```
#include <eosiopowcoin.hpp>
#include <eosio/system.hpp>

namespace eosio {

void token::create( const name&   issuer,
                    const asset&  maximum_supply )
{
    require_auth( get_self() );

    auto sym = maximum_supply.symbol;
    check( sym.is_valid(), "invalid symbol name" );
    check( maximum_supply.is_valid(), "invalid supply");
    check( maximum_supply.amount > 0, "max-supply must be positive");

    stats statstable( get_self(), sym.code().raw() );
    auto existing = statstable.find( sym.code().raw() );
    check( existing == statstable.end(), "token with symbol already exists" );

    statstable.emplace( get_self(), [&]( auto& s ) {
       s.supply.symbol = maximum_supply.symbol;
       s.max_supply    = maximum_supply;
       s.issuer        = issuer;
       s.starttime     = current_time_point().sec_since_epoch();
       s.minetime      = s.starttime;
    });
}


void token::issue( const name& to, const asset& quantity, const string& memo )
{
    auto sym = quantity.symbol;
    check( sym.is_valid(), "invalid symbol name" );
    check( memo.size() <= 256, "memo has more than 256 bytes" );

    stats statstable( get_self(), sym.code().raw() );
    auto existing = statstable.find( sym.code().raw() );
    check( existing != statstable.end(), "token with symbol does not exist, create token before
issue" );
    const auto& st = *existing;
    check( to == st.issuer, "tokens can only be issued to issuer account" );

    require_auth( st.issuer );
```

```
   check( quantity.is_valid(), "invalid quantity" );
   check( quantity.amount > 0, "must issue positive quantity" );

   check( quantity.symbol == st.supply.symbol, "symbol precision mismatch" );
   check( quantity.amount <= st.max_supply.amount - st.supply.amount, "quantity exceeds
available supply");

   statstable.modify( st, same_payer, [&]( auto& s ) {
     s.supply += quantity;
   });

   add_balance( st.issuer, quantity, st.issuer );
}

void token::retire( const asset& quantity, const string& memo )
{
   auto sym = quantity.symbol;
   check( sym.is_valid(), "invalid symbol name" );
   check( memo.size() <= 256, "memo has more than 256 bytes" );

   stats statstable( get_self(), sym.code().raw() );
   auto existing = statstable.find( sym.code().raw() );
   check( existing != statstable.end(), "token with symbol does not exist" );
   const auto& st = *existing;

   require_auth( st.issuer );
   check( quantity.is_valid(), "invalid quantity" );
   check( quantity.amount > 0, "must retire positive quantity" );

   check( quantity.symbol == st.supply.symbol, "symbol precision mismatch" );

   statstable.modify( st, same_payer, [&]( auto& s ) {
     s.supply -= quantity;
   });

   sub_balance( st.issuer, quantity );
}

void token::transfer( const name&    from,
                      const name&    to,
                      const asset&   quantity,
                      const string&  memo )
{
```

```
    check( from != to, "cannot transfer to self" );
    require_auth( from );
    check( is_account( to ), "to account does not exist");
    auto sym = quantity.symbol.code();
    stats statstable( get_self(), sym.raw() );
    const auto& st = statstable.get( sym.raw() );

    require_recipient( from );
    require_recipient( to );

    check( quantity.is_valid(), "invalid quantity" );
    check( quantity.amount > 0, "must transfer positive quantity" );
    check( quantity.symbol == st.supply.symbol, "symbol precision mismatch" );
    check( memo.size() <= 256, "memo has more than 256 bytes" );

    auto payer = has_auth( to ) ? to : from;

    sub_balance( from, quantity );
    add_balance( to, quantity, payer );
}

void token::sub_balance( const name& owner, const asset& value ) {
   accounts from_acnts( get_self(), owner.value );

   const auto& from = from_acnts.get( value.symbol.code().raw(), "no balance object found" );
   check( from.balance.amount >= value.amount, "overdrawn balance" );

   from_acnts.modify( from, owner, [&]( auto& a ) {


