBySecretary function is a write function in the TandaPay smart contract that allows the secretary to cancel a previously initiated manual collapse of the community. This function does not require any arguments. By invoking this function, the secretary can revert the community from a collapsed state back to its previous operational state, ensuring that the community can continue its activities normally. **Use Case** The primary use case of the cancelManualCollapsBySecretary function is to provide a way to undo the manual collapse initiated by the secretary. This action is crucial for:   * **Crisis Management:** Allowing the community to return to normal operations if the reason for the manual collapse has been resolved. * **Operational Governance:** Ensuring that the community can quickly recover from a collapsed state. * **Member Confidence:** Providing assurance to members that the community can revert back to normal operations if a collapse was initiated in error or if conditions have improved.  **Why It's Important**  * **Operational Continuity:** Helps in maintaining continuity in community operations by allowing a reversal of the collapsed state. * **Crisis Resolution:** Ensures that the community can quickly recover from a collapsed state if the issues causing the collapse have been addressed. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications that manage collapse and recovery processes.  **Example Code Block Using Ethers.js** Here is an example of how the cancelManualCollapsBySecretary function can be called using Ethers.js to interact with the TandaPay smart contract:  async function cancelManualCollapsBySecretary() {  console.log(`Canceling the manual collapse of the community...`);  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Request accounts from the user  await provider.send("eth\_requestAccounts", []);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the cancelManualCollapsBySecretary function  const transactionResponse = await contract.cancelManualCollapsBySecretary();  await listenForTransactionMine(transactionResponse, provider);  console.log("Manual collapse canceled successfully.");  } catch (error) {  console.error("Error canceling the manual collapse:", error);  }  } else {  console.log("Please install MetaMask");  }  }  async function listenForTransactionMine(transactionResponse, provider) {  console.log(`Mining ${transactionResponse.hash}...`);  return new Promise((resolve, reject) => {  provider.once(transactionResponse.hash, (transactionReceipt) => {  console.log(`Completed with ${transactionReceipt.confirmations} confirmations.`);  resolve();  });  });  } **Example Code Block Using Wagmi** Here is an example of how the cancelManualCollapsBySecretary function can be called using Wagmi to interact with the TandaPay smart contract:  import { getAccount, prepareWriteContract, writeContract, waitForTransaction } from '@wagmi/core';  const cancelManualCollapsBySecretary = async () => {  try {  const account = getAccount();  const config = await prepareWriteContract({  address: contractAddress,  abi: contractABI,  functionName: "cancelManualCollapsBySecretary",  chainId: expectedChainId,  account: account,  });  const { hash } = await writeContract(config);  const data = await waitForTransaction({  confirmations: 1,  chainId: expectedChainId,  hash: hash,  });  console.log("Manual collapse canceled successfully:", data);  } catch (error) {  console.error("Error canceling the manual collapse:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to cancel the manual collapse of the community through a user interface, ensuring that the community can revert to normal operations.  <button id="cancelManualCollapsBySecretaryBtn">Cancel Manual Collapse</button>  <script>  document.getElementById("cancelManualCollapsBySecretaryBtn").addEventListener("click", async () => {  await cancelManualCollapsBySecretary();  });  </script> |
| --- |

| **Function Name:** AdvanceToTheNextPeriod  **Description:**  The AdvanceToTheNextPeriod function is a write function in the TandaPay smart contract that allows the secretary to advance the community to the next operational period. This function does not require any arguments. By invoking this function, the secretary can transition the community from the current period to the next one, enabling the community to progress and start a new cycle. **Use Case** The primary use case of the AdvanceToTheNextPeriod function is to manage the progression of periods within the TandaPay community. This action is crucial for:   * **Period Management:** Ensuring the community transitions smoothly from one period to the next. * **Operational Continuity:** Maintaining the operational flow and continuity of the community. * **Member Participation:** Allowing members to participate in new periods and fulfill their obligations for the upcoming cycle.  **Why It's Important**  * **Period Transition:** Helps in managing the transition between periods effectively, ensuring all necessary activities are completed. * **Operational Readiness:** Ensures that the community can continue to operate smoothly with defined periods. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications that manage period transitions.  **Example Code Block Using Ethers.js** Here is an example of how the AdvanceToTheNextPeriod function can be called using Ethers.js to interact with the TandaPay smart contract:  async function AdvanceToTheNextPeriod() {  console.log(`Advancing to the next period...`);  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Request accounts from the user  await provider.send("eth\_requestAccounts", []);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the AdvanceToTheNextPeriod function  const transactionResponse = await contract.AdvanceToTheNextPeriod();  await listenForTransactionMine(transactionResponse, provider);  console.log("Advanced to the next period successfully.");  } catch (error) {  console.error("Error advancing to the next period:", error);  }  } else {  console.log("Please install MetaMask");  }  }  async function listenForTransactionMine(transactionResponse, provider) {  console.log(`Mining ${transactionResponse.hash}...