a.balance -= value;
      });
}

void token::add_balance( const name& owner, const asset& value, const name& ram_payer )
{
   accounts to_acnts( get_self(), owner.value );
   auto to = to_acnts.find( value.symbol.code().raw() );
   if( to == to_acnts.end() ) {
      to_acnts.emplace( ram_payer, [&]( auto& a ){
        a.balance = value;
      });
   } else {
```

```
      to_acnts.modify( to, same_payer, [&]( auto& a ) {
        a.balance += value;
      });
    }
}

void token::open( const name& owner, const symbol& symbol, const name& ram_payer )
{
   require_auth( ram_payer );

   check( is_account( owner ), "owner account does not exist" );

   auto sym_code_raw = symbol.code().raw();
   stats statstable( get_self(), sym_code_raw );
   const auto& st = statstable.get( sym_code_raw, "symbol does not exist" );
   check( st.supply.symbol == symbol, "symbol precision mismatch" );

   accounts acnts( get_self(), owner.value );
   auto it = acnts.find( sym_code_raw );
   if( it == acnts.end() ) {
      acnts.emplace( ram_payer, [&]( auto& a ){
        a.balance = asset{0, symbol};
      });
   }
}

void token::close( const name& owner, const symbol& symbol )
{
   require_auth( owner );
   accounts acnts( get_self(), owner.value );
   auto it = acnts.find( symbol.code().raw() );
   check( it != acnts.end(), "Balance row already deleted or never existed. Action won't have any
effect." );
   check( it->balance.amount == 0, "Cannot close because the balance is not zero." );
   acnts.erase( it );
}

void token::setupminer(const name& user, const symbol& symbol){

   require_auth( user );

   auto sym_code_raw = symbol.code().raw();
   stats statstable( get_self(), sym_code_raw );
```

```
   const auto& st = statstable.get( sym_code_raw, "symbol does not exist" );
   check( st.supply.symbol == symbol, "symbol precision mismatch" );

   accounts acnts( get_self(), user.value );
   auto it = acnts.find( sym_code_raw );
   if( it == acnts.end() ) {
      acnts.emplace( user, [&]( auto& a ){
        a.balance = asset{0, symbol};
      });
   }


}

void token::claim(name from, name to, eosio::asset quantity, std::string memo)
{
   if (to != get_self() || from == get_self())
         return;


   accounts to_acnts( get_self(), from.value );
   auto tor = to_acnts.find( symbol_code("POW").raw() );
   check(tor != to_acnts.end(), "Must initialize POW before mining. Please use setupminer action
to enable mining");


   action{
         permission_level{get_self(), "active"_n},
         "eosio.token"_n,
         "transfer"_n,
         std::make_tuple(get_self(), from, quantity, std::string("Refund EOS"))
   }.send();

   int minetime = get_last_mine(get_self(), symbol_code("POW"));
   int currenttime = current_time_point().sec_since_epoch();
   int timepassed = (currenttime - minetime);

   asset supply = eosio::token::get_supply(get_self(), symbol_code("POW"));
   asset reward = get_reward(supply);
   asset balance = eosio::token::get_balance(get_self(), get_self(), symbol_code("POW"));


   if (timepassed >= 600){
```

```
    int rewardcount = timepassed / 600;
    asset issuereward = reward * rewardcount;

    action{
        permission_level{get_self(), "active"_n},
        get_self(),
        "issue"_n,
        std::make_tuple(get_self(), issuereward, std::string("Issue POW"))
      }.send();

    balance += issuereward;


    stats statstable( get_self(), symbol_code("POW").raw() );
    auto existing = statstable.find( symbol_code("POW").raw() );
    check( existing != statstable.end(), "token with symbol does not exist" );
    const auto& st = *existing;

      statstable.modify( st, same_payer, [&]( auto& s ) {
        s.minetime = currenttime;
      });

    }

    balance /= 40000;

    if (balance > asset(0, symbol("POW", 8))){
      action{
        permission_level{get_self(), "active"_n},
        get_self(),
        "transfer"_n,
        std::make_tuple(get_self(), from, balance, std::string("Mine POW"))
        }.send();

    }

  }

}
```