`);  return new Promise((resolve, reject) => {  provider.once(transactionResponse.hash, (transactionReceipt) => {  console.log(`Completed with ${transactionReceipt.confirmations} confirmations.`);  resolve();  });  });  } **Example Code Block Using Wagmi** Here is an example of how the AdvanceToTheNextPeriod function can be called using Wagmi to interact with the TandaPay smart contract:  import { getAccount, prepareWriteContract, writeContract, waitForTransaction } from '@wagmi/core';  const AdvanceToTheNextPeriod = async () => {  try {  const account = getAccount();  const config = await prepareWriteContract({  address: contractAddress,  abi: contractABI,  functionName: "AdvanceToTheNextPeriod",  chainId: expectedChainId,  account: account,  });  const { hash } = await writeContract(config);  const data = await waitForTransaction({  confirmations: 1,  chainId: expectedChainId,  hash: hash,  });  console.log("Advanced to the next period successfully:", data);  } catch (error) {  console.error("Error advancing to the next period:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to advance the community to the next period through a user interface, ensuring that the community can continue to operate and progress.  <button id="AdvanceToTheNextPeriodBtn">Advance to Next Period</button>  <script>  document.getElementById("AdvanceToTheNextPeriodBtn").addEventListener("click", async () => {  await AdvanceToTheNextPeriod();  });  </script> |
| --- |

**All the functions members will use to send transaction**

**To the smart contract**

| **Function Name:** joinToCommunity  **Description:**  The joinToCommunity function is a write function in the TandaPay smart contract that allows a user to join the community. This function does not require any arguments. By invoking this function, a user transfers 11/12th of their individual savings amount's fund to the community, officially becoming a member of TandaPay. **Use Case** The primary use case of the joinToCommunity function is to enable users to join the TandaPay community by contributing their funds. This action is crucial for:   * **Community Growth:** Allowing new members to join and contribute to the community pool. * **Fund Accumulation:** Ensuring that the community pool grows with the addition of new members' contributions. * **Member Participation:** Providing a mechanism for users to become official members and participate in the community's operations.  **Why It's Important**  * **Community Expansion:** Helps in expanding the community by enabling new members to join. * **Fund Collection:** Ensures that the community pool has sufficient funds contributed by members. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications that manage member onboarding.  **Example Code Block Using Ethers.js** Here is an example of how the joinToCommunity function can be called using Ethers.js to interact with the TandaPay smart contract:  async function joinToCommunity() {  console.log(`Joining the community...`);  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Request accounts from the user  await provider.send("eth\_requestAccounts", []);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the joinToCommunity function  const transactionResponse = await contract.joinToCommunity();  await listenForTransactionMine(transactionResponse, provider);  console.log("Joined the community successfully.");  } catch (error) {  console.error("Error joining the community:", error);  }  } else {  console.log("Please install MetaMask");  }  }  async function listenForTransactionMine(transactionResponse, provider) {  console.log(`Mining ${transactionResponse.hash}...`);  return new Promise((resolve, reject) => {  provider.once(transactionResponse.hash, (transactionReceipt) => {  console.log(`Completed with ${transactionReceipt.confirmations} confirmations.`);  resolve();  });  });  } **Example Code Block Using Wagmi** Here is an example of how the joinToCommunity function can be called using Wagmi to interact with the TandaPay smart contract:  import { getAccount, prepareWriteContract, writeContract, waitForTransaction } from '@wagmi/core';  const joinToCommunity = async () => {  try {  const account = getAccount();  const config = await prepareWriteContract({  address: contractAddress,  abi: contractABI,  functionName: "joinToCommunity",  chainId: expectedChainId,  account: account,  });  const { hash } = await writeContract(config);  const data = await waitForTransaction({  confirmations: 1,  chainId: expectedChainId,  hash: hash,  });  console.log("Joined the community successfully:", data);  } catch (error) {  console.error("Error joining the community:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to allow users to join the TandaPay community through a user interface, ensuring that the community can grow and accumulate funds.  <button id="joinToCommunityBtn">Join Community</button>  <script>  document.getElementById("joinToCommunityBtn").addEventListener("click", async () => {  await joinToCommunity();  });  </script> |
| --- |

| **Function Name:** approveSubGroupAssignment  **Description:**  The approveSubGroupAssignment function is a write function in the TandaPay smart contract that allows members to approve their assignment to a specific subgroup. This function requires one argument:   * **approval:** A boolean value indicating whether the member approves (true) or disapproves (false) of the subgroup assignment.   By invoking this function, members can formally approve their assignment to a subgroup, ensuring that they are grouped correctly for operational purposes. **Use Case** The primary use case of the approveSubGroupAssignment function is to confirm member assignments to subgroups within the TandaPay community. This action is crucial for:   * **Member Consent:** Allowing members to agree to their subgroup assignments. * **Operational Organization:** Ensuring that members are properly grouped and can participate effectively in subgroup activities. * **Community Governance:** Maintaining a transparent and consent-based process for subgroup assignments.  **Why It's Important**  * **Member Agreement:** Ensures that members are placed in subgroups with their consent, fostering trust and cooperation. * **Operational Efficiency:** Helps in organizing members into subgroups for better management and collaboration. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications that manage subgroup assignments.  **Example Code Block Using Ethers.js** Here is an example of how the approveSubGroupAssignment function can be called using Ethers.js to interact with the TandaPay smart contract:  async function approveSubGroupAssignment(approval) {  console.log(`Approving subgroup assignment with status: ${approval}`);  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Request accounts from the user  await provider.send("eth\_requestAccounts", []);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the approveSubGroupAssignment function  const transactionResponse = await contract.approveSubGroupAssignment(approval);  await listenForTransactionMine(transactionResponse, provider);  console.log("Subgroup assignment approved successfully.");  } catch (error) {  console.error("Error approving subgroup assignment:", error);  }  } else {  console.log("Please install MetaMask");  }  }  async function listenForTransactionMine(transactionResponse, provider) {  console.log(`Mining ${transactionResponse.hash}...`);  return new Promise((resolve, reject) => {  provider.once(transactionResponse.hash, (transactionReceipt) => {  console.log(`Completed with ${transactionReceipt.confirmations} confirmations.`);  resolve();  });  });  } **Example Code Block Using Wagmi** Here is an example of how the approveSubGroupAssignment function can be called using Wagmi to interact with the TandaPay smart contract:  import { getAccount, prepareWriteContract, writeContract, waitForTransaction } from '@wagmi/core';  const approveSubGroupAssignment = async (approval) => {  try {  const account = getAccount();  const config = await prepareWriteContract({  address: contractAddress,  abi: contractABI,  functionName: "approveSubGroupAssignment",  args: [approval],  chainId: expectedChainId,  account: account,  });  const { hash } = await writeContract(config);  const data = await waitForTransaction({  confirmations: 1,  chainId: expectedChainId,  hash: hash,  });  console.log("Subgroup assignment approved successfully:", data);  } catch (error) {  console.error("Error approving subgroup assignment:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to allow members to approve their subgroup assignments through a user interface, ensuring that the community remains organized and consent-based.  <button id="approveSubGroupAssignmentBtn">Approve Subgroup Assignment</button>  <input type="checkbox" id="approvalInput" />  <script>  document.getElementById("approveSubGroupAssignmentBtn").addEventListener("click", async () => {  const approval = document.getElementById("approvalInput").checked;  await approveSubGroupAssignment(approval);  });  </script> |
| --- |

| **Function Name:** approveNewSubgroupMember  **Description:**  The approveNewSubgroupMember function is a write function in the TandaPay smart contract that allows existing members of a subgroup to approve the inclusion of a new member. This function requires three arguments:   * **subGroupId:** The ID of the subgroup to which the new member is being assigned. * **newMemberId:** The ID of the new member being added to the subgroup. * **approval:** A boolean value indicating whether the new member is accepted (true) or not (false).   By invoking this function, existing subgroup members can vote on whether to accept a new member into their subgroup, ensuring that the subgroup remains cohesive and aligned with its members' preferences. **Use Case** The primary use case of the approveNewSubgroupMember function is to manage the inclusion of new members in subgroups within the TandaPay community. This action is crucial for:   * **Member Approval:** Allowing existing subgroup members to have a say in the inclusion of new members. * **Operational Organization:** Ensuring that new members are accepted by the majority of existing subgroup members. * **Community Governance:** Maintaining a transparent and democratic process for subgroup membership changes.  **Why It's Important**  * **Member Inclusion:** Ensures that new members are only added to subgroups with the approval of existing members, fostering trust and cooperation. * **Operational Efficiency:** Helps in managing the membership of subgroups effectively, ensuring smooth operations. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications that manage subgroup memberships.  **Example Code Block Using Ethers.js** Here is an example of how the approveNewSubgroupMember function can be called using Ethers.js to interact with the TandaPay smart contract:  async function approveNewSubgroupMember(subGroupId, newMemberId, approval) {  console.log(`Approving new subgroup member with ID: ${newMemberId} for subgroup: ${subGroupId} with approval status: ${approval}`);  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Request accounts from the user  await provider.send("eth\_requestAccounts", []);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the approveNewSubgroupMember function  const transactionResponse = await contract.approveNewSubgroupMember(subGroupId, newMemberId, approval);  await listenForTransactionMine(transactionResponse, provider);  console.log("New subgroup member approved successfully.");  } catch (error) {  console.error("Error approving new subgroup member:", error);  }  } else {  console.log("Please install MetaMask");  }  }  async function listenForTransactionMine(transactionResponse, provider) {  console.log(`Mining ${transactionResponse.hash}...`);  return new Promise((resolve, reject) => {  provider.once(transactionResponse.hash, (transactionReceipt) => {  console.log(`Completed with ${transactionReceipt.confirmations} confirmations.`);  resolve();  });  });  } **Example Code Block Using Wagmi** Here is an example of how the approveNewSubgroupMember function can be called using Wagmi to interact with the TandaPay smart contract:  import { getAccount, prepareWriteContract, writeContract, waitForTransaction } from '@wagmi/core';  const approveNewSubgroupMember = async (subGroupId, newMemberId, approval) => {  try {  const account = getAccount();  const config = await prepareWriteContract({  address: contractAddress,  abi: contractABI,  functionName: "approveNewSubgroupMember",  args: [subGroupId, newMemberId, approval],  chainId: expectedChainId,  account: account,  });  const { hash } = await writeContract(config);  const data = await waitForTransaction({  confirmations: 1,  chainId: expectedChainId,  hash: hash,  });  console.log("New subgroup member approved successfully:", data);  } catch (error) {  console.error("Error approving new subgroup member:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to approve new subgroup members through a user interface, ensuring that the community remains organized and member-driven.  <button id="approveNewSubgroupMemberBtn">Approve New Subgroup Member</button>  <input type="text" id="subGroupIdInput" placeholder="Enter Subgroup ID" />  <input type="text" id="newMemberIdInput" placeholder="Enter New Member ID" />  <label for="approvalInput">Approve:</label>  <input type="checkbox" id="approvalInput" />  <script>  document.getElementById("approveNewSubgroupMemberBtn").addEventListener("click", async () => {  const subGroupId = document.getElementById("subGroupIdInput").value;  const newMemberId = document.getElementById("newMemberIdInput").value;  const approval = document.getElementById("approvalInput").checked;  await approveNewSubgroupMember(subGroupId, newMemberId, approval);  });  </script> |
| --- |

| **Function Name:** exitSubGroup  **Description:**  The exitSubGroup function is a write function in the TandaPay smart contract that allows a member to exit from a specific subgroup. This function does not require any arguments. By invoking this function, a member can leave their assigned subgroup, which might be necessary if they no longer wish to participate in that subgroup or need to reassign themselves to another one. **Use Case** The primary use case of the exitSubGroup function is to manage the departure of members from subgroups within the TandaPay community. This action is crucial for:   * **Member Flexibility:** Allowing members to leave a subgroup if they no longer wish to be part of it. * **Operational Management:** Ensuring that subgroup membership is accurately maintained and updated. * **Community Dynamics:** Providing a mechanism for members to manage their participation in different subgroups.  **Why It's Important**  * **Member Autonomy:** Ensures that members have the freedom to leave subgroups as needed. * **Operational Accuracy:** Helps in maintaining accurate records of subgroup memberships. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications that manage subgroup memberships.  **Example Code Block Using Ethers.js** Here is an example of how the exitSubGroup function can be called using Ethers.js to interact with the TandaPay smart contract:  async function exitSubGroup() {  console.log(`Exiting the subgroup...`);  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Request accounts from the user  await provider.send("eth\_requestAccounts", []);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the exitSubGroup function  const transactionResponse = await contract.exitSubGroup();  await listenForTransactionMine(transactionResponse, provider);  console.log("Exited the subgroup successfully.");  } catch (error) {  console.error("Error exiting the subgroup:", error);  }  } else {  console.log("Please install MetaMask");  }  }  async function listenForTransactionMine(transactionResponse, provider) {  console.log(`Mining ${transactionResponse.hash}...`);  return new Promise((resolve, reject) => {  provider.once(transactionResponse.hash, (transactionReceipt) => {  console.log(`Completed with ${transactionReceipt.confirmations} confirmations.`);  resolve();  });  });  } **Example Code Block Using Wagmi** Here is an example of how the exitSubGroup function can be called using Wagmi to interact with the TandaPay smart contract:  import { getAccount, prepareWriteContract, writeContract, waitForTransaction } from '@wagmi/core';  const exitSubGroup = async () => {  try {  const account = getAccount();  const config = await prepareWriteContract({  address: contractAddress,  abi: contractABI,  functionName: "exitSubGroup",  chainId: expectedChainId,  account: account,  });  const { hash } = await writeContract(config);  const data = await waitForTransaction({  confirmations: 1,  chainId: expectedChainId,  hash: hash,  });  console.log("Exited the subgroup successfully:", data);  } catch (error) {  console.error("Error exiting the subgroup:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to allow members to exit their subgroups through a user interface, ensuring that the community remains organized and members can manage their subgroup participation.  <button id="exitSubGroupBtn">Exit Subgroup</button>  <script>  document.getElementById("exitSubGroupBtn").addEventListener("click", async () => {  await exitSubGroup();  });  </script> |
| --- |

| **Function Name:** defects  **Description:**  The defects function is a write function in the TandaPay smart contract that allows a member to defect from the community. This function does not require any arguments. By invoking this function, a member can leave the community, which might be necessary if they no longer wish to participate in the community or need to withdraw from it for personal reasons. **Use Case** The primary use case of the defects function is to manage the departure of members from the TandaPay community. This action is crucial for:   * **Member Autonomy:** Allowing members to leave the community if they no longer wish to be part of it. * **Operational Management:** Ensuring that membership records are accurately maintained and updated. * **Community Dynamics:** Providing a mechanism for members to manage their participation and exit if necessary.  **Why It's Important**  * **Member Freedom:** Ensures that members have the freedom to leave the community as needed. * **Operational Accuracy:** Helps in maintaining accurate records of community membership. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications that manage membership statuses.  **Example Code Block Using Ethers.js** Here is an example of how the defects function can be called using Ethers.js to interact with the TandaPay smart contract:  async function defects() {  console.log(`Defecting from the community...`);  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Request accounts from the user  await provider.send("eth\_requestAccounts", []);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the defects function  const transactionResponse = await contract.defects();  await listenForTransactionMine(transactionResponse, provider);  console.log("Defected from the community successfully.");  } catch (error) {  console.error("Error defecting from the community:", error);  }  } else {  console.log("Please install MetaMask");  }  }  async function listenForTransactionMine(transactionResponse, provider) {  console.log(`Mining ${transactionResponse.hash}...`);  return new Promise((resolve, reject) => {  provider.once(transactionResponse.hash, (transactionReceipt) => {  console.log(`Completed with ${transactionReceipt.confirmations} confirmations.`);  resolve();  });  });  } **Example Code Block Using Wagmi** Here is an example of how the defects function can be called using Wagmi to interact with the TandaPay smart contract:  import { getAccount, prepareWriteContract, writeContract, waitForTransaction } from '@wagmi/core';  const defects = async () => {  try {  const account = getAccount();  const config = await prepareWriteContract({  address: contractAddress,  abi: contractABI,  functionName: "defects",  chainId: expectedChainId,  account: account,  });  const { hash } = await writeContract(config);  const data = await waitForTransaction({  confirmations: 1,  chainId: expectedChainId,  hash: hash,  });  console.log("Defected from the community successfully:", data);  } catch (error) {  console.error("Error defecting from the community:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to allow members to defect from the community through a user interface, ensuring that membership records are updated accurately.  <button id="defectsBtn">Defect from Community</button>  <script>  document.getElementById("defectsBtn").addEventListener("click", async () => {  await defects();  });  </script> |
| --- |

| **Function Name:** payPremium  **Description:**  The payPremium function is a write function in the TandaPay smart contract that allows valid members to pay the premium for the upcoming period. This function does not require any arguments. By invoking this function, members can ensure their continued coverage and participation in the community's operations for the next period. **Use Case** The primary use case of the payPremium function is to facilitate the payment of premiums by members for the upcoming period. This action is crucial for:   * **Continued Coverage:** Ensuring that members remain covered by paying their premiums on time. * **Operational Funding:** Providing the necessary funds to the community pool to cover claims and other expenses. * **Member Participation:** Allowing members to maintain their active status and participate in the community's activities.  **Why It's Important**  * **Member Coverage:** Ensures that members continue to receive coverage by paying their premiums. * **Community Funding:** Helps in maintaining a well-funded community pool to handle claims and other financial obligations. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications that manage premium payments.  **Example Code Block Using Ethers.js** Here is an example of how the payPremium function can be called using Ethers.js to interact with the TandaPay smart contract:  async function payPremium() {  console.log(`Paying premium for the upcoming period...`);  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Request accounts from the user  await provider.send("eth\_requestAccounts", []);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the payPremium function  const transactionResponse = await contract.payPremium();  await listenForTransactionMine(transactionResponse, provider);  console.log("Premium paid successfully.");  } catch (error) {  console.error("Error paying premium:", error);  }  } else {  console.log("Please install MetaMask");  }  }  async function listenForTransactionMine(transactionResponse, provider) {  console.log(`Mining ${transactionResponse.hash}...`);  return new Promise((resolve, reject) => {  provider.once(transactionResponse.hash, (transactionReceipt) => {  console.log(`Completed with ${transactionReceipt.confirmations} confirmations.`);  resolve();  });  });  } **Example Code Block Using Wagmi** Here is an example of how the payPremium function can be called using Wagmi to interact with the TandaPay smart contract:  import { getAccount, prepareWriteContract, writeContract, waitForTransaction } from '@wagmi/core';  const payPremium = async () => {  try {  const account = getAccount();  const config = await prepareWriteContract({  address: contractAddress,  abi: contractABI,  functionName: "payPremium",  chainId: expectedChainId,  account: account,  });  const { hash } = await writeContract(config);  const data = await waitForTransaction({  confirmations: 1,  chainId: expectedChainId,  hash: hash,  });  console.log("Premium paid successfully:", data);  } catch (error) {  console.error("Error paying premium:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to allow members to pay their premiums for the upcoming period through a user interface, ensuring that the community remains well-funded and members stay covered.  <button id="payPremiumBtn">Pay Premium</button>  <script>  document.getElementById("payPremiumBtn").addEventListener("click", async () => {  await payPremium();  });  </script> |
| --- |

| **Function Name:** secretaryAcceptance  **Description:**  The secretaryAcceptance function is a write function in the TandaPay smart contract that allows valid members who are in line for the secretary successor list and are being voted on to accept the secretary role. This function does not require any arguments. By invoking this function, a member can formally accept the role of secretary if they have been chosen as the successor. **Use Case** The primary use case of the secretaryAcceptance function is to facilitate the acceptance process for members being voted as the new secretary. This action is crucial for:   * **Leadership Transition:** Allowing a smooth transition of the secretary role to the next member in line. * **Operational Governance:** Ensuring that the community always has a formally accepted and active secretary. * **Member Participation:** Providing a mechanism for members to accept leadership roles within the community.  **Why It's Important**  * **Leadership Continuity:** Helps in maintaining a continuous and smooth transition of leadership within the community. * **Operational Stability:** Ensures that the community always has an active and accepted secretary to manage operations. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications that manage leadership transitions.  **Example Code Block Using Ethers.js** Here is an example of how the secretaryAcceptance function can be called using Ethers.js to interact with the TandaPay smart contract:  async function secretaryAcceptance() {  console.log(`Accepting the secretary role...`);  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Request accounts from the user  await provider.send("eth\_requestAccounts", []);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the secretaryAcceptance function  const transactionResponse = await contract.secretaryAcceptance();  await listenForTransactionMine(transactionResponse, provider);  console.log("Secretary role accepted successfully.");  } catch (error) {  console.error("Error accepting the secretary role:", error);  }  } else {  console.log("Please install MetaMask");  }  }  async function listenForTransactionMine(transactionResponse, provider) {  console.log(`Mining ${transactionResponse.hash}...`);  return new Promise((resolve, reject) => {  provider.once(transactionResponse.hash, (transactionReceipt) => {  console.log(`Completed with ${transactionReceipt.confirmations} confirmations.`);  resolve();  });  });  } **Example Code Block Using Wagmi** Here is an example of how the secretaryAcceptance function can be called using Wagmi to interact with the TandaPay smart contract:  import { getAccount, prepareWriteContract, writeContract, waitForTransaction } from '@wagmi/core';  const secretaryAcceptance = async () => {  try {  const account = getAccount();  const config = await prepareWriteContract({  address: contractAddress,  abi: contractABI,  functionName: "secretaryAcceptance",  chainId: expectedChainId,  account: account,  });  const { hash } = await writeContract(config);  const data = await waitForTransaction({  confirmations: 1,  chainId: expectedChainId,  hash: hash,  });  console.log("Secretary role accepted successfully:", data);  } catch (error) {  console.error("Error accepting the secretary role:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to allow members to accept the secretary role through a user interface, ensuring that the community remains well-governed and operational.  <button id="secretaryAcceptanceBtn">Accept Secretary Role</button>  <script>  document.getElementById("secretaryAcceptanceBtn").addEventListener("click", async () => {  await secretaryAcceptance();  });  </script> |
| --- |

| **Function Name:** emergencyHandOverSecretary  **Description:**  The emergencyHandOverSecretary function is a write function in the TandaPay smart contract that allows valid members who are in the line of the secretary successors list to set up another valid member, who is also in the line of the secretary successors list, as the secretary in emergency situations. This function requires one argument:   * **successorAddress:** The wallet address of the preferred successor to be appointed as the new secretary.   By invoking this function, a member can ensure a quick and seamless transition of the secretary role in case of emergencies. **Use Case** The primary use case of the emergencyHandOverSecretary function is to facilitate the swift transition of the secretary role during emergencies. This action is crucial for:   * **Emergency Management:** Allowing the community to quickly appoint a new secretary when immediate leadership is needed. * **Operational Continuity:** Ensuring that the community remains operational and well-managed even during crises. * **Leadership Flexibility:** Providing a mechanism for the quick reassignment of the secretary role to maintain stability.  **Why It's Important**  * **Crisis Response:** Helps in managing leadership transitions effectively during emergencies. * **Operational Stability:** Ensures that the community always has a designated leader to maintain operations. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications that manage emergency leadership transitions.  **Example Code Block Using Ethers.js** Here is an example of how the emergencyHandOverSecretary function can be called using Ethers.js to interact with the TandaPay smart contract:  async function emergencyHandOverSecretary(successorAddress) {  console.log(`Handing over secretary role to ${successorAddress} in emergency...`);  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Request accounts from the user  await provider.send("eth\_requestAccounts", []);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the emergencyHandOverSecretary function  const transactionResponse = await contract.emergencyHandOverSecretary(successorAddress);  await listenForTransactionMine(transactionResponse, provider);  console.log("Secretary role handed over successfully.");  } catch (error) {  console.error("Error handing over secretary role:", error);  }  } else {  console.log("Please install MetaMask");  }  }  async function listenForTransactionMine(transactionResponse, provider) {  console.log(`Mining ${transactionResponse.hash}...`);  return new Promise((resolve, reject) => {  provider.once(transactionResponse.hash, (transactionReceipt) => {  console.log(`Completed with ${transactionReceipt.confirmations} confirmations.`);  resolve();  });  });  } **Example Code Block Using Wagmi** Here is an example of how the emergencyHandOverSecretary function can be called using Wagmi to interact with the TandaPay smart contract:  import { getAccount, prepareWriteContract, writeContract, waitForTransaction } from '@wagmi/core';  const emergencyHandOverSecretary = async (successorAddress) => {  try {  const account = getAccount();  const config = await prepareWriteContract({  address: contractAddress,  abi: contractABI,  functionName: "emergencyHandOverSecretary",  args: [successorAddress],  chainId: expectedChainId,  account: account,  });  const { hash } = await writeContract(config);  const data = await waitForTransaction({  confirmations: 1,  chainId: expectedChainId,  hash: hash,  });  console.log("Secretary role handed over successfully:", data);  } catch (error) {  console.error("Error handing over secretary role:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to quickly hand over the secretary role to a successor during emergencies through a user interface, ensuring that the community remains operational.  <button id="emergencyHandOverSecretaryBtn">Emergency Hand Over Secretary</button>  <input type="text" id="successorAddressInput" placeholder="Enter Successor Address" />  <script>  document.getElementById("emergencyHandOverSecretaryBtn").addEventListener("click", async () => {  const successorAddress = document.getElementById("successorAddressInput").value;  await emergencyHandOverSecretary(successorAddress);  });  </script> |
| --- |

| **Function Name:** withdrawRefund  **Description:**  The withdrawRefund function is a write function in the TandaPay smart contract that allows a member to withdraw their available funds. This function does not require any arguments. By invoking this function, a member can retrieve any funds they have contributed that are available for withdrawal. **Use Case** The primary use case of the withdrawRefund function is to facilitate the withdrawal of available funds by members. This action is crucial for:   * **Fund Retrieval:** Allowing members to access their funds when needed. * **Operational Flexibility:** Ensuring that members can withdraw funds without needing special permissions or steps. * **Member Trust:** Providing a mechanism for members to manage their financial contributions transparently.  **Why It's Important**  * **Member Access:** Ensures that members have easy access to their funds. * **Operational Efficiency:** Helps in managing fund withdrawals effectively, maintaining smooth operations. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications that manage financial transactions.  **Example Code Block Using Ethers.js** Here is an example of how the withdrawRefund function can be called using Ethers.js to interact with the TandaPay smart contract:  async function withdrawRefund() {  console.log(`Withdrawing refund...`);  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Request accounts from the user  await provider.send("eth\_requestAccounts", []);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the withdrawRefund function  const transactionResponse = await contract.withdrawRefund();  await listenForTransactionMine(transactionResponse, provider);  console.log("Refund withdrawn successfully.");  } catch (error) {  console.error("Error withdrawing refund:", error);  }  } else {  console.log("Please install MetaMask");  }  }  async function listenForTransactionMine(transactionResponse, provider) {  console.log(`Mining ${transactionResponse.hash}...`);  return new Promise((resolve, reject) => {  provider.once(transactionResponse.hash, (transactionReceipt) => {  console.log(`Completed with ${transactionReceipt.confirmations} confirmations.`);  resolve();  });  });  } **Example Code Block Using Wagmi** Here is an example of how the withdrawRefund function can be called using Wagmi to interact with the TandaPay smart contract:  import { getAccount, prepareWriteContract, writeContract, waitForTransaction } from '@wagmi/core';  const withdrawRefund = async () => {  try {  const account = getAccount();  const config = await prepareWriteContract({  address: contractAddress,  abi: contractABI,  functionName: "withdrawRefund",  chainId: expectedChainId,  account: account,  });  const { hash } = await writeContract(config);  const data = await waitForTransaction({  confirmations: 1,  chainId: expectedChainId,  hash: hash,  });  console.log("Refund withdrawn successfully:", data);  } catch (error) {  console.error("Error withdrawing refund:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to allow members to withdraw their funds through a user interface, ensuring that the community remains financially transparent and members have control over their contributions.  <button id="withdrawRefundBtn">Withdraw Refund</button>  <script>  document.getElementById("withdrawRefundBtn").addEventListener("click", async () => {  await withdrawRefund();  });  </script> |
| --- |

| **Function Name:** submitClaim  **Description:**  The submitClaim function is a write function in the TandaPay smart contract that allows eligible members to submit a claim for coverage. This function does not require any arguments. By invoking this function, a member can request a claim based on the coverage they are eligible for. **Use Case** The primary use case of the submitClaim function is to facilitate the process of submitting claims by eligible members. This action is crucial for:   * **Claim Submission:** Allowing members to submit claims for coverage. * **Operational Transparency:** Ensuring that the process of submitting claims is straightforward and accessible. * **Member Participation:** Providing a mechanism for members to request their eligible coverage.  **Why It's Important**  * **Member Support:** Ensures that members can easily submit claims when they are eligible for coverage. * **Operational Efficiency:** Helps in managing the submission of claims effectively. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications that manage claim submissions.  **Example Code Block Using Ethers.js** Here is an example of how the submitClaim function can be called using Ethers.js to interact with the TandaPay smart contract:  async function submitClaim() {  console.log(`Submitting claim...`);  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Request accounts from the user  await provider.send("eth\_requestAccounts", []);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the submitClaim function  const transactionResponse = await contract.submitClaim();  await listenForTransactionMine(transactionResponse, provider);  console.log("Claim submitted successfully.");  } catch (error) {  console.error("Error submitting claim:", error);  }  } else {  console.log("Please install MetaMask");  }  }  async function listenForTransactionMine(transactionResponse, provider) {  console.log(`Mining ${transactionResponse.hash}...`);  return new Promise((resolve, reject) => {  provider.once(transactionResponse.hash, (transactionReceipt) => {  console.log(`Completed with ${transactionReceipt.confirmations} confirmations.`);  resolve();  });  });  } **Example Code Block Using Wagmi** Here is an example of how the submitClaim function can be called using Wagmi to interact with the TandaPay smart contract:  import { getAccount, prepareWriteContract, writeContract, waitForTransaction } from '@wagmi/core';  const submitClaim = async () => {  try {  const account = getAccount();  const config = await prepareWriteContract({  address: contractAddress,  abi: contractABI,  functionName: "submitClaim",  chainId: expectedChainId,  account: account,  });  const { hash } = await writeContract(config);  const data = await waitForTransaction({  confirmations: 1,  chainId: expectedChainId,  hash: hash,  });  console.log("Claim submitted successfully:", data);  } catch (error) {  console.error("Error submitting claim:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to allow members to submit claims for coverage through a user interface, ensuring that the process is transparent and accessible.  <button id="submitClaimBtn">Submit Claim</button>  <script>  document.getElementById("submitClaimBtn").addEventListener("click", async () => {  await submitClaim();  });  </script> |
| --- |

| **Function Name:** withdrawClaimFund  **Description:**  The withdrawClaimFund function is a write function in the TandaPay smart contract that allows a whitelisted claimant to withdraw the claim amount. This function does not require any arguments. By invoking this function, a claimant can retrieve the funds that have been approved for their claim. **Use Case** The primary use case of the withdrawClaimFund function is to enable whitelisted claimants to withdraw their approved claim amounts. This action is crucial for:   * **Fund Retrieval:** Allowing claimants to access the funds they are entitled to after a claim has been approved. * **Operational Transparency:** Ensuring that the process of withdrawing claim funds is straightforward and accessible. * **Member Support:** Providing a mechanism for members to receive financial support as per their claims.  **Why It's Important**  * **Member Access:** Ensures that claimants have easy access to their approved claim funds. * **Operational Efficiency:** Helps in managing the withdrawal of claim funds effectively, maintaining smooth operations. * **Integration:** Simplifies the integration of the TandaPay smart contract with front-end applications that manage financial transactions.  **Example Code Block Using Ethers.js** Here is an example of how the withdrawClaimFund function can be called using Ethers.js to interact with the TandaPay smart contract:  async function withdrawClaimFund() {  console.log(`Withdrawing claim fund...`);  if (typeof window.ethereum !== "undefined") {  // Create a provider using Web3  const provider = new ethers.providers.Web3Provider(window.ethereum);  // Request accounts from the user  await provider.send("eth\_requestAccounts", []);  // Get the signer  const signer = provider.getSigner();  // Create a contract instance  const contract = new ethers.Contract(contractAddress, abi, signer);  try {  // Call the withdrawClaimFund function  const transactionResponse = await contract.withdrawClaimFund();  await listenForTransactionMine(transactionResponse, provider);  console.log("Claim fund withdrawn successfully.");  } catch (error) {  console.error("Error withdrawing claim fund:", error);  }  } else {  console.log("Please install MetaMask");  }  }  async function listenForTransactionMine(transactionResponse, provider) {  console.log(`Mining ${transactionResponse.hash}...`);  return new Promise((resolve, reject) => {  provider.once(transactionResponse.hash, (transactionReceipt) => {  console.log(`Completed with ${transactionReceipt.confirmations} confirmations.`);  resolve();  });  });  } **Example Code Block Using Wagmi** Here is an example of how the withdrawClaimFund function can be called using Wagmi to interact with the TandaPay smart contract:  import { getAccount, prepareWriteContract, writeContract, waitForTransaction } from '@wagmi/core';  const withdrawClaimFund = async () => {  try {  const account = getAccount();  const config = await prepareWriteContract({  address: contractAddress,  abi: contractABI,  functionName: "withdrawClaimFund",  chainId: expectedChainId,  account: account,  });  const { hash } = await writeContract(config);  const data = await waitForTransaction({  confirmations: 1,  chainId: expectedChainId,  hash: hash,  });  console.log("Claim fund withdrawn successfully:", data);  } catch (error) {  console.error("Error withdrawing claim fund:", error);  }  }; **Example Use Case in a Front-End Application** In a front-end application, this function can be used to allow whitelisted claimants to withdraw their claim funds through a user interface, ensuring that the process is transparent and accessible.  <button id="withdrawClaimFundBtn">Withdraw Claim Fund</button>  <script>  document.getElementById("withdrawClaimFundBtn").addEventListener("click", async () => {  await withdrawClaimFund();  });  </script> |
| --